Formal Verification of Pipelined Machines with Out-of-order Execution *

Jun Sawada
University of Texas at Austin, Computer Science Dept.
email: sawada@cs.utexas.edu

Abstract

This paper discusses the technical details of the design verification of a pipelined processor with out-of-order execution. We have developed new techniques to verify pipelined processors with complex control logic. Our principal technique is modeling the stream of instructions using a table representation, which allows us to directly express many machine-relevant properties. Using this representation, we have verified pipeline properties incrementally, and eventually verified a complete pipelined machine design, whose correctness is defined using the idea of pipeline flushing. The proof has been mechanically checked by ACL2 theorem prover.

1 Introduction

Since the principles of formal hardware verification were established in the 80's [4, 8, 10], there have been many attempts to apply these techniques to larger and larger examples[2, 9]. Although some measure of success has been achieved, formal hardware verification is not yet widely applied to general-purpose microprocessor design. A serious impediment to applying such techniques is the failure of current proof approaches to address the verification of today's processors which adopt radical performance optimizations, including superscalar super-pipelining, out-of-order execution, and speculative execution [5, 7, 11].

Pipelining is a key feature of today's microprocessor designs. There have been several earlier successes to formally verify pipelined machines [6, 14, 15, 16]; however, these machines have a short and simple pipeline structure. The lack of complex control logic, which is one of the hardest parts to verify, is something our approach addresses.

In our project, we have been studying and developing techniques to verify the correctness of pipelined machines with relatively complex pipeline control.

*This paper is a technical report attached to the dissertation proposal.

To achieve this goal, we designed an example processor, which includes features like out-of-order instruction completion, speculative fetching of instructions, dynamic resolution of RAW(read-after-write, true) dependencies and WAW (write-after-write, output) dependencies.

Our method introduces a new technique we call the instruction table. With help of the instruction table, we can define and prove various properties of our pipelined machine. The correctness criterion of our pipelined machine is, then, proved from these pipeline properties. In Section 2, we describe our pipelined machine. In Section 3, we discuss how to represent the correctness of the pipelined machine and why it is difficult to verify. In Sections 4 and 5, we describe our verification methods using the instruction table. In Section 6, we discuss the mechanical proof of our verification.

2 Design of a Pipelined Machine

We have designed an example pipelined machine to investigate verification techniques for today's complex pipelined processors. In order to achieve this goal, our machine should have a rather complex structure and control logic, not just a simple linear pipeline with virtually no interlocks. We want to study various performance optimization techniques which are interesting to verify. At the same time, our example machine should be simple enough so that we can experiment with new verification techniques without too much effort. In order to facilitate our work, we have implemented a pipelined machine with various features including out-of-order execution, but with a small number of instructions and no external interactions.

Our pipelined machine design has only 6 instructions: addition, multiplication, conditional branch, register load, register store, and a synchronization instruction. Each kind of instruction behaves differently in the pipeline. For instance, an addition instruction takes typically 4 cycles to complete including the fetch cycle, while a multiplication takes at least 6 cycles.

The organization of our processor is shown in Figure 1. We have multiple execution units: a branch unit, an adder, a multiplier, and a load-store unit. The register file is written by the write-back unit, and read by the decoding unit. The register file has three read ports to support simultaneous reading of multiple source operands, but has only one write port. The memory is accessed only by the instruction-fetch unit and the load-store unit. The memory is divided into two parts: the last half is readable and writable, while the first half is read-only. Instructions can be fetched only from the first half memory; this restriction eliminates the problem of self-modifying code as long as code is fetched from the first half of memory. In our model, the memory always responses in one cycle. Also, simultaneous accesses by the instruction fetch unit and the load-store unit are allowed, as is often the case with a split cache. These assumptions simplified the design of the instruction-fetch unit and the load-store unit. Pipeline latches,

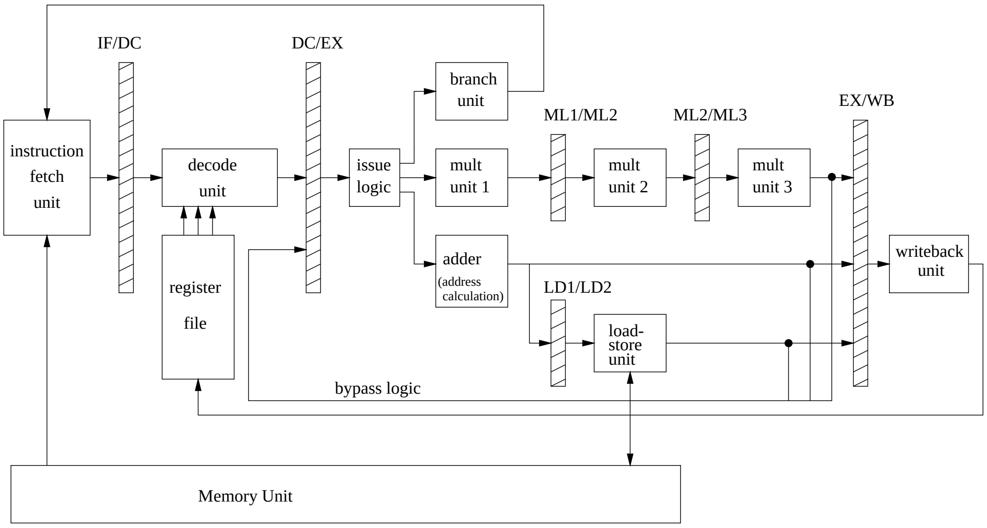

Figure 1: Block Diagram of the Example Pipelined Machine.

shown by pattern filled rectangles in Figure 1, split the pipeline into stages. They are named **IF/DC**, **DC/EX**, **ML1/ML2**, **ML2/ML3**, **LD1/LD2** and **EX/WB**.

Each instruction is processed as follows. First, the instruction-fetch unit reads the instruction from the memory and sends it to the decode unit. Then the issuing logic dispatches the decoded instruction to the appropriate execution unit. Our design can issue one instruction per clock cycle, and the issue is in-order. The issuing logic checks *RAW-dependencies* [7] between the issued instruction and instructions issued earlier. If there is such an RAW-dependency, the issue is suspended until the appropriate data is ready. Bypassing logic helps to reduce the number of stall cycles. It also checks *WAW-dependencies* between instructions. Because addition takes fewer cycles than multiplication, an addition instruction may complete earlier than a multiplication instruction, even though the multiplication may have been issued first. If this kind of out-of-order completion occurs with instructions that share the same destination register, a WAW-hazard happens and the resulting register file is incorrect. This is the purpose of the WAW-dependency check. The issuing logic also schedules instructions so that there are no *structural hazards* at the writeback stage.

Our pipelined machine also performs *speculative instruction fetching*. Our branch prediction algorithm is static and it always predicts that a branch will not be taken. We have no exception mechanisms.

We have also defined an equivalent *sequential machine* as the specification for our pipelined machine. Since a non-pipelined machine completes an instruction before starting another, it is easier to specify the effect of a single instruction with a sequential machine model than with a pipelined machine.

