View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

provided by CiteSeerX

The Formal Verification of a Pipelined Double-Precision IEEE Floating-Point Multiplier

Mark D. Aagaard and Carl-Johan H. Seger
Dept. of Comp. Sci, Univ. of British Columbia
Vancouver B.C. V6T 1Z4 Canada

Abstract

Floating-point circuits are notoriously difficult to design and verify. For verification, simulation barely offers adequate coverage, conventional model-checking techniques are infeasible, and theorem-proving based verification is not sufficiently mature. In this paper we present the formal verification of a radix-eight, pipelined, IEEE double-precision floating-point multiplier. The verification was carried out using a mixture of model-checking and theorem-proving techniques in the Voss hardware verification system. By combining model-checking and theorem-proving we were able to build on the strengths of both areas and achieve significant results with a reasonable amount of effort.

1 Introduction

This paper describes the verification of a pipelined, IEEE compliant [7], double-precision floating-point multiplier. The features of the multiplier include:

- based on high-performance commercial designs from Digital Equipment Corp. [3]
- radix-eight multiplier array with carry-save adders
- round-to-nearest rounding mode
- optional non-IEEE-compliant mode: treats denormalised numbers as zero
- four stage pipeline
- three 56-bit carry-select adders with carry-propagate circuitry
- over 33 000 two-input gate equivalents

The top-level specification of the circuit is written in terms of arithmetic operations on integers. The design was done in structural VHDL, then synthesised to a unit-delay gate-level model using a cell-library. The verification was carried out in the Voss hardware verification system [9].

Voss includes an efficient implementation of ordered binary decision diagrams (BDDs); an event driven symbolic simulator with comprehensive delay and race analysis capabilities; a set of theorem-proving style inference rules; and a general purpose, functional programming language. The simulator implements symbolic trajectory evaluation, which offers a good compromise between expressibility of specifications and rapid verification. The inference rules allow the composition of verification results and support abstract data-types, such as integers. This enables Voss to overcome the limitations inherent in BDD-based model-checking.

Large parts of the IEEE floating-point standard have been formalised by Barrett [2] in the Z specification language and by Carreño and Miner [5] in the HOL and PVS theorem provers. Some work has recently been done in formally verifying complex integer circuits. Bryant and Chen have used Binary Moment Diagrams (BMDs) [4] to verify a sixty-two bit combinational multiplier. Claesen *et al.* and O'Leary *et al.* have used theorem provers to verify an SRT integer divider [10] and an SRT integer square-root circuit [8], respectively.

2 Background and Theory

In Voss, specifications consist of an antecedent, a consequent, and an optional relation (used for relational, but not functional, verification). Typically the antecedent is used to initialise inputs to the circuit. In functional verification, the consequent specifies the values of the outputs as functions of the inputs. In relational verification the relation gives the correctness condition in terms of the variables appearing in the antecedent and consequent. (Section 4.1 has an example of relational verification.) The antecedent and consequent are temporal formulas. The key to the efficiency of trajectory evaluation is the restricted language of the temporal formulas: there is no negation, the only temporal operator is "next", and there is only a restricted form of disjunction.

Hazelhurst and Seger [6] have defined a set of inference rules for composing verification results in Voss. These rules include: pre-condition strengthening, post-condition weakening, structural composition, and instantiation of symbolic and temporal variables.

One of the most powerful ramifications of these rules is that abstract data types (ADTs) for integers and other objects can be defined and related to primitive trajectory formulas. This allows specifications to be written in terms of the ADTs (*e.g.* integers). Verification can be carried out either by mapping the specification down to bit-vectors or by manipulating the ADTs. As an example of the second method, a linear-programming package has been added to Voss. Section 4.2 illustrates how we use arithetic decision procedures to simplify integer expressions. [1]

Trajectory evaluation is automatic, but can exceed the capacity of compute resources. In comparison, using inference rules requires human interaction, but is computationally less intensive. Our normal verification technique is to rely primarily on trajectory evaluation and use inference rules only when necessary. We use a mixture of top-down and bottom-up verification: top-down to isolate bugs and bottom-up to compose successful verifications. When doing divide-and-conquer verification with trajectory evaluation, only the specification needs to be partitioned. Trajectory evaluations can be carried out on the complete circuit, but only those parts related to the specification are exercised, which makes it very efficient.

