Proving the Correctness of Pipelined Micro-Architectures

Daniel Kroening[*] and Wolfgang J. Paul
Dept. 14: Computer Science, University of Saarland
Post Box 151150, D-66041 Saarbruecken, Germany
email: {kroening,wjp}@cs.uni-sb.de

Silvia M. Mueller[†]
IBM Deutschland Entwicklung GmbH, Dept. 3173, AdrS. 7103-19
Post Box 1380, D-71003 Boeblingen, Germany
email: SMM@de.ibm.com

Abstract

This paper presents how to generate the implementation of a pipelined microprocessor from an arbitrary sequential specification. All necessary forwarding and stalling logic is created automatically. The implementation is provided in the language of the theorem proving system (PVS). This implementation is translated to the Verilog hardware description language. Furthermore, a mathematical correctness proof for the machine is supplied. This proof is verified by the theorem proving system.

1 Introduction

Automation is crucial for the design process of todays microprocessors, since it cuts design time and cost. A high level of automation is desired during the design and the verification of processors.

There has been much progress in the verification of given designs. Advanced model checking and theorem proving techniques allow for checking an almost complete processor in a reasonable amount of time for sequential machines [21], for pipelined machines [3, 18, 4, 12, 8] and for machines with out of order execution [6, 8, 13, 2, 10, 14].

However, the complete design is usually not covered. Furthermore, most proofs rely on a high-level specification of the scheduling algorithm or the processor control. The correctness of the implementation of the algorithm is usually also not covered.

Contribution

The paper presents a method to design in-order pipelined microprocessors with a large scale of automation by transforming a given sequential machine into a pipelined machine. This is done as follows (figure 1):

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

[†]Research was performed while affiliated with the University of Saarland, Computer Science Department.

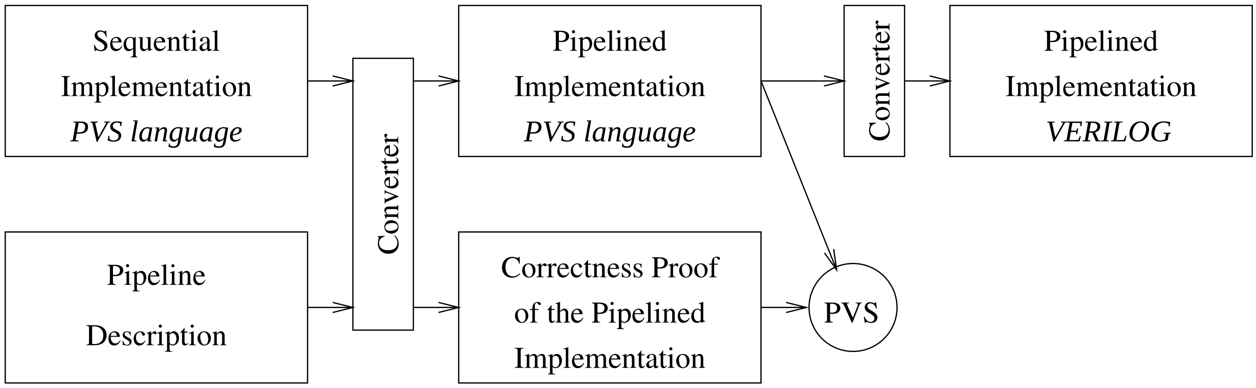

Figure 1: Generation of the implementations and of the proof

1. The first step is to specify the reference machine as sequential mathematical machine. This can be done even for complex instruction set architectures with ease.

2. The second step is to make a *sequential* implementation of this reference design with the following premises: the machine has the same stage structure as the pipelined design. However, the stages are never clocked simultaneously but in a round robin fashion. This design is called *sequential prepared* machine. The correctness of this design is easily shown using the theorem proving system PVS [5].

3. In the third step, this sequential design is transformed into a pipelined machine. The transformation is completely automated. Forwarding hardware is added where necessary. If forwarding is not possible, appropriate stall signals are generated.

