Proving the Correctness of a Complete Microprocessor

Christian Jacobi, Daniel Kroening*

Dept. 14: Computer Science, University of Saarland
Post Box 151150, D-66041 Saarbruecken, Germany
email: {cj,kroening}@cs.uni-sb.de

Abstract. This paper presents status results of a microprocessor verification project. The authors verify a complete 32-bit RISC microprocessor including the floating point unit and the control logic of the pipeline. The paper describes a formal definition of a "correct" microprocessor. This correctness criterion is proven for an implementation using formal methods. All proofs are verified mechanically by means of the theorem proving system PVS.

1 Introduction

Microprocessor design is an error-prone process. With increasing complexity of current microprocessor designs, formal verification has become crucial. In order to achieve completely verified designs, adjusting the design process itself plays an important role: the more high-level information on the design is available, the faster the verification can be done.

The authors re-designed a simple RISC processor, the DLX [1], with respect to verifiability. The design includes the complete pipe control and forwarding logic. The function units are fully featured including a floating point unit. They are not abstracted by means of uninterpreted functions. The proofs for the glue logic, the ALU, and floating point unit are verified using the theorem proving system PVS [2].

Related Work Recent papers show the correctness of complex designs or schedulers in theorem proving systems such as PVS. Hosabettu et al. [3] prove both safety and liveness of Tomasulo's algorithm using PVS. Swada and Hunt [4] provide an ACL2 [5] proof of a complete design implementing a Tomasulo scheduler with reorder buffer.

Henzinger et al. [6] verify a simple pipelined processor using a model checker. McMillan [7] partly automates the proof by refinement of Tomasulo's algorithm presented in [8] with the help of compositional model checking. This technique is improved in [9] by theorem proving methods to support an arbitrary register size and number of function units.

There are many publications on the verification of (parts of) floating point units. Bryant and his group verified different function units using model-checking [10–12]. Aagaard and Seger verified a multiplier using model-checking combined with theorem proving [13]. Claesen et.al. and O'Leary et.al. have used theorem provers to verify

* supported by the DFG graduate program 'Effizienz und Komplexität von Algorithmen und Rechenanlagen'

an SRT integer divider [14], and an SRT integer square root circuit [15], respectively. Russinoff has proven the correctness of the multiplication, division and square root algorithms of the AMD K7 processor [16]. Most of the publications cited do not cover denormal numbers.

Project Status The verification of the pipeline and forwarding logic has reached a high level of automation. However, the process of verifying the function units is not automated at all. The fundamentals of the floating point mathematics are verified already. The verification of the individual floating point circuits is work-in-progress.

2 The Specification Machine

2.1 Hardware Model

Both the specification design and the hardware are modeled as *mathematical machine*. Mathematical machines are a common method to model the behavior of arbitrary microprocessor systems. For this paper, the definition of the mathematical machine from [17] is used: a mathematical machine is a triple $M = (C, c^0, \delta)$ which consists of the following components:

- C is the set of all possible configurations of M. An element c of C is called configuration or state of the machine.
- The initial configuration $c^0 \in C$ is a configuration of M.
- The transition function $\delta : C \to C$ maps a configuration c^T to its successor c^{T+1}.

A sequence $c^0, c^1, \ldots$ of configurations is called computation of M iff $c^{T+1} = \delta(c^T)$ holds.

Notation Registers are used in both the specification and the implementation of a microprocessor. Let $\mathcal{R} = \{R_1, ..., R_n\}$ be a finite set of registers. Each register R can have a value within a finite domain $\mathcal{W}(R)$.