The behavioral description of the sequential and pipelined machines are given by executable ACL2 functions. ACL2 [13] is both an executable specification language and a theorem-proving system, and it can be considered a "next generation" Boyer-Moore theorem prover [1]. The ACL2 logic is a subset of Common Lisp. In the body of this paper, we do not use the Common Lisp style syntax, although the appendices include real definitions and theorems in ACL2.

We define p-step() as the state transition function for the pipelined machine. Taking a pipeline state s, p-step(s) returns the next pipeline machine state after one clock cycle. The sequential specification is given by s-step(), which returns the machine state after executing one instruction. Note the difference in time granularity between s-step() and p-step().

3 Pipeline Correctness Diagram

Our research objective is proving the equivalence between our sequential specification and pipelined design; however, this equivalence is hard to define. Intuitively, we want to verify that the pipelined design executes all instructions in any context just as the sequential machine does. A central problem is how to define the effect of a single instruction in the pipelined machine. In a pipelined design, the machine starts the execution of an instruction before an earlier instruction completes, blurring the execution boundary between instructions. On the other hand, a sequential machine completes each instruction before starting the next, making the boundary between instructions clear.

A time abstraction is the most frequently used solution to relate different levels of behavioral description. However, a simple time abstraction cannot relate sequential and pipelined machines, as we typically do not have a pipelined machine state that we can directly compare to the sequential specification state.

Some earlier pipeline verification research[6, 15] used abstraction mappings combining different machine components at different times. For instance, an abstraction mapping of a pipelined machine may combine the program counter at time t and the register file and the memory at time $t+2$ to form the corresponding specification state. Sometimes the pipeline may stall, forcing us to pick the register file from time $t+3$ instead of $t+2$. This concept is described in Figure 2. This technique of abstraction mapping works well if the pipeline structure is simple and limited pipeline interlocks that cause few pipeline stalls. However, as the pipeline becomes more complex and more interlocks are introduced, selecting the right times for different components becomes very difficult. Since our objective is the verification of pipelined machines with complex structure and control, we chose not to use this approach. Instead, we have to find an alternative method to represent the correctness which is applicable to the

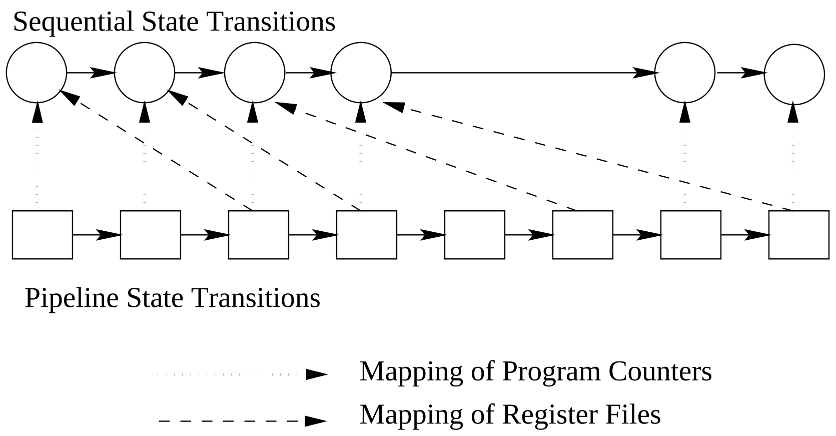

Figure 2: An example time abstraction combining different components at different times.

pipelined machines in general.

A more general approach for comparing a pipelined design with a sequential specification was proposed by Burch and Dill [3]. Figure 3 shows the basic idea of such a comparison. We map a pipelined machine state to a sequential machine state by flushing instructions out of the pipeline without fetching any new instructions, and then projecting the synchronized pipeline state to a sequential machine state. Using this mapping, we can define a correctness criterion for our pipelined machine by the diagram shown in Figure 3. We call this diagram *Burch and Dill's pipeline correctness diagram* or simply the pipeline correctness diagram.

This diagram holds only when the pipelined machine fetches an instruction during the left-hand side pipeline state transition, so we have to prove a similar diagram for the case when no instruction is fetched. Once the diagrams for these two cases are proved, we can prove an n-cycles version of the diagram, which

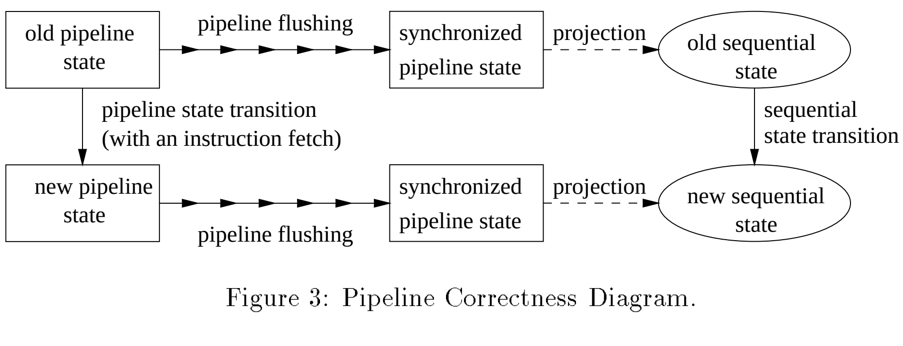

Figure 3: Pipeline Correctness Diagram.

implies that the state transitions of the sequential machine can be emulated by the pipelined machine as long as the initial and final states of the pipeline state transitions are synchronized.

The pipeline correctness diagram is relatively succinct and applicable to a wide range of implementations. We want to note that a pipelined machine design with out-of-order instruction completion should satisfy the pipeline correctness diagram in Figure 3. The reader can easily see this, since the same set of instructions must be completed on the two paths from the "old pipeline state" to the "new sequential state", even though the order of completion may be different.

In the original work by Burch and Dill, they proved the pipeline correctness diagram by symbolically simulating the pipelined machine design in their logic of uninterpreted functions with equalities. Just like our example, their behavioral machine specifications are given by next state functions. By unrolling these functional specifications, they compare the traversal of the paths in Figure 3. The comparison is done for any initial pipeline configurations by an automatic decision procedure with case analysis and simplification. This method is much faster than the ordinary simulation, because only the control part of the pipelined machines is analyzed, while the datapath is manipulated symbolically. We call their approach the *symbolic execution* method in this paper.

We found two problems with the symbolic execution approach when we attempted to verify another example machine. First, the cost of verification increases dramatically as the pipeline control becomes complex and the pipeline is lengthened. Because the pipeline correctness diagram involves flushing which takes many machine cycles, slight increase in complexity of the pipeline causes an explosion in the number of examined cases. We can improve the efficiency of the verification process using heuristics and other techniques[12], but it does not solve the problem of case explosion. It is our belief that a naive application of the symbolic execution method cannot verify a pipelined machine with complex pipeline-control logic with an acceptable computational cost.

The second problem is that the diagram is not valid for all pipeline states, but just the reachable states. Our machine contains a bookkeeping register to record activated execution units. If the bookkeeping record and the actual pipeline state are inconsistent in the "old pipeline state" in Figure 3, the pipeline correctness diagram may not hold. Usually, a reachable pipeline machine state does not contain such an inconsistency. Thus, we restrict the "old pipeline state" to the reachable machine states and exclude inconsistent states. Burch and Dill suggested that they can use invariants to restrict the "old pipeline state". However, it is not clear from the paper how to prove the closure property of the invariants during the pipeline state transition, or even how to express such invariants.

