Formal Verification of Pipelined Processors*

Randal E. Bryant

Carnegie Mellon University, Pittsburgh PA 15213, USA

Abstract. Correspondence checking formally verifies that a pipelined microprocessor realizes the serial semantics of the instruction set model. By representing the circuit state symbolically with Ordered Binary Decision Diagrams (OBDDs), this correspondence checking can be performed directly on a logic-level representation of the circuit. Our ongoing research seeks to make his approach practical for real-life microprocessors.

1 Motivation

Microprocessors are among the most complex electronic systems created today. High performance processors require millions of transistors and employ complex techniques such as pipelining, multiple instruction issue, branch prediction, speculative and/or out-of-order execution, register renaming, and many forms of caching. When correctly implemented, these implementation artifacts should be invisible to the user. The processor should produce the same results as if it had executed the machine code in strict, sequential order.

Design errors can often lead to violations of the sequential semantics. For example, an update to a register or memory location by one instruction may not be detected by an instruction following too closely in the pipeline. An instruction following a conditional branch may be executed prematurely, modifying a register even though the processor later determines that the branch is taken. Such *hazard* possibilities increase dramatically as the instruction pipelines increase in both depth and width.

Historically, microprocessor designs have been validated by extensive simulation. Instruction sequences are executed, in simulation, on two different models: a high-level model describing the desired effect of each instruction and a low-level model capturing the detailed pipeline structure. The results from these simulations are then compared for discrepancies. The instruction sequences may be taken from actual programs or synthetically generated to exercise different aspects of the pipeline structure [5].

Validation by simulation becomes increasingly costly and unreliable as processors increase in complexity. The number of tests required to cover all possible pipeline interactions becomes overwhelming. Furthermore, simulation test generators suffer from a fundamental limitation due to their use of information about the pipeline structure in determining the possible interactions in an instruction sequence that need to be

* This research was supported in part by the Defense Advanced Research Project Agency, Contract DABT63-96-C-0071, and in part by the Semiconductor Research Corporation, Contract 97-DC-068.

simulated. A single conceptual design error can yield both an improperly-designed pipeline and a failure to test for a particular instruction combination.

As an alternative to simulation, a number of researchers have investigated using formal verification techniques to prove that a pipelined processor preserves the semantics of the instruction set model. Formal verification has the advantage that it demonstrates correct execution for all possible instruction sequences. Our interest is in developing automated techniques that apply powerful symbolic evaluation techniques to analyze the behavior of the processor over all possible operating conditions. We believe that high degrees of automation are essential to gaining acceptance by chip designers.

2 Verification Methodology

Our task is to verify that a processor will execute all possible instruction sequences properly. Since there is an infinite number of possible sequences, this condition cannot be proved directly. Instead, we show that each possible individual instruction will be executed correctly, regardless of the preceding and following instruction sequences. The correct execution of a complete sequence then follows by induction on its length. One methodology for proving the correctness of individual instructions is based on proving the invariance of an abstraction function between processor and program states by each instruction execution. A similar methodology was proposed by Hoare for proving the correctness of each operation in the implementation of an abstract data type [4].

We model the processor as having states in the set Q_{pipe}, and the behavior of the processor for each clock cycle of operation by a next-state function δ_{pipe}: $Q_{\text{pipe}} \rightarrow Q_{\text{pipe}}$. Similarly, the state visible to the assembly language programmer (typically the main memory, integer and floating point registers, program counter, and other status registers) is modeled by a state set Q_{prog} and the execution of a single instruction by a next-state function δ_{prog}: $Q_{\text{prog}} \rightarrow Q_{\text{prog}}$. In our simplified formulation, we we do not consider the input or output to the processor, but rather that the action taken on each step is determined by the program or pipeline state.

Our task is to show a correspondence between the transformations on the pipeline state by the processor and on the program state by the instruction execution model. This correspondence can be described by an *abstraction function* Abs: $Q_{\text{pipe}} \rightarrow Q_{\text{prog}}$ identifying which program state is represented by a given pipeline state. Typically, this corresponds to the effect of completing any instructions in the pipeline without fetching any new instructions. For each pipeline state, there must be a value k indicating the number of clock cycles required after fetching the most recent instruction until the next instruction can be fetched. Our correctness condition states that the effect of fetching and executing a single instruction should match the effect of performing the corresponding instruction on the program state. That is, for all $Q_{\text{pipe}} \in Q_{\text{pipe}}$, there must be a k such that

$$\delta_{\text{prog}}(Abs(Q_{\text{pipe}})) \;=\; Abs(\delta^{k}_{\text{pipe}}(Q_{\text{pipe}})) \qquad (1)$$

In addition to the invariance property described in Equation 1, we require Abs to be surjective to guarantee that all program behaviors can be realized. That is, for every program state Q_{prog}, there must be a state Q_{pipe} such that $Abs(Q_{\text{pipe}}) = Q_{\text{prog}}$. Beyond this requirement, the abstraction function can be arbitrary, as long as it satisfies Equation 1.