The configuration set consists of the domains of the registers:

$$C = \mathcal{W}(R_1) \times \mathcal{W}(R_2) \times ... \times \mathcal{W}(R_n)$$

The projection function φ_{R_i} extracts the value of a register R_i from a configuration. Let c be $(a_1, a_2, ..., a_n)$.

$$\varphi_{R_i} : C \to \mathcal{W}(R_i), \qquad \varphi_{R_i}(c) = a_i$$

Let $c = c^T$ be part of a computation of a mathematical machine. R^T is a shorthand for $\varphi_R(c^T)$. Let $c.R$ be a shorthand for the following projection on c:

$$c.R = \varphi_R(c)$$

In analogy to that, let $\delta.R$ be a shorthand for the following projection on a state transition function:

$$\delta.R : C \to \mathcal{W}(R), \qquad \delta.R = \varphi_R \circ \delta$$

A signal s is defined as a mapping from the set of configurations into an arbitrary domain $\mathcal{W}(s)$:

$$s : C \to \mathcal{W}(s)$$

2.2 DLX Architecture

Our design implements the DLX architecture. The DLX architecture [1] features a RISC instruction set included both integer and floating point instructions. The integer core is taken from [17] and extended by a floating point register file (FPR) and floating point instructions as described in [18].

2.3 Correct IEEE Floating Point Arithmetic

Our primary goal is the verification of a complete processor. Thus, we formally verify the correctness of a floating point unit (FPU). In the processor framework, the FPU is a multi-cycle function unit, and can (almost) be seen as a black box. The FPU supports the operations addition, subtraction, multiplication, division and square root. The FPU handles normal and denormal numbers, special values, traps, and interrupts. This is in contrast to most previous results, where denormal numbers, traps and interrupts are disregarded.

The correctness criterions for the FPU are given by the IEEE standard 754 [19]. The standard is informal which makes it unusable for the formal verification of the FPU. One therefore has to formalize the IEEE standard; this formalization has to preserve the notion of the standard. Inherently, one cannot prove the equivalence of the formal and the informal specification. The formal specifications have to convince anybody of their correctness. We will give the specification of the IEEE rounding mode *to_nearest* as an example. The three other rounding modes *round_up*, *round_down*, and *to_zero* are not as complicated as the mode to_nearest. Nevertheless, they are covered.

For this, we first have to introduce some notations, which are taken from [18]. In contrast to [18], we spent reasonable effort on the definition of the rounding function itself, since this simplifies the verification of the FPU (see section 3.5). Due to lack of space, we omit the PVS specifications and proofs, which are available on request.

We abstract IEEE numbers, as they are defined in the standard, to *factorings*. A factoring is a triple (s, e, f) with sign bit $s \in \{0, 1\}$, exponent $e \in \mathbb{Z}$, and significant $f \in \mathbb{R}, f \geq 0$. The value of such a factoring is $[s, e, f] := (-1)^s \cdot 2^e \cdot f$. We use constants $e_{min}, e_{max} \in \mathbb{Z}$ as lower and upper bounds for the exponent, as they are defined in the standard.

We call a factoring (s, e, f) *normal*, if $e \geq e_{min}$ and $f \in [1, 2)$; we call (s, e, f) *denormal*, if $e = e_{min}$, $f \in [0, 1)$, and $f = 0 \Rightarrow s = 0$ holds. A factoring is called an *IEEE-factoring*, if it is normal or denormal. Note that $e \geq e_{min}$ holds for IEEE-factorings.

Lemma 1. *Each number* $x \in \mathbb{R}$ *has a unique IEEE-factoring* (s, e, f) *with* $[s, e, f] = x$.

Let η be the function which maps reals to IEEE-factorings. We call η the normalization shift.

Let P be the precision as defined in the standard. The significant f is called representable, if f is an integral multiple of 2^{-P}, i.e., $2^P \cdot f \in \mathbb{N}_0$. We call an IEEE-factoring (s, e, f) representable, if its significant f is representable, and $e \leq e_{max}$ holds.

We call an IEEE-factoring semi-representable, if f is representable. We call a real x (semi-)representable, if $\eta(x)$ is (semi-)representable.

Representable numbers exactly correspond to the representable numbers as defined in the standard. In the following, we will only investigate semi-representable factorings. In order to "round" semi-representable factorings to representable ones, one just has to decide whether one has to round to infinity or not. This can basically be done by a comparison of e with e_{max}.

