Trace Table Based Approach for Pipelined Microprocessor Verification

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

[1] Department of Computer Sciences, University of Texas, Austin, TX 78712, USA
E-mail: sawada@cs.utexas.edu
[2] Computational Logic, Inc., 1717 West 6th Street, Suite 290, Austin, TX 78703, USA E-mail: hunt@cli.com

Abstract. This paper presents several techniques for formally verifying pipelined microprocessor implementations that contain out-of-order execution and dynamic resolution of data-dependent hazards. Our principal technique models the trace of executed instructions using a table-based representation called a MAETT. We express invariant properties of pipelined implementations by specifying relations between fields in the MAETT. To show the viability of this technique, we have proved the correctness of a simple out-of-order completion pipelined microprocessor design using the ACL2 theorem prover. This verification was performed incrementally by proving that the specified relations hold for all micro-architectural states reachable from a flushed implementation state, eventually permitting us to prove that the entire pipelined machine design implements its ISA specification.

1 Introduction

In our project, we have been developing techniques to verify the correctness of pipelined microprocessor implementations with relatively complex pipeline control logic. To achieve this goal, we designed an example microprocessor which features out-of-order instruction completion, speculative instruction fetch, and dynamic resolution of *read-after-write (RAW) dependencies* (also called *true dependencies*) and *write-after-write (WAW) dependencies* (also called *antidependencies*) [6, 8]. Our method employs a technique we call a *Micro-Architectural Execution Trace Table (MAETT)*. With the help of the MAETT, we define various properties of our pipelined implementation, and incrementally prove that each of them holds for all the reachable pipeline states. The correctness for our pipelined implementation is then proved from these pipeline properties.

Our motivation is to permit the verification of complex pipelined microprocessor implementations. Although some measure of success has been achieved[7], formal verification has not been widely applied to the micro-architectural design of general-purpose microprocessors. Specifically, for pipelined machine designs

* This research was supported in part by the Semiconductor Research Corporation under contract 96-DP-388.

there have been several efforts aimed at verifying pipelined implementations; however, most of the formally verified designs have a short and simple pipeline structure. An impediment to applying such proof techniques to commercial designs is the failure of current formal approaches to address the verification of today's processors with modern performance optimizations, including superscalar super-pipelining, out-of-order execution, and speculative execution.

2 Background

There have been a number of earlier efforts to verify pipelined microprocessor designs with interactive theorem provers[5, 10, 11, 13]. Typically in these projects, they show the equivalence of an *instruction-set architecture (ISA)* and a corresponding *micro-architectural* design. An ISA is a non-pipelined abstract machine which specifies the effects of individual instructions, while a micro-architectural design exposes the pipeline structure of an actual machine implementation. The equivalence between these levels is defined by a commutative diagram with an abstraction function mapping a micro-architectural state to an ISA state. In many cases, however, a pipeline state may not be directly mapped to an ISA state because of the *latency* of pipeline events.

One way to avoid this problem is to define a *skewed* [11] abstraction function, which relates multiple pipeline states to a single ISA state. Skewed abstraction functions are often complex, and they are vulnerable to minor design modifications because skewed abstraction functions are defined with the expectation that very specific timing properties are preserved between pipeline events.

Instead of defining an artificially-constructed skewed abstraction function, Burch and Dill[3] proposed a correctness criterion where they first flush the pipeline of the micro-architecture by stalling and then map the resulting micro-architectural state to an ISA-state. Diagram(a) in Fig. 1 shows their correctness criterion. Using induction, Diagram(a) can be extended to the more general criterion shown in Diagram(b). Diagram(b) suggests that the n-cycle micro-state transition which starts and ends with flushed pipeline states is equivalent to the m-cycle state transition at the ISA level, where m is the number of instructions executed in the micro-state transition.

Burch and Dill's approach is more robust to design modifications in the micro-architecture than the method with a skewed abstraction function, because it does not depend on specific timing between pipeline events. It is also applicable to pipelined machines with out-of-order execution without any modification.

