Formal Verification of Iterative Algorithms in Microprocessors

Mark D. Aagaard, Robert B. Jones, Roope Kaivola,
Katherine R. Kohatsu, Carl-Johan H. Seger
Intel Corporation, Hillsboro, Oregon, USA

Abstract

Contemporary microprocessors implement many iterative algorithms. For example, the front-end of a microprocessor repeatedly fetches and decodes instructions while updating internal state such as the program counter; floating-point circuits perform divide and square root computations iteratively. Iterative algorithms often have complex implementations because of performance optimizations like result speculation, re-timing and circuit redundancies. Verifying these iterative circuits against high-level specifications requires two steps: reasoning about the algorithm itself and verifying the implementation against the algorithm. In this paper we discuss the verification of four iterative circuits from Intel microprocessor designs. These verifications were performed using Forte, a custom-built verification system; we discuss the Forte features necessary for our approach. Finally, we discuss how we maintained these proofs in the face of evolving design implementations.

1 Introduction

In this paper we describe the verification of Intel circuits with three attributes that make them particularly difficult to verify: *full-custom industrial circuits* from *active designs* that implement *iterative algorithms*. We have verified these circuits with Forte, our custom-built verification system[1].

In full-custom industrial design flows, a great deal of energy is expended to maximize performance while efficiently using area and power. Achieving these goals relies on optimizations such as result speculation, re-timing and redundant circuitry. These optimizations make verification more difficult because the semantic gap between an "obviously correct" specification and the actual implementation. The complexity of the circuits and the algorithms they implement mean that circuits within the capacity of fully automated model checkers are too small to contain clean interfaces or behavior that can be concisely specified.

Much verification research is intended to reduce the time and memory resources needed to verify correct implementations. Our experiences have demonstrated that optimizing for the infrequent case when a circuit is finally "verified" is of little value in improving productivity. The quality of capabilities of CAD tools for exploring the structure and behavior of new circuits, debugging counter-examples, and regression testing (re-running verification scripts after design modification) are as important as the raw power of verification engines. This is particularly true for active designs, which are subject to frequent change and often poorly documented.

Verifying a circuit that implements an iterative algorithm can be decomposed into three phases: verifying the circuit against a low-level invariant that is designed for model-checking efficiency; relating the low-level invariant to a higher-level, more natural invariant; and applying an inductive argument to prove that the high-level invariant guarantees the input-output behavior of the circuit. Iterative circuits are particularly challenging to verify because they require the construction of invariants that are often difficult to find and that bridge a large semantic gap. Bridging this gap between the natural specification of input-output behavior and a specification that is within the capacity limits of model checking requires that model-checking results be related to the theorem proving environment where abstraction and induction can be used to carry out the higher levels of the verification.

In the process of verifying iterative algorithms on Intel microprocessors, we have built an effective verification system called Forte. We have identified some important features of an effective industrital verification system:

- a judiciously chosen collection of high-capacity verification engines with complementary strengths,
- a general-purpose programming language that glues the verification engines together and acts as the primary user interface to the system,
- a seamless connection between model checking and theorem proving,
- extensive support for behavioral and structural queries of circuits, and
- integrated symbolic simulation and visualization of circuit behavior and structure.

This paper is about verifying iterative circuits, not the Forte system. However, we discuss the relevant features of Forte that enabled these verifications.

In Section 3 we describe the use of customized and automated reachability analysis to compute an over-approximated invariant for a floating-point square-root circuit. This circuit had resisted several months of effort that involved fully automated reachability analysis and manual invariant finding. Section 4 uses a floating-point remainder circuit to demonstrate how the model checking and theorem-proving facilities of Forte were combined to create an application-specific environment for pre-postcondition style verification of iterative algorithms. This verification approach abstracts the proof above the hardware level and creates a more natural, algorithmic verification. Section 5 describes a previously-verified circuit that marks the boundaries between IA-32 instructions [1].

[1] Forte evolved from Voss, a verification system developed by Seger [19].

DAC 2000, Los Angeles, California

This circuit implements an algorithm that iterates over a stream of bytes, in contrast with the loop counter used by the floating-point circuits. We describe our experience porting the verification script to a new microprocessor design and maintaining the verification as the circuit implementation evolved. The effort required to port and maintain the verification script was greatly mitigated by the high capacity of our verification engines and by techniques that minimize implementation-dependent details in the verification script.

