Decomposing the Verification of Pipelined Microprocessors with Invariant Conditions

Jun Sawada[1] and Warren A. Hunt, Jr.[2]

[1] Department of Computer Sciences, University of Texas at Austin
Austin, TX 78712, USA
E-mail: sawada@cs.utexas.edu
[2] IBM Austin Research Laboratory
11400 Burnet Road MS/9460, Austin, TX 78758, USA
E-mail: whunt@austin.ibm.com

Abstract. We have developed an approach to mechanically verify a microprocessor design that includes speculative execution, out-of-order instruction completion using Tomasulo's algorithm, precise exception handling, and other features found in modern microprocessors. Our approach defines invariant conditions that characterize the valid behavior of micro-architectural components. Given the validity of these invariants, we proved the correctness of our microprocessor design. Invariant conditions are defined and verified incrementally using a strengthening procedure, which works by starting with an initial set of invariants and produces a sufficiently strong set. This procedure effectively decomposes the pipeline verification problem into the verification of invariants. We can further decompose the verification of certain invariant conditions with mathematical induction and case-analysis. Our invariant verification procedure uncovered numerous bugs that were not found by simulation.

1 Introduction

The goal of formally verifying pipelined microprocessor designs [1, 14] is to show that pipelined implementations operate exactly as specified by their non-pipelined specification[2]. In this paper we discuss our approach of reducing the verification problem into a number of simpler pieces [3, 9, 6, 12, 15] by introducing various invariant conditions that characterize the valid behavior of a pipeline. Using our approach, we have verified the correctness of a microprocessor model, whose features include: speculative execution, Tomasulo's algorithm [4, 8], and precise exception handling [13]. Identifying and verifying invariant conditions are the key to our approach. We seed our approach by first defining a small set of invariant conditions that are needed to verify the final correctness criterion. Then we incrementally strengthen this set of conditions until we obtain a sufficiently strong set of invariants. This strengthening process also ensures that the

* This research was supported in part by the Semiconductor Research Corporation under contract 98-DJ-388.

validity of the invariants are maintained by the design under verification and, in our case, was the place where we discovered faults in our example design.

For certain invariant conditions, we can further reduce the verification problem into even smaller pieces, by employing mathematical induction and case analysis. We describe this kind of reduction by presenting our verification of an invariant that specifies that certain conditions for intermediate values are maintained by our pipelined processor design.

We have defined a microprocessor design to which we have applied our invariant verification approach, and we describe our correctness criteria in Section 2. We present our invariant strengthening procedure in Section 3, and summarize the different invariants we defined to verify our design. We then elaborate on the verification of one of our invariants by showing how we further simplified the verification of our "intermediate data values" invariant in Section 4. We finally present a breakdown of the total effort required to verify this design in Section 5, and then present our conclusions.

2 Machine Model and Correctness Criterion

We designed a new microprocessor model [11] for the purpose of our research. The model contains several features which make the verification problem challenging. Some of the interesting features included are speculative execution, branch prediction, exceptions, and external interrupts. It contains four pipelined execution units that execute instructions in parallel. Reservation stations and a re-order buffer are used to implement Tomasulo's algorithm, which allows out-of-order execution of instructions [5].

The processor is modeled at two levels; the *instruction-set architecture* (ISA) level and the *micro-architecture* (MA) level. The ISA model contains only the programmer visible components such as the program counter, the register files and the memory. The ISA model executes instructions sequentially, and it serves as our specification. The MA model is an abstract model of the actual pipelined implementation, and it is our target of verification. This model specifies how individual micro-architectural components behave in each clock cycle.

Given the current ISA state ISA_i and the external signals SIG_i^{ISA}, our ISA specification function $\mathit{ISA\text{-}step}(ISA_i, SIG_i^{ISA})$ returns the ISA state after executing one instruction. Similarly, given the current MA state MA_i and its external signals SIG_i^{MA}, $\mathit{MA\text{-}step}(MA_i, SIG_i^{MA})$ returns the MA state after one clock cycle of execution. Notice the difference in time granularity between the two levels. The input signals to the ISA and the MA are also different, because certain implementation details such as memory responses are abstracted away from the ISA-level input signals. We define $\mathit{ISA\text{-}stepn}(ISA_i, SIG_*^{ISA}, n)$ and $\mathit{MA\text{-}stepn}(MA_i, SIG_*^{MA}, n)$ as recursive functions that apply the corresponding single-cycle next state functions n times with the sequence of external signals SIG_*^{ISA} and SIG_*^{MA}, respectively. Projection function $\mathit{proj}(MA_i)$ strips off the implementation dependent components from MA_i, and returns the ISA state consisting of only the programmer visible components.