Burch and Dill's automated verification procedure[3] is effective in verifying their correctness criteria for processors with a simple pipeline control logic, because it treats the data-path symbolically and only verifies the control. However, the procedure can fail for large and complex processor designs, because the number of examined cases explodes as the control part becomes complicated. Various studies have been done to improve the capability of the technique [8, 14]. Especially, [14] decomposed Diagram (a) to simpler diagrams, which were used in superscalar verification in [4]. However, it is still difficult to see how we can apply

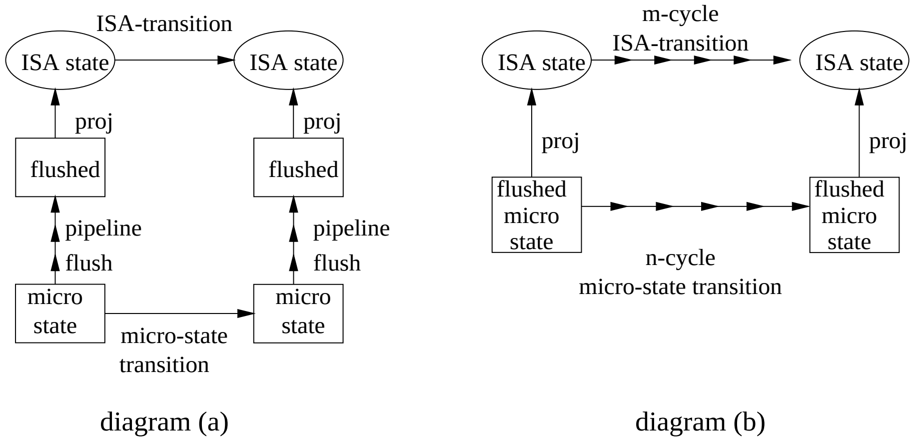

Fig. 1. Pipeline Correctness Criteria

the procedure to the verification of complex micro-architectural designs used in today's microprocessors.

Since we want to study the verification of out-of-order pipelined micro-architectural designs, we believe it will be necessary to use general and robust correctness criteria similar to the one used by Burch and Dill. We also think that dividing the complete verification problem into smaller subproblems is the key to the successful verification of complex processor implementations. In the verification of our example pipelined machine, we incrementally verify various pipeline properties first, from which we can prove our correctness criteria. An examples of such properties will be given in Sec. 3.2

Several earlier projects have focused on the verification of pipeline properties. Tahar and Kumar[12] verified that their DLX implementation does not have RAW hazards and other pipeline conflicts, but they didn't verify the equivalence between their ISA and micro-architecture. Agaard and Leeser[1] proposed a framework for pipeline verification focusing on structural hazards, which is similar in character to what we describe here; however, they did not mechanize its application. Our verification is complete in the sense that we mechanically verified pipeline properties and the more general criteria shown in Fig. 1.

3 Verification Approach

We first present our example machine in Subsection 3.1. Using the example, we discuss how to define and verify pipeline properties in Subsection 3.2, and how to prove the correctness criteria in Subsection 3.3. We briefly discuss the mechanization of the verification in Subsection 3.4.

3.1 An Example Pipelined Machine Design

We designed a new pipelined machine to study the verification of pipelined machines with out-of-order execution, where instructions may not be executed in the ISA program order. We did not use a published pipelined implementation such as DLX, because we wanted to study machine designs with a more complex control logic. A typical DLX implementation has a simple pipeline control with few pipeline interlocks. On the other hand, our example machine includes out-of-order execution and speculative instruction fetch. We also wanted our example machine to be as small as possible, so that we can rapidly try different approaches. Our example design has a very small instruction set and has neither external interactions nor interrupts.