We proceed with the definition of the rounding function. The standard defines the rounding mode *to_nearest* as follows:

> *... In this mode the representable value nearest to the infinitely precise result* $[$*of any floating point operation*$]$ *shall be delivered; if the two nearest representable values are equally near, the one with its least significant bit zero shall be delivered. ...*

The correspondence between this specification and the following definitions is not obvious. We will focus on this in the theorems below. We start with the definition of a function which rounds reals x to integers [20]:

$$r_{int}(x) := \begin{cases} \lfloor x \rfloor & \text{if } x - \lfloor x \rfloor < \lceil x \rceil - x \\ \lceil x \rceil & \text{if } x - \lfloor x \rfloor > \lceil x \rceil - x \\ x & \text{if } \lfloor x \rfloor = \lceil x \rceil \\ 2 \lfloor \lceil x \rceil / 2 \rfloor & \text{otherwise} \end{cases}$$

By scaling the input by 2^P, one rounds reals to rationals with P fractional bits:

$$r_{rat}(x) := 2^{-P} r_{int}(x \cdot 2^P).$$

Let (s, e, f) be an IEEE-factoring, and let $x := [s, e, f]$ be its value. One defines the IEEE rounding function for rounding mode *to_nearest* as follows:

$$r_{ne}(x) := 2^e r_{rat}(x \cdot 2^{-e}).$$

Now we have a definition relatively close to the hardware but far away from the specification in the standard. On one hand, this enables simpler implementation and verification of the rounder, as we will see in section 3.5. On the other hand, it is not obvious that these definitions conform to the the IEEE standard. We give three theorems which justify this claim.

Theorem 1. *For any real x, $r_{ne}(x)$ is semi-representable.*

The next theorem states that the result of the rounding function indeed is a nearest representable number.

Theorem 2. *For any real x, and any semi-representable IEEE-factoring (s, e, f), it holds $|x - [s, e, f]| \geq |x - r_{ne}(x)|$.*

The third theorem states that a number with least significant bit zero is chosen in case of a tie between the two nearest representable numbers. Thus, we first bound the distance between x and $r_{ne}(x)$. We then show that the significant is even if the maximum distance is reached.

Theorem 3. *For any real x, it holds $|x - r_{ne}(x)| \leq 1/2 \cdot 2^{e-P}$. If $|x - r_{ne}(x)| = 1/2 \cdot 2^{e-P}$ and $(s, e, f) = \eta(r_{ne})$, then $2^P \cdot f$ is even.*

We will give a theorem in section 3.5 which simplifies the verification of the rounding unit by decomposing it into smaller parts. This theorem will seem fairly obvious just because we invested reasonable effort in the definition of the rounding function. In contrast to our definition, the rounding result in [18] is defined as *"a representable number y closest to x. If there are two such numbers y, one chooses the number with even significant"*. This coincides obviously with the IEEE standard. Nevertheless, it is impractical to verify the rounder with this informal definition. The effort we have spent on the definition of the rounding function pays off when verifying the hardware implementation.

3 Implementing the Processor

3.1 Forwarding and Stalling Logic

The design uses a common five stage pipeline as presented in [1, 18]. The pipelined machine is generated by an automatic transformation from a sequential prepared machine as described in [17].

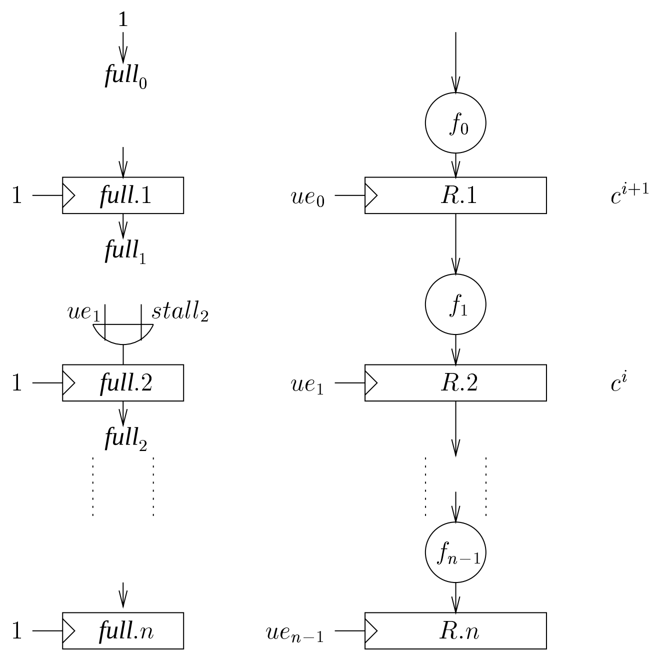

Fig. 1. The registers of a n-stage pipeline. The functions f_0 to f_{n-1} represent the data paths.