Our verification goal is to show that the MA model correctly implements our ISA specification, by proving the following correctness criterion [11].

Definition 1. *(Correctness Criterion) Suppose we run the MA model for n machine cycles from an initial flushed state MA_0 with a sequence of external signals SIG_*^{MA}. If the resulting state $MA_n = MA\text{-}stepn(MA_0, SIG_*^{MA}, n)$ is also flushed, there exists a corresponding ISA execution of m instructions with an appropriate sequence of external signals SIG_*^{ISA}, and*

$$proj(MA\text{-}stepn(MA, SIG_*^{MA}, n)) = ISA\text{-}stepn(proj(MA), SIG_*^{ISA}, m) \ ,$$

provided that the ISA execution $ISA\text{-}stepn(proj(MA), SIG_*^{ISA}, m)$ does not cause self-modification of the program.

The existence of m and SIG_*^{ISA} for arbitrary n and SIG_*^{MA} is shown by constructing witness functions.

3 Invariant Strengthening Procedure

Our verification process is divided into two steps.

Step 1: Define and verify invariant conditions of the target design.
Step 2: Prove the correctness criterion from the invariant conditions.

These verification steps may be carried out in any order. Actually, we first completed Step 2, and then Step 1. In this paper, we mainly discuss Step 1.

We can prove our correctness criterion in Step 2, if the set of invariant conditions, S, defined in Step 1 satisfies the following requirements [11]:

Requirements for the Invariants.

1. If S holds for a flushed MA state, then each instruction is completely executed or not executed at all;
2. If S holds and the MA is not executing instructions speculatively, the program counter points to the instruction to be fetched next;
3. If S holds, the register files contain the results of all committed instructions;
4. If S holds, the memory contains the results of all completed memory operations;
5. S holds for all initial MA states; and
6. If S holds for the current state MA_n, it also holds for the next state MA_{n+1}.

Finding a set of invariant conditions satisfying these requirements is the most difficult part of our verification. Rather than defining such a set directly, we find invariant conditions incrementally using a strengthening procedure. First we define a small number of conditions, from which we can directly or easily show Requirements 1 through 4. We call these conditions our *initial invariant conditions*. We strengthen the initial invariant conditions with the following procedure:

Invariant Strengthening Procedure.

1. Let S_0 be the set of the initial invariant conditions.
2. Let S be an empty set.
3. Pick a condition C from S_0. Assume $S_0 \cup S$ for the current state MA_n and prove C for the next state MA_{n+1}. If C is not provable from $S_0 \cup S$, we may add additional invariant conditions to S_0 until we can prove C.
4. Prove C for initial MA states.
5. Remove C from S_0 and add it to S.
6. Goto 3 if S_0 is not empty.
7. Return S.

We carried out this process manually while proving each invariant condition C using the ACL2 theorem prover [7]. The set of conditions returned by the procedure satisfies the six requirements given above.

This procedure is not an algorithm in a strict sense, because adding a new condition to set S_0 is done by hand. When we encounter a condition C that we are unable to prove from $S_0 \cup S$, we have to analyze how our mechanical verification failed, understand what kind of additional invariant conditions are needed for the proof, and add the new conditions. Sometimes, we find that C is not valid, because C may be an incorrectly defined invariant condition, or the target machine does not satisfy C because of design faults. In this way, we found several bugs in our design that we discuss later. The quality of the newly added invariant conditions is critical for the procedure to terminate in a reasonably short time.

In Table 1, we list the invariants obtained by the strengthening procedure for our verified machine. In this list, the initial invariant conditions are numbered 0 through 6. We applied our invariant strengthening procedure and obtained the remaining 15 invariant conditions. In reality, we did not follow the procedure strictly. Sometimes when we found that we had to add additional invariants, we instead modified or strengthened some of the already existing conditions, rather than simply adding a new invariant condition to set S_0. This enabled us to keep the list of invariant conditions simple and organized. For instance, the definition of *MT-init-invariants* was very simple when we started the procedure, but it was modified many times into our most complex invariant.

Figure 1 shows the dependencies between the invariant conditions. This dependency graph was extracted from the output of the ACL2 theorem prover. The connecting edge shows which conditions the ACL2 theorem prover used to verify each condition.