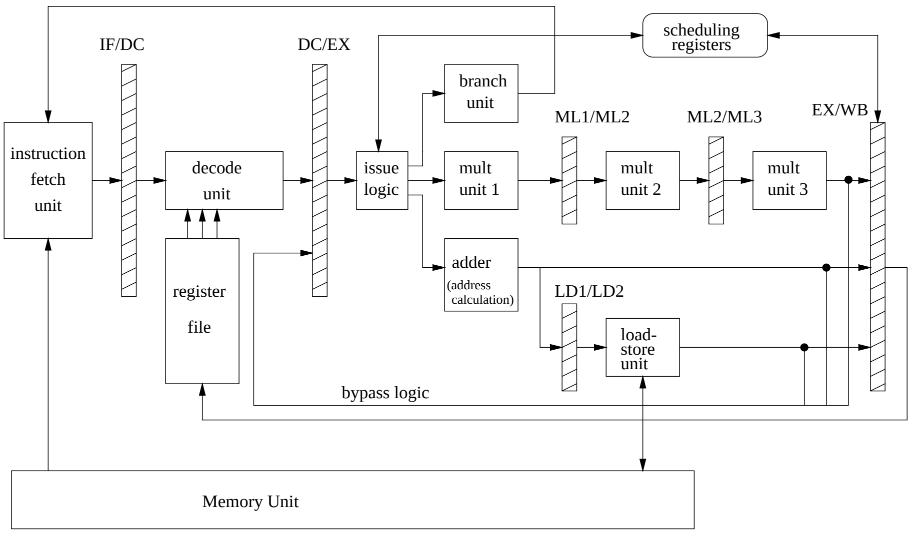

Fig. 2. A block diagram of our example processor.

A block diagram of our example pipelined machine is shown in Fig. 2. The maximum length of the pipeline is six stages and each stage is separated by pipeline latches shown as patterned rectangles. This machine has multiple execution units, including a simple adder and a pipelined multiplier. Different execution units have different numbers of pipeline stages; one stage for an addition, two for a load-store instruction, and three for a multiplication. Consequently, our example machine may complete the execution of instructions in an out-of-order fashion, even though the instructions are issued in order. Since our design does not have a re-order buffer, instructions retire immediately after their execution completes, and the register file may be updated in an out-of-order fashion.

A pipelined design with out-of-order register file updates may exhibit certain hazards. Consider a code sequence including three assignment instructions:

```
R1 := R2 * R3      (i)
R5 := R1 + R4      (ii)
R1 := R6 + R7      (iii)
```

In this code, instruction (ii) uses the new value of register `R1` which is returned by (i), and we say (ii) has a *read-after-write(RAW) dependency* on (i). A correct pipelined design has to postpone the execution of (ii) and/or bypass the result of (i) to earlier stages, so that (ii) obtains the correct value of `R1`. Otherwise, (i) may calculate an incorrect result, causing a *read-after-write(RAW) hazard*.

We can also see that (iii) updates the same register `R1` as (i) does. We say (iii) has a *write-after-write(WAW) dependency* on (i). If (iii) updates register `R1` earlier than (i) by out-of-order execution, (i) destroys the value stored by (iii), leaving `R1` containing a wrong value. To prevent this incorrect behavior, which is called a *write-after-write(WAW) hazard*, (i) and (iii) must update `R1` in program order.

A *structural hazard* occurs when multiple instructions simultaneously try to use the same hardware resource. With regard to our machine design, instructions may simultaneously occupy the write-back stage due to the various lengths of the execution units. Without proper arbitration or scheduling, the results of the execution of instructions may be corrupted.

In order to avoid these hazards, our issuing logic suspends the issue of any instruction that may cause a hazard. Once an instruction is issued, it is guaranteed that no hazard will occur due to the instruction. The scheduling registers shown in Fig. 2 keep track of the instructions in the execution units, and the issuing logic refers to them when deciding whether it can issue an instruction.