Our design features a complete *stall engine* [21, 18]. In contrast to the stall engine presented in [18], it allows stalling all stages individually. The stall engine is taken from [17] with small changes: a clock enable signal is no longer used. The full bits are updated in every cycle instead (figure 1).

The transition function for the full bits is changed accordingly; the full bit of each stage is set iff the stage is updated or stalled.

$$\delta.\mathit{full}.k(c) = ue_{k-1}(c) \vee stall_k(c)$$

The calculation of the signals ue and $stall$ is not changed and taken from [17]. The signal ue_k is the clock enable signal of the output registers of stage k: the registers are updated iff the stage is full and not stalled:

$$ue_k = \mathit{full}_k \wedge \overline{stall_k}$$

The generation of both the stall engine logic and the forwarding logic is done by a program based on the algorithms described in [17]. Furthermore, the program generates a correctness proof for both the forwarding and stalling logic, which is verified by the theorem proving system PVS.

3.2 Data Consistency

In order to formalize the data consistency criterion, a scheduling function $sI(k, T)$ is defined which specifies the index i of the instruction which is in the registers of stage k at time T [17]. Let R_I denote the value of a register in the implementation and R_S denote the value of the same register in the specification machine.

Theorem 4. *Let an instruction i be in the output registers of stage k at time T. Then the values in a specification register R of stage k of the implementation machine match those in the configuration of the specification machine after the execution of instruction i:*

$$sI(k,T) = i \quad \Longrightarrow \quad R_I^T = R_S^i$$

During cycle 0, all stages are in the initial configuration, which has index 0:

$$\forall k : sI(k, 0) = 0$$

The scheduling function for $T > 0$ of the machine is taken from [17]:

$$sI(k,T) = \begin{cases} sI(k, T-1) & \text{if } ue_k(c^{T-1}) = 0 \\ sI(0, T-1) + 1 & \text{if } ue_k(c^{T-1}) = 1 \wedge k = 0 \\ sI(k-1, T-1) & \text{if } ue_k(c^{T-1}) = 1 \wedge k \neq 0 \end{cases}$$

Theorem 4 relies on the following lemmas:

Lemma 2. *If the update enable signal of a stage is active in cycle T, the value of the scheduling function for that stage increases by one. If the update enable signal of a stage is not active, the value does not change. For $T > 0$:*

$$sI(k,T) = \begin{cases} sI(k, T-1) & \textit{if } ue_k(c^{T-1}) = 0 \\ sI(k, T-1) + 1 & \textit{if } ue_k(c^{T-1}) = 1 \end{cases}$$

Lemma 3. *Given a cycle T, the values of the scheduling functions of two adjoining stages are either equal or the value of the scheduling function of the later stage is greater by one.*

Lemma 4. *The values are equal iff the full bit of the later stage is not set.*

$$full_k^T = 0 \Leftrightarrow sI(k-1,T) = sI(k,T)$$

Negating both sides of the last equation and applying lemma 3 results in:

$$full_k^T = 1 \Leftrightarrow sI(k-1,T) = sI(k,T) + 1$$

Proof The proof of the lemmas above depends on the stall engine. It is an invariant proof by induction. Lemma 2 for cycle T is shown using lemma 4 for cycle $T-1$. Lemma 3 for cycle T is shown using lemma 2 in cycle T and lemma 4 in cycle $T-1$. Lemma 4 is shown using lemma 2 and 3 in cycle T.

Due to lack of space, only the induction step for lemma 2 is shown here: The claim for the case $ue_k^{T-1} = 0$ holds by definition. Let $ue_k^{T-1} = 1$ hold. For the case $k = 0$, the claim follows from the definition of sI. For $k > 0$ and $T > 1$ the claim is shown using lemma 4 for cycle $T-1$, which states that the claim is equivalent to $full_k^{T-1} = 1$. This is true because of the definition of the ue signals.