2 The Forte System

Symbolic trajectory evaluation (STE) [20] is the core of Forte. It is used as a model checker and as a symbolic simulation engine. The STE model checker verifies that if a finite-length circuit behavior satisfies given assumptions, it also satisfies given specification obligations. STE is the primary semantic link between the circuit model and all of the verification algorithms in Forte. Using a single source for information on circuit behavior helps ensure that all of the formal verification algorithms use the same semantics for circuits.

The Forte scripting language, FL, is a functional language in the ML family that includes binary decision diagrams (BDDs) as first-class objects. Forte includes a connection between FL and Tcl/Tk. Forte components are typically implemented in FL using STE for symbolic simulation and including graphical interfaces via the Tcl/Tk connection. To illustrate, consider the combination of FL and STE for a reachability-based model checker. STE is used as a symbolic simulator to extract the transition relation for the circuit. Because BDDs are first-class objects in FL, computing a fix-point for a transition relation requires only a few lines of FL code.

A continuing focus in the evolution of Forte is an extensive suite of graphical and textual debugging tools. Our experiences with real-world verification problems have heavily influenced this aspect of Forte. We have discovered that it is surprisingly difficult to anticipate which debugging features users will find most useful and which will remain largely unused. Some of the heavily-used capabilities we have implemented are an integrated circuit browser and waveform viewer, facilities for structural queries on circuits (computing transitive fanin/fanout, determining the type of a combinational gate or latch, etc.), BDD-based counter-example analysis, and a theorem proving proof visualizer.

FL supports a form of reflection that we call "lifting" [3]. Lifted FL allows users to access abstract syntax trees generated by the FL parser. FL is the specification language for STE. Lifted FL is the specification, or term, language for ThmTac, the proof tool in Forte. Lifting a functional language has allowed us to seamlessly integrate theorem proving with multiple model checkers without compromising model-checking usability or efficiency. This results in an unprecedented ability to verify complex RTL hardware against natural, high-level specifications. Additionally, theorem proving increases user confidence that combinations of model checking results are stitched together correctly.

3 Floating-Point Square Root (FSQRT)

Most IEEE-compliant floating-point divide and square-root algorithms are iterative [12]. Hardware implementing such algorithms performs three distinct operations: initialization, iteration, and rounding. During initialization the circuit loads predefined values into registers and configures control circuitry. The data computation is performed iteratively, where the number of iterations may be fixed or data dependent. Finally, the result of the iteration is passed to a rounding circuit that produces the IEEE-compliant result.

A number of case studies on the verification of division and square root circuits have been reported previously [4, 17, 5, 8, 7, 15, 18, 16]. Model checking approaches that verify actual implementations (instead of an abstraction) verify a high-level invariant or verify an invariant recurrence relation. Regardless of the model-checking technique, success depends on finding a suitable invariant. However, it is considerably more difficult to construct the appropriate invariant for complex designs with parallelism, speculation, and other optimizations. The square-root verification we report demonstrates a novel technique for finding invariants where both manual and automated techniques have previously failed.

Intuitively, the overall input-output correctness of the square root circuit can be expressed as: *if N is the input radicand and W is the output computed by the circuit, then W is $\sqrt{N}$ rounded correctly*. Our top-level specification is a formalization of this statement. Although superficially simple, the formalization is non-trivial and requires great care and floating-point expertise. In particular, "rounded correctly" is a fairly involved statement. The approach we took in our verification follows closely the work in [16].

The high-level specification is naturally decomposed into two requirements. First, the final unrounded root computed by the loop in the circuit, Z_{fin}, bounds the specified result as: $Z_{fin} \leq \sqrt{N} < Z_{fin} + \varepsilon$, where ε depends on the precision of the calculation. Second, the circuit rounds Z_{fin} correctly to produce W.

At the highest level of the loop computation, there is a natural loop invariant that relates the partial root Z_i, remainder R_i, and iteration number i as $N = Z_i * Z_i + 2^i R_i$. The partial root is the root of N, after i iterations, while the remainder is the remainder computed after iteration i. The multiplication operator in this high-level invariant means that verifying this property by direct model-checking is difficult. Hence, we decompose the problem by introducing two intermediate properties. The lowest level property is a bit-vector invariant that is optimized for model checking efficiency. The second property is the recurrence relation that the loop is supposed to compute, *i.e.*, an equation relating current and previous loop values.

