Doctoral Dissertation Proposal

Design Verification of Advanced Pipelined Machines

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

abstract

We have developed formal verification techniques for the high-level design of pipelined microprocessors. Our technique represents the instruction stream in such a way that pipeline properties, such as RAW hazards do not exist, can be directly expressed. This approach allows us to prove the correctness of the machine design incrementally, which reduces the computational cost of the verification. Thus, we believe we can apply our techniques to designs with more complex pipeline control than those previously verified. We are enhancing our techniques to permit the verification of microprocessors containing advanced pipeline features, such as precise exception handling and the out-of-order issue of instructions.

1 Overview

Today's microprocessors are built of 50 to 200 cooperating micro-architectural components, which may be designed by separate engineers. The lack of a rigorous, abstract, high-level specification may cause misunderstandings, which can result in design errors. Our approach stresses the use of specification and verification at the most abstract levels for microprocessor design.

Our view of the processor verification is given in Figure 1 The highest level description of a microprocessor is given as an instruction-set specification, which describes the effect of individual instructions. The micro-architectural design is given as a behavioral description, which exposes micro-architectural design choices such as pipelining. A high-level verification checks the equivalence between these two levels. Once a micro-architectural design is verified, we believe that we can check an RTL implementation against it. Therefore, we have chosen to focus our research on the verification of micro-architectural designs.

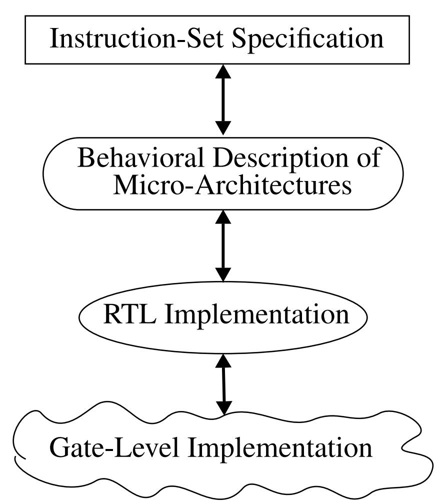

FIGURE 1.

A major difficulty in the high-level verification of today's microprocessor is verifying that a pipelined microprocessor executes each instruction correctly. Instruction execution in modern pipelined microprocessors is often parallelized using multiple execution units, which are themselves pipelined[8]. Verifying the correct operation of a microprocessor that has a speculative, out-of-order, multi-issue, pipelined micro-architecture design is a difficult and essentially unsolved problem.

A major difficulty in verifying pipelined machine designs is concurrency. When verifying a non-pipelined design, we can check its correctness by examining the machine state after completion of each instruction. This is usually done by defining a *time abstraction* mapping from a machine implementation to its specification. Since pipelined machines can execute more than one instruction concurrently, the boundary of execution between instructions is often blurred making it difficult to determine their correctness.

We have been developing new techniques for verifying the correctness of pipelined machines. One correctness criteria of a pipelined machine is defined using the idea of flushing instructions out of the pipeline. Although this definition of correctness criteria was proposed and used by earlier research[2], our new techniques have enabled us to verify pipelined machine designs containing more complex control logic than those previously verified.

One of the major techniques in our approach is representing the stream of executed instructions with a table. There are many properties of the pipelined machine and instructions that can be more easily expressed and reasoned about with help of the table. For example, a *RAW dependency*[5] between instructions is much easier to represent using our instruction table instead of lifting the necessary information from the actual pipeline design.

Using our instruction table approach, we have verified two example machine designs. Our first design was a very simple pipelined machine with two memory units; it simply copies the contents of one memory to the other. Our second example is a simple microprocessor design with out-of-order execution. In our latter example, we showed that our methodology can verify various features like data bypass logic, dynamic resolution of instruction dependencies, out-of-order register writeback, and speculative execution of instructions.

We know of no other earlier work which has verified a pipelined design with out-of-order completion. Our methodology has not suffered the severe case explosion that prevented automatic verification techniques [2][9] from being applied to a bigger example.

We plan to enhance our verification techniques for pipelined machine designs with advanced features in the following ways:

1. Verification of a larger example with advanced features,
2. Verification of exception handling mechanisms in pipelined machines, and
3. Integration of automatic verification techniques into our methodology.