Theorem 4 is then shown by induction on T: the claim is obvious for stages k which are not updated in a given cycle. If the stage is updated (i.e., $ue_k^{T-1} = 1$), the correctness of these values is argued by showing the correctness of the input values of the stage. An example proof using the lemmas above for the instruction fetch stage is in [17].

3.3 Liveness

The liveness criterion is formalized as follows: for any given configuration c_S^i of the specification machine, we prove that the implementation machine calculates these values within a finite amount of time, i.e., there is a finite T such that $sI(k,T) = i$ holds. The proof is made by showing that any active stall signal becomes de-active within a finite amount of time. This is a proof by induction on the number of stages beginning with the last stage.

3.4 Integer Unit

Our design features an integer unit (ALU). It supports addition, subtraction, shift and compare operations, and bit-wise operations (AND, OR, XOR). The ALU is verified completely with the theorem proving system PVS. This includes an arbitrary-sized carry lookahead adder. However, the implementation and the proof for the carry lookahead adder is included only in order to achieve completeness. In order to create hardware, a pre-defined adder from the vendor library is used.

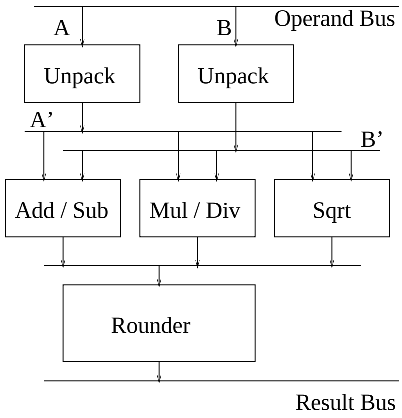

Fig. 2. Top level schematics of the FPU

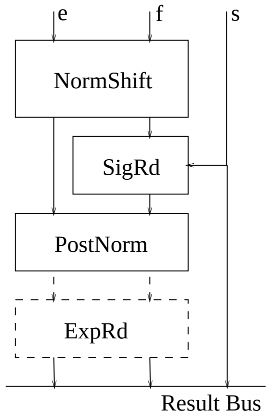

Fig. 3. Top level schematics of the rounder

3.5 Floating Point Unit

Figure 2 shows the top-level schematic of the FPU. The processor feeds packed IEEE numbers [19] A and B into the FPU. The unpacker circuit converts these numbers into the factoring format described in section 2.3. Depending on the operation, the operands A' and B' are fed into one of the function units. The last stage rounds the result of the operation to a representable and packed IEEE number, and places the result on the result-bus of the processor.

The design is pipelined, i.e., the design includes registers which store intermediate results. The division is carried out using the Newton-Raphson method. Thus, the function unit for multiplication and division contains loops to feed back intermediate results for the next Newton-Raphson iteration.

Complete hardware schematics at the gate level can be found in [18]. We will focus on the rounder. We demonstrate a part of the verification of the rounding unit exemplary. We give a theorem which decomposes the rounding function into three simpler functions which then serve as a basis for the implementation of the rounder. The three functions are the normalization shift η, the significant round r_{sig} and the post-normalization pn. Figure 3 shows a decomposition of the rounding hardware in corresponding sub-circuits. The sub-circuit "ExpRd" rounds to infinity, if an overflow occurs. This part is not yet formalized.

For reals x, $\eta(x)$ was defined as the unique IEEE-factoring (s, e, f) with $[s, e, f] = x$ in section 2.3.

Lemma 5. *For any real x and $(s, e, f) = \eta(x)$, it holds*

1. $s = 0$ *iff* $x \geq 0$,
2. $e = \max(\lfloor log_2(x) \rfloor, e_{min})$, *and*
3. $f = |x|/2^e$.

Lemma 6. *For any factoring (not necessarily IEEE-factoring)* (s, e, f) *with* $(s', e', f') = \eta([s, e, f])$, *it holds*

1. $s' = s$,
2. $e' = \max(e + \lfloor log_2(f) \rfloor, e_{min})$, *and*
3. $f' = f/2^{e'-e}$.

We assume that the input to the rounder is encoded as a factoring, but not necessarily as an IEEE-factoring. The normalization shift can then be computed in hardware by a *leading zero counter* to compute the logarithm of f, a *left/right shifter* to compute f', and an *adder* to adjust the exponent.