We notice that some invariant conditions are mutually dependent. For instance, *MT-inst-invariants* is assumed when *in-order-dispatch-commit-p* is verified, and vice versa. This, however, does not lead to an invalid proof of the invariant conditions, because the conditions are assumed only for the current state MA_n and verified for the next state MA_{n+1}. We did not assume any conditions for the next state MA_{n+1} as was done by McMillan's verification of Tomasulo's algorithm [8].

Table 1. List of the complete invariant conditions and their description.

#	Invariant Name	Brief Description
0	**weak-invariants:**	A well-formedness predicate for a MAETT.
1	**pc-match-p:**	Correct state of the program counter.
2	**sregs-match-p:**	Correct state of the special register file.
3	**regs-match-p:**	Correct state of the general register file.
4	**mem-match-p:**	Correct state of the memory.
5	**no-speculative-commit-p:**	No speculatively executed instruction commits.
6	**MT-inst-invariants:**	Valid intermediate data values in the pipeline.
7	**correct-speculation-p:**	Instructions following a mis-predicted branch are speculatively executed.
8	**correct-exintr-p:**	Externally interrupted instructions retires immediately.
9	**in-order-dispatch-commit-p:**	Instructions dispatch and commit in program order.
10	**in-order-DQ-p:**	The dispatch queue is a FIFO queue.
11	**in-order-ROB-p:**	The re-order buffer is a FIFO queue.
12	**no-stage-conflict:**	No structural conflict for pipeline stages.
13	**no-robe-conflict:**	No structural conflict for the re-order buffer.
14	**in-order-LSU-inst-p:**	Certain orders are preserved for the instructions in the load-store unit.
15	**consistent-RS-p:**	Reservation stations keep track of instruction dependencies.
16	**consistent-reg-tbl-p:**	The register reference table keeps track of the newest instruction that updates each general register.
17	**consistent-sreg-tbl-p:**	The register reference table keeps track of the newest instruction that updates each special register.
18	**consistent-MA-p:**	The conjunction of miscellaneous invariants.
19	**misc-invariants:**	The conjunction of miscellaneous invariants.

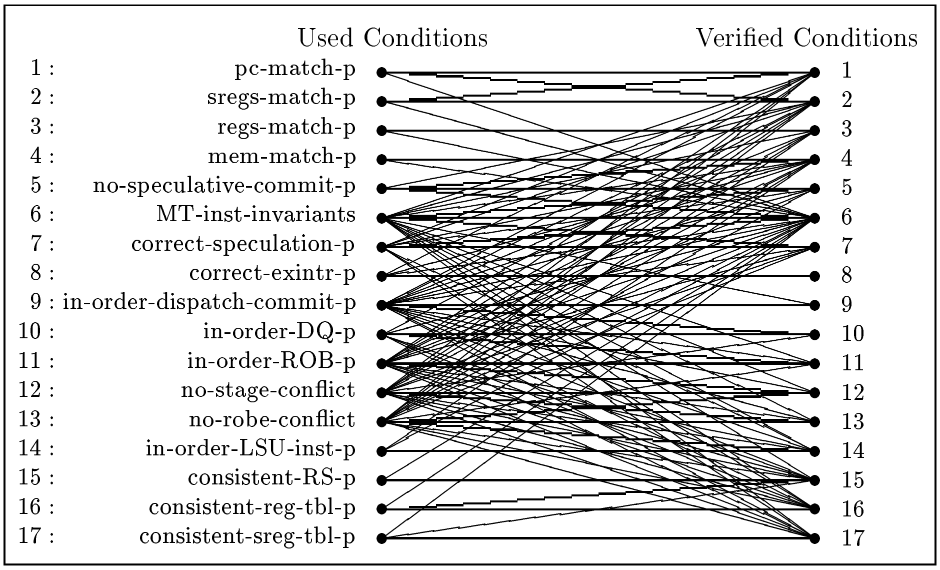

Fig. 1. Dependency between the invariant conditions. We omitted conditions 0, 18, and 19 of Table 1 for presentation purposes.

Some of the initial invariant conditions such as *pc-match-p* and *regs-match-p* are rarely used in the proof of other invariant conditions. This is natural, because the initial invariant conditions are useful to prove our correctness criterion, but they are not intended to characterize how the pipelined implementation works in detail. The invariant conditions 7–13 are used heavily in the proof of other conditions. These invariant conditions are important properties for the correct pipeline operation and were added to the set of invariants at a relatively early stage of the verification. On the other hand, the invariant conditions 14–17 are rarely used in the verification of other conditions. These conditions were added to the set relatively late, and they are not frequently used in our proof.