Each of the topics will be discussed in more detail in Section 4.

2 Background

In order to verify the high-level micro-architectural design of a machine, a user must provide the behavioral descriptions of both the instruction-set specification and the micro-architectural design of interest. The behavioral description of a machine is given as a function, which takes the current machine state and returns the next state. It is also necessary to define the abstraction mapping from an implementation state to a specification state. Then, we can show the equivalence between implementation and specification by proving that a commutative diagram, like that shown in Figure 2, exists. Let us note that the abstraction mapping may likely abstract the time sequence[7] as well as the data. For example, a machine design may take more than one cycle to execute an instruction. The abstraction mapping will obviously need to take this fact into account.

In general, however, pipelined machines do not satisfy the commutative diagram shown in Figure 2 Pipelined machines usually start the execution of the next instruction before the previous one is completed, and there is no clear boundary of execution between instructions. Thus, there is no natural abstraction of the time sequence which makes the diagram in Figure 2 hold for pipelined designs. Since the instruction-set specification runs instructions one-by-one, we call it the *sequential specification* as opposed to the *pipeline implementation*, in order to make clear the difference in styles of execution.

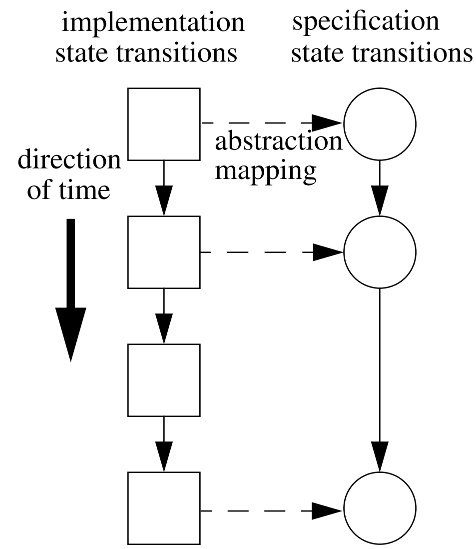

FIGURE 2.

There are several earlier pipelined processor verification efforts[4][11][12][13] in which they proved commutative diagrams like Figure 2 using a peculiar abstraction mapping. In many of them[4][12][13], they combine the states of different components at different times to produce a corresponding sequential specification state. For instance, the program counter at time t and the register file and the memory at time $t + 2$ are combined to form a corresponding sequential specification state. In general, earlier pipelined processor verification efforts verified machines with very simple pipeline control and limited interlocks. We believe that their solution to the problem of time abstraction will be difficult to employ when verifying a complex pipelined machine. For instance, it is obvious that the state mapping combining different component states at different times becomes very difficult for a pipelined machine with out-of-order execution.

A more general method to express the correctness of the pipeline behavior has been proposed by Burch and Dill[2]. Mapping of a pipelined machine state to a sequential machine state is done by first flushing instructions out of the pipeline and then projecting the synchronized pipeline state to a sequential specification state. *Pipeline flushing* is done by running the machine without fetching a new instruction, until the pipeline is empty. Using this approach, our proof obligation for verifying a pipelined design is shown in Figure 3. We call it *Burch and Dill's pipeline correctness diagram*, or simply the pipeline correctness diagram.

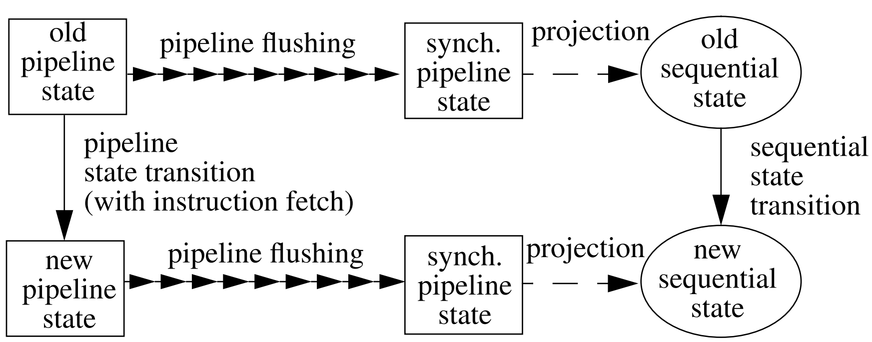

FIGURE 3.