Using the ACL2 logic[2, 9], we have defined a ISA-level next-state function, `ISA-state-step()`, as an instruction interpreter and a micro-architectural next-state function, `micro-state-step()`, as a clock-by-clock cycle interpreter. The control of the pipeline is specified concretely, while the data-path is specified with abstract functions. For instance, our adder is represented by the ACL2 integer addition function, which is treated as an uninterpreted function during the verification process. One exception is our pipelined Wallace-tree multiplier, which is specified more concretely using carry-save adders and carry-propagate adders as primitive blocks. This is for studying data-path verification in the context of pipeline verification.

3.2 The MAETT and Pipeline Properties

Let us first discuss why the MAETT is useful with our verification approach for pipelined micro-architectural designs. We often wish to define invariants of the target design, but some pipeline properties are difficult to define as a predicate taking a micro-architectural state as its sole argument. This is because micro-architectural designs are usually optimized for speed and size, and a micro-architectural state may not contain enough information to directly define such predicates.

For example, an out-of-order pipelined machine design may suffer WAW hazards, and we would want to check that no such hazards occur in our micro-architectural design. However, a WAW hazard may involve an already retired instruction, so it is difficult to check for such a hazard by simply examining a micro-architectural state, because the machine may not contain information about retired instructions. This is why writing a WAW hazard-free predicate is easier if we have a richer and more redundant representation of the execution trace of the machine, that is, the MAETT.

A MAETT M is an *unbounded* list of all instructions $I_1, I_2, \ldots, I_n$ which are currently being executed or have already retired since the machine started:

$$M = (I_1\ I_2\ \cdots\ I_n)\ . \tag{1}$$

The ISA executes $I_1, I_2, \ldots, I_n$ in that order. Each instruction I_i is represented by a record like:

$$I_i = (\mathit{Flg}_i, \mathit{PC}_i, \mathit{Inst}_i, \mathit{RA}_i, \mathit{RB}_i, \mathit{RC}_i, \mathit{Stg}_i, \mathit{Regs}_i, \mathit{Mem}_i, \mathit{Misc}_i)\ , \tag{2}$$

where PC_i, Inst_i, RA_i, RB_i, RC_i, Stg_i, Misc_i are respectively the instruction address; instruction word; operand register identifiers of RA, RB and RC; current stage of the instruction; and miscellaneous stage-dependent information. Flg_i is a flag indicating whether I_i is speculatively fetched. Regs_i and Mem_i are the correct register file and memory states after completing all previous instructions $I_1, \ldots, I_{i-1}$ and before executing I_i with the ISA. Let `ISA-state`$(\mathit{PC}, \mathit{Reg}, \mathit{Mem})$ denote the ISA state with PC, Reg, Mem as its program counter, register file and memory, respectively. We can define an "ideal" ISA state S_i corresponding to I_i by:

$$S_i = \texttt{ISA-state}(\mathit{PC}_i, \mathit{Reg}_i, \mathit{Mem}_i)\ . \tag{3}$$

The next ISA state S_{i+1} is related to S_i by:

$$S_{i+1} = \texttt{ISA-state-step}(S_i)\ . \tag{4}$$

In this sense, we say that a MAETT records an ISA execution sequence.

At the same time, a MAETT represents the current micro-architectural state, by recording the current status of each instruction. We define a MAETT updating function `MAETT-step()` to simulate the micro-architectural state transition `micro-state-step()`. If s is a micro-architectural state and M is its MAETT representation, `MAETT-step(`M`,`s`)` gives a MAETT representation of the next micro-state `micro-state-step(`s`)`. `MAETT-step` updates appropriate fields of (2) for instructions already in a MAETT. It may also add a newly fetched instruction to a MAETT or delete speculatively fetched instructions if they are aborted.