3 Multiplication Implementation

The IEEE standard defines six different classes of floating-point data: infinity, normalised, denormalised, zero, quiet NaNs, and signalling NaNs. Multiplication is only performed if both operands are either normalised or denormalised. If multiplication is performed, the result may overflow, underflow, be normalised, or be denormalised.

IEEE floating-point numbers are represented as bit-vectors with three fields: sign, exponent and significand. *Denormalised* numbers (or denorms) represent values that, if normalised, would require that the exponent field be less than the minimum representable value.

Due to the cost in both area and performance required to support denormalised numbers in hardware, we rely on software support when a fully IEEE compliant result is needed. When our multiplier (which we call the "ADK" multiplier) is in IEEE compliant mode, it generates an *emulation exception* when multiplication is to be performed on a denorm input or when the result will be a denorm. (Underflows are always handled in hardware.) We improve performance in non-compliant mode by treating denorms as zeros.

The operations performed in each stage of the pipeline are summarised in Table 1.

Table 1: Pipeline stages and datapath operations

1.	**Significand**	Multiplier: Booth recode. Multiplicand: multiply by three
	Exponent	Add exponents together
	Special	Detect input denorms, infinities, NaNs, zeroes
2.	**Significand**	Multiplier array with carry-save adders
	Exponent	Subtract bias
3.	**Significand**	Add carry & sum vectors; normalise
	Exponent	Decrement if significand shifted
4.	**Significand**	Round-to-nearest; renormalise
	Exponent	Increment if significand shifted
	Special	Detect overflow, denorm out, underflow; set exception signals

Each row in the multiplier array calculates the product of a digit from the recoded multiplier and the multiplicand. Digits in the radix-eight recoded multiplier are in the range $-4\ldots+4$ and are in sign-magnitude format. The magnitude is used to select the desired multiple of the multiplicand. The sign is used to negate the product if needed. Negating the product is done by inverting it and including the extra "plus-one" needed for the two's complement in the initial partial product. The least-significant cell in each row computes the sticky and guard bits used in rounding.

4 Verification

Our verification of the multiplier relies on a hierarchy of specifications (Figure 1). IEEE Rel is a relational formalization of the IEEE standard for multiplication (including NaNs, infinities, *etc.*). ADK Rel is a relational specification of our multiplier. It is identical to IEEE Rel except for those cases where our multiplier raises an emulation exception. ADK Fun is a functional specification describing exactly what result our multiplier should produce for any set of inputs. Below ADK Fun are specifications for subparts of the circuit, such as the Booth recoder and rounding circuitry.

Our formalization of the IEEE standard is *relational*, not *functional*. The IEEE standard is non-functional in several cases, in that it specifies properties that the result must satisfy but not the exact value that must be produced. For example, if one of the operands is a

[1] We have not formally verified the ADTs or the linear programming package. Nonetheless, these techniques offer a practical and relatively sound method for manipulating integer and Boolean expressions

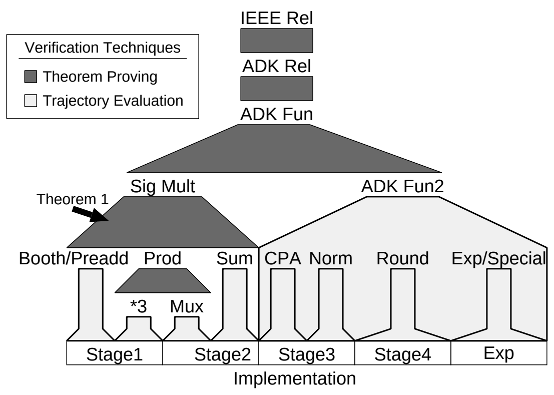

Figure 1: Hierarchy of specifications

Table 2: Design and verification effort

	Des	Spec	Ver	Rdes	Tot
ADK Fun2	—	3.8	2.2	6.0	12.0
Sig Mult	—	2.0	3.0		5.0
Booth	1.0	1.5	0.5		3.0
Prod/Sum	7.0	5.0	5.5	5.0	22.5
Carry-Prop Add	1.5	0.5	0.3		2.3
Normalization	0.1		0.2		0.3
Rounding	1.0		0.3		1.3
Exp/Special	6.6		4.6	1.0	12.2
Interconnect	1.7		3.0		4.7
Total	18.9	12.8	19.6	12.0	63.3