The validity of the verification is not compromised by an incorrect abstraction function. That is, an invalid abstraction function will not cause the verifier yield a "false positive" result, declaring a faulty pipeline to be correct. We can let user provide us with the abstraction function [1, 6], but this becomes very cumbersome with increased pipeline complexity. Alternatively, we can attempt to derive the abstraction function directly from the pipeline structure [3]. Unlike simulation test generation, using information about the pipeline structure does not diminish the integrity of the verification.

3 Automated Correspondence Checking

Burch and Dill [3] first proposed using the pipeline description to automatically derive its abstraction function. They do this by exploiting two properties found in many pipeline designs. First, the programmer-visible state is usually embedded within the overall processor state. That is, there are specific register and memory arrays for the program registers, the main memory, and the program counter. Second, the hardware has some mechanism for "flushing" the pipeline, i.e., to complete all instructions in the pipeline without fetching any new ones. For example, this would occur when the instruction cache misses and hence no new instructions could be fetched. A symbolic simulator, which can model the behavior of the circuit over symbolically-represented states, can automatically derive the abstraction function. First, we initialize the circuit to an arbitrary, symbolic state, covering all the states in Q_{pipe}. We then symbolically simulate behavior of a processor flush. We then examine the state in the program visible register and memory elements and declare these symbolic values to represent the mapping Abs. Using similar symbolic simulation techniques, we can also compute the effect of the processor on an arbitrary pipeline state δ_{pipe} and the effect of executing an arbitrary program instruction δ_{prog}. Thus, a symbolic simulator can solve the key problems related to verifying pipeline processors.

Burch and Dill use symbolic simulation tools based on a logic of uninterpreted functions with equality, a weakened form of first order predicate calculus. Using this weak form, they can guarantee a complete decision procedure. Using their approach requires having a model of the circuit that abstracts away many details, including the sizes of the memory and register arrays, the bit widths of the data paths, and even the functionality of the data operations. This abstract model allows them to concentrate on the key issues of pipeline structure and control. Unfortunately, it can be difficult and time-consuming to derive such a model from the circuit description and to maintain it as the circuit design evolves. In addition, determining the correctness of some aspects of the circuit behavior require more detailed information about the data operations.

4 Verifying at the Bit-Level

Our recent research has focussed on adapting Burch and Dill's verification methodology to operate directly on a low-level model of the circuit, in which state is explicitly represented by sets of Boolean values and the state transformations are described by Boolean functions over this state. Such a model can be derived directly from a logic-level

description of the circuit, avoiding the need to manually create a more abstract model. Instead of manipulating symbolic variables representing abstract state and uninterpreted functions, we use Ordered Binary Decision Diagrams (OBDDs) [2] to represent the symbolic circuit state. OBDDs have the advantage over other approaches to symbolic Boolean manipulation of being canonical as well as reasonably compact for many of the functions encountered in modeling digital circuits.

Many difficult hurdles must be overcome to make bit-level correspondence checking practical. One problem is to find an efficient representation of the initial, but arbitrary pipeline state, given the large number of memory elements found in a processor. A naive approach would be to introduce a distinct Boolean variable for each bit in each register or memory array, but this could require thousands, or even millions of variables. Instead, we have developed techniques to introduce only as many variables as are needed to represent the states of the symbolic locations that are actually accessed during the execution of the instruction sequence [7]. Since the sequences we symbolically simulate are relatively short, this leads to a greatly reduced number of variables. Other problems are related to the OBDD complexities caused when modeling the interactions between successive instructions, such as when the value generated by one instruction becomes the target address by a later jump instruction. We have made some progress in this area, but much more is required before we will be able to handle full scale microprocessors.

References

1. S. Bose, and A. L. Fisher, "Verifying Pipelined Hardware Using Symbolic Logic Simulation," *International Conference on Computer Design (ICCD '89)*, 1989, pp. 217–221.
2. R. E. Bryant, "Symbolic Boolean Manipulation with Ordered Binary Decision Diagrams," *ACM Computing Surveys*, Vol. 24, No. 3 (September, 1992), pp. 293–318.
3. J. R. Burch, and D. L. Dill, "Automated Verification of Pipelined Microprocessor Control," *Computer-Aided Verification (CAV '94)*, LNCS 818, Springer-Verlag, June, 1994, pp. 68–80.
4. C. A. R. Hoare, "Proof of Correctness of Data Representations," *Acta Informatica* Vol. 1, 1972, pp. 271–281.
5. M. Kantrowitz, and L. M. Noack, "I'm Done Simulating; Now What? Verification Coverage Analysis and Correctness Checking of the DECchip 21164 Alpha Microprocessor," *33rd Design Automation Conference (DAC '96)*, 1996, pp. 325–330.
6. K. L. Nelson, A. Jain, and R. E. Bryant, "Formal Verification of a Superscalar Execution Unit," *34th Design Automation Conference (DAC '97)*, June, 1997.
7. M. N. Velev, and R. E. Bryant, "Verification of Pipelined Microprocessors by Comparing Memory Execution Sequences in Symbolic Simulation," *Asian Computer Science Conference (ASIAN '97)*, LNCS 1345, Springer-Verlag, December 1997, pp. 18–31.