We will use the pipeline correctness diagram as our verification goal in this paper, because it is the best and most general method we know of to represent the correctness of a complex pipelined machine. In the following two sections, we discuss our method to prove the correctness diagram and our approach for the problems discussed above.

4 Instruction Stream Table

Our approach is based on analytical reasoning methods using a theorem prover, rather than testing the control logic for all possible pipeline configurations with an automatic decision procedure. By verifying the machine incrementally at an abstract level, we aim at reducing the computational cost of verification.

As is often done in theorem prover-based verification research, we first prove invariant properties of our target pipelined machine design. We have found that writing invariant conditions using only the pipelined machine state is difficult. One of the reasons is that pipelined machine designs are often highly optimized and contain only the information necessary to execute instructions. For instance, the instruction word is thrown away once the instruction is decoded, and the operand register name is abandoned after the source operand value is read from the register file. In this case, it is difficult to write a predicate stating the existence of RAW-dependencies between instructions, because we cannot lift the necessary information out of the pipelined machine state.

As an answer to this difficulty, we model the stream of instructions with a table, which we call the *instruction table*. This new representation, which can be considered as another level of machine abstraction, is devised so that we can directly reason about the effect of instructions as they flow through the pipeline. An instruction table contains all the instructions which are either currently being executed in the pipeline or that have completed execution. The order of the instructions in the table is the same as the order that instructions are processed by the sequential specification.

`valid`	A valid instruction or a speculatively executed invalid instruction.
`stg`	Pipeline stage at which the instruction is in.
`inst`	Instruction word.
`rnames`	Operand registers. Each instruction has three operand registers.
`pc`	Program counter in the logical sequential state.
`regs`	Register file in the logical sequential state.
`mem`	Memory in the logical sequential state.
`misc`	Miscellaneous information.

Table 1: The fields of a table entry.

Each instruction is expressed with the data structure shown in Table 1. Field `valid` contains **false** if the instruction is speculatively executed and will be abandoned later; otherwise, it is **true**. Field `stg` contains the current pipeline stage of the instruction. The pipeline stage of an instruction is represented by the name of the pipeline latch that stores the instruction. After the instruction is completed, field `stg` is set to **RETIRE**.

Fields `pc`, `regs` and `mem` are defined with the corresponding *logical sequential state* of the instruction. Intuitively, a logical sequential state of an instruction is the ideal sequential state in which the instruction should be executed. From programmer's viewpoint, instructions are executed in a sequential execution model. So, we can say that a logical sequential state of an instruction is what programmers have in mind as the machine state at the time when the instruction is executed.

Let us give a more formal definition of the logical sequential states. In our machine design, a sequential machine state is represented by a triple (pc, $regs$, mem), where pc, $regs$ and mem represent the program counter, the register file, and the memory, respectively. Assume that we are considering the execution of an instruction sequence $I_0, I_1, \ldots I_i, \ldots$ from an initial sequential state (pc_0,$regs_0$,mem_0). We define (pc_0,$regs_0$,mem_0) to be the logical sequential state of the instruction I_0. For $i > 0$, the logical sequential state (pc_i, $regs_i$, mem_i) of the instruction I_i is defined by the following equation:

$$(pc_{i+1}, regs_{i+1}, mem_{i+1}) = \text{s-step}((pc_i, regs_i, mem_i)),$$

where s-step() is the next state function for our sequential machine, as discussed in Section 2. The fields `pc`, `regs` and `mem` for the table entry representing instruction I_i contain pc_i, $regs_i$ and mem_i, respectively.

The logical sequential states are useful for representing intermediate values of partially executed instructions. For instance, we may want to verify whether the bypass logic is forwarding the "correct" source-register value to an instruction in an earlier stage of the pipeline. This "correct" register value for an instruction can be defined as the value of the register in the logical sequential state of that instruction.

Since an instruction table abstracts a pipeline state, we can also abstract a pipeline state transition by an update of the instruction table. We define the updating function of the instruction table as step-table(tbl,s), which takes an instruction table tbl and a pipeline state s, and returns the new instruction table which corresponds to the next pipeline machine state p-step(s).

Step-table() is defined using another function step-entry(), which specifies the behavior of each instruction in the pipeline. For instance, if an instruction advances to the next pipeline stage, step-entry() updates the field `stg` of the table entry to the new stage. When the pipeline fetches a new instruction or abandons speculatively executed instructions, step-table() adds or deletes table entries.

Figure 4 shows the relation between the table-updating function and the state transitions of both the sequential specification and the pipelined implementation. Instruction tables $T_1, T_2, \ldots, T_5$ are shown in the middle of the

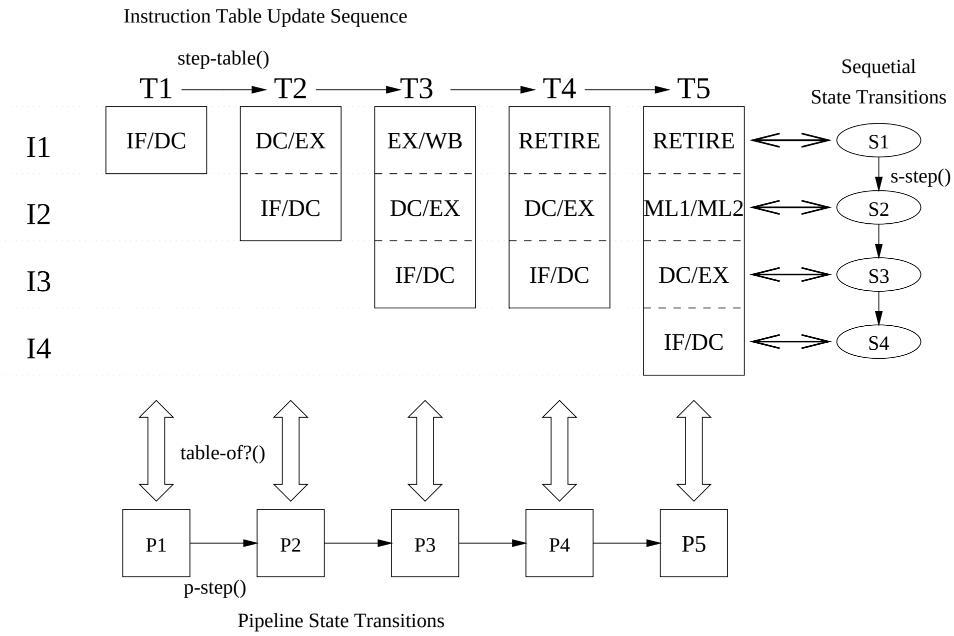

Figure 4: Relation between instruction tables and machine states.

figure. They correspond to the pipeline states $P_1, P_2, \ldots, P_5$, respectively. Each table entry is updated to the next table to the right by the table-updating function step-table(). Each row of the instruction table represents a specific instruction. For instance, the top row represents the instruction I_1, whose stage is updated from **IF/DC** to **DC/EX**, **EX/WB** and **RETIRE** as the table is updated. The entries corresponding to I_2, I_3 and I_4 are added as the table is updated. The sequential states S_0, S_1, S_2 and S_4 are also related to the instructions $I_1, \ldots, I_4$, because these sequential states are the logical sequential states of the instructions.