The richness and regularity of the MAETT representation help us to define various pipeline properties. We can see this by looking at the definition of our WAW hazard-free predicate `no-WAW-hazards?`. The WAW hazard between the instructions I_k and I_l is defined as:

$$\begin{aligned}\texttt{WAW-violation?}(I_k,I_l) = (&\texttt{reg-writeback-inst?(entry-inst}(I_k)\texttt{)}\\ &\land\ \texttt{reg-writeback-inst?(entry-inst}(I_l)\texttt{)}\\ &\land\ \texttt{same-destination-reg?}(I_k,I_l)\\ &\land\ \texttt{out-of-order-retire?}(I_k,I_l))\ .\end{aligned}$$

This definition assumes that I_k is an earlier instruction than I_l, with regard to the ISA-execution order. The definition says that a WAW hazard between I_k and I_l occurs when both of the instructions are write-back instructions, they share the same destination register, and I_l retires earlier than I_k. Each of the predicates in the definition of `WAW-violation?` is further defined as a relation of the fields of I_k and I_l. For example, the `out-of-order-retire?` is defined as:

$$\begin{aligned}&\texttt{out-of-order-retire?}(I_k,I_l)\\ &= \textbf{let } Stg_k \textbf{ be } \texttt{entry-stg}(I_k) \textbf{ and}\\ &\qquad Stg_l \textbf{ be } \texttt{entry-stg}(I_l)\\ &\quad \textbf{in}\\ &\quad (\textbf{case } Stg_k \textbf{ of}\\ &\qquad\quad \text{EX/WB:}\quad Stg_l \in \{\text{RETIRE}\}\\ &\qquad\quad \text{LD1/LD2:}\quad Stg_l \in \{\text{EX/WB RETIRE}\}\\ &\qquad\quad \text{ML2/ML3:}\quad Stg_l \in \{\text{EX/WB RETIRE}\}\\ &\qquad\quad \text{ML1/ML2:}\quad Stg_l \in \{\text{LD1/LD2 ML2/ML3 EX/WB RETIRE}\}\\ &\qquad\quad \textbf{otherwise:}\ \text{false}\ )\ ,\end{aligned}$$

where $\texttt{entry-stg}(I_i)$ returns Stg_i for I_i in (2). This definition checks for out-of-order retirements by considering different stages Stg_k for I_k. The first case specifies an out-of-order retirement occurring when I_k is still in the EX/WB latch while I_l has already retired. Finally, we can define our global WAW hazard-free property `no-WAW-hazards?` as a recursive predicate checking `WAW-violation?` for all the pairs of instructions in MAETT M:

$$\texttt{no-WAW-hazards?}(M) = \forall I_k, I_l \in M \text{ s.t. } I_k <_M I_l, \neg\texttt{WAW-violation}(I_k, I_l)\ ,$$

where $I_k <_M I_l$ means that I_k appears in M earlier than I_l.

In a similar manner, we have used the MAETT representation to define other pipeline properties which are essential to the correct operation of our design:

`no-structural-hazard?` : No structural hazards occur.

`intermediate-values-ok?` : Predicate checking that the intermediate values of the instruction at each stage are correct. This predicate entails the non-existence of RAW hazards. For instance, an instruction at the issue stage should have obtained the correct source register values from either the register itself or the bypass logic. The correct value of source register RA can be expressed as $\texttt{read-reg}(RA_i, Regs_i)$ for I_i in (2).

`all-insts-in-MAETT?` : All the partially executed instructions in the pipeline are in the MAETT.

`consistent-scheduler?` : The scheduling registers correctly keep track of instructions in the execution units.

`speculative-inst-not-issued?` : No speculatively fetched instructions are issued to the execution units.

`speculative-fetching-ok?` : All speculatively fetched instructions are abandoned before instructions are fetched from the correct branch.

`in-order-issue?` : Instruction issues are in-order.

`no-WAW-hazard?` : No WAW hazards occur.

`in-order-memory-accs?` : No out-of-order memory accesses are performed.

`sync-retires-insts?` : All instructions retire before a subsequent SYNC instruction retires.

`consistent-ISA-states?` : Equation (4) holds for all instructions except those speculatively fetched.

`correct-pc?` : The program counter is pointing to the next instruction to be fetched.