This diagram elegantly defines a correctness criteria of processors with complex pipeline interlocks. As we discuss later, we showed that this method is also applicable to the processors with out-of-order execution or speculative execution. Also, we can easily prove an *n*-cycle version of the diagrams, which states that we can emulate the sequential execution of the specification with the pipelined machine design, as long as the initial and final machine states are synchronized. Although we are still looking for an alternative to represent the pipeline correctness, Burch and Dill's pipeline correctness diagram is the best method we know of for representing the correctness property of a complex pipelined machine.

The verification of the pipeline correctness diagram shown in Figure 3 is not a easy problem. Originally, Burch and Dill verified this diagram by symbolically simulating a machine design in a logic of uninterpreted functions with equality. In this paper, we call it the *symbolic execution method*. The symbolic execution method is highly automated and much faster than ordinary simulations, because it manipulates the datapath symbolically, and verifies only the control logic of the pipeline. However, the cost of verification increases dramatically as the control logic becomes complex and the number of cases to be examined explodes. It is possible to use heuristics to improve the speed of the verification[9], but it still does not solve the case explosion problem.

3 Method of Pipeline Verification and its Examples

We concluded that Burch and Dill's pipeline correctness diagram is a strong and succinct way to represent the correctness of the pipelined machines. We have developed new techniques to verify the diagram for a pipelined processor with complicated control logic. Our techniques can be summarized as:

1. A table-based representation of an instruction stream,
2. Expressing pipeline properties using the instruction stream table,
3. Theorem-prover based verification to prove pipeline properties, and
4. Proof of the pipeline correctness diagram.

A major difficulty in verifying the Burch and Dill's diagram is caused by flushing, which takes many pipeline machine cycles. Unrolling the behavioral description of the pipelined machine to examine this cycles leads to an explosion of the verification cost. Because of this, the symbolic execution method cannot verify a big design. With our approach, we first prove invariant pipeline properties over the pipeline state transitions, and then we prove the correctness diagram by induction on the number of flushing cycles, with help of the proven properties

We found that it is not easy to define many properties of a pipelined machine design without reasoning about the instructions. For instance, in order to show that all the issued instructions obtain the correct operand values, we have to reason about RAW dependencies between instructions. We also noticed that pipelined machines do not hold necessary information about all the instructions being executed currently or in the past. For example, a pipeline state does not supply enough information to reason about *WAW dependency*[5] of a machine with out-of-order execution. Suppose we want to reason about WAW dependency between an instructions I1 and its following instruction I2. If I2 has retired earlier than I1, the pipeline state no longer contains knowledge about I2, making it difficult to reason about the dependency between I1 and I2. Thus, we need another level of abstraction of the pipelined machine which enables us to reason about instructions easily.

Our approach introduces a table-based representation of the instruction stream, or what we call the *instruction table*. An instruction table is a list of instructions, which are currently executed in the pipeline or have been completed in the past. The order of instructions in the table is the same order as sequential execution. It can be said that our instruction table corresponds to a column of the reservation table.

Each entry of the table contains information relevant to an instruction, such as the instruction word, the operand registers, the current pipeline stage, and what we call the *logical sequential state* for the instruction. The logical sequential state of an instruction is the sequential specification state in which the instruction should logically be executed. We can say that it is the state which programmers have in mind for the execution of the instruction.

In our method, we also define a *table update function* which simulates the behavior of the pipeline. If a table correctly represents a pipeline state, the updated table should also represents the next pipeline state.

Using our table representation of the instruction stream, it is straightforward to represent various properties of a pipeline. Table 1 contains examples of the pipeline properties verified in Example 2, which we discuss later. For example, the property defined with `no-stage-conflict?` is that no instructions crash into the same pipeline stage, causing a structural hazard. We proved these properties invariantly hold during the pipeline state transition. The typical proof of them can be conducted independently, even though some properties have to be proved before the proof of another. These properties are usually proved by examining the pipeline state transition during only one machine cycle. Thus, the proofs of the pipeline properties breaks down the whole problem into simpler problems, allowing us to attack the verification problem incrementally.