Furthermore, a proof that the pipelined design simulates the reference machine is generated. This proof relies on the forwarding and stalling logic generated in step three and has the premise that the sequential prepared machine simulates the reference machine.

A simple RISC processor, the DLX [7], is used as an example for this paper. However, the methods and tools presented in this paper are not limited to DLX-like designs. The specification and the complete proof of the DLX in PVS language is omitted here due to lack of space. The files are available on the Internet instead [1]. In order to verify the proofs, make sure that PVS version 2.3 is used.

Related Work

Recent papers show the correctness of complex designs or schedulers in theorem proving systems such as PVS. Hosabettu et al. [10] prove both safety and liveness of Tomasulo's Algorithm with PVS. Swada and Hunt [17] provide an ACL2 [11] proof of a complete design implementing a Tomasulo scheduler with reorder buffer.

Henzinger et al. [8] verify a simple pipelined processors with a model checker. McMillan [13] partly automates the proof presented in [6] with the help of compositional model checking. This technique is improved in [14] by theorem proving methods to support an arbitrary register size and number of functional units.

2 Specifying the Reference Machine

The reference design is specified as *mathematical machine*. Mathematical machines are a common method to model the behavior of arbitrary microprocessor systems. There are different definitions of mathematical machines. For this paper, a single concept of a mathematical machine is used to specify both the hardware and the instruction set architecture. The correctness criterion and its proof relies on arguments on these two mathematical machines. A similar approach is used in [6] with the concept of synchronous transition systems.

A mathematical machine, as used in this paper, is a three-tuple $M = (C, c^0, \delta)$ which consists of the following components:

- C is a 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 is a configuration of M.
- The transition function $\delta : C \rightarrow C$ maps one configuration c^T to its successor c^{T+1}.

The sequence c^0, c^1, ... of configurations is called computation of M. The configurations of M for $T \geq 1$ are defined recursively as follows:

$$c^T = \delta(c^{T-1})$$

The reference machine processes exactly one instruction with each transition, no matter how complex this instruction is. It contains only registers which are part of the instruction set specification. This sequential reference machine is assumed to be correct.

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

In order to allow an easy identification of the value of a register in the configuration of a mathematical machine, all valid configurations in C are expected to be a tuple of the values of all registers:

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

The value of a given register R_i can be extracted from a configuration c with a projection function φ_i. Let c be $(a_1, a_2, ..., a_n)$.

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

Let $c = c^T$ be part of a computation of a mathematical machine. In this case, let R^T be a shorthand for $\varphi_{R_i}(c^T)$. 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 restriction of a state transition function to a register value:

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

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

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

Signals are therefore a shorthand for a calculation on a given configuration.

3 The Sequential Prepared Machine

The sequential prepared machine [15] has the same stage structure as the pipelined design. The process of partitioning the sequential step into stages is not automated. We use a common five stage pipeline for the DLX processor as found in many textbooks. The processor has a data memory interface but no IEEE floating point arithmetic.

In order to realize the sequential prepared machine, *implementation registers* are added which buffer intermediate results between the stages. Each stage is only allowed to read the registers from the specification machine and the implementation registers of the stage above. However, the stages are not clocked simultaneously but in a round robin fashion. Thus, the machine is still sequential. The execution of each instruction takes exactly as many cycles as there are stages.

With the premise that the sequential reference design is correct, the correctness (data consistency) of this sequential five-stage DLX design is easily shown with the theorem proving system. The largest part of the proof is the verification of the ALU implementation. This proof is omitted here, but available for download [1].

4 The Pipelined Machine

In the pipelined machine, registers of multiple stages can be updated simultaneously. The clocking of a stage k is controlled by the signal ue_k. Iff $ue_k(c)$ is one in a given configuration c, the stage k is updated.

The pipelined machine is generated by an automatic transformation from the sequential prepared machine. This is done by a program written in C++ which takes a list of the specification registers and the stage they are in, and the original transition functions of the sequential prepared design. The program outputs the following:

- an implementation of a stall engine, which controls the update enable signals ue of the stages,
- an implementation of forwarding logic, wherever necessary,
- an overall transition function for the complete pipelined design,
- and a PVS proof which shows the correctness criterion for the pipelined design.

