Processor Verification with Precise Exceptions and Speculative Execution

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 describe a framework for verifying a pipelined microprocessor whose implementation contains precise exceptions, external interrupts, and speculative execution. We present our correctness criterion which compares the state transitions of pipelined and non-pipelined machines in presence of external interrupts. To perform the verification, we created a table-based model of pipeline execution. This model records committed and in-flight instructions as performed by the microarchitecture. Given that certain requirements are met by this table-based model, we have mechanically verified our correctness criterion using the ACL2 theorem prover.

1 Introduction

We have studied the verification of a pipelined microprocessor whose implementation contains speculative execution, external interrupts and precise exceptions. The verification of pipelined microprocessors has been studied[1, 12, 6, 13], but complicated features, such as exception mechanisms, are often simplified away from the implementation model. Several verified microprocessor designs contain exception mechanisms[4, 11]; however, they contain only one kind of exception and require only a few cycles before exception handling starts. Modern microprocessors have multiple exception types, which can occur simultaneously in its pipeline. Correct handling of an exception requires synchronizing and saving the machine state, which may take many clock cycles. This synchronization process may itself cause further exceptions.

Modern processors often execute a large number of instructions speculatively using branch prediction mechanisms. The processor has to keep track of these instructions correctly so that speculatively executed instructions following a mispredicted branch have no side-effect. Also speculatively executed instructions may themselves cause exceptions, which may need to be ignored.

* This research was supported in part by the Semiconductor Research Corporation under contract 97-DJ-388.

To investigate these issues, we designed a processor model which can speculatively execute instructions and simultaneously detect multiple exceptions while executing instructions out-of-order. This machine has been specified at the instruction-set architecture level and micro-architecture level. We discuss the machine specification in Sect. 2.

Previously, we used a correctness criterion for verifying a pipelined microprocessor which did not contain exceptions[10]. In Sect. 3, we have extended this correctness criterion to permit the verification of a design containing speculative execution and external interrupts.

We have modeled the behavior of our processor using an intermediate model, called a MAETT, which records all executed instructions. This model, given in Sect. 4, presents an abstraction of the behavior of our pipelined design on speculative execution and exceptions. Using this model, we wrote an invariant condition that meets several requirements, and show that these requirements are strong enough to prove the correctness criterion. The proof has been carried out with ACL2 theorem prover[9]. A brief proof sketch is given Sect. 5. The verification of the invariant condition is in progress.

2 Hardware Specifications

Our processor model has been specified at two levels: its micro-architecture (*MA*) and its instruction-set architecture (*ISA*). At the ISA level, we only describe the states of the components visible to the programmer, which are shown as shaded boxes in Fig. 1. We specify the ISA behavior with an instruction interpreter function ISA-step(), which takes a current ISA state and an external interrupt input and returns the state after executing a single instruction. At the MA level, we describe the behavior of all components shown in Fig. 1. The behavioral function MA-step() takes a current MA state and its external inputs, and returns the state after one clock cycle of execution. The ISA model is a non-pipelined machine specification while the MA model is pipelined.

Our ISA model implements eleven instructions, each in a different instruction class. For instance, ADD is the only integer operation instruction. For the purpose of our investigation, parameters such as the number of instructions, registers, and the register width are not critical. The ISA specification describes the action for external interrupts and internal exceptions. When an exception occurs, the processor saves some states in special registers, switches to supervisor mode, and jumps to the address specific to the exception type.

The MA specification gives an abstract description of the complete design shown in Fig. 1, as well as the exception mechanism, branch prediction unit, and memory-write buffers. It fetches and commits instructions in program order, but it has the capability to issue up to three instructions to the execution units simultaneously and does execute instructions in an out-of-order manner. The machine can hold as many as 15 instructions in the pipeline, and 12 instructions can be speculatively executed.