The relations between the instruction table entries and the pipeline states, which are shown by the vertical arrows in the Figure 4, are defined by the predicate table-of?(). Table-of?() is the conjunction of the various pipeline properties listed in Appendix A. Every reachable pipelined machine state and its correct table representation should satisfy all of these pipeline properties. For example, the predicate match-each-entry() in Appendix A checks whether all the partially executed instructions listed in an instruction table have correct intermediate values in the pipeline. Another predicate stage-order-preserved?() checks if all the instructions before the issuing stage are in-order with respect to the pipeline stages.

The reader can see the expressive power of the instruction table from the definition of the pipeline properties listed in Appendix A. The ACL2 definition of the predicate stage-order-preserved?() is given in Appendix B as an example. The recursive definition of stage-order-preserved?() *systematically* checks whether any combination of instructions in an instruction table violates the issuing order. Furthermore, stage-order-preserved?() can examine already retired instructions. The instruction table makes it easier to write a predicate involving retired instructions, because it contains instructions executed in the past; however, a similar predicate is hard to define only using the pipelined machine state, which does not contain the information about retired instructions. This expressive power of the instruction table becomes especially useful in the verification of a machine with out-of-order completion of instructions.

Another definition of a pipeline property is given in Appendix C . Exist-output-violation?() is a predicate checking whether there is a WAW-hazard in the execution. A WAW-hazard between two instructions occurs if they share the same destination register and write their results back to it out-of-order. Exist-output-violation?() recursively checks these conditions for all pairs of instructions in a table. It is also helpful that we can check WAW-hazards involving an already retired instruction.

We prove that the properties in Appendix A hold for all the reachable machine states. In order to prove that, we need to show that the property table-of?() is an invariant during the pipeline state transition in the sense shown in Figure 5. If the "old table" correctly represents the "old pipeline state" with respect to table-of?(), the "new table" after a table update should also be the correct representation of the "new pipelined state" after a machine clock cycle. More formally:

Lemma 1 (Invariant property of table-of?()) *Assume s is a pipeline state and tbl is an instruction table, and they satisfy* table-of?*(tbl,s). Then the next pipeline state* p-step*(s) and the updated table* step-table*(tbl,s) satisfy:*

$$\text{table-of?}(\text{step-table}(tbl, s), \text{p-step}(s))$$

Because the predicate table-of?() is a conjunction of the pipeline properties listed in Appendix A, we were able to prove Lemma 1 by proving the properties one-by-one. Typically, each property can be proven independently. Furthermore, the proof of each property requires examining only the related part of the pipeline design. For example, the predicate pc-correspondence() determines whether the program counter points to the newly fetched instruction, and it is closely related to only the program counter and the branch unit. Examining these units is enough to prove that the predicate holds for the new pipeline state. The proof of another predicate exist-output-violation?() is done by examining mostly the issuing logic and the later stages of the pipeline.

It is also worthwhile to note that we need to examine only one cycle of the pipeline state transition to prove Lemma 1. This is obvious from Figure 5, which involves one cycle of machine transition. This is why the proof of Lemma 1 is

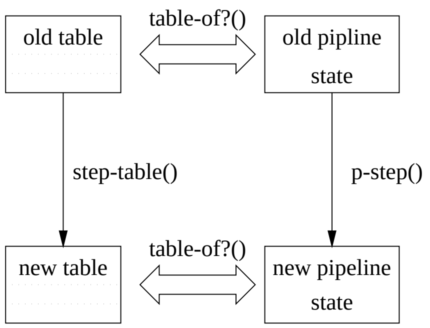

Figure 5: Invariant Relation of Table-of?()

easier than that of the pipeline correctness diagram in Figure 3, which involves many cycles of stalling. It can be said that the invariant proof of table-of?() divides the big problem of proving the pipeline correctness diagram into small sub-problems in terms of the number of machine cycles to be examined, as well as the size of the related micro-architectural components.

Appendix D gives a sketch of the proof of non-existence of WAW-hazards, which is defined by the predicate exist-output-violation?(). The separate proofs for all the properties in Appendix A are combined to form the proof of Lemma 1.

We can also prove the existence of the correct table representation for any synchronized state. In our definition, the empty instruction table `nil` is always a correct table representation for any synchronized state.

Lemma 2 (Table Representation for a Synchronized State) *Suppose s is a synchronized pipeline machine state; that is, no partially executed instruction is in the pipeline of s. The empty instruction table* `nil` *is the correct table representation of s, and* table-of?(`nil`, s) *holds.*

By Lemma 1 and Lemma 2, we can conclude that, for any pipeline state s which is reachable from a synchronized pipeline state, there is an instruction table representation *tbl* satisfying table-of?(s,*tbl*). From the definition of table-of?(), we can see that any reachable state satisfies all the properties listed in Appendix A. This fact will be used in the proof of our correctness diagram in the next section.

The largest portion of our verification effort has been directed to the proof of Lemma 1. Actually, we believe that most of the "correctness" of our pipeline machine is proved here. The properties listed in Appendix A are what we believe the most crucial properties for the issuing logic, the bypass logic, the branch unit, and the data-path. And their proof requires profound analysis of the pipeline design. The rest of the work is how to combine these verification results together to prove the pipeline correctness diagram.

5 Proof of the Pipeline Correctness Diagram

Before proceeding to the proof of the pipeline correctness diagram, we first have to specify the pipeline flushing process. Pipeline *stalling* is a newly defined behavior of the pipeline in which the machine does not fetch a new instruction, but works normally otherwise. We define p-drain-step() as the function specifying one clock cycle of stalling. Then, the entire pipeline flushing process is defined as a recursive function p-drain(), which repeatedly calls p-drain-step() until all the instructions are flushed out of the pipeline.

We also define a new table-updating function drain-step-table() which corresponds to the stalling function p-drain-step(). The predicate table-of?() should be invariant during the stalling cycle in the following sense:

Lemma 3 (Invariant Property of table-of?() in Stalling) *Assume s is a pipeline state and tbl is an instruction table, and they satisfy* table-of?*(tbl,s). The pipeline state* p-drain-step*(s) after one cycle of stalling and the corresponding updated table* drain-step-table*(tbl,s) satisfy:*

$$\text{table-of?}(\text{drain-step-table}(tbl, s), \text{p-drain-step}(s))$$

We also require that the pipeline flushing process satisfies the following lemmas.

Lemma 4 (Termination of Flushing) *For any pipeline state s, if there exists a correct table representation tbl satisfying* table-of?*(s,tbl), then the recursively defined function* p-drain*(s) terminates and returns a synchronized pipeline state.*

Lemma 5 (Flushing of Synchronized Pipeline) *For any synchronized pipeline state s,*

$$\text{p-drain}(s) = s$$

Lemmas 3, 4 and 5 are necessary to complete the proof of the pipeline correctness diagram and other relevant theorems. Since these lemmas are about the stalling process and the corresponding table-updating function, they may not hold for certain definitions of pipeline stalling. We easily found a simple definition of stalling for our pipeline machine which satisfies Lemmas 3, 4 and 5. However, it may not be easy to define such a stalling process for some machine designs. Since we don't want to rely on the existence of a "good" definition of stalling, we are looking for an alternative approach to express the pipeline correctness without using pipeline flushing.