The forwarding logic affects how register values are read by a stage. Read access to implementation registers never requires forwarding logic. However, accessing registers from the specification machine might require forwarding logic. Let stage k read specification register R, and let the new values of R be calculated in stage w. There are three cases:

1. If the read access is done after the new value is already calculated, i.e., $k > w$, the register is already overwritten. The original value is buffered in an implementation register. This implementation register is read instead of R.
2. If the read access is done in the stage which writes the register, i.e., $k = w$, it is sure that the register still contains the value from the previous configuration. Nothing has to be changed in this case.

3. If the read access is done in a stage before the stage which writes the register, i.e., $k < w$, the access cannot be done, since the desired value is not calculated yet.

There are two methods to overcome the limitation in the last case: forwarding and, if this fails, stalling.

Forwarding

Microprocessor instruction sets usually offer different kinds of instructions, such as ALU and memory instructions. The value which is to be forwarded is the result of these operations. The different instructions are processed by different stages, e.g., by an execute and by a memory stage. The result is available in an early stage therefore.

In these implementations, the result is buffered in an implementation register. The value of this implementation register is written into the register file in the last stage. However, in most implementations, this implementation register does not always hold the final result, e.g., if the desired result is calculated by different stages depending on the instruction. This applies for load instructions. The data from memory has to be shifted and masked before it can be written into the register file. This is done in an extra stage.

In order to realize forwarding of results which are available in an early stage, the following additions are necessary:

- It is necessary to specify which implementation register holds the intermediate result of a specification register.
- In order to specify whether the implementation register holds the final value of the specification register or not, a valid function is added for each register and each stage.

The following condition is added to the premise for each such register: if the valid function of a register returns one, the data in the implementation register matches the final value of the specification register.

In order to determine the stage which holds the desired value, extra signals hit_k are added for each input register of each stage. The signal hit_k is active iff the stage k contains an instruction which modifies the desired register. These signals are calculated with an equality tester and a check for the full bit of the stage: if the stage is not full, the hit_k signal is also not active. The first stage with an active hit signal is the stage where to forward from. Let $firsthit$ denote the number of this stage. This is calculated by an unary find first one circuit.

The forwarding is now done as follows:

- If there is no stage with a hit, the value is read directly from the register.
- If there is a stage with a hit and the valid function returns one for this stage, the output value from this stage for the register is taken.
- If neither case applies, forwarding fails.