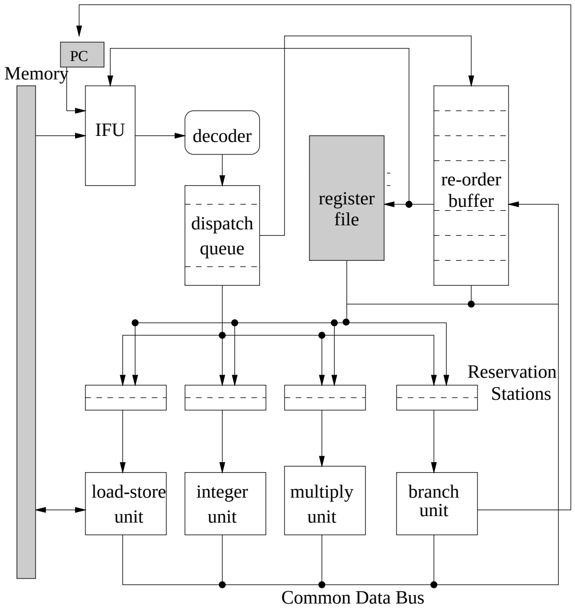

Fig. 1. Block Diagram of Our Pipeline Machine Design

Instructions are executed as follows. A fetched instruction is decoded and dispatched to an appropriate reservation stations, where the instruction waits for its operands. Once an instruction has all necessary values, it is issued to the corresponding execution unit, and the result is written to the re-order buffer[7]. Finally, instructions are committed in program order. Committing is the point where the instruction actually takes its effect. Speculatively executed instructions may reach the re-order buffer, but are only committed if appropriate.

Our MA deals with four types of exceptions: fetch errors, decode errors, data access errors, and external interrupts. The first three exceptions have internal causes, and they are called *internal exceptions*. All exceptions are *precise*; that is, the correct machine state is saved so that the executed program can be restarted from the point where the exception occurred. To achieve this, our machine satisfies the following properties for precise exceptions:

1. All instructions preceding an exception must complete their operation.
2. All partially executed instructions following an exception must be abandoned with no side-effect.

The machine may take a large number of machine cycles before it actually starts exception handling, because the first condition requires completion of partially executed instructions that precede the exception in program order. The re-order buffer is used to sort out the instructions to be completed from those to be abandoned[5]. If multiple exceptions are detected in the pipeline, only the earliest exception in program order is processed. Our MA design does not contain any imprecise exceptions, and we have not considered the verification a processor with imprecise exceptions.

3 Correctness Criterion

Our verification objective is to show that the MA design correctly executes instructions as specified by the ISA. Various ways to show the equivalence between the two levels have been presented. Burch and Dill verified pipelined designs using a correctness criterion that involves pipeline flushing[2]. Although this criterion with flushing has been extended to cover superscalar processors[3, 14], it does not address speculative execution and external exceptions.

We previously used the correctness criterion shown as diagram (a) in Fig. 2 to verify a pipelined design[10]. This diagram compares two paths. The lower path runs the MA design for an arbitrary number of clock cycles from a flushed pipeline state MA_0 to another flushed state MA_n, which causes m ISA instructions to be executed. By stripping off states not visible to the programmer, we can project MA_n to ISA_m. The upper path first projects MA_0 to an initial ISA state ISA_0 and then runs the ISA specification for m cycles to get the final state ISA_m. By comparing ISA_m obtained by following the different paths, we can check whether the MA design conforms to the ISA specification.

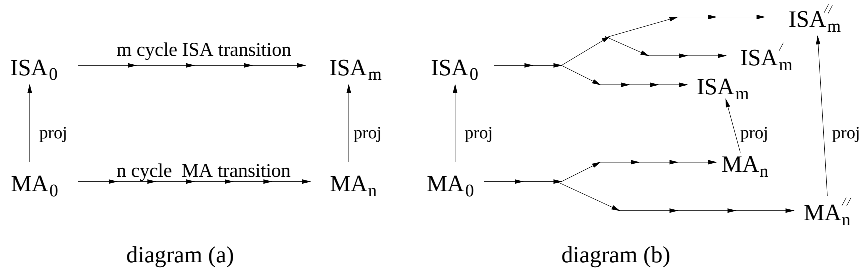

Fig. 2. Correctness Diagrams

In a correctly implemented MA design, speculatively executed instructions after a mispredicted branch should have no side-effect on the programmer visible state. This can be checked by verifying diagram (a), because the ISA executes instructions one-by-one. The correctness diagram shows that instructions are executed correctly independently of how branches are predicted.