For IEEE-factorings (s, e, f), we define the significant round

$$r_{sig}(s, f) := |r_{rat}((-1)^s \cdot f)|$$

as the significant rounded to P fractional binary digits. The multiplication with the sign is necessary since the rounding decision depends on the sign. In hardware, the significant round is computed by the examination of the low-order bits of the significant. This technique is called *sticky bit computation* [18].

Lemma 7. *For any IEEE-factoring* (s, e, f), *it holds*

1. $r_{sig}(s, f) \in [0, 1]$, *if* (s, e, f) *is denormal,*
2. $r_{sig}(s, f) \in [1, 2]$, *if* (s, e, f) *is normal.*

The lemma is proven by unfolding the definitions, and applying the following lemma:

Lemma 8. *For any integers* a, b *and any real* x *with* $a \leq x \leq b$, *it holds* $a \leq \lfloor x \rfloor \leq \lceil x \rceil \leq b$.

In PVS, this lemma is proven automatically by the powerful proof-strategy `grind`.

Let (s, e, f) be an IEEE-factoring, and let $f' := r_{sig}(s, f)$. If the significant round yields a significant $f' = 2$, the result has to be post-normalized; the significant is set to 1, and the exponent is incremented. This is accomplished by the function pn:

$$pn(s, e, f) := \begin{cases} (s, e, f') & \text{if } f' \neq 2 \\ (s, e + 1, 1) & \text{if } f' = 2 \end{cases}.$$

The value of the factorings is obviously preserved by the function pn. The function is implemented by an *incrementer* for the exponent and an *multiplexer* for the significant.

Assume that the sub-circuits in figure 3 indeed compute the corresponding functions. Then the correctness of the whole rounder follows from the next theorem:

Theorem 5. *For any factoring* (s, e, f) *(not necessarily an IEEE-factoring), it holds*

$$\eta(r_{ne}([s, e, f])) = pn(\eta([s, e, f])).$$

This theorem is proven by definition unfolding, the use of the lemmas above, and some rules on exponentiation.

Theorem 5 decomposes the verification problem into smaller sub-problems such that the sub-circuits from figure 3 can be verified separately. These sub-circuits are further decomposed in [18].

4 Converting Mathematical Machines to Verilog HDL

The implementation above is specified as mathematical machine in the PVS language. All proofs rely on this specification. This specification is converted into a synthesizable subset of Verilog HDL [22]. This is done automatically by a program. A similar approach is made in [23].

The program is limited to convert mathematical machines, i.e., it takes a configuration set, an initial configuration, and a transition function as inputs. This tool is not limited to in-order designs.

5 Future Work

We are in progress of extending the design with a mechanism for speculative execution and precise interrupts. Furthermore, out-of-order execution capabilities are added by means of a Tomasulo scheduler.

The mathematics of the floating point arithmetic have been verified completely. Our future work is to verify the corresponding circuits.

Acknowledgment

The authors would like to thank Michael Bosch, Michael Klein, and Jochen Preiss for valuable discussions.

References

1. J.L. Hennessy and D.A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann Publishers, INC., San Mateo, CA, 2nd edition, 1996.
2. D. Cyrluk, S. Rajan, N. Shankar, and M. K. Srivas. Effective theorem proving for hardware verification. In *2nd International Conference on Theorem Provers in Circuit Design*, 1994.
3. Ravi Hosabettu, Ganesh Gopalakrishnan, and Mandayam Srivas. A proof of correctness of a processor implementing Tomasulo's algorithm without a reorder buffer. In *Correct Hardware Design and Verification Methods: IFIP WG 10.5 Internatinal Conference on Correct Hardware Design and Verification Methods (CHARME)*, pages 8–22. Springer, 1999.
4. Jun Sawada and Warren A. Hunt. Results of the verification of a complex pipelined machine model. In *Correct Hardware Design and Verification Methods: IFIP WG 10.5 Internatinal Conference on Correct Hardware Design and Verification Methods (CHARME)*, pages 313–316. Springer, 1999.
5. Matt Kaufmann and J. S. Moore. ACL2: An industrial strength version of nqthm. In *Proc. of the Eleventh Annual Conference on Computer Assurance*, pages 23–34. IEEE Computer Society Press, 1996.
6. Thomas A. Henzinger, Shaz Qadeer, and Sriram K. Rajamani. You assume, we guarantee: Methodology and case studies. In *Proc. 10th International Conference on Computer-aided Verification (CAV)*, 1998.
7. K.L. McMillan. Verification of an implementation of Tomasulo's algorithm by composition model checking. In *Proc. 10th International Conference on Computer Aided Verification*, pages 110–121, 1998.