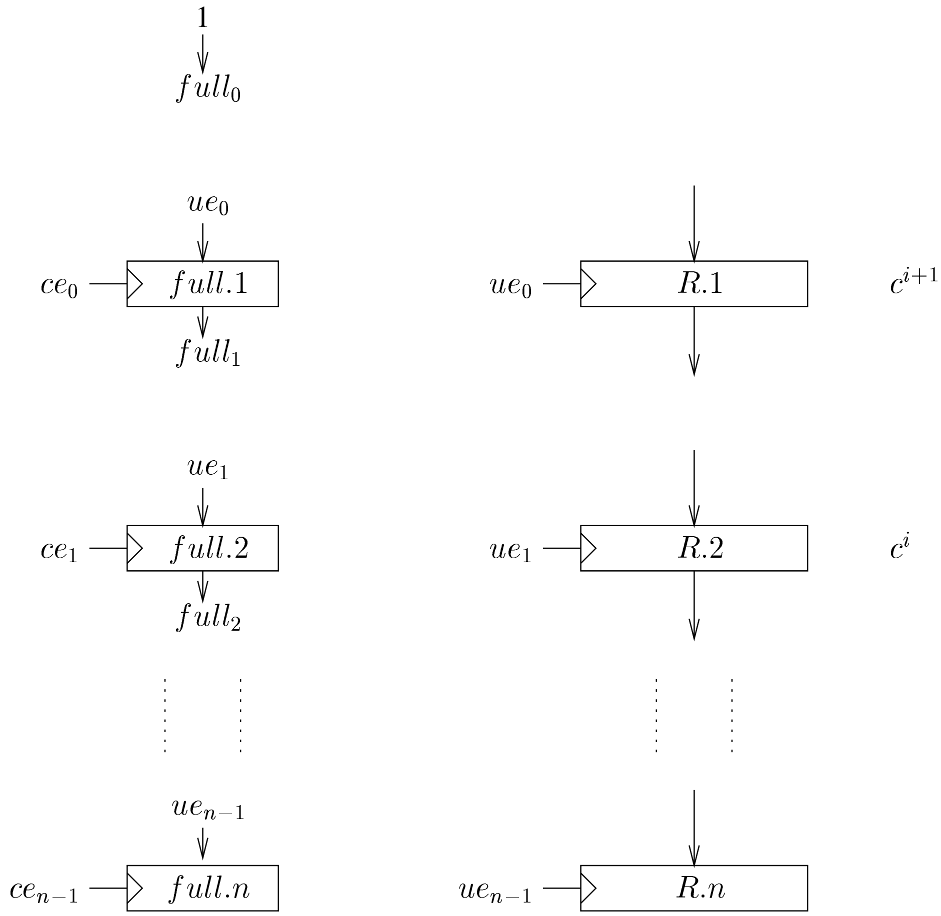

Figure 2: The clocking of the registers of a n-stage pipelined machine

Stalling

If forwarding fails, the calculation of the new values of the registers of a stage is not possible until the input registers are available. The instruction has to wait. This is realized by the stall engine (figure 2).

For each stage, new signals are introduced. The signal $stall_k$ controls whether stage k has to be stalled. The calculation of the update enable signal ue_k is changed in an obvious way: the output registers of a stage are only updated iff the stage is full and not stalled.

$$ue_k = full_k \wedge \overline{stall_k}$$

In order to calculate the stall signals, a signal is required which indicates whether a given stage has to wait for an input value. The signal $dhaz_k$ is active iff stage k is waiting for an input operand. The stage k must be stalled if $dhaz_k$ is active and if the stage is full. Furthermore, the stage must be stalled if the next stage (stage $k+1$) is stalled because there is no space to store the results of stage k in this case. Since the last stage has no next stage, the calculation of the signal $stall_k$ depends on the stage number:

$$stall_k = full_k \wedge \begin{cases} dhaz_k & \text{if } k \text{ is the last stage} \\ (dhaz_k \vee stall_{k+1}) & \text{otherwise} \end{cases}$$

The full bit register of a stage must not be updated if the stage is stalled. This is controlled by a clock enable signal ce_k for each stage. Iff ce_k is active, the full bit $full.k+1$ is updated.

