Automatic Test Program Generation for Pipelined Processors

Hiroaki Iwashita, Satoshi Kowatari, Tsuneo Nakata, and Fumiyasu Hirose

Fujitsu Laboratories Ltd.
1015 Kamikodanaka, Nakahara-ku, Kawasaki 211, Japan
hiroaki@flab.fujitsu.co.jp

Abstract

Simulation-based verification has both advantages and disadvantages compared with formal verification. Our demand is to find a practical way to verify actual microprocessors. This paper presents an efficient test program generation method for simulation-based verification using techniques developed for formal verification. Our test program generator enumerates all reachable states of a processor pipeline and generates instruction sequences for every reachable test case. The program covers complicated test cases that are difficult to cover with random instructions and impossible to cover with conventional test program generation methods. Our test program generator also works for larger microprocessor designs than formal verifiers have done.

1 Introduction

Logic simulation is still widely used to verify that an implementation conforms with its specifications, while various formal verification approaches have also been proposed in recent years. High-speed scalar and superscalar microprocessors use highly sophisticated pipelining[1, 2]. Pipeline complexity increases the number of possible design errors, and also makes design verification more difficult. It is difficult for formal verification methods proposed to handle the entire designs of today's complex pipelined processors.

Microprocessors are verified by running test programs through a logic simulators for an implementation and for a specification, and comparing the results. Since exhaustive logic simulator is impossible, it is very important to generate such test programs that is executable in terms of length and reliable in terms of verification coverage.

Some papers have presented test program generation methods for pipelined processor verification[3, 4, 5]. These methods focus on pipeline hazards[1] and can automatically generate effective test programs for target cases. Pipeline behavior when and after a hazard is detected is not considered, so these methods cannot cover cases reachable only after a hazard has occured. Moreover, they cannot avoid unexpected hazards that prevent reaching the target cases. This paper presents a new approach to generating test programs for any processor state.

2 Microprocessor verification

First, we compare characteristics of simulation-based verification and formal verification, and then consider how to incorporate formal verification features into simulation-based verification.

2.1 Simulation-based verification against formal verification

Successful works have been done in validating logic circuits using formal verifiers. Much academic attention has recently been directed to formal verification for microprocessors[6, 7, 8]. The methods proposed, however, are still weak in manipulating large circuits and have difficulty in handling the entire designs of a today's complex pipelined processors.

Even if formal verification becomes applicable to actual microprocessors, we prefer logic simulation to formal verification in an early design stage with many design errors. Verification and debugging can be repeated in a short cycle by logic simulation without changing test programs. Formal verification is very expensive for repeated use, since formal verification for a large design generally needs an abstracted version of the implementation to reduce complexity. When a design error is found in a original implementation, it is only required for simulation-based verification to modify the implementation, while we also have to modify the abstracted verification for formal verification.

Both formal verification and simulation-based verification sacrifice completeness to reduce computation complexity. In formal verification, complexity is reduced by focusing on a target mechanism to simplify the implementation. Logic simulation, in a sense, also reduces the complexity by restricting input sequences, while it can handle

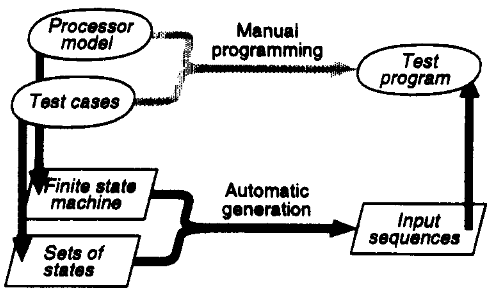

Figure 1: Automatic test program generation

a large scale of microprocessor design as it is. We have to make clear what a test program verifies and what it neglects to choose a verification method properly.

2.2 Simulation-based verification criteria

We make two assumptions for considering simulation-based verification criteria.

- Independent processor mechanisms can be verified separately.
- A design error in a processor mechanism is detected by a large percentage of those input sequences that activate the mechanism.