Pipeline Property	**Comments**
`(no-stage-conflict? tbl)`	No instructions in the instruction table are in the same pipeline stage.
`(match-each-entry tbl s)`	Checks whether each partially executed instruction in the instruction table is correctly represented in the pipelined machine.
`(stage-order-preserved? tbl)`	Our machine issues instructions in-order. All the instructions before the issuing stage should be in order in the pipeline.
`(regs-match? tbl (p-state-regs s))`	Property asserting that the register file reflects the side-effects of all the retired instructions, but not the partially executed instructions.

TABLE 1. Pipeline properties verified in Example 2.

Once these pipeline properties are proved to hold for any reachable pipeline state, we can prove the pipeline correctness diagram using the induction over the number of machine cycles for pipeline flushing, under the assumption that the “old pipeline state” in Figure 3 on page 3 is a reachable state. The expressive power of the instruction table helps the proof of this diagram.

Because the pipeline properties are proven incrementally, and also because the pipeline correctness diagram is logically proved at an abstract level with help of various pipeline properties, our methods does not suffer from the case explosion which severely cramps the automatic symbolic execution method. Another merit of our approach is that the pipeline properties can be used to restrict the “old pipeline state” in Figure 3 to the reachable machine states, so that the diagram is correct. The original work by Burch and Dill[2] suggests that they can supply an invariant condition to restrict “the old pipeline state”, but they don’t discuss how to represent the condition or how to verify its closure property.

We have verified two examples so far. We will briefly discuss each of them.

Example 1. Our first example we verified was a simple design built of pipelined latches and two memory units. This machine merely copies the content of one memory unit to another by incrementing a counter. The design of this machine was organized to make it easy to change the length of pipeline and the configuration of interlocks easily. We experimented with a variety of configurations.

We verified this design with various approaches, including automatic symbolic execution method and our theorem-proving approach based on the instruction table. The verification for the latter method was mechanically checked using the Nqthm theorem prover[1]. We concluded that our instruction-table method will likely scale to permit the verification of a large pipelined machine with complex control logic.

Example 2. We have shown that our instruction-table method can be applied to a larger machine in this example. This example machine has an in-order issue, out-of-order completion design. Since different instruction types require a different number of cycles to complete, in-order issued instructions may retire in an out-of-order fashion. Thus, the pipeline control logic has a capability to resolve WAW dependency, as well as the RAW dependency, of instructions. The design of the pipeline also fetches instructions speculatively after conditional branches.

The pipeline correctness diagram can be easily extended to our second example machine, even if the machine performs speculative execution and out-of-order execution. We successfully verified the diagrams for this machine using our methodology. The proof was mechanically checked by ACL2 theorem prover[10]. The major difficulty in this verification exercise was proving the properties about out-of-order and speculative execution. The clarity of the instruction table in expressing abstract concepts greatly helped us to verify their correctness.

With respect to the pipeline control logic, this example is more complex than others verified. No other effort has ever verified a pipelined machine with all of out-of-order execution, dynamic resolution of RAW and WAW dependencies, and speculative execution.

4 Future Work Proposal

Our recent research has been focused on basic techniques for the formal verification of pipelined machines, especially those with complex control logic. We have identified several important concepts and techniques for high-level verification. In our future research, we are going to apply them to more realistic examples, and study if our approach is applicable to practical verification problems.

We will discuss three major problems as the goals of our future work. We discuss our major goal in Section 4.1, then talk about two derived topics in 4.2 and 4.3.

4.1 Verification of a larger example with advanced features

We want to test our techniques on a larger processor design which contains advanced implementation features. Example 2 discussed in Section 3 is still a small machine design, even though it contains various interesting features. It can execute only 6 instructions and it has no external inputs, no exception, and no memory interaction. It does not have many interesting features of today's microprocessors, such as out-of-order issue of instructions, renaming registers and reorder buffers[3][5][8].

For a larger example, we plan to target a subset of the instruction-set architecture of a commercial microprocessor. We plan to construct a machine design that will include the following features:

1. Out-of-order issue, and possibly multi-issue, of instructions based on Tomasulo's algorithm;
2. Speculative execution based on dynamic branch prediction;
3. Processor-memory interactions and external interrupt; and
4. Precise exception handling.