Let us consider how internal exceptions affect the diagram. The ISA specification describes the machine behavior for internal exceptions; it specifies what states are stored in special registers, what the next PC value is, and so on. We want to show the MA design implements this action correctly, but we also want to check it implements precise exceptions. Since the ISA specification executes instructions one-by-one, it captures the requirements for precise exceptions given in Sect. 2. The correct behavior on multiple exceptions in the pipelined MA is also implied by the ISA specification, because it always processes exceptions in program order. These are our reasons to claim that verifying diagram (a) demonstrates precise handling of internal exceptions, as well as the correct action on exceptions. We do not check how exceptions are handled by exception handlers, since this is a software verification problem[11].

External exceptions make the problem more complicated. The ISA specification function *ISA-step*() takes an external interrupt signal as its argument, and describes the action of an external interrupt as it does for internal exceptions. The problem is that the non-determinism introduced by the external signal can lead to different final ISA states, as shown in diagram (b). The commutative diagram holds only for the ISA state transitions which interrupt the same instructions as the MA does. Since supplying different environments to the MA will cause different instructions to be executed and interrupted, we need to find the corresponding ISA sequence for each MA state sequence with different input signals.

> **Correctness Criterion:** For an arbitrary MA execution sequence from a flushed state MA_0 to another flushed state MA_n, there exists a corresponding ISA execution sequence from ISA_0 to ISA_m. This sequence executes and interrupts the same instructions as occur in the MA execution sequence, and satisfy $ISA_0 = proj(MA_0)$ and $ISA_m = proj(MA_n)$.

The problem of self-modifying code is inseparable from pipelined processor verification, because instructions can be fetched from the main memory prior to the completion of writes by previous instructions. As a part of the statement of our correctness criterion, we assume that the program executed between the initial flushed MA state and the final flushed MA state does not modify itself.

Our correctness criterion does not imply the complete correctness of a microprocessor design. Intuitively, our correctness criterion only suggests that the execution of instructions is correct if they are in fact executed. The liveness of the processor is not part of our criterion, but can be proven separately. The criterion suggests that external interrupt signals are processed correctly, but it does not guarantee that all the interrupt signals actually interrupt the machine. For a real time system, we may further want to show that the processor responds to an external signal in a bounded amount of time.

4 MAETT for Speculative Execution and Exceptions

We have extended our Micro-Architectural Execution Trace Table (*MAETT*)[10] to model the behavior for speculative execution, internal exceptions and external interrupts. A MAETT is an abstraction of an MA state, which contains redundant information that makes it straightforward to specify machine invariants.

A MAETT is a list whose entries correspond to either a committed or in-flight instruction. Each entry represents an instruction with a data structure whose fields are shown in Table 1. A MAETT records all instructions that are executed from the initial MA state, and the size of a MAETT is unbounded. A MAETT grows as more instructions are fetched, and shrinks when speculatively executed instructions are abandoned. Instructions are recorded in the ISA execution order. The MAETT corresponding to a flushed MA state contains only committed instructions.

In the rest of the paper, we write $(I_1, \ldots, I_l)$ to designate a MAETT which records instructions $I_1, \ldots, I_l$. $ISA_0 \xrightarrow{I_1} ISA_1 \xrightarrow{I_2} \cdots \xrightarrow{I_l} ISA_l$ designates an ISA state sequence that executes instructions $I_1, \ldots, I_l$. The arrow labeled with I_i means state ISA_{i-1} changes to ISA_i under the action of I_i. Since each MAETT entry contains the ISA states before and after executing the corresponding instruction, it is easy to reconstruct the ISA state sequence $ISA_0 \xrightarrow{I_1} \cdots \xrightarrow{I_l} ISA_l$ from a MAETT $(I_1, \ldots, I_l)$.