Many formal verification techniques also make the first assumption. The second is introduced for simulation-based verification. It is valid when the processor mechanisms are classified precisely enough considering the remaining errors. On these assumptions, the most important factor for reliable verification is to classify processor mechanisms properly and activate all of them with test programs.

2.3 Test program generation

A situation examined by a test program is called a *test case*. Test program generation is a process for finding instruction sequences that cause the microprocessor to encounter the test cases. Designers who create test programs manually know how the processor works. They traces instruction flow in the processor and constructs instruction sequences for test cases.

We suggest a method to automate such manual jobs (Figure 1). The processor model is written as a finite state machine (FSM). The test cases are translated into sets of states on the FSM. Once these model translations are completed, various conventional techniques, especially techniques for formal verification, become applicable. A test program is composed of a series of input sequences that satisfy the test cases.

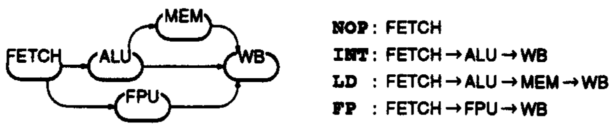

Figure 2: Processor P1

3 Test program generator for pipelined processors

We concentrated on verifying the pipeline control parts of processors and have developed an automatic test program generator. The modeling method and algorithm are shown in this section.

3.1 Test case

Pipeline control mechanisms are activated in cases of pipeline hazards, so the test cases for pipeline control parts are pipeline hazard situations. Although there are numerous hazard cases for an actual microprocessor, they can be enumerated automatically from a simple description that contains pipelining information for each instruction. We have presented the method in previous papers[4, 5].

A hazard case can be represented by conflicting instructions and their locations immediately before causing the hazard. It does not matter what instructions are in the rest of the places, because they do not affect whether the target error appears or not. Therefore, a hazard case generally contains multiple states.

3.2 FSM for processor pipelines

The processor model can be simplified as long as it keeps the instruction flow information and it can represent the test cases. On verifying pipeline control parts, we do not have to care how operand data and result data are processed in the functional modules. Thus, we simplify processor hardware to a set of *pipeline units*. A pipeline unit corresponds to a hardware block which can hold an instruction for one clock cycle. A hardware block that produces results in one cycle, such as an integer ALU, is modeled as a pipeline unit, and a hardware block that produces results in n clock cycles, such as a FPU, is divided into n sub-blocks and modeled as a series of n pipeline units.

If two kinds of instructions behave equally from the standpoint of the verification, we treat them as the same instruction type. Register addresses are also taken into account for distinguishing between instruction types when we consider data hazards[1]. A pipeline unit that can hold k different instructions has $k + 1$ states, *i.e.*, k states when it holds an instruction, plus one state when it holds nothing. A state of the whole FSM is defined by states of all pipeline

```
Read an input description
  and construct state transition functions;
Enumerate test cases $T_1, \ldots, T_n$;
Let $C$ be the set of the initial state;
while ($C \neq \phi$) {
  foreach $T_i$ ($i = 1, \ldots, n$) {
    if ($C \cap T_i \neq \phi$) {
      Add a set of input sequences
        that satisfy $C \cap T_i$ to $S_i$;
      Exclude $C \cap T_i$ from $T_i$;
    }
  }
  Advance the time and update $C$ to the image of $C$;
  Exclude the states already enumerated from $C$;
}
foreach $S_i$ ($i = 1, \ldots, n$) {
  Choose user-specified number of input sequences
    randomly from $S_i$;
}
```

Figure 3: Basic procedure for automatic test program generation

units. If the unit i can hold k_i kinds of instructions, the FSM has $\prod(k_i + 1)$ states.