Finally, we prove the pipeline correctness diagram. Normally, the proof of the diagram in Figure 3 has to show that the same "new sequential state" can be obtained from any configuration of the "old pipeline state" by following the two paths in the diagram. More formally:

Theorem 1 (Pipeline Correctness Diagram (Instruction Fetch Version)) *If a pipelined machine state s and its instruction table tbl satisfy* table-of?*(tbl,s), and the pipeline machine s fetches a new instruction in the next clock cycle, then:*

$$\mathrm{proj}(\text{p-drain}(\text{p-step}(s), \text{step-table}(tbl, s))) = \text{s-step}(\mathrm{proj}(\text{p-drain}(s, tbl))),$$

where proj*() is the projection function from pipeline states to sequential states.*

As we discussed in Section 3, the pipeline correctness diagram may not hold for some unreachable states. In Theorem 1, we constrain "old pipeline state" s by assuming the existence of a correct table representation tbl which satisfies table-of?(tbl,s). This excludes some unreachable states from "old pipeline states", but Theorem 1 still covers all the reachable pipeline states. This is justified by Lemma 1 and Lemma 2.

The proof of Theorem 1 is split into separate proofs about the program counter, the register file, and the memory. We display the sketch of the diagram proof with respect to the register file in Appendix E. Let us give an informal discussion on our approach used in the proof.

One of the major difficulties in the proof of the pipeline correctness diagram is reasoning about the effect of pipeline flushing, which takes a number of clock cycles. Our approach for this problem is this: represent the effect of pipeline flushing with the instruction table, and then reason about it with help of the pipeline properties which are invariant during the stalling cycle.

For example, in our proof shown in Appendix E, the function final-regs() represents the register file state after pipeline flushing. This representation is defined directly using the instruction table and it does not examine the actual pipeline stalling process. However, we can prove that final-regs() is a correct representation of the effect of pipeline flushing, using the invariant property of table-of?() during stalling. The rest of the proof can be conducted by a relatively easy reasoning about the instruction table. The proof steps for the register file equivalence are shown in detail in Appendix E.

Our method of diagram proof using the instruction table solves the problems for the symbolic execution method discussed in Section 3. It does not suffer the case explosion problem, because we logically proved the correctness of the diagram at an abstract level using various pipeline properties. And our method excludes the inconsistent pipeline states from the "old pipeline state" by assuming the existence of a correct instruction table representation for the pipeline state.

6 Mechanical Check of the Proof

Our pipeline correctness proof has been mechanically verified by ACL2 theorem prover. The number of the lines of the proof scripts and the CPU time for the proof are given in Table 2. It took less than 1 man-month to design the machine

File Name	Lines	Time(sec)	Comment
b-ops-aux-def.lisp	39	1.68	Supplementary definitions of bit operation.
b-ops-aux.lisp	298	22.08	Basic Lemmas about Bit Operations.
basic-def.lisp	224	40.74	Definition of Register File and Memory.
diagrams.lisp	1292	215.68	Proof of Pipeline Correctness Diagram.
drain-step.lisp	139	4.45	Definition of Stalling Cycle.
invariants.lisp	27	0.41	Proof of Lemma 1 from Sub-lemmas.
invariants1.lisp	5171	1742.39	Proof of Properties in Appendix A.
invariants2.lisp	1222	340.78	Proof of Properties in Appendix A.
invariants3.lisp	1256	228.74	Proof of Properties in Appendix A.
p-drain.lisp	489	48.75	Definition of Pipeline Flushing.
pipeline-def.lisp	1351	109.97	Definition of Pipeline Design.
reg-modifier.lisp	299	132.65	Theory of Register Modifying Instructions.
seq-def.lisp	157	10.57	Definition of Sequential Machine.
table-def.lisp	1628	103.51	Definition of Instruction Table.
table-lemmas.lisp	3871	549.72	Basic Lemmas about Instruction Table.
table-misc.lisp	192	29.12	Miscellaneous Lemmas including Lemma 2.

Table 2: Lines and Proof Time for the Script Files. The CPU time for definition and proof is measured on a Pentium 133MHz, Solaris 2.5 Machine. It does not include time for loading ACL2 certified books.

and debug it, then took 3 man-months to verify it. Since we have tried many experimental techniques in our project, and the machine is very small compared with practical commercial processors, it is too early to make a conclusion on the labor cost. In our future work, we will pay close attention to the labor involved.

There are a few other points we want to discuss here.

The execution capability in ACL2 allows us to run the behavioral description of our pipelined machine. Using this feature, we have run small programs on our machine design and discovered most of our design errors before applying formal verification techniques. Actually, all the fatal error except one were eliminated in this stage. The only fatal error uncovered by the formal verification concerned undefined instructions, which we forgot to test in the debugging phase. We believe it is important to remove most of the design errors using techniques like simulation before attempting to apply formal verification techniques.

The actual verification work did not proceed quite as directly as we described in the last few sections. We first proved the pipeline correctness diagram under the assumption that Lemma 1 holds, then we proceeded to the proof of Lemma 1, which took most of our verification effort. During this proof, we often discovered that we needed to prove additional properties about the pipeline and instructions, or that some naively defined properties do not hold with our machine. In order to reflect these findings, we continuously modified the definition of the table-updating function and properties in Appendix A. The predicate table-of?(), which is the conjunction of all the resulting properties, is strong enough to prove the pipeline correctness diagram, but weak enough to hold for any reachable pipeline states. This definition is a result of our interaction with the theorem prover.

Our method checks whether the micro-architectural components work cooperatively. One of the most interesting minor defects we found involved a bookkeeping register for the issuing logic. The issuing logic reads and updates a register to record the pipeline configuration, but the writeback unit also refers to this register to determine the source of the next written-back datum. To our surprise, the designs of the issuing logic and the writeback unit used different assumptions about the bookkeeping register. Although the problem did not turn out to be an major error, it demonstrated that such conflicting assumptions in the design of micro-architectural components can be uncovered by our approach.

7 Conclusion

We have verified the behavioral equivalence of our pipelined machine with respect to its sequential specification using ACL2 theorem prover. Our machine design is certainly not the most complex pipelined machine that has been formally specified and verified. However, our machine has more complex pipeline control and interlocks than those in earlier verification research we are aware of. Our example machine features out-of-order completion of instructions, speculative instruction fetching and dynamic resolution of RAW-hazards, WAW-hazards and structural hazards.

For the sake of the verification, we introduced the instruction table as a new level of machine abstraction. It allows us to directly reason about the instructions executed by the pipeline machine. With help of the table representation, we defined various properties about the pipelined machine design. These properties are proved to hold for reachable pipelined states. The proof are conducted incrementally by examining the related micro-architectural components.