Field name	Brief description
`ID`	Identity of I_i.
`word`	Instruction word.
`stg`	Current pipeline stage of I_i.
`robe`	Reorder buffer entry where I_i is stored.
`modify?`	Flag to show whether I_i is a modified instruction.
`speculative?`	Flag to show whether I_i is speculatively executed.
`br-predict?`	Outcome of branch prediction if I_i is a conditional branch.
`exintr?`	Flag to show whether I_i is interrupted.
`pre-ISA`	ISA_{i-1}, i.e., ISA state before executing I_i.
`post-ISA`	ISA_i, i.e., ISA state after executing I_i.

Table 1. Data structure for representing an instruction.

We define a MAETT step function *MAETT-step*() to simulate the MA state transition. *MAETT-step*() takes the current MA state, its corresponding MAETT, and external inputs, and returns a new MAETT representing the MA state one cycle later.

Each clock cycle, a MAETT is updated in concert with the MA state transition. Suppose the current MAETT is $(I_1, \ldots, I_l)$. If the MA fetches a new instruction I_{l+1}, *MAETT-step*() returns an extended MAETT $(I_1, \ldots, I_l, I_{l+1})$. The fields of each in-flight instruction are modified to reflect its progress in the pipeline.

When the MA abandons instructions following a mispredicted branch or an exception, MAETT entries corresponding to these instructions are eliminated. Figure 3 shows branching of an ISA state transition sequence due to an external interrupt. If instruction I_i is not interrupted, state ISA_{i-1} changes to ISA_i. If I_i is interrupted, it changes to ISA'_i. Before an external interrupt occurs, the MA executes instructions along the normal execution path, and the MAETT contains instructions $(I_0, \ldots, I_i, I_{i+1}, \ldots, I_k)$, and looks like MAETT (a). When an exter-

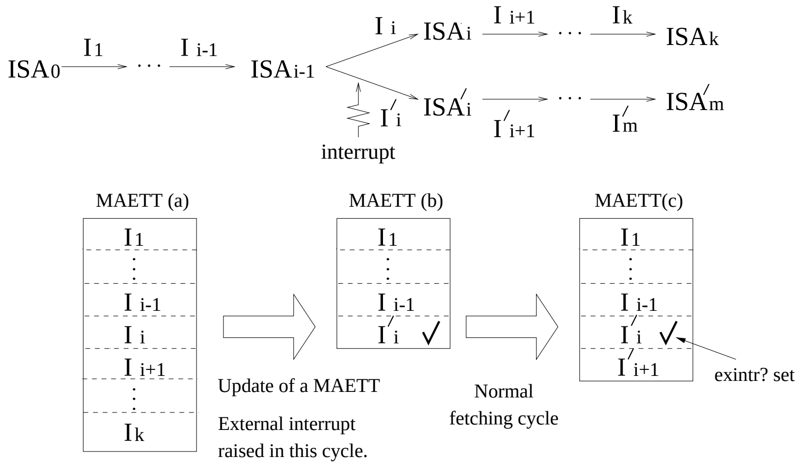

Fig. 3. Branching of ISA state sequence on an external interrupt. Corresponding MAETTs are also shown.

nal interrupt signal is received, the MA design starts synchronizing the machine and picks an instruction to be interrupted. If the interrupted instruction is I_i, instructions $I_i, \ldots, I_k$ are abandoned. When this happens, *MAETT-step*() eliminates the abandoned instruction and returns $(I_0, \ldots, I_{i-1}, I'_i)$, which is shown as MAETT (b). *MAETT-step*() replaces I_i with I'_i, whose **post-ISA** field contains ISA'_i and **exintr?** flag is set to record the fact that I'_i was where the interrupt occurred. By the time we reach the final MA state, the MAETT will contain a history of instructions that shows where the interrupts occurred. From the MAETT, we can easily reconstruct the ISA execution sequence that satisfies the commutative diagram of our correctness criterion. Similarly, we use the MAETT to model speculative execution and internal exceptions.

5 Invariants Conditions and Correctness Criterion

We have defined various invariant properties about our pipeline implementation. Instead of discussing a complete list of invariant properties and techniques to define them, we present the minimum requirements that our invariant condition should satisfy, and we give a sketch of the proof of our correctness criterion using them.