This decomposition divides the verification task into six steps:

1. Use model checking to prove that the circuit satisfies a low-level invariant over bit-vectors that is optimized for model checking efficiency.
2. Prove that the low-level bit-vector invariant implies a numerical recurrence relation.
3. Prove that the numerical recurrence relation maintains a high-level invariant.
4. Prove that the high-level invariant guarantees that the final result emerging from the loop is the correct (unrounded) result.
5. Prove that the circuit rounds the final result correctly.
6. Prove that the final result from the loop and the rounded result is IEEE-compliant.

There is typically a large semantic gap between the bit-level properties in step 1 and the natural high-level invariant in step 4. Step 2 requires reasoning about the correspondence between bit-vector operations and their numerical counterparts. Steps 3 and 4 rely purely on arithmetic reasoning. Section 4 discusses how all six tasks were accomplished within the Forte system, although steps 5 and 6 were vacuous for the floating-point remainder operation.

A more detailed flow of the six sub tasks relating to the square root verification is shown in Figure 1. Step 1 was carried out by using STE. Step 2 was done by manually inspecting the properties that were model checked against the properties defined in the theorem prover. Steps 3 and 4 were originally accomplished with a detailed hand-proof, but to increase our level of confidence, were later translated into a mechanized proof system and verified.

Completing step 5 consists of connecting the unrounded result Z_{fin} produced by the loop with the IEEE-compliant rounded result and comparing that to the value returned by the circuit. We accomplished this using a combination of arithmetic and bit-level reasoning. Since the rounder is not an iterative algorithm and the proof is fairly routine, this verification will not be discussed further.

We now go into some detail on step 1 for the square root verification. The discussion of the floating-point remainder in Section 4 focuses on steps 2–4.

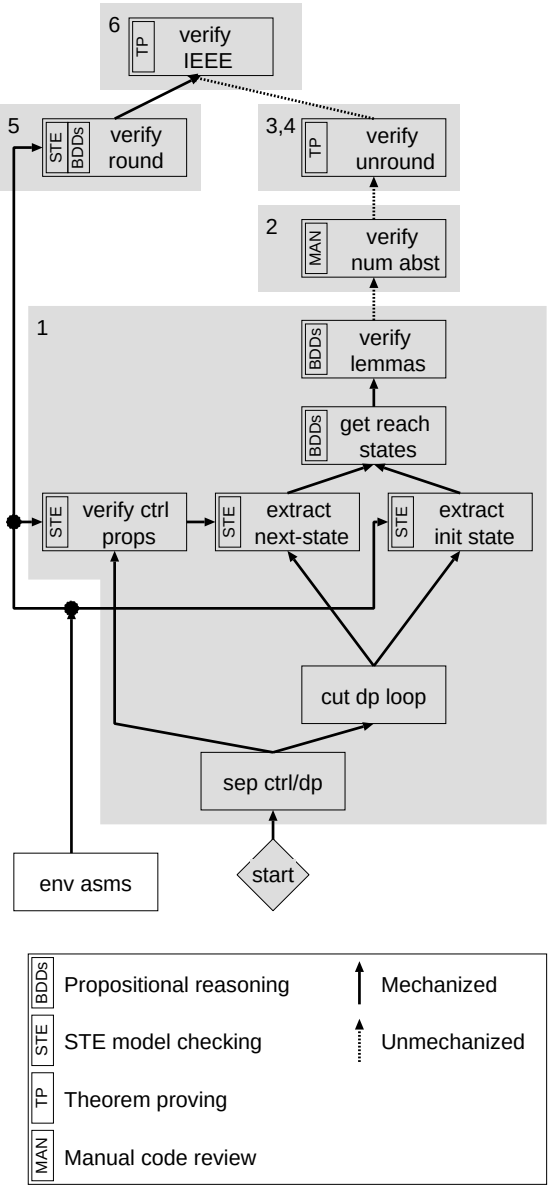

Figure 1: Outline of FSQRT Verification

Step 1, verifying that the circuit satisfies the bit-level properties, was accomplished by using model-checking to verify that the circuit satisfies two lemmas:

1. The next remainder and next partial root are particular functions of the previous remainder, the previous partial root, and the next root bits.
2. After each iteration, the remainder is within certain bounds.