We adopted a correctness criteria that is defined using the idea of pipeline flushing. This allowed us to express the correctness of the pipelined machine containing complex control logic. The correctness criteria is verified using the pipeline properties of the reachable pipeline states. Since this approach does not appear to suffer the computational cost explosion of the other methods, we hope that our methodology will allow us to verify bigger and more complex examples.

References

[1] R. S. Boyer and J S. Moore. A Computational Logic Handbook. Academic

Press, Boston, 1988.

[2] B. Brock, W. A. Hunt, Jr., M Kaufmann, The FM9001 Microprocessor Proof. Technical Report 86, Computational Logic, December, 1994.

[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] A. Cohn., A Proof of the VIPER Microprocessor: The First Level; In: G. Birtwistle and P. A. Subrahmanyam Editors, VLSI Specification, Verification and Synthesis, page 27-72 Kluwer Academic Publishers, Boston, MA, 1988.

[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.: Microprocessor Design Verification; Journal of Automated Reasoning, 5(4), December 1989, page 429-460.

[9] 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

[10] W. A. Hunt, Jr., FM8501: A Verified Microprocessor. Lecture Notes in Computer Science 795, Springer-Verlag, 1994.

[11] M. Johnson, Superscalar Microprocessor Design, Prentice Hall, Englewood Cliffs, NJ, 1991.

[12] 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.

[13] 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.

[14] M. Srivas, M. Bickford, Formal Verification of a Pipelined Microprocessor, IEEE Software, September 1990, page 52-64.

[15] M. K. Srivas, S. P. Miller, Formal Verification of a Commercial Microprocessor, Technical Report SRI-CSL-95-12, SRI Computer Science Laboratory, July 1995.

[16] 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.

A Pipeline Properties in table-of?()

The predicate table-of?() defines various pipeline properties using the instruction table representation. It is defined as:

```
(defun table-of? (tbl s)
  (and (table-p tbl)
       (no-stage-conflict? tbl)
       (match-each-entry tbl s)
       (exist-matching-entry tbl s)
       (cntl-matching tbl s)
       (valid-inst-stream? tbl)
       (speculative-inst-in-IF/DC tbl)
       (valid-speculation tbl)
       (stage-order-preserved? tbl)
       (regs-match? tbl (p-state-regs s))
       (mem-match? tbl (p-state-mem s))
       (not (exist-output-violation? tbl))
       (not (exist-mem-dep-violation? tbl))
       (pc-correspondence tbl s))),
```

where arguments `tbl` and `s` are an instruction table and a pipelined state, respectively . Each property and its simple description are listed below:

`(table-p tbl)` :
: Well-typed predicate of the instruction table.

`(no-stage-conflict? tbl)` :
: Predicate checking non-existence of stage conflicts between instructions. No instructions in the table should share the same pipeline stage.

`(match-each-entry tbl s)` :
: Property that each partially executed instruction has correct intermediate values in the pipeline.

`(exist-matching-entry tbl s)` :
: Property that the instruction table contains all the instructions which are currently executed in the pipeline.

`(cntl-matching tbl s)` :
: Property that the bookkeeping register for the issuing logic is consistent with the pipeline state. See discussion in Section 6.

`(valid-inst-stream? tbl)` :
: Property that the logical sequential states in the table satisfy the defining equation: $(pc_{i+1}, regs_{i+1}, mem_{i+1}) = \text{s-step}(pc_i, regs_i, mem_i)$. See the argument about the logical sequential state in Section 4.

`(speculative-inst-in-IF/DC tbl)` :
: Property that no speculative instruction should proceed beyond the decoding stage. IF/DC is the only stage which a speculatively executed instruction is allowed to be in.

`(valid-speculation tbl)` :
: Property that a speculatively executed instruction follows a conditional branch instruction or a synchronization instruction. In our machine, a synchronization instruction is treated as if it is a conditional branch.

`(stage-order-preserved? tbl)` :
: Property that all the instructions are issued in order, and that all the instructions before the issuing stage is queuing up in-order in the pipeline.

`(regs-match? tbl (p-state-regs s))` :
: Property that the register file reflects written-back values of all the retired instructions in the instruction table.

`(mem-match? tbl (p-state-mem s))` :
: Property that the memory state reflects all the retired STORE instructions in the table.

`(not (exist-output-violation? tbl))` :
: Non-existence of WAW-hazards with respect to the register file.

`(not (exist-mem-dep-violation? tbl))` :
: Non-existence of WAW-hazards and RAW-hazards with respect to the memory.

`(pc-correspondence tbl s)))` :
: Property that the program counter in the pipeline machine correctly points to the next instruction to be executed, except when the machine is executing instructions speculatively.

B Definition of stage-order-preserved?()

Stage-order-preserved?() is the predicate checking that instructions are issued in-order. It also checks that all the instructions before the issuing stage are lining up in-order in the pipeline.

The stage of an instruction is represented by the latch name at which the instruction is stored in. There are latches named **IF/DC**, **DC/EX**, **EX/WB**, **LD1/LD2**, **ML1/ML2**, **ML2/ML3**. **RETIRE** is a special stage constant to indicate that the instruction is retired.

The recursively defined predicate stage-order-preserved?() systematically checks every combination of instructions in the table against the issuing-order constraints. Stage-order-preserved?() repeatedly calls the function state-order-earlier(), which checks the constraints for each pair.

```
; The next predicate defines the constraints that should be
; satisfied by a pair of instructions so that they are issued
; in-order.
(defun stage-order? (stg1 stg2)
  (case stg1
    (DC/EX (if (equal stg2 *IF/DC*) nil T))
    ((LD1/LD2 ML1/ML2 ML2/ML3 EX/WB RETIRE)
      (if (member stg2 (list *IF/DC* *DC/EX*)) nil T))
    (otherwise T)))

; The predicate stage-order-earlier checks whether instructions
; represented by entry1 and entry2 satisfy the issuing order
; constraints.
(defun stage-order-earlier (entry1 entry2)
  (stage-order? (entry-stg entry1) (entry-stg entry2)))

(defun stage-order-earlier-for-all (tbl entry)
  (if (endp tbl)
      T
      (and (stage-order-earlier (car tbl) entry)
           (stage-order-earlier-for-all (cdr tbl) entry))))

(defun stage-order-preserved? (tbl)
  (if (endp tbl)
      T
      (and (stage-order-earlier-for-all (cdr tbl) (car tbl))
           (stage-order-preserved? (cdr tbl)))))
```

C Definition of exist-output-violation?()