In the following argument, we assume that MT_k is the MAETT representing an MA state MA_k, and MT_k contains l instructions $I_1, \ldots, I_l$. MT_k essentially represents the ISA transitions $ISA_0 \stackrel{I_1}{\to} \cdots \stackrel{I_l}{\to} ISA_l$, by storing ISA states in fields **pre-ISA** and **post-ISA**.

We defined an invariant condition as $Inv(MT_k, MA_k)$ that should satisfy following requirements.

Requirement 1. *If* $Inv(MT_k, MA_k)$ *holds and* MA_k *is a flushed pipeline state, then every instruction* I_i *in* MT_k *is committed.*

A mispredicted conditional branch and an error-causing instruction will eventually cause instructions to be abandoned. $MAETT\text{-}speculative?(MT_k)$ is a predicate to check whether MT_k contains such an uncommitted mispredicted branch or an uncommitted error-causing instruction. The program counter in MA_k should correctly point to the next instruction I_{l+1} to be fetched by ISA_l, unless it is fetching instructions speculatively.

Requirement 2. *If* $Inv(MT_k, MA_k)$ *holds and* $MT_k = (I_1, \ldots, I_l)$*, then*

$$\neg MAETT\text{-}speculative?(MT_k) \Rightarrow MA\text{-}pc(MA_k) = ISA\text{-}pc(ISA_l).$$

Results of instructions are written back to the register file when the instructions commit.

Requirement 3. *If* $Inv(MT_k, MT_k)$ *is true and* I_{i+1}*,* $i < l$*, is the first uncommitted instruction in* $MT_k = (I_1, \ldots, I_l)$*, then*

$$MA\text{-}regs(MA_k) = ISA\text{-}regs(ISA_i).$$

Requirement 4. *If* $Inv(MT_k, MA_k)$ *is true and* I_{i+1}*,* $i < l$*, is the first memory store instruction whose memory write is not completed, then*

$$MA\text{-}mem(MA_k) = ISA\text{-}mem(ISA_i).$$

Requirement 5. *For an arbitrary flushed initial state* MA_0 *and its MAETT* MT_0*,* $Inv(MT_0, MA_0)$ *holds.*

We must show that the invariant condition *Inv*() is preserved during MAETT updates; however, if self-modified code is executed, the pipelined MA may not work correctly with respect to ISA specification. To characterize this problem, we defined a predicate *commit-self-modified-inst-p*(MT) to check whether any instruction in MT is self-modified and also committed. Our invariant is preserved only when there is no such instruction. The machine can speculatively execute self-modified instructions, if they are eventually abandoned and have no effect on the programmer visible state.

Requirement 6. *Suppose* MA_{k+1} *and* MT_{k+1} *are the next MA state and next MAETT, that is:*

$$\begin{aligned} MA_{k+1} &= MA\text{-}step(MA_k, Inputs_k), \\ MT_{k+1} &= MAETT\text{-}step(MT_k, MA_k, Inputs_k). \end{aligned}$$

Then

$$Inv(MT_k, MA_k) \Rightarrow Inv(MT_{k+1}, MA_{k+1}) \vee commit\text{-}self\text{-}modified\text{-}inst\text{-}p(MT_{k+1})\ .$$

Requirements 5 and 6 assure that *Inv*() is true for all reachable states. Requirements 2, 3 and 4 constrain the relation between an MA state and the ISA state sequence represented by its MAETT. An example of this relation is shown in Fig. 4. Let us assume that, at the state MA_i, instruction I_1 is committed, I_2 is waiting for its memory operation to complete, I_3 and I_4 are being executed, and I_5 is not fetched yet. Requirement 2 implies that the program counter in MA_i is equal to that of ISA_4, because it should point to instruction I_5 in both states. Examples of Requirement 3 and 4 are also shown in the figure; the register file in MA_i is equal to the register file in ISA_2, and the memory of MA_i is equal to the memory in ISA_1. If all instructions $I_1, \ldots, I_5$ are committed, then the skewed dashed lines align to relate the final MA state and the final ISA state.