$$ce_k = (\overline{stall_k} \wedge full_k) \vee ue_{k+1}$$

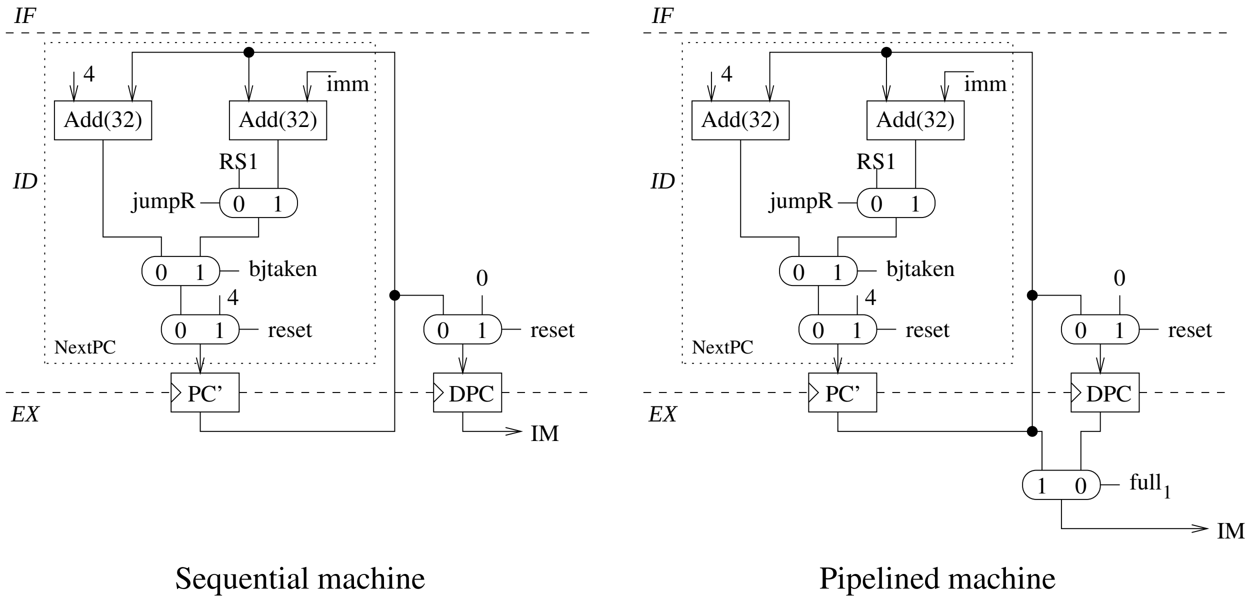

Figure 3: Transformation of the PC environment

The calculation of the new values for the full bit registers has to be changed also. It is now no longer possible to copy the full bit of the previous stage, since the current stage might be stalled. In this case, it is necessary to clear the full bit. The update enable bit is used as new value therefore.

Transformation of the Sequential DLX

As example, this section presents the transformation of the instruction fetch mechanism of a DLX with delayed branch. The sequential semantics for a machine with delay slot are given in [16, 15]: Let $nextPC(PC)$ calculate the new PC in the sequential machine (figure 3). New instructions are fetched from the address provided by a register DPC, which is a delayed version of the regular PC register, which is named PC':

$$DPC := PC' \tag{1}$$

$$PC' := nextPC(PC') \tag{2}$$

Figure 3 shows how the value of DPC in the sequential machine is calculated by the pipelined machine: according to the forwarding rules, this depends on the full bit:

$$DPC_S = \begin{cases} PC'.2 & \text{if } full_1 = 1 \\ DPC.2 & \text{if } full_1 = 0 \end{cases} \tag{3}$$

5 Correctness of the Pipelined Machine

In order to formalize the data consistency criterion, a scheduling function $I(k, T)$ is defined which specifies the index i of instruction which is in the registers of stage k at time T. During cycle 0, all stages are in the initial configuration, which has index 0:

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

The scheduling function for $T > 0$ of the machine is:

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

Let R_I denote the value of a register in the implementation and R_S denote the value of a register in the specification machine. Data consistency means that each time a register R of stage k is written, the value written must match the value which the register has in the specification machine:

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

This proof relies on the following lemmas:

1. 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 $T > 0$:

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

2. 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 one higher.

3. Iff the values are equal, the full bit of the later stage is not set.

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

   Negating both sides of the last equation results in:

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

This argument is extended inductively to multiple stages. The correctness criterion is then shown by induction on T: for stages k which do not have new values in this cycle (i.e., $ue_k^{T-1} = 0$) the claim is obvious. If the stage got new values (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. A complete proof for a five stage DLX RISC processor is available for download [1]. Due to lack of space, only the correctness of the forwarding done for stage IF is presented here. The claim is:

$$I(0,T) = i \Rightarrow IR_I^T = IM[DPC_S^{i-1}]$$

Due to $ue_0^{T-1} = 1$, $I(0,T-1) = i-1$ holds by lemma 1. Because of lemma 2 there are two cases for $I(1,T-1)$:

- If $I(1,T-1) = i-2$ holds, lemma 3 implies $full_1^{T-1} = 1$. By (3) the register $PC'.2$ is used as address for the instruction fetch. By the induction premise, $PC'.2^{T-1} = PC'^{i-2}$ holds. This is equal to DPC_S^{i-1} by (1).