The first property verifies that the algorithm is implemented correctly and the second verifies that the algorithm converges. Note that our decomposition does not require that we specify the algorithm used for selecting the root bits. By not specifying this, the verification becomes more robust to implementation changes.

Before starting step 1, some environmental assumptions were made about this circuit (see Figure 1). For example, we assumed the circuit was not being reset, it was given a square root instruction, and the square root completed. These assumptions only dealt with primary inputs to the circuit. Using the STE engine of Forte, they were then used to verify some internal control properties, which were then used to compute the initial state of the iteration loop.

Next, we partitioned the iteration circuit into control and data path. The cyclic data path was further cut at locations with relatively clear algorithmic definition. STE was then used as a symbolic simulator by invoking it using internal control node assumptions and the current state of the cut points as antecedents. STE thus computed the BDDs representing the next-state functions on the data path cut nodes within the iterative loop. These next-state extractions were performed on all versions, formats, and rounding modes of the square root instruction. By using the same BDD variables for each run, we compared these next-state functions with a simple equivalence check. The check proves that the next-state functions for the iterative part of the loop is the same for all modes of square root. With this proof of equivalence of the loop's next-state functions, we only had to prove the lemmas for one version of square root. Illustrating the flexibility of Forte, performing this equivalence check for about 400 signals required two lines of FL code.

By using STE, we were able to compute the next state functions for *any cut of the data path*, rather than being limited to the traditional latch boundaries. This made it possible to re-align parts of the computation that had been implemented out-of-phase with each other. Secondly, this reduced the number of "state holding" variables needed, since it was possible to avoid cutting in the middle of speculated data. Finally, this allowed the next state functions to more closely match the algorithm. All this contributed greatly to the feasibility of the remaining verification tasks.

For the STE extraction above to be possible, a number of assumptions were made on the behavior of internal control and clock signals that were inputs to the data path. Following an "assume-guarantee" approach, these internal assumptions were verified as consequents of the environmental assumptions. These control and clock signals are internal signals with complex, poorly documented behavior. The structural query facilities and integrated simulation and visualization environment of Forte were used extensively for this part of the verification.

Referring to Figure 1, the next step in the verification flow after "extract init state" and "extract next state", is to compute the reachable state space. The square root implementation is complex due to the use of redundant circuitry and result speculation to improve performance. Finding the relationships between the redundant pieces of circuitry is critical because none of the lemmas will hold without them. Due to the very large size of the iterative data path (close to 1k latches), using a traditional symbolic model checker, in which the fix-point computation derives the reachable state space automatically, was not feasible. The complexity of the implementation also defied several months of work aimed at manually constructing an explicit invariant.

Our solution was to compute several over-approximations of the reachable state space. Each of these approximations focused on some particular aspect of the circuit. For example, one of the

approximations focused on the relation between the root selection logic and the most significant digits of the remainder. Another characterized a register that was used as a one-hot counter. A final over-approximation for the reachable state space was obtained by intersecting these approximations with BDD operators. Since we were only interested in verifying safety properties, this approach is safe (no false positives)[2].

Finally, with an over-approximation of the reachable state space computed, completing step 1, the verification of the two lemmas, was straightforward. Using BDD operations, we proved the base and inductive cases of the range recurrence relations: first, the initial state of the data path satisfies the invariant. Secondly, assuming that the current state satisfies the invariant and that the current state is within the reachable state space, the next state, as computed by the extracted next state functions, also satisfies the invariant.

4 Floating Point Division and Remainder

In Section 3, we concentrated on the model-checking aspects of verifying an iterative square-root circuit. In this section we describe the verification of a floating-point division and remainder circuit and focus on steps 2, 3 and 4 in the proof, *i.e.*, composing the model-checked invariant lemmas to prove the overall input-output correctness of the circuit. The primary purpose of the circuit is division computation, and it could be argued that the remainder is simply a by-product of this. Nevertheless, here we concentrate on the remainder calculation, as this allows us to ignore the largely orthogonal issue of specifying formally what correct rounding means. The techniques discussed here are applicable to division, as well, so the choice is merely a matter of presentation.