Des	Initial Design	Ver	Verification/bug fixes
Spec	Specification	Rdes	Redesign

All times are measured in workdays

NaN, the standard requires that result is a NaN, but it does not specify which NaN.

The IEEE standard is informal and written in natural language, so formalizations of the standard can only be verified against it informally. Our formalization [1] is only a few pages and (we believe) quite readable. Thus, we claim that others can inspect our formalization and convince themselves that it conforms to the standard.

We began the verification by using test-vector simulation as a quick and effective way of catching many bugs and then relied on trajectory evaluation to verify individual components (the lowest layer in Figure 1). Once the components had been verified, we needed to compose the results to verify the complete circuit. We used a single trajectory evaluation to verify the significand datapaths in stages three and four and Exp/Special against ADK Fun2. We used inference rules to combine the verification results from Booth/Preadd, Prod, and Sum to prove that the circuit multiplies correctly (Sig Mult). The verification results for Sig Mult and ADK Fun2 were combined together using inference rules to complete the verification of the multiplier against ADK Fun. We used BDDs and inference rules to verify that ADK Fun implies ADK Rel and ADK Rel implies IEEE Rel.

The total design and verification effort took approximately seventy work days (Table 2). At the time of writing, some of the theorem-proving parts of Voss are still evolving. A few aspects of the verification of ADK Fun and ADK Rel are not complete and are not included included in Table 2.

Each of the trajectory evaluations for the lowest level specifications took under a minute on a Sparc 10/51 with 64M of memory. ADK Fun2 required approximately ten minutes. The automated decision procedures used in the theorem proving verification ran in under three minutes each. Variable re-ordering was automatically done once for each of the trajectory evaluations in Figure 1. Most runs took several hours on a DEC 3000 with 512M of memory.

In Sections 4.1 and 4.2 we briefly describe the verification of the Booth recoder and significand multiplication datapath. These examples are two of the most complicated verifications. They illustrate the use of relational verification and arithmetic decision procedures respectively.

4.1 Booth Recoder

The Booth recoder in stage one was verified against the relational specification in Equation 1. The input is the multiplier (m) and the outputs are eighteen sign-magnitude digits in the range $-4...+4$ (sgn_i and mag_i for $0 \leq i \leq 17$). A functional specification would require separate equations defining sgn_i and mag_i in terms of m, which would clearly be much more difficult to write than the relational specification.

$$m = \sum_{i=0}^{17} 8^i * (1-2*sgn_i)*mag_i \qquad (1)$$

4.2 Significand Multiplication

Theorem 1 says that the composition of the specifications for the components in the significand datapaths in stages one and two implies that the sum of the carry (C) and sum (S) vectors output from the multiplier array is the upper fifty-five bits of the product of the multiplier (M_1) and multiplicand (M_2). The theorem was proved automatically by Voss' arithmetic decision procedures in three minutes.

The first line describes the Booth recoding of the multiplier (M_1). The second line is for the preaddition

Theorem 1: *Composition of significand specifications*

$$\vdash \left(\sum_{i=0}^{17} 8^i * (1-2*sgn_i)*mag_i = M_1 \right) \wedge$$

$$\left(P = \sum_{i=0}^{17} 8^i * (sgn_i) \right) \wedge$$

$$\left(p_i = M_2 * (1-2*sgn_i)*mag_i - sgn_i \right) \wedge$$

$$\left(C + S = p_{17} + (p_{16} + (\cdots(p_0+P)/8\cdots)/8) \right) \Longrightarrow$$

$$\left(C + S = (M_1 * M_2)/2^{51} \right)$$

of the plus-ones into the initial partial product (P) for the generation of two's complement products in the multiplier array (see Section 3). The third and fourth lines describe the calculation of the product terms (p_i) and the summation of the partial products using carry-save addition (C and S).