Our strengthening procedure effectively divides the verification problem into the smaller problems of verifying individual invariant conditions. The verification of some invariant conditions can be further divided into smaller pieces. In the next section, we explain this by examining the definition and verification of invariant *MT-inst-invariants*.

4 Decomposition of Invariant Verification

Most of the invariant conditions discussed in the previous section are defined as predicates that take an MA state and its corresponding *MAETT* [10]. The MAETT is an abstract representation of an MA state. A MAETT records various information about the current MA state and, most importantly, the history of completed and in-flight instructions. The data-structure of a MAETT is abstractly depicted in Fig. 2. The list pointed to from MAETT field `trace` records instructions in program order. Each entry represents an instruction by recording its current stage, the corresponding re-order buffer entry, whether it is speculatively being executed, etc. Fields `pre-ISA` and `post-ISA` point to the ideal ISA states before and after executing instruction I_i in the ISA model.

Using this data-structure, we defined *MT-inst-invariants*. A schematic definition of *MT-inst-invariants* is given below:

$$\mathit{MT\text{-}inst\text{-}invariants}(MT, MA) \stackrel{def}{=} \forall I \in MT, \mathit{inst\text{-}invariants}(I, MA)$$

$$\begin{aligned}\mathit{inst\text{-}invariants}(I, MA) \stackrel{def}{=} &\ \textbf{if} \quad\quad \mathit{stage}(I) = \mathit{Stage}_1 \ \textbf{then}\ \mathit{Stage}_1\text{-}\mathit{inst\text{-}inv}(I, MA)\\ &\ \textbf{else if}\ \mathit{stage}(I) = \mathit{Stage}_2 \ \textbf{then}\ \mathit{Stage}_2\text{-}\mathit{inst\text{-}inv}(I, MA)\\ &\quad\quad\quad\quad \vdots\end{aligned}$$

$$\mathit{Stage}_i\text{-}\mathit{inst\text{-}inv}(I, MA) \stackrel{def}{=} \bigwedge_k \{\mathit{Hyps}_{ik} \Rightarrow \mathit{Field\text{-}Val}_{ik}(MA) = \mathit{Ideal\text{-}Val}_{ik}(I)\}.$$

Here, I is an instruction represented by an entry in MAETT MT. Predicate Stage_i-*inst-inv*(I, MA) defines the valid intermediate values at pipeline stage Stage_i as a conjunction of conditional equalities. The recursively defined predicate *MT-inst-invariants*(MT, MA) holds if each instruction I satisfies the corresponding Stage_i-*inst-inv*(I, MA), depending on I's current stage.

We defined Stage_i-*inst-inv*(I, MA) for each pipeline stage Stage_i. For example, *MU-RS0-inst-inv*(I, MA) defines the valid intermediate values at reservation station `MU-RS0`. One of the fields in the reservation station is `RS-val1`, which should

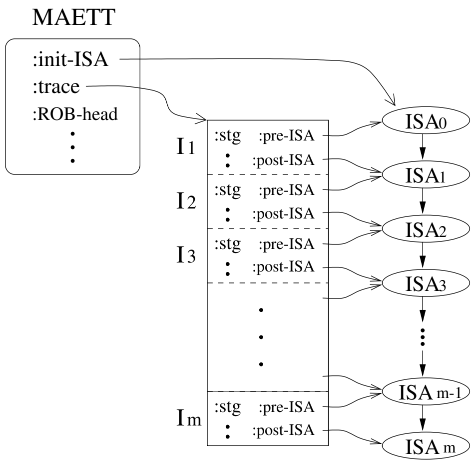

Fig. 2. Structure of MAETT

hold the first operand register value when operand ready flag `RS-ready1?` is asserted. We define function *INST-src-val1*(I) to calculate the correct first operand value for I using the `pre-ISA` field of the MAETT entry. Thus an instruction I at stage `MU-RS0` should satisfy the conditional equality[1]:

$$\texttt{RS-ready1?}(\texttt{MU-RS0}(MA)) \Rightarrow \texttt{RS-val1}(\texttt{MU-RS0}(MA)) = \textit{INST-src-val1}(I).$$

MU-RS0-inst-inv(I, MA) is defined as the conjunction of such conditional equalities for all fields in `MU-RS0`. We define other stage dependent conditions similarly.