Figure 2 shows an example of a simple processor, P1. Its pipeline units are FETCH, ALU, FPU, MEM, and WB. P1 has four types of instructions: `NOP`, `INT`, `LD`, and `FP`. Data hazards are not considered in this example. FETCH can hold four kinds of instructions, ALU can hold two kinds, FPU and MEM can each hold one kind, and WB can hold three kinds. Thus, the FSM for P1 has $5 \times 3 \times 2 \times 2 \times 4 = 240$ states.

3.3 Automatic generation algorithm

The FSM is represented by Boolean state variables and state transition functions. We use reduced ordered binary decision diagrams (*ROBDD*'s)[9] to represent functions and sets.

Our basic procedure is shown briefly in Figure 3. The initial state is a empty pipeline, or the state after executing a long `NOP` sequence. The procedure enumerates reachable states from the initial state. Efficient state enumeration algorithms have been proposed for formal verification[10]. When one or more reachable states are included in a test case, a set of input sequences that satisfy the states is calculated. The set is calculated by recursive substitution of the state transition functions.

The techniques required are already common in formal verification. However, this test program generator can work

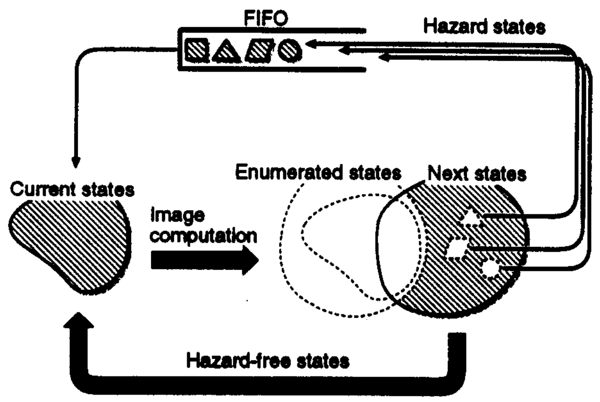

Figure 4: Hazard-free-first state enumeration

for larger microprocessor designs than formal verifiers have done. While formal verifiers handle an FSM that corresponds to the implementation, the FSM for the test program generator can be much more simplified as long as it keeps the instruction flow information and it can represent the test cases.

We reduced memory requirement for the basic procedure by optimizing the state enumeration order. State enumeration from pipeline hazard states is not performed until after that from hazard-free states ends. We named this the hazard-free-first procedure (Figure 4). Since state transitions from hazard-free states are simple for a pipelined processor, the BDD size can be kept small by using the hazard-free-first procedure.

4 Experimental results

The test program generator is written in Perl and runs on a special Perl interpreter linked with a BDD package writ-

Table 1: Execution summary of basic/hazard-free-first procedures

	P1	P2	P3
Pipeline units	5	9	11
Instructions per cycle	1	1	2
FSM states	240	190512	9.335×10^6
Test cases	12	61	497
Reachable FSM states	125	16747	1.851×10^6
— only after hazards	28	7236	1.244×10^5
Reachable test cases	8	25	285
— only after hazards	3	9	0
Test program length	27/ 27	127/ 127	2289/2516
CPU time (seconds)	10/ 13	83/ 90	495/ 579
Max. BDD nodes	2K/599	37K/2658	117K/5797

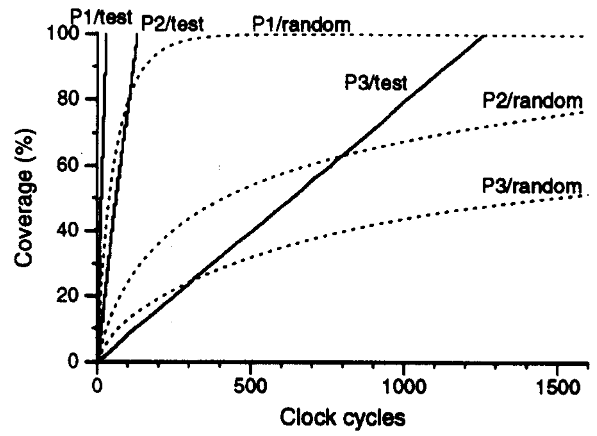

Figure 5: Test coverage by test programs and random instructions

ten in C. Execution results for three pipelined microprocessors P1, P2, and P3 are summarized in Table 1. To construct these test programs, we generated one test sequence for each reachable test case. CPU times are measured on a SPARCstation2.