Exist-output-violation?() checks the existence of WAW-hazards. Exist-output-violation?() calls output-violation?() repeatedly to check every combination of instructions against the WAW-hazard condition. A WAW-hazard occurs if two conditions hold; if the instructions share the same destination register, and if they write back their results out-of-order.

```
(defun reg-output-dependent? (entry1 entry2)
  (equal (rnames-rcn (entry-rnames entry1))
         (rnames-rcn (entry-rnames entry2))))

(defun reg-output-stg-out-of-order? (stg1 stg2)
  (case stg1
    (EX/WB (member-equal stg2 (list *RETIRE*)))
    (LD1/LD2 (member-equal stg2 (list *EX/WB* *RETIRE*)))
    (ML2/ML3 (member-equal stg2 (list *EX/WB* *RETIRE*)))
    (ML1/ML2 (member-equal stg2 (list *LD1/LD2* *ML2/ML3* *EX/WB* *RETIRE*)))
    (otherwise nil)))

(defun reg-output-out-of-order? (entry1 entry2)
  (reg-output-stg-out-of-order? (entry-stg entry1) (entry-stg entry2)))

; Output-violation? returns non-nil value if there is a WAW-hazard
; between instructions represented by entry1 and entry2.
(defun output-violation? (entry1 entry2)
  (if (and (member-equal (opcode (entry-inst entry1))
                         (list 0 1 3)) ; add mult or ld
           (member-equal (opcode (entry-inst entry2))
                          (list 0 1 3)))
      (and (reg-output-dependent? entry1 entry2)
           (reg-output-out-of-order? entry1 entry2))
      nil))

(defun list-output-violation? (entry tbl)
  (if (endp tbl)
      nil
      (if (output-violation? entry (car tbl))
          T
          (list-output-violation? entry (cdr tbl)))))

(defun exist-output-violation? (tbl)
  (if (endp tbl)
      nil
```

```
    (or (list-output-violation? (car tbl) (cdr tbl))
        (exist-output-violation? (cdr tbl)))))
```

D Proof of Non-Existence of WAW-hazards

The proof of Lemma 1 is split into separate sub-proofs, in which we prove each property in Appendix A holds invariantly during pipeline state transitions. In this section, we give a proof sketch of the property asserting that no WAW-hazard occurs in the pipeline state transitions. This property is defined with the predicate exist-output-violation?() in Appendix C.

Our proof goal is given as the theorem NOT-EXIST-OUTPUT-VIOLATION-PRESERVED shown below, which asserts that there is no WAW-hazard for any combination of instructions. Simple inductions reduce this theorem to a sub-lemma NOT-OUTPUT-VIOLATION-STEP-ENTRY, which states that no WAW-hazard occurs between a particular pair of instruction entries `e1` and `e2`.

NOT-OUTPUT-VIOLATION-STEP-ENTRY is proved by case analysis on the pipeline stage of the instruction `e2`. Lemma NOT-OUTPUT-VIOLATION-STEP-ENTRY-DC/EX covers the most critical case, where the instruction `e2` is stored in DC/EX latch and waiting for the issue. Other cases are strait-forward from definitions and basic lemmas.

A WAW-hazard occurs if an out-of-order writeback happens between instructions that share the same destination register. In our machine, an out-of-order writeback happens only if an adding instruction is issued one machine cycle after a multiplying instruction is issued, because an addition takes one cycle in the execution unit while a multiplication takes three cycles. In order to prevent WAW-hazards, our issuing logic suspends the issue of an adding instruction when a multiplying instruction is in the latch **ML1/ML2**. This fact is proven in the lemma NO-ADD-RELEASE-IF-OUTPUT-DEPENDENT. Using this lemma, NOT-OUTPUT-VIOLATION-STEP-ENTRY-DC/EX is proved by cases analysis on the stage of `e1`.

Major lemmas are shown below:

```
; (inst-at stg tbl) returns the instruction at the stage stg.
; (reg-output-dependent? inst1 inst2) is non-nil if there is a
; WAW-dependency between inst1 and inst2.
; (release-add? s) returns the value of the control line
; release-add?, which is asserted when an adding instruction is
; issued.
;
; The lemma intuitively states that the control line release-add?
; is not asserted if there is a WAW-dependency between the
; instructions at DC/EX and ML1/ML2.
(defthm NO-ADD-RELEASE-IF-OUTPUT-DEPENDENT
```

```
    (implies (and (table-of? tbl s)
                  (p-state-p s)
                  (inst-at *DC/EX* tbl)
                  (inst-at *ML1/ML2* tbl)
                  (reg-output-dependent? (inst-at *ML1/ML2* tbl)
                                         (inst-at *DC/EX* tbl)))
             (not (b1p (release-add? s)))))

; This lemma tells that no WAW-hazard occurs between instructions
; e1 and e2, if e2 is in the stage DC/EX.  The proof requires the
; lemma NO-ADD-RELEASE-IF-OUTPUT-DEPENDENT
(defthm NOT-OUTPUT-VIOLATION-STEP-ENTRY-DC/EX
    (implies (and (table-of? tbl s)
                  (p-state-p s)
                  (entry-p e1)
                  (entry-p e2)
                  (member-equal e1 tbl)
                  (member-equal e2 tbl)
                  (match-entry e1 s)
                  (match-entry e2 s)
                  (equal (entry-stg e2) *DC/EX*)
                  (stage-order-earlier e2 e1)
                  (not (output-violation? e1 e2)))
             (not (output-violation? (step-entry e1 s)
                                     (step-entry e2 s)))))

; The following lemma tells that no WAW-hazard occurs between two
; instructions represented by e1 and e2.
(defthm NOT-OUTPUT-VIOLATION-STEP-ENTRY
    (implies (and (p-state-p s)
                  (table-of? tbl s)
                  (match-entry e1 s)
                  (match-entry e2 s)
                  (member-equal e1 tbl)
                  (member-equal e2 tbl)
                  (stage-order-earlier e2 e1)
                  (not (output-violation? e1 e2)))
             (not (output-violation? (step-entry e1 s)
                                     (step-entry e2 s)))))

(defthm NOT-EXIST-OUTPUT-VIOLATION-PRESERVED
    (implies (and (table-of? tbl s)
                  (p-state-p s))
             (not (exist-output-violation? (step-table tbl s)))))
```

E Proof of the Pipeline Correctness Diagram

The theorem COMMUTATIVE-LEMMA shown below formally expresses the concept presented by the pipeline correctness diagram in Figure 3. The hypothesis `(new-inst-readin? s tbl)` in COMMUTATIVE-LEMMA assumes that a new instruction is fetched in the pipelined state transition at the left-hand side of the diagram. IDEMPOTENT-LEMMA is a similar theorem about the case where no instruction is fetched in the state transition. From these theorems, we can prove the theorem STEPN-LEMMA, which is an n-cycle version of the pipeline correctness diagram.

The proof of COMMUTATIVE-LEMMA is split into three sub-proofs, in which the program counter, the register file and the memory are separately shown to satisfy the diagram. For instance, in the proof of DRAIN-REGS-COMMUTE, we calculate the states of the register file by following the two paths in Figure 3 and show their equivalence. We discuss this example in the rest of this section.

Before examining the proof of DRAIN-REGS-COMMUTE, let us discuss the functions final-regs() and virtual-new-state(). As we discussed in Section 4, each instruction has its logical sequential state. Suppose instructions $I_1, I_2, \ldots, I_n$ are in an instruction table *tbl*. Each instruction I_i has its logical sequential state s_i. In the ideal execution of the instructions in the sequential machine, s_i is the machine state *before* I_i is executed. When we add an new instruction I_{n+1} to the table, we need to calculate the new logical sequential state s_{n+1} from s_n by the defining equation discussed in Section 4. This logical state s_{n+1} is the ideal machine state immediately after completing I_n and before executing I_{n+1}. Function virtual-new-state() calculates this state s_{n+1} from s_n.