For the invariant strengthening procedure, we need to show Requirement 6 for *MT-inst-invariants*. More concretely, we have to prove:

$$\textit{invariants}(MT_n, MA_n) \Rightarrow \textit{MT-inst-invariants}(MT_{n+1}, MA_{n+1}), \qquad (1)$$

where *invariants* is the conjunction of all invariant conditions listed in Table 1. By mathematical induction, it becomes sufficient to show:

$$\textit{invariants}(MT_n, MA_n) \Rightarrow \textit{inst-invariants}(I, MA_{n+1})$$

for an arbitrary instruction I in MT_{n+1}. By the definition of *inst-invariants*, this problem is further broken down into:

$$\textit{invariants}(MT_n, MA_n) \wedge \textit{stage}(I) = \textit{Stage}_i \Rightarrow \textit{Stage}_i\textit{-inst-inv}(I, MA_{n+1}). \quad (2)$$

The verification complexity of this formula is far smaller than the original formula (1). Since a single instruction I and a single stage $\textit{Stage}_i$ are relevant in the conclusion of (2), we don't have to consider all instructions and the whole pipeline at once.

[1] This is an example, and the actual definition contains more complex hypotheses.

We incrementally defined *MT-inst-invariants*. We were unable to define it correctly from the beginning because of its complexity. While we verified (2) for each $Stage_i$, we sometimes had to modify incorrect definitions of $Stage_i$*-inst-inv*. We also had to strengthen $Stage_j$*-inst-inv* for some other stage $Stage_j$, especially when data is forwarded from $Stage_j$ to $Stage_i$, and the validity of the forwarded data was needed to verify (2). The verification of *MT-inst-invariants* uncovered various bugs, which are reported in the next section.

5 Verification Summary

Using the ACL2 theorem prover, we have completely verified our invariant conditions and our correctness criterion for our pipelined machine design. Our correctness criterion implies that the MA model executes each instruction correctly with respect to the ISA model. However, there are some properties that have not been verified. For instance, liveness properties are not yet verified as we have done previously [10]. We verified that external interrupts are processed correctly, but we have not verified how long it takes for an interrupt signal to be processed.

The verification cost is always a serious practical concern. Since our verification was carried out solely by the ACL2 theorem prover, we had to manually write many lemmas to guide the prover. Table 3 shows the size of the ACL2 proof script files and the time to run ACL2 on these files with 200MHz PentiumPro processor. The reader can assume that the human effort devoted to the corresponding task is almost proportional to the size of the ACL2 script, not the CPU time. We wrote our ISA and MA specifications in one month, but the whole verification project took about 15 months.

We notice that a large portion of ACL2 script files are for proving invariants and shared lemmas. Since, most of the shared lemmas are, in fact, used for the proof of invariants, we can safely say that the verification of invariants required most of our effort. This is not surprising because the verification of invariants, which corresponds to Step 1 as defined in Section 3, is the core of our verification process. We found several bugs in the machine design, and all bugs were detected during the verification of invariant conditions. On the other hand, the proof of our correctness criterion, which corresponds to Step 2, was a much smaller portion of our verification effort.

Although the verification is labor intensive, our technique scales well with the size of machine. We compare the size of our machine specification and the verification script with our two other projects [10], in which we employed a similar approach, but with different example machine sizes. The ratio of the verified machine design and verification script does not change much. We also point out that our machine design discussed here contains more complex features such as exceptions and speculative execution, which were not part of other two machine designs.

We found 14 design faults in our machine design during the verification. Most of these design faults were hard bugs, since easy bugs were eliminated by executing test programs on the machine design before applying formal verification

Table 2. ACL2 script size and CPU time for different verification phases.

Type of ACL2 Script	ACL2 Script Size	CPU Time to Certify
Definitions of ISA and MA	140 KBytes	14 minutes
MAETT modeling	55 KBytes	6 minutes
Definitions of invariants	89 KBytes	7 minutes
Proof of Shared Lemmas	481 KBytes	58 minutes
Proof of Invariants	1034 KBytes	211minutes
Proof of Criterion	37 KBytes	11 minutes

Table 3. Sizes of ACL2 proof scripts for different machines.

Verified Machine	Machine Spec	Total Verification
Small Example Machine	13 KBytes	169 KBytes
Pipelined Design presented in CAV '97	78 KBytes	757 KBytes
Design discussed in this paper	140 KBytes	1909 KBytes

techniques. We think that some of the bugs would have been very difficult to discover with simulation techniques. For instance, some design faults produced an incorrect behavior only when a certain combination of instructions were issued in a rare timing. Another bug caused a mal-functioning only when the outcome of branch prediction for the same instruction changed after a pipeline stall. We also found other bugs involving speculative execution, and exceptions. For instance, exceptions were not correctly raised if an access was attempted to a non-existing special register, or if a privileged instruction was executed in the user mode.