The test program generator enumerated all reachable pipeline states and distinguished reachable test cases from unreachable ones. It is difficult to analyze test cases manually, and impossible for conventional test program generation methods to distinguish them. The test programs generated by these procedures covered test cases that are reachable only after hazards. These are difficult cases to handle manually, and cannot be covered by conventional test program generation methods.

Results show that computations completed in reasonable CPU/memory requirements, and also show that the hazard-free-first procedure is comparable to the basic procedure in CPU time, and superior in memory requirements.

The system can also analyze reachability of test cases. The number of test cases expected to be covered by random instructions in each clock cycle is calculated by the system. Percentages of reachable test cases covered by the test programs and random instructions are plotted in Figure 5. About 360 clock cycles of random simulation is needed for P1 to guarantee 99% coverage, 9,600 cycles for P2, and 90,000 cycles for P3. Our test programs are 13 to 76 times smaller than 99% coverage random instructions.

5 Conclusion

We have demonstrated the necessity of an effective automatic test program generator and presented our realization of it for pipelined processors.

We compared characteristics of simulation-based verification and formal verification, and then considered how to incorporate formal verification features into simulation-based verification.

A automatic test program generator for pipelined processors is implemented by utilizing techniques developed for formal verification. We also presented the basic procedure and the improved procedure to reduce memory requirement called the hazard-free-first procedure.

Our method can generate test programs that are difficult to code by hand and impossible to generate with conventional test program generation methods. Experimental results have shown that the hazard-free-first procedure is much more efficient in memory usage than the basic procedure. Results also demonstrated that while random simulation needs a large number of clock cycles to achieve high test coverage, our test programs can achieve perfect test coverage in a small number of clock cycles.

Our automatic test program generation system becomes more powerful if combined better modeling methods for processors and test cases.

References

[1] J. L. Hennessy and D. A. Patterson, *Computer Architecture: A Quantitative Approach*, Morgan Kaufmann, 1990.

[2] M. Johnson, *Superscalar Microprocessor Design*, Prentice Hall, 1991.

[3] D. C. Lee and D. P. Siewiorek, "Functional Test Generation for Pipelined Computer Implementations," *FTCS21*, pp. 60–67, 1991.

[4] H. Iwashita, T. Nakata, and F. Hirose, "Behavioral Design and Test Assistance for Pipelined Processors," *IEEE The First Asian Test Symposium*, pp. 8–13, 1992.

[5] H. Iwashita, T. Nakata, and F. Hirose, "Integrated Design and Test Assistance for Pipeline Controllers," *IEICE Transactions on Information and Systems*, Vol. E76-D, No. 7, pp. 747–754, 1993.

[6] J. R. Burch and D. L. Dill, "Automatic Verification of Pipelined Microprocessor Control," *Proc. Conf. on Computer-Aided Verification*, pp. 68–80, 1994.

[7] D. L. Beatty and R. E. Bryant, "Formally Verifying a Microprocessor Using a Simulation Methodology," *Proc. 31st DAC*, pp. 596–602, 1994.

[8] V. Bhagwati and S. Devadas, "Automatic Verification of Pipelined Microprocessors," *Proc. 31st DAC*, pp. 603–608, 1994.

[9] R. E. Bryant, "Graph Based Algorithm for Boolean Function Manipulation," *IEEE Transactions on Computers*, C-35(8), pp. 677–691, 1986.

[10] H. J. Touati, H. Savoj, B. Lin, R. K. Brayton, A. Sangiovanni-Vincentelli, "Implicit State Enumeration of Finite State Machines using BDD's", *ICCAD-90*, pp. 130–133, 1990.