- If $I(1,T) = i-1$ holds, lemma 3 implies $full_1^{T-1} = 0$. By (3) the register $PC'.2$ is used as address for the instruction fetch. By the induction premise, $DPC.2^{T-1} = DPC_S^{i-1}$ holds.

6 Converting Mathematical Machines to Verilog

The implementations above are all specified as mathematical machine in the PVS language. All proofs rely on these specifications. In order to get real hardware, e.g. which can be put on an ASIC or FPGA, this specification is converted into a subset of Verilog [19]. This is done automatically by a program. A similar approach is made by [9].

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

7 Future Work

This paper does only covers data consistency and does not provide a proof that the machine is deadlock-free. The transformation tool is limited to in-order pipelines, but the current research goal is to extend it to support out-of-order schedulers such as the Tomasulo scheduler[20]. Furthermore, speculative execution can be added with ease which is necessary for branch prediction and precise interrupts.

References

[1] Example: Implementation and correctness proof of a five stage DLX. http://www-wjp.cs.uni-sb.de/~kroening/pipe/.

[2] Arvind and Xiaowei Shen. Using term rewriting systems to design and verify processors. *IEEE Micro Special Issue on Modeling and Validation of Microprocessors, May/June*, 1999.

[3] Mark Bickford and Mandayam Srivas. Verification of a pipelined microprocessor using Clio. In M. Leeser and G. Brown, editors, *Proceedings of the Mathematical Sciences Institute Workshop on Hardware Specification, Verification and Synthesis: Mathematical Aspects*, volume 408 of *LNCS*, pages 307–332. Springer, 1990.

[4] Jerry R. Burch and David L. Dill. Automatic verification of pipelined microprocessor control. In *Proc. International Conference on Computer Aided Verification*, 1994.

[5] 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.

[6] 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.

[7] J.L. Hennessy and D.A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann Publishers, INC., San Mateo, CA, 2nd edition, 1996.

[8] 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.

[9] James C. Hoe and Arvind. Hardware synthesis from term rewriting systems. In *In Proc. of VLSI'99, Lisbon, Portugal*, 1999.

[10] 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.

[11] Matt Kaufmann and J. S. Moore. ACL2: An industrial strength version of nqthm. In *In Proc. of the Eleventh Annual Conference on Computer Assurance*, pages 23–34. IEEE Computer Society Press, 1996.

[12] Jeremy Levitt and Kunle Olukotun. A scalable formal verification methodology for pipelined microprocessors. In *33rd Design Automation Conference (DAC'96)*, pages 558–563. Association for Computing Machinery, 1996.

[13] 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.

[14] 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.

[15] Silvia M. Müller and Wolfgang Paul. *Complexity and Correctness of Computer Architectures*. Springer, 2000. Draft.

[16] S.M. Mueller, W.J. Paul, and D. Kroening. Proving the correctness of processors with delayed branch using delayed PC. In I. Althoefer, N. Cai, G. Dueck, L. Khachatrian, M. Pinsker, A. Sarkozy, I. Wegener, and Zhang Z., editors, *Numbers, Information and Complexity*. Kluwer, 1999.

[17] 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.

[18] James B. Saxe, Stephen J. Garland, John V. Guttag, and James J. Horning. Using transformations and verification in circuit design. Technical report, Digital Systems Research Center, 1991.

[19] Donald E. Thomas and Philip Moorby. *The Verilog Hardware Description Language*. Kluwer, Boston;Dordrecht;London, 1991.

[20] R.M. Tomasulo. An efficient algorithm for exploiting multiple arithmetic units. In *IBM Journal of Research and Development*, volume 11 (1), pages 25–33. IBM, 1967.

[21] Phillip J. Windley. Formal modeling and verification of microprocessors. *IEEE Transactions on Computers*, 44(1):54–72, 1995.