`correct-regs?` : The register file is correct. The correct register file is defined by scanning the MAETT and sequentially accumulating all the side-effects by the retired instructions on the initial register file. In other words, the register file should record all the updates caused by the retired instructions.

`correct-mem?` : The memory reflects all the memory writes by retired STORE instructions.

We define the conjunction of all of these properties to be predicate `MAETT-ok?`. Using this predicate, we proved two lemmas:

Lemma 1 Invariant property of MAETT-ok? . *Suppose s is a micro-architectural state and M is a corresponding MAETT satisfying* `MAETT-ok?(`M`,`s`)`. *Then the next micro-architectural state* `micro-state-step(`s`)` *and the updated MAETT* `MAETT-step(`M`,`s`)` *satisfy:*

$$\texttt{MAETT-ok?}(\texttt{MAETT-step}(M, s), \texttt{micro-state-step}(s)).$$

Lemma 2 MAETT for a Flushed State. *Suppose s is a flushed micro-architectural state; that is, no partially executed instruction is in the pipeline of s. The empty MAETT* `nil` *is a correct MAETT representing s, and* `MAETT-ok?`$($`nil`$, s)$ *is true.*

From these two lemmas, we can conclude that `MAETT-ok?` is true for all the micro-architectural states reachable from a flushed state. Since the `MAETT-ok?` is a conjunction of pipeline properties, we can prove Lemma 1 and Lemma 2 by proving the pipeline properties one at a time. We do not have to examine the whole processor design to prove an individual property, because each property is relevant to a small part of the complete design. For instance, when we prove that our design invariantly satisfies the property `no-WAW-hazards?`, we can concentrate our effort on the issuing logic and ignore, for instance, the branching unit. Then, assuming no WAW hazard occurs, we can prove `correct-regs?` by concentrating our effort on the write-back stage to the register file. In this way, the complete verification problem is divided into local problems, which are verified incrementally. This incremental verification approach helped us avoid case explosions in our proof and reduced the computational cost of the verification process.

The proofs of Lemma 1 and 2 are the core part of our verification, because it required a great portion of our verification effort and profound analysis of the machine design. The proofs of individual properties vary, but it is often the case that the top-level proof goals are reduced, by induction, into simpler subproblems, which are then solved by case analysis and rewriting. We have been trying to verify these subproblems by more automatic procedures including BDD's, but with little success. We think improvements in this step will be important to make our approach more practical.

3.3 Equivalence Proof of ISA and Micro-architecture

We have proved the equivalence of our ISA and our micro-architectural design in two ways; one approach proves Diagram(a) in Fig. 1, and then we verify Diagram(b) from (a) by induction. Our second approach is a direct proof of the correctness criteria given in Diagram(b) without proving Diagram(a). Here we discuss only the latter approach, because it turned out to be simpler than the former approach.

We pictorially portray the proof of the correctness criteria using Fig. 3. Let us consider n micro-architectural state transitions $P_0, P_1, \ldots, P_n$. We assume that the initial state P_0 and the final P_n are flushed, as shown in Diagram (b). We further assume that m instructions $I_0, \ldots, I_{m-1}$ are executed in the micro-state transitions from P_0 to P_n. Let $S_0 = \texttt{proj}(P_0)$, that is, the ISA state projected from P_0. And let $S_1, \ldots, S_m$ be the m-step ISA-state transitions from S_0. Now we need to prove $\texttt{proj}(P_n) = S_m$ to satisfy Diagram(b).