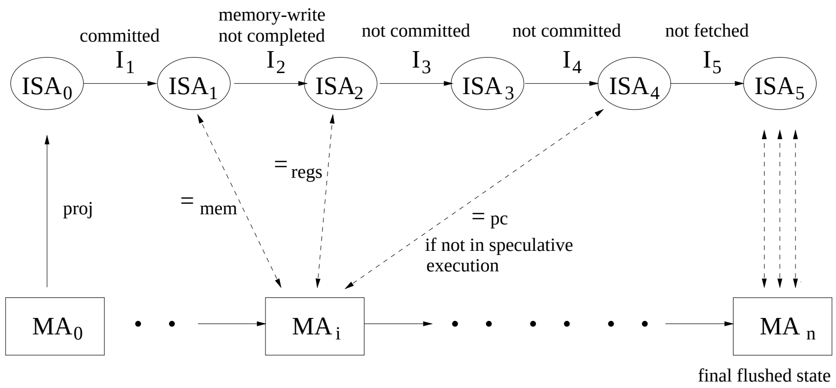

Fig. 4. Relations between MA and ISA sequences.

Checking the first five requirements is easy, since they don't involve a state transition of MA. However, checking Requirement 6 takes extensive analysis of MA state transitions, and this is where the actual verification activity of the hardware design takes place. In the rest of this section, we summarize the proof of our correctness criterion, assuming that *Inv*() satisfies Requirement 6.

From Requirements 2, 3 and 4 and the definition of *MAETT-speculative?*(), it is straight forward to get the following lemma.

Lemma 1. *Suppose that invariant condition* $Inv(MA_n, MT_n)$ *holds, and every instruction* I_i *in* $MT_n = (I_1, \ldots, I_m)$ *is committed, then*

$$proj(MA_n) = ISA_m.$$

Lemma 2. *Let* MA_0 *be a flushed MA state,* MA_n *be the state after* n *MA transitions, and* MT_n *be the MAETT of* MA_n*. Then*

$$\neg commit\text{-}self\text{-}modified\text{-}inst\text{-}p(MT_n) \Rightarrow Inv(MT_n, MA_n).$$

Proof. Although a MAETT trace grows and shrinks, committed instructions are never removed from a MAETT. So any instruction in the intermediate MAETT MT_i for $i < n$ is also in the final MAETT MT_n. This implies

$$\neg \textit{commit-self-modified-inst-p}(MT_n) \Rightarrow \neg \textit{commit-self-modified-inst-p}(MT_i).$$

Hence, if $\textit{commit-self-modified-inst-p}(MT_n)$ is false, we can combine Requirement 5 and 6 and get $Inv(MT_n, MA_n)$ by induction. ◇

Corollary 3. *If MAETT MT_n does not contain any modified instructions, then $Inv(MT_n, MA_n)$ is true.*

From Requirement 1, Lemma 1 and Corollary 3, we get the following theorem.

Theorem 4. *Suppose MA_0 is a flushed MA state and $ISA_0 = proj(MA_0)$. After n cycles of MA state transitions with arbitrary input sequence Inputs, we arrive at another flushed MA state $MA_n = \textit{MA-stepn}(MA_0, Inputs, n)$. Then there exists a corresponding ISA transition sequence $ISA_0 \xrightarrow{I_1} \cdots \xrightarrow{I_m} ISA_m$, and if this sequence is not self-modifying, then $proj(MA_n) = ISA_m$.*

The ISA sequence $ISA_0 \xrightarrow{I_1} \cdots \xrightarrow{I_m} ISA_m$ can be constructed by calculating the MAETT of state MA_n with MAETT step function *MAETT-step*().

The proof of Theorem 4 has been mechanically checked with ACL2 theorem prover. At this point, we have not yet completed the verification of Requirement 6. As pointed out earlier, the real verification problem is finding and verifying the invariant conditions. A merit of using the MAETT is that it helps us to define various pipeline invariants. For example, one invariant is that instructions are dispatched and committed in the ISA execution order. This is nicely defined as a recursive function over the list of instructions in a MAETT. We have verified 6 out of 18 invariant conditions of *Inv*(). So far, this verification process found three bugs in the design, even though the design had been simulated.