8. W. Damm and A. Pnueli. Verifying out-of-order executions. In H.F. Li and D.K. Probst, editors, *Advances in Hardware Design and Verification: IFIP WG 10.5 Internatinal Conference on Correct Hardware Design and Verification Methods (CHARME)*, pages 23–47. Chapmann & Hall, 1997.
9. M.L. McMillan. Verification of infinite state systems by compositional model checking. In *Correct Hardware Design and Verification Methods: IFIP WG 10.5 Internatinal Conference on Correct Hardware Design and Verification Methods (CHARME)*, pages 219–233. Springer, 1999.
10. Y.-A. Chen and R. E. Bryant. Verification of floating-point adders. *Lecture Notes in Computer Science*, 1427, 1998.
11. Y.-A. Chen and R. E. Bryant. PHDD: An efficient graph representation for floating point circuit verification. In *IEEE/ACM International Conference on Computer Aided Design; Digest of Technical Papers (ICCAD '97)*, pages 2–7, Washington - Brussels - Tokyo, November 1997. IEEE Computer Society Press.
12. Y.-A. Chen, E. Clarke, P.-H. Ho, and Y. Hoskote. Verification of all circuits in a floating-point unit using word-level model checking. *Lecture Notes in Computer Science*, 1166, 1996.
13. M. D. Aagaard and C.-J. H. Seger. The formal verification of a pipelined double-precision IEEE floating-point multiplier. In *International Conference on Computer Aided Design*, pages 7–10, Los Alamitos, Ca., USA, November 1995. IEEE Computer Society Press.
14. L. Claesen, D. Verkest, and H. De Man. A proof of the non-restoring division algorithm and its implementation on an ALU. In *Formal Methods in System Design, vol. 5*, pages 5–31, 1994.
15. J. O'Leary, M. Leeser, J. Hickey, and M. Aagaard. Non-restoring integer square root: A case study in design by principled optimization. *Lecture Notes in Computer Science*, 901, 1995.
16. David M. Russinoff. A mechanically checked proof of IEEE compliance of the floating point multiplication, division and square root algorithms of the AMD-K7 processor. *LMS Journal of Computation and Mathematics*, 1:148–200, 1998.
17. Daniel Kröning, Wolfgang Paul, and Silvia M. Müller. Proving the correctness of pipelined micro-architectures. In Klaus Waldschmidt and Christoph Grimm, editors, *Proc. of ITG/GI/GMM-Workshop "Methoden und Beschreibungssprachen zur Modellierung und Verifikation von Schaltungen und Systemen"*, pages 89–98. VDE Verlag, 2000.
18. Silvia M. Müller and Wolfgang Paul. *Computer Architecture: Complexity and Correctness*. Springer, 2000.
19. Institute of Electrical and Electronics Engineers. *ANSI/IEEE standard 754–1985, IEEE Standard for Binary Floating-Point Arithmetic*, 1985. for a readable account see the article by W.J. Cody et al. in the IEEE MICRO Journal, Aug. 1984, 84–100.
20. Paul S. Miner. Defining the IEEE-854 floating-point standard in PVS. Technical report, NASA, Langley Research Center, 1995.
21. Wolfgang Paul. Recherarchitektur II SS98, 1998. Lecture Notes.
22. Donald E. Thomas and Philip Moorby. *The Verilog Hardware Description Language*. Kluwer, Boston;Dordrecht;London, 1991.
23. James C. Hoe and Arvind. Hardware synthesis from term rewriting systems. In *Proc. of VLSI'99, Lisbon, Portugal*, 1999.