Although the square-root proof reported in Section 3 was carefully scrutinized and many aspects of the proof machine-checked , we were concerned about the use of bit-vector operations to justify statements about the numerical recurrence relation. Previous work [1, 2, 3, 16] had demonstrated Forte's capability in combining theorem proving and model checking for unified top-to-bottom verifications. The goal of this floating-point remainder verification was to use the pre-postcondition reasoning technique from sequential program verification to create a natural proof. This approach results in an attractive connection between the overall correctness criteria and the circuit: when input-output correctness is expressed as a pre-postcondition pair, the statement is in a form that is directly model-checkable—given unlimited resources and access to true mathematical operations.

To illustrate our approach, consider an iterative circuit as a sequential program. Pre-postcondition triples $\{P\}S\{Q\}$ (where P and Q are logical properties and S is a program) formalize the statement: *if the execution of program S starts in a configuration satisfying P, then the execution terminates in a finite time and leads to a configuration satisfying Q* [10, 13]). Pre-postconditions are quite amenable to theorem-proving and share similarities with the inference rules about STE [11]. This led us to create an application-specific extension to Forte for theorem proving and STE model checking with pre-postcondition statements.

Verifying a loop in the pre-postcondition paradigm involves formulating an explicit loop invariant and verifying that the invariant is true in each loop iteration. For the remainder circuit, we wrote two concise bit-vector invariants that could be verified directly against the circuit. These properties are: (1) a bit-vector version of the recurrence relation the circuit should compute and (2) a low-level bit-vector invariant that limits the range of values manipulated by the loop. The second invariant also keeps track of various implementation-related values that are necessary for the proper operation of the loop.

A direct proof of the bit-level properties against the numerical recurrence relation would require tedious reasoning to show that the bit-vector operations do not overflow. To improve productivity for future verifications, we created a bit-vector arithmetic library where each operation is augmented with a validity bit that indicates overflow and loss of precision. We used ThmTac, the Forte theorem prover, to prove general theorems that relate bit-vector properties to properties about unbounded integers.

To connect the pre-postcondition approach with the circuits, we used trajectories to bind input and output variables to circuit nodes and computed the relation between inputs and outputs in the circuit using STE with parametric representations [2]. Verifying that each precondition-satisfying input leads to a postcondition-satisfying output was accomplished with standard BDD operations. This process was fully encapsulated in FL functions, enabling the user to work exclusively at the level of pre-postcondition statements. No direct interaction with STE was necessary. Because the underlying verification engine is STE, all of the standard Forte debugging facilities were available within this pre-postcondition environment.

The pre-postcondition environment we created provides familiar inference rules like sequential composition, precondition strengthening, and postcondition weakening. These rules were formally derived from a few axioms about STE using first-order logical inference rules in ThmTac.

The number of loop iterations that can occur in the circuit is bounded. We exploited this fact to simplify the verification of the control circuitry; we were able to use sequential composition instead of true induction. This is similar to the approach taken in the square-root verification of the previous section, with the difference that the square root reasoning was done with BDD operations in an FL script rather than inside ThmTac.

Since the completion of this remainder verification, our pre-postcondition framework has been applied to other division operations. These subsequent experiences indicate that the approach described here is quite robust—it can be applied to a variety of classes of iterative circuits. We are currently investigating an extension of the floating-point remainder proof "downwards" to include reasoning about special-case control circuit behaviors like reset.

5 IA-32 Instruction-Length Decoder

We have previously verified an 11,000-gate IA-32 instruction-length decoder [1]. Since that time, we have ported the verification to a new microprocessor and maintained the script during the evolution of the new design. The length decoder (IM) operates iteratively over fixed-length sequences of bytes containing IA-32 instructions. The input is packet of bytes to decode, the output is a bit-vector marking the boundaries between IA-32 instructions. The internal state is a *wrap pointer* that keeps track of the byte offset in the subsequent packet where the next IA-32 instruction begins.

The IM top-level specification describes the expected values of the primary outputs and wrap pointer over an unbounded sequence of inputs. The decoder circuit iterates over unbounded sequences, in constrast to the fixed iteration counts of the arithmetic examples in the previous sections. Following our standard approach, the IM verification was decomposed into verifying certain invariant properties of the circuit and then using the invariant with induction to verify the input-output behavior of the circuit. The IM verification used *data-space decomposition*, instead of multiple levels of invariants. The range of possible values for the primary inputs and internal state were decomposed into 28 different cases, each of which

[2] The idea of using multiple over-approximations is not novel [9, 6, 14]. The novel aspect is the ease with which a user of Forte can do this type of model checking and combine the results.