From Lemma 1 and Lemma 2, micro-architectural states $P_0, P_1, \ldots, P_n$ have corresponding MAETT representations $M_0, M_1, \ldots, M_n$, and $\texttt{MAETT-ok?}(M_i, P_i)$ is satisfied by each pair of P_i and M_i. Figure 3 shows MAETT's $M_0, \ldots, M_n$, which contain instructions $I_0, \ldots, I_{m-1}$. In the figure, the current stages of instructions are shown; for instance, I_0 advances from stage IF/DC to DC/EX, and RETIRE's by the time it reaches M_n. M_n contains all of $I_0, \ldots, I_{m-1}$, and they are all retired since the corresponding micro-state P_n is flushed. The figure also shows the relations of instructions $I_0, \ldots, I_{m-1}$ and ideal ISA states $S_0, \ldots, S_m$ satisfying (3) and (4) ; the ISA-machine executes instruction I_i in the transition from S_i to S_{i+i}. In other words, S_{i+1} is the ideal ISA state after executing instruction I_i. We define a function $\texttt{final-state}(P_i, M_i)$ which returns the ideal ISA state after executing the last instruction in M_i. So $\texttt{final-state}(P_n, M_n) = S_m$, because I_{m-1} is the last instruction in M_n. The definition of `final-state` is easy as a MAETT contains the ideal ISA states corresponding to the instructions. We then can prove:

Lemma 3. *If a micro state P_i and its MAETT M_i satisfy* $\texttt{MAETT-ok?}(P_i, M_i)$, *and all the instructions in M_i are retired, then* $\texttt{final-state}(P_i, M_i) = \texttt{proj}(P_i)$.

Lemma 3 can be proven from the invariant properties `correct-pc?`, `correct-regs?` and `correct-mem?`, which are conjuncts of `MAETT-ok?`. Using Lemma 3

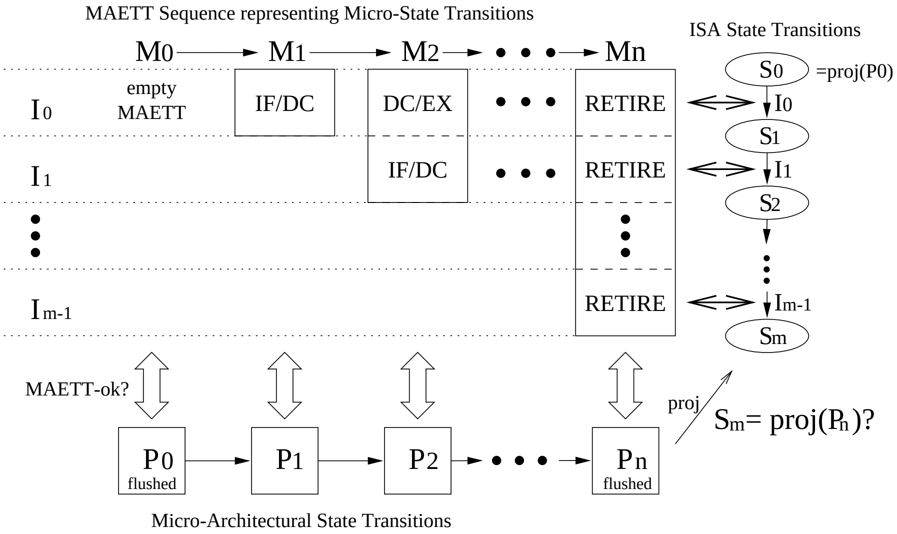

Fig. 3. Pictorial Description of the Proof of Diagram (b)

with M_n and P_n, we will get:

$$S_m = \texttt{final-state}(P_n, M_n) = \texttt{proj}(P_n).$$

The proof of the correctness criteria is performed by theorem proving techniques, and Diagram (b) in Fig. 1 has been shown to be valid for arbitrarily long micro-state transitions with any instruction sequences, provided that the initial and final states are flushed.

We also have to verify the liveness property of the pipelined machine, because the correctness criteria are vacuously true if the pipelined machine cannot reach a flushed state. We have shown that our pipelined design completes the execution of an arbitrary instruction sequence in finite cycles. The proof is done by constructing a measure function which returns the sum of the number of stages before each instruction retires, and then showing that the measure function decreases in each cycle. Such a measure function can be defined as a simple recursive function over a MAETT.