5 Conclusion

As shown in Table 2, we found many bugs, both in our design and specifications. Many of these could have been found through extensive use of test-vectors, but it is doubtful that they could have been found as quickly as with trajectory evaluation. Most of the bugs were related to the multiplier array or the special cases. Because of the regularity of the multiplication implementation, we were able to find many of the bugs using test vectors. However, the control circuitry for the special cases is very irregular, making test vectors impractical. Our most subtle bug illustrates the need for relational and high-level specificiations. We were very confident in our specification ADK Fun, but verifying it against ADK Rel revealed that a particular NaN value would sometimes produce a result of infinity, rather a NaN. This error was in both our implementation and functional specification and would very likely have remained undetected in test-vector simulation.

Our long-term goal is to develop practical and rigourous formal-verification techniques. From experience with a variety of model-checking and theorem-proving techniques, we have concluded that trajectory evaluation and built-in support for debugging hardware is a very effective verification process. Composing verification results using both trajectory evaluation and inference rules provides the freedom to choose the most appropriate technique for each situation. Using a general-purpose programming language as an interface makes it easy to automate repetitive tasks and customize interfaces. More experience with combined model-checking and theorem-proving based verification is clearly needed, but even at this early stage, we are very optimistic that the combination offers the promise of practical formal verification, scalability, and high-level specifications.

References

[1] M. D. Aagaard and C.-J. H. Seger, "The design and verification of a radix-eight, pipelined, IEEE double-precision floating-point multiplier," tech. rep., Dept. of Comp. Sci, Univ. of British Columbia, 1995.

[2] G. Barrett, "Formal methods applied to a floating-point number system," *IEEE Trans. Soft. Eng.*, vol. 15, no. 5, pp. 611–621, 1989.

[3] B. J. Benschneider, *et al.* , "A pipelined 50-MHz CMOS 64-bit floating-point arithmetic processor," *IEEE Jour. of Solid-State Circuits*, vol. 24, pp. 1317–1323, Oct. 1989.

[4] R. E. Bryant and Y.-A. Chen, "Verification of arithmetic functions with binary moment diagrams," Tech. Rep. CMU / / CS-94-160, Dept. of Comp. Sci, Carnegie-Mellon Univ. Aug. 1994.

[5] V. A. Carreño and P. S. Miner, "Specification of the IEEE-854 floating-point standard in HOL and PVS," in *Higher Order Logic Theorem Proving and Its Applications*, Sept. 1995.

[6] S. Hazelhurst and C.-J. H. Seger, "A simple theorem prover based on symbolic trajectory evaluation and BDDs," *IEEE Trans. on CAD*, Apr. 1995.

[7] IEEE, *IEEE Standard for binary floating-point arithmetic.* ANSI / IEEE Std 754-1985, 1985.

[8] J. W. O'Leary, M. E. Leeser, J. Y. Hickey, and M. D. Aagaard, "Non-restoring integer square root: A case study in design by principled optimization," in *Theorem Provers in Circuit Design*, Springer Verlag; New York, Sept. 1994.

[9] C.-J. Seger, "Voss — A formal hardware verification system user's guide," Tech. Rep. 93-45, Dept. of Comp. Sci, Univ. of British Columbia, 1993.

[10] D. Verkest, L. Claesen, and H. De Man, "A proof of the nonrestoring division algorithm and its implementation on an ALU," *Formal Methods in System Design*, vol. 4, pp. 5–31, Jan. 1994.

Acknowledgments

We would like to thank Mark Greenstreet, Scott Hazelhurst, Catherine Leung, Andy Martin, David Weih, and the Semiconductor Engineering Group at Digital Equipment Corp. This research was supported by operating grant OGPO 109688 from the Natural Sciences and Engineering Research Council of Canada, a fellowship from the B.C. Advanced Systems Institute, Research Contract DJ-295 from the Semiconductor Research Corporation, and equipment grants from Sun Microsystems Inc, Canada, and Digital Equipment Corp, Canada.