6 Conclusion

Defining and verifying the invariant conditions is the most difficult and important part of our pipelined machine verification approach. Using our technique to break the verification problem into pieces with invariant conditions, we have completely verified our microprocessor model, which implements various features of modern microprocessors, such as exceptions, external interrupts, speculative execution, and out-of-order execution using Tomasulo algorithm. We have mechanically proved that our MA model executes each instruction as specified in the ISA model.

During the verification process, we extracted from our machine design various invariant conditions, which are defined as 20 predicates. We incrementally found these invariant conditions using our invariant strengthening procedure. During the strengthening process, we verified all the invariant conditions independently, effectively breaking down the problem of verification into pieces. Individual invariant conditions are often defined with recursive predicates using our MAETT abstraction of an MA state. Because of the way they are defined, the verification of such conditions can be further broken down to smaller problems by using mathematical induction and case analysis.

Our verification technique seems to scale well with respect to the size of the implementation, even though the process is labor intensive. We found 14 subtle bugs in the design, which had been tested using simulation before applying formal verification.

References

1. B. Brock and W. Hunt, Jr. Formally Specifying and Mechanically Verifying Programs for the Motorola Complex Arithmetic Processor DSP. In *1997 IEEE International Conference on Computer Design*, pages 31–36, IEEE Computer Society. October 13–15, 1997.
2. J. R. Burch and D. L. Dill. Automatic Verification of Pipelined Microprocessor Control. In *Computer Aided Verification, CAV '94*, LNCS 818, pages 68-80, Springer Verlag, 1994.
3. J. R. Burch. Techniques for verifying superscalar microprocessors. In *Design Automation Conference, DAC '96*, June 1996.
4. W. Damm and A. Pnueli. Verifying out-of-order executions. In D. Probst, editor, *CHARME '97*. Chapman & Hall, 1997.
5. J. Hennessey and D. Patterson. *Computer Architecture: a Quantitative Approach*, Morgan Kaufmann Publishers, Inc., 1996.
6. R. Jones, J. Skakkebæk, and D. Dill. Reducing Manual Abstraction in Formal Verification of Out-of-Order Execution. In *Formal Methods in Computer-Aided Design, FMCAD'98*, LNCS 1522, pages 2-17, Springer-Verlag, 1998.
7. M. Kaufmann and J S. Moore. ACL2: An Industrial Strength Version of Nqthm. Proceedings of the *Eleventh Annual Conference on Computer Assurance (COMPASS-96)*, pages 23-34, IEEE Computer Society Press, June 1996.
8. K. L. McMillan. Verification of an Implementation of Tomasulo's Algorithm by Compositional Model Checking. *Computer Aided Verification, CAV '98*, LNCS 1427, pages 110-121, Springer Verlag, 1998.
9. Ravi Hosabettu, Mandayam Srivas and Ganesh Gopalakrishnan. Decomposing the Proof of Correctness of Pipelined Microprocessors. *Computer Aided Verification, CAV'98*, LNCS 1427, pages 122-134, Springer Verlag, 1998.
10. J. Sawada and W. Hunt, Jr. Trace Table Based Approach for Pipelined Microprocessor Verification. *Computer Aided Verification, CAV'97*, LNCS 1254, pages 364-375, Springer Verlag, 1997.
11. J. Sawada and W. Hunt, Jr. Processor Verification with Precise Exceptions and Speculative Execution. *Computer Aided Verification, CAV'98*, LNCS 1427, pages 135-146, Springer Verlag, 1998.
12. J. Skakkebæk, R. Jones and D. Dill. Formal Verification of Out-of-Order Execution Using Incremental Flushing. *Computer Aided Verification, CAV'98*, LNCS 1427, pages 98-109, Springer Verlag, 1998.
13. M. Srivas and M. Bickford. Formal Verification of a Pipelined Microprocessor. *IEEE Software*, pages 52-64, September 1990.
14. M. K. Srivas and S. P. Miller. *Formal Verification of a Commercial Microprocessor.* Technical Report SRI-CSL-95-12, SRI Computer Science Laboratory, July 1995.
15. P. J. Windley and J. R. Burch. Mechanically Checking a Lemma Used in an Automatic Verification Tool. *Formal Methods in Computer-Aided Design, FMCAD '96*, LNCS 1166, pages 362-376, Springer Verlag, 1996.