Circuit	Latch Count	Verification Time	Peak BDD Size
FDIV	7 k	8 hr	44 MB
FSQRT	7 k	24 hr	56 MB
FPREM	4 k	36 hr	20 MB
IM	2.5 k	10 hr	6 MB

Table 1: Verification results

was verified with an STE run.

Two facets of formal verification that are rarely encountered outside of an industrial setting are script reuse in new implementations and regression over design changes. In general, a major boost in verification productivity could be realized if verification scripts from earlier designs could be reused on new circuits implementing similar functionality. For this to be feasible, scripts should reason about circuit *functionality*, rather than circuit *implementation*. For example a decomposition that is based on input or data-space partitioning is generally more portable than one based on structural decomposition.

The IM verification was ported to a new processor design that used a completely different algorithm. The only implementation similarity between the two circuits was the use of a wrap pointer. While the initial IM specification and verification took almost six months to develop, the port to the new design took less than two weeks. This remarkable result was possible because the verification focused on input-to-output verification: the signals in the wrap pointer are the only internal signals mentioned in the script.

The most difficult parts of the initial verification (formulating the induction strategy, identifying the key pieces of internal state for the invariant, and carrying out the theorem proving) did not need to be redone for the new implementation. Only the implementation verification by model checking had to be performed again.

In an active design environment, a successful verification is not the end, but just another milestone. Verification scripts need to be be *regressed* against the design changes at regular intervals. This helps to ensure that previously verified functionality is not broken by bugs introduced as the design evolves toward completion. Our experience shows that this process can be a significant overhead in the verification effort, sometimes one-third of the total effort.

The input-to-output nature of the IM verification and Forte's debugging facilities were instrumental in achieving a major reduction in regression overhead. The small number of internal signals mentioned in the script meant that most implementation changes that did not affect high-level functionality also did not affect the script. This eliminated a great deal of effort that is typically spent chasing false negatives during regression. When a regression run failed, the locations of the relevant design changes were easily identified with Forte's circuit and waveform visualizers. Once the location of the changes had been found, the the Fortecircuit query support made it straightforward to determine new signal names and timing.

The initial IM verification yielded eight first-found bugs. Regular regression testing has resulted in discovery of three additional bugs that were introduced during circuit changes. Only one logic bug has been found in the decoder data-path using other techniques; it was discovered between verification regressions. When the regression was run, the same bug was detected and the bug fix was then later confirmed.

6 Conclusion

Each of the verifications discussed in this paper earlier were performed directly on the RTL description of active Intel designs. In table 1, we summarize the designs and verification effort. The verifications were performed on .

The difficulty in verifying an iterative algorithm implemented in hardware is that it requires reasoning about both the algorithm and its implementation. Finding a succesful verification strategy requires a solid understanding of the underlying verification tools as well as intimate knowledge of the algorithm being verified. For example, it was important to be able to reliably predict whether a particular part of an algorithm was likely to be verifiable by direct model checking or would require decomposition, and then picking a succesful decomposition strategy.

One somewhat suprising lesson was that we often were required to focus on a relatively small part of a circuit in great detail, while large sections could be treated as black boxes. A few bits of redundant information in the square-root circuit, and a change to their implementation part way through the verification effort, caused more than a month of intense work and detailed scrutiny.

The complexity of the implementations and the lack of precise documentation led us to sometimes reverse engineer the intended behavior. In one example, some internal state was supposedly represented in a one-hot encoding, but only after debugging a number of perplexing counterexamples did we discover that there were a few special cases in which all of the bits in the internal state were intended to be turned off.

Despite the challenges of verifying hardware of this complexity against high-level specifications, we are quite optimistic about the continued growth of formal verification. We believe that several key aspects of the Forte system make it useful for this type of verification and can point the way for further evolution in verification tools. Perhaps two most important features of Forte are the use of an general-purpose, functional language as its interface and the choice of symbolic trajectory evaluation as the underlying model checking engine.

Having a general-purpose programming language as the interface has provided the infrastructure for building application-specific verification environments on top of a very general system. This has increased automation and made it easier for new users to learn the system. The fact that the language is interpreted means that the system is interactive, rather than batch mode, which greatly improves productivity. Having the programming language be a functional language enable us to implemement a theorem prover in the same environment that model checking occurs.