Our proof presented here reduces the problem of checking the correctness criterion to the problem of verifying our requirements. In this sense, what we have presented here is a framework for verifying a microprocessor. Our requirements are strong enough to prove our correctness criterion, since we have carried out the mechanical proof by assuming only those conditions. This suggests the possibility that we can reuse the structure of the proof for other hardware designs which satisfy these requirements, even though the construction of *MAETT-step*() and the verification of the requirements are design dependent.

6 Conclusion

We have described a framework for verifying pipelined machine designs at the micro-architectural level. Our correctness criterion compares MA state transitions between two flushed states to the corresponding ISA state transitions. We discussed why our correctness criterion implies correct speculative execution and precise exceptions. The non-determinism at the ISA level introduced by external interrupts requires us to dynamically construct corresponding ISA transitions. This construction is done by modeling the execution of the MA design with a MAETT, which is essentially a history of committed and in-flight instructions. We defined an invariant condition that satisfies several requirements, and proved our correctness criterion under the assumption that the invariant conditions are preserved during MA state transitions. The proof has been mechanically checked by the ACL2 theorem prover. We have shown that our requirements are strong enough to carry out the proof.

We are currently verifying the invariant condition. We also would like to check whether an external interrupt is guaranteed to be processed. In our MA design, some external interrupts are dropped because internal exceptions have higher priority or because multiple interrupts are received within too short of an interval. It is an open question whether the MAETT model can help us to prove properties such as that an isolated external interrupt is guaranteed to be serviced.

References

1. Bishop C. Brock and Warren A. Hunt, Jr. Formally Specifying and Mechanically Verifying Programs for the Motorola Complex Arithmetic Processor DSP. In *1997 IEEE International Conference on Computer Design*, IEEE Computer Society. pp. 31–36, October 13–15, 1997.
2. J. R. Burch, D. L. Dill. Automatic Verification of Pipelined Microprocessor Control, Computer Aided Verification, Lecture Notes in Computer Science 818, Springer Verlag, pages 68-80, 1994.
3. J. R. Burch. Techniques for verifying superscalar microprocessors. In Design Automation Conference, June 1996.
4. M. Coe. Results from Verifying a Pipelined Microprocessor, Master's Thesis, University of Idaho, 1994.
5. H. G. Cragon. Memory Systems and Pipelined Processors, Jones and Bartlett Publishers, Inc., 1996.
6. D. Cyrluk. Microprocessor verification in PVS: A methodology and simple example, Technical Report SRI-CSL-93-12, SRI Computer Science Laboratory, Dec. 1993.
7. J. Hennessey, D. Patterson. Computer Architecture a Quantitative Approach, Morgan Kaufmann Publishers, Inc., 1996.
8. W. A. Hunt, Jr., B. Brock. A Formal HDL and Its Use in the FM9001 Verification. In C.A.R. Hoare and M.J.C. Gordon, editors, Mechanized Reasoning and Hardware Design, pages 35-48. Prentice-Hall International Series in Computer Science, Engle wood Cliffs, N.J., 1992.
9. M. Kaufmann, 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.
10. J. Sawada, W. Hunt, Jr. Trace Table Based Approach for Pipelined Microprocessor Verification, Computer Aided Verification, Lecture Notes in Computer Science 1254, Springer Verlag, pages 364-375, 1997.
11. M. Srivas, M. Bickford. Formal Verification of a Pipelined Microprocessor, IEEE Software, pages 52-64, September 1990.

12. M. K. Srivas, S. P. Miller. Formal Verification of a Commercial Microprocessor, Technical Report SRI-CSL-95-12, SRI Computer Science Laboratory, July 1995.
13. S. Tahar, R. Kumar. Formal Verification of Pipeline Conflicts in RISC Processors, Proc. European Design Automation Conference (EURO-DAC94), Grenoble, France, IEEE Computer Society Press. pages 285-289, September 1994.
14. P. J. Windley, J. R. Burch. Mechanically Checking a Lemma Used in an Automatic Verification Tool, Formal Methods in Computer-Aided Design, Lecture Notes in Computer Science 1166, Springer Verlag, pages 362-376, 1996.