3.4 Mechanical Verification

Using ACL2 theorem prover, we have verified that the micro-architectural design of our example machine implements its ISA specification, by following the steps discussed in the previous subsections. The control logic of our micro-architectural design is proven correct with respect to the correctness criteria, but we did not concern ourselves with the correctness of the data-path elements, as in Burch and Dill's work. However, the pipelined multiplier was independently verified by showing the output from the unit is a wrap-around product of the inputs.

We demonstrated that this result can be integrated in our pipeline verification technique by proving the correctness criteria for the micro-architectural design with the detailed multiplier specification.

The verification effort took three months of manual effort. The replay of the whole proof takes about one hour of CPU time using a Pentium 133MHz. During the verification effort, we found two bugs which had not been discovered by running the ACL2 executable specification. One was a deadlock caused by undefined instructions, and the other was in the multiplier which returned wrong answers for certain inputs.

4 Conclusion

We have verified the correctness of the micro-architectural specification of a pipelined microprocessor with respect to its ISA-level specification by defining a table-based execution abstraction. Our technique introduced the MAETT as an execution trace model, which allows us to easily define various pipeline properties. We proved that such properties are invariantly held by all micro-architectural states reachable from a flushed state, and then used that fact to prove the final correctness criteria showing the micro-architecture implements the ISA.

Our proof approach is incremental. Properties of the micro-architectural design are identified and formally defined; for example, that the program counter should be pointing to the correct instruction. The definition of these properties often only involves a portion of the micro-architecture. Thus, proofs to ensure that these properties are maintained by the implementation do not require analyzing the entire implementation. Once we have established the correctness of these properties, we compose them to establish the correctness of the entire design.

We believe that this approach will allow us to verify deeper and more complex designs than is currently possible with automated approaches. The immediate focus of our future research is integrate exceptions and interrupts into our trace table based verification approach.

References

1. M. Agaard, M. Leeser, Reasoning About Pipelines with Structural Hazards, Pipelined Microprocessors, Theorem Provers in Circuit Design : theory, practice, and experience, Lecture Notes in Computer Science 901, Springer Verlag, 1995, page 13-32.
2. R. S. Boyer and J S. Moore. A Computational Logic Handbook. Academic Press, Boston, 1988.
3. J. R. Burch, D. L. Dill: Automatic Verification of Pipelined Microprocessor Control, In D. Dill Editor, Computer Aided Verification, Lecture Notes in Computer Science 818, Springer Verlag, 1994, page 68-80.
4. J. R. Burch. Techniques for verifying superscalar microprocessors. In Design Automation Conference, June 1996.

5. D. Cyrluk. Microprocessor verification in PVS: A methodology and simple example, Technical Report SRI-CSL-93-12, SRI Computer Science Laboratory, Dec. 1993
6. J. Hennessey, D. Patterson, Computer Architecture a Quantitative Approach, Morgan Kaufmann Publishers, Inc., 1996.
7. 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, page 35-48. Prentice-Hall International Series in Computer Science, Engle wood Cliffs, N.J., 1992
8. R. B. Jones, D. L. Dill, J. R. Burch, Efficient Validity Checking for Processor Verification, 1995 IEEE/ACM International Conference on Computer-Aided Design, pages 2-6.
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. M. Srivas, M. Bickford, Formal Verification of a Pipelined Microprocessor, IEEE Software, September 1990, page 52-64.
11. M. K. Srivas, S. P. Miller, Formal Verification of a Commercial Microprocessor, Technical Report SRI-CSL-95-12, SRI Computer Science Laboratory, July 1995.
12. S. Tahar, R. Kumar, Formal Verification of Pipeline Conflicts in RISC Processors, Proc. European Design Automation Conference (EURO-DAC94), Grenoble, France, September 1994, IEEE Computer Society Press. page 285-289.
13. P. J. Windley, M. L. Coe, A Correctness Model for Pipelined Microprocessors, Theorem Provers in Circuit Design : theory, practice, and experience, Lecture Notes in Computer Science 901, Springer Verlag, 1995, page 33-51.
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, 1996, page 362-376.