Tomasulo's algorithm is a widely used mechanism to implement today's superscalar pipelined processors. The re-order buffer has a critical role in this algorithm. We also want to attack the problem of multi-issue of instructions. However, this may complicate the verification task too much. We will consider this problem and decide later whether we will study this problem.

We will implement speculative execution based on dynamic branch prediction; however, we do not have to verify the accuracy of the branch prediction. We only have to prove that the instructions are executed correctly regardless of the outcome of the branch prediction. We will also make the processor accept external inputs such as external interrupts and memory interactions. The external inputs are represented as a list of oracles to the functional behavioral description. Precise exception handling will be discussed in the following subsection.

The verification of this new example is our primary goal in the dissertation research. It will also be a workbench for the subordinate problems discussed in the following two subsections. It may be out of the scope of this research to answer all of the problems discussed below. Thus we want to present the following topics as the possible derivatives of the verification research discussed in this subsection.

4.2 Verification of Exception Handling

Exception handling mechanism is a rich source of design errors of microprocessors, and also a difficult part to verify using simulation. Very little work has been reported regarding the verification of exception handling mechanisms by pipelined machines.

Srivas and Bickford[11] verified a pipelined machine with external interrupts. They have noted that the correctness of the software interrupt handler is not important for the verification of hardware. They verified that the behavior of their pipelined machine fielding an interrupt is equivalent to that of the specification. However, their machine was simple enough to have an abstraction mapping satisfying the diagram in Figure 2 on page 3 Thus, it is not clear whether we can directly apply their technique to a machine with complex pipeline control, whose correctness can be represented by the pipeline correctness diagram in Figure 3 on page 3, but not the one shown in Figure 2 Since the pipeline correctness diagram does not address the issue of exception handling, we will need to reconsider the basic framework of our verification approach.

Another important issue about exception handling is *precise exception handling*[3]. In order to restart the machine after an exception, the pipelined machine has to complete all the partially executed instructions preceding the exception-causing instruction, abandon all the side-effects by any following instructions, and then save the machine state. This is called the precise exception handling. No earlier verification work, including [11], has worked on this issue.

As a preliminary research for exception verification, we will study this issue using a simple machine. For this purpose, we can modify the machine used in Example 2 and add a simple exception, like an undefined instruction or an external interrupt. With this example, we will study how to represent the correctness of exception handling and how to verify it.

We will also implement precise exception mechanism in the machine discussed in Subsection 4.1. We will implement more than one kind of exception occurring in different stages so we may study the handling of simultaneously raised exceptions. The combination of a memory access error and an undefined instruction can be an example of such simultaneous exceptions

4.3 Integration of Automatic Verification Techniques.

Since our method is based on theorem proving techniques, it is a labor-intensive process. This is a major problem in applying our techniques to a practically-sized examples. The automatic verification techniques are not so labor-intensive, but they work successfully on limited problems. They are usually very inefficient if applied directly to a large scale problem, like the one we are working on. However, it is possible that combining automatic techniques to our methodology will improve the efficiency of our verification.

One such example is using automatic verification methods like BDD and model-checking to prove properties about sub-components of a processor design, and then using the result in the reasoning by the theorem prover. The problem we have to consider is how to isolate properties that can be verified by automatic procedures and how to integrate them into our more general approach.

References

[1] R. S. Boyer and J S. Moore. A Computational Logic Handbook. Academic Press, Boston, 1988.

[2] 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.

[3] H. G. Cragon, Memory Systems and Pipelined Processors, Jones and Bartlett Publishers, Inc., 1996.

[4] D. Cyrluk. Microprocessor verification in PVS: A methodology and simple example, Technical Report SRI-CSL-93-12, SRI Computer Science Laboratory, Dec. 1993

[5] J. Hennessey, D. Patterson, Computer Architecture a Quantitative Approach, Morgan Kaufmann Publishers, Inc., 1996.

[6] W. A. Hunt, Jr.: Microprocessor Design Verification; Journal of Automated Reasoning, 5(4), December 1989, page 429-460.

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

[9] 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.

[10] 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.

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

[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, September 1994, IEEE Computer Society Press. page 285-289.