Let us consider a pipeline state p which is represented by the table *tbl*. We want to define functions final-pc(), final-regs() and final-mem() as table-based representations of the program counter, the register file and the memory, respectively, of the synchronized state after flushing the pipeline p. The flushing of p completes all the partially executed instructions in the instruction sequence $I_1, I_2, \ldots, I_n$. The synchronized state after pipeline flushing must be equivalent to the logical state s_{n+1}, because the next instruction to be executed is I_{n+1}. We define final-pc(), final-regs() and final-mem() as functions calculating s_{n+1} and extracting the corresponding component from it. We, then, prove that they correctly represent the effect of flushing for each component. For instance, the theorem P-DRAIN-FINAL-REGS confirms that the final-regs() returns the register file after flushing.

Note that the return value of virtual-new-state() is usually equal to the values of final-pc(), final-regs() and final-mem() in the following sense:

```
(virtual-new-state tbl s)
= (s-state (final-pc tbl s) (final-regs tbl s) (final-mem tbl s)),
```

where s-state() is the function constructing a sequential state out of component states. This equality is proven in FINAL-S-IS-VIRTUAL-NEW-STATE. Due to the subtlety regarding the speculatively executed instructions, this equation does not always hold, but we will ignore it in this proof sketch.

Now the sketch of a proof of DRAIN-REGS-COMMUTE is here:

```
(p-state-regs (p-drain (p-step s) (step-table tbl s)))             (1)
= (final-regs (step-table tbl s) (p-step s))                       (2)
= (final-regs (append (step-all-entries tbl s)                     (3)
                      (list
                        (new-entry s (virtual-new-state tbl s))))
              (p-step s))
= (final-regs (list (new-entry s (virtual-new-state tbl s)))       (4)
              (p-step s))
= (s-state-regs (s-step (virtual-new-state tbl s)))                (5)
= (s-state-regs (s-step (s-state (final-pc tbl s)                  (6)
                                 (final-regs tbl s)
                                 (final-mem tbl s)))
=(s-state-regs (s-step (sp-proj (p-drain s tbl))))                 (7)
```

Step `(1)` $\rightarrow$ `(2)`: By using lemma P-DRAIN-FINAL-REGS, We replace `(p-state-regs (p-drain ...))` by `(final-regs ...)`. As we discussed above, `(final-regs ...)` is a table-based representation of the register state after pipeline flushing.

Step `(2)` $\rightarrow$ `(3)`: This step is done by unrolling the definition of step-table(). Because we have the assumption that the pipeline fetches an instruction in the next cycle, a new instruction entry is added to the updated table. We can see it because `(step-table tbl s)` in (2) is replaced by `(append ...)` in (3), in which `(new-entry ...)` represents the new entry.

Step `(3)` $\rightarrow$ `(4)`: As we can see from the definition, the function final-regs() calculates its value from the logical sequential state of the last entry of the table. So, we can drop the first half of the concatenated instruction table. Lemma FINAL-REGS-APPEND justifies this step.

Step `(4)` $\rightarrow$ `(5)`: This is immediate from the definition of final-regs. See the definition of final-regs.

Step `(5)` $\rightarrow$ `(6)`: By lemma FINAL-S-IS-VIRTUAL-NEW-STATE. See the discussion above about the relation of virtual-new-state() and final-pc(), final-regs() and final-mem().

Step `(6)` $\rightarrow$ `(7)`: This step is justified by P-DRAIN-FINAL-REGS and similar lemmas about the program counter and the memory, which are not shown in this appendix. Also note that sp-proj() is the projection from pipeline states to sequential states satisfying:

```
(sp-proj s)=(s-state (p-state-pc s) (p-state-regs s) (p-state-mem s))
```

Major lemmas are:

```
; Final-regs represents the register file state after pipeline
; flushing.  The definition simply returns the logical sequential
; state after the last instruction.
(defun final-regs (s tbl)
  (if (endp tbl)
      (p-state-regs s)
      (if (or (endp (cdr tbl)) (speculative-step? (car tbl)))
          (s-state-regs (s-step (s-state (entry-pc (car tbl))
                                         (entry-regs (car tbl))
                                         (entry-mem (car tbl)))))
          (final-regs s (cdr tbl)))))

; This theorem says that final-regs correctly represents the
; register file state after flushing.
(defthm P-DRAIN-FINAL-REGS
    (implies (and (p-state-p s)
                  (table-of? tbl s))
             (equal (p-state-regs (p-drain s tbl))
                    (final-regs s tbl))))

; The first half of the concatenated instruction table is
; irrelevant to final-regs.
(defthm FINAL-REGS-APPEND
    (implies (and (not (in-speculative-branch? tbl1))
                  (consp tbl2))
             (equal (final-regs s (append tbl1 tbl2))
                    (final-regs s tbl2))))

; Unless we are speculatively executing instructions, the final
; state after pipeline flushing should be equal to that of logical
; sequential state of a newly fetched instruction.
(defthm FINAL-S-IS-VIRTUAL-NEW-STATE
    (implies (and (p-state-p s)
                  (table-of? tbl s)
                  (not (in-speculative-branch? tbl)))
             (equal (s-state (final-pc s tbl)
                             (final-regs s tbl)
                             (final-mem s tbl))
                    (virtual-new-state tbl s))))

; The pipeline correctness diagram holds with respect to the
; register file.  This is one of three cases we have to prove for
```

```
; COMMUTATIVE-LEMMA.
(defthm DRAIN-REGS-COMMUTE
    (implies (and (p-state-p s)
                  (table-of? tbl s)
                  (new-inst-readin? s tbl))
             (equal (p-state-regs (p-drain (p-step s) (step-table tbl s)))
                    (s-state-regs (s-step (sp-proj (p-drain s tbl)))))))

(defthm COMMUTATIVE-LEMMA
    (implies (and (p-state-p s)
                  (table-of? tbl s)
                  (new-inst-readin? s tbl))
             (equal (sp-proj (p-drain (p-step s) (step-table tbl s)))
                    (s-step (sp-proj (p-drain s tbl))))))

(defthm IDEMPOTENT-LEMMA
    (implies (and (p-state-p s)
                  (table-of? tbl s)
                  (not (new-inst-readin? s tbl)))
             (equal (sp-proj (p-drain (p-step s) (step-table tbl s)))
                    (sp-proj (p-drain s tbl)))))

; Final theorem. (p-stepn s n) runs the pipelined machine n
; cycles.  P-drain flushes the pipeline.  Sp-proj is the
; projection from pipelined machine states to sequential machine
; states.  The lemma says that N cycles of pipelined machine
; execution is equivalent to (number-of-inst-fetched s tbl n)
; cycles of sequential machine.
(defthm STEPN-LEMMA
    (implies (and (p-state-p s)
                  (table-of? tbl s))
             (equal (sp-proj (p-drain (p-stepn s n) (stepn-table tbl s n)))
                    (s-stepn (sp-proj (p-drain s tbl))
                             (number-of-inst-fetched s tbl n)))))
```