Symbolic trajectory evaluation's success as a model checking engine for datapath intensive circuits has been well publicized. However, we are still discovering new benefits to its use as a general purpose symbolic simulation engine. Using trajectory evaluation to extract out the initial state and next-state functions is only one example. Symbolic simulation also plays an integral role in our counter-example analysis work, where the link between simulation and the graphical interface for circuit browsing and waveform display has revolutionized the way we approach debugging.

Acknowledgments

We thank John Harrison for mechanizing the hand proof of FDIV and FSQRT and Bob Brennan for providing the opportunity to carry out these case studies. John O'Leary contributed to this work during insightful discussions on rounding, division, and square-root verification.

References

[1] M. D. Aagaard, R. B. Jones, and C.-J. H. Seger. Combining theorem proving and trajectory evaluation in an industrial environment. In *DAC*, pages 538–541. ACM/IEEE, July 1998.

[2] M. D. Aagaard, R. B. Jones, and C.-J. H. Seger. Formal verification using parametric representations of Boolean constraints. In *DAC*, July 1999.

[3] M. D. Aagaard, R. B. Jones, and C.-J. H. Seger. Lifted-fl: A pragmatic implementation of combined model checking and theorem proving. In L. Thery, editor, *Theorem Proving in Higher Order Logics*. Springer Verlag; New York, Sept. 1999.

[4] G. Barrett. Formal methods applied to a floating-point number system. *IEEE Trans. on Soft. Eng.*, 15(5):611–621, May 1989.

[5] R. E. Bryant. Bit-level analysis of an SRT divider circuit. In *DAC*, pages 661–665, New York, June 1996. ACM.

[6] J. R. Burch. *Trace Algebra for Automatic Verification of Real-Time Concurrent Systems*. PhD thesis, Comp. Sci. Dept, Carnegie Mellon Univ., 1992.

[7] Y.-A. Chen, E. Clarke, P.-H. Ho, Y. Hoskote, T. Kam, M. Khaira, J. O'Leary, and X. Zhao. Verification of all circuits in a floating-point unit using word-level model checking. In M. Srivas and A. Camilleri, editors, *Formal Methods in CAD*, volume 1166 of *LNCS*, pages 19–33, Palo Alto, CA, USA, Nov. 1996.

[8] E. M. Clarke, S. M. German, and X. Zhao. Verifying the SRT division algorithm using theorem proving techniques. In Rajeev Alur and Thomas A. Henzinger , editors, *CAV*, volume 1102 of *LNCS*, pages 111–122, New Brunswick, NJ, USA, July 1996.

[9] E. M. Clarke, O. Grumberg, and D. E. Long. Model checking and abstraction. *ACM Trans. on Prog. Lang. and Systems*, 16(5):1512–1542, Sept. 1994.

[10] D. Gries. *The Science of Programming*. Springer-Verlag, 1981.

[11] S. Hazelhurst and C.-J. H. Seger. Symbolic trajectory evaluation. In T. Kropf, editor, *Formal Hardware Verification*, chapter 1, pages 3–78. Springer Verlag; New York, 1997.

[12] J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann Publishers, Inc., 1990. Second edition, 1995.

[13] A. Kaldewaij. *Programming: The Derivation of Algorithms*. Prentice-Hall, 1990.

[14] R. P. Kurshan. *Computer-Aided Verification of Coordinating Processes: The Automata-Theoretic Approach*. Princeton Univ. Press, 1994.

[15] J. S. Moore, T. W. Lynch, and M. Kaufmann. A mechanically checked proof of the AMD K-5 86 floating point division program. *IEEE Trans. on Comp.*, 47(9):913–926, Sept. 1998.

[16] J. O'Leary, X. Zhao, R. Gerth, and C.-J. H. Seger. Formally verifying IEEE compliance of floating-point hardware. *Intel Technology Journal*, Q1, Feb. 1999.

[17] 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.

[18] D. M. Russinoff. A mechanically checked proof of IEEE compliance of the floating point multiplication, division and square root algorithms of the AMD-K7 processor. *J. of Comp. Math.*, 1:148–200, 1998. London Mathematical Soc.

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

[20] C.-J. H. Seger and R. E. Bryant. Formal verification by symbolic evaluation of partially-ordered trajectories. *Formal Methods in System Design*, 6(2):147–189, Mar. 1995.