Coverage-Directed Verification of Microprocessor Units Based on Cycle-Accurate Contract Specifications

Alexander Kamkin
Software Engineering Department
Institute for System Programming of Russian Academy of Sciences
25, B. Kommunisticheskaya, Moscow, 109004, Russia
E-mail: kamkin@ispras.ru

Abstract

In this paper we describe a method for simulation-based verification of microprocessor units based on cycle-accurate contract specifications. Such specifications describe behavior of a unit in the form of preconditions and postconditions of microoperations. Test sequence generation is based on traversal of FSM constructed automatically from specifications and test coverage definition. We have successfully applied the method to several units of the industrial MIPS64-compatible microprocessor.

1. Introduction

Microprocessors play an important role in present-day life. They underlie all digital computer systems including safety-critical ones, such as airplanes control systems, medical systems of life support, etc. The most commonly used way to ensure functional correctness of a microprocessor is a *simulation-based verification* of its register-transfer-level (RTL) model [1]. High cost of a bug, increasing design complexity, and shrinking time to market make functional verification a key component of the microprocessor development cycle.

To improve the efficiency of verification it is performed not only for a full-chip model, but for individual units as well. Unit-level verification gives the following key advantages. First and foremost, it provides observability and controllability that is lacking at the chip level. Second, it allows to make early verification of a microprocessor even before a full-chip model is available [2]. It also should be emphasized that unit-level verification provides significant return in terms of quantity and quality of bugs removed [3].

Nowadays, it is next to impossible to develop high-quality tests for a microprocessor manually. Moreover, it is impracticable to develop tests for a sufficiently complex unit of a microprocessor. The need of automated testbench development technologies is widely recognized. Development of such technologies and supporting tools has separated to a special branch of electronic design automation (EDA) industry which is known as *testbench automation*.

In this work we consider a method of simulation-based verification of microprocessor units based on *cycle-accurate contract specifications*. Such specifications describe behavior of a unit in the form of *preconditions* and *postconditions* of *microoperations*. They allow to represent cycle-accurate requirements in comprehensible declarative form and to automate simulation-based verification.

The rest of the paper is organized as follows. The second section describes the suggested approach to formal specification of microprocessor units. In the third section application of cycle-accurate specifications to simulation-based verification is considered. In the fourth section related work is outlined. The fifth section describes our experience in functional verification of microprocessor units. Finally, the sixth section concludes the paper.

2. Specification of Microprocessor Units

Operations implemented by microprocessor units are generally multi-cycle ones, i.e., they are executed during several clock cycles. A part of an operation executed during one clock cycle is called a *microoperation*.

The suggested approach to testbench automation is based on *contract specifications* in the form of preconditions and postconditions. We propose to define contracts for separate microoperations and to obtain contract for the entire operation by means of *temporal composition of contracts*. Here under temporal composition we mean a special mechanism that in each cycle of simulation calculates a set of contracts to be fulfilled.

The process of specification of an operation can be outlined as follows. First, a precondition restricting situations in which the operation can be supplied for execution is determined. Based on the analysis of the requirements, functional decomposition of the operation into a set of microoperations is carried out. For each of them a postcondition describing corresponding requirements is defined. Then, the postcondition is associated with the cycle when it should be fulfilled. Thus, contract of the operation consisting of n microoperations is formalized by the structure:

$$Contract = \langle pre, \{(post_i, \tau_i)\}_{i=1,n}\rangle,$$

where pre is the precondition of the operation, $post_i$ is the postcondition of the i^{th} microoperation, and τ_i is the number of the cycle when the i^{th} microoperation is executed.

For the purpose of clearness hereinafter we assume that microoperations are executed sequentially, i.e., $\tau_1 = 1, \ldots, \tau_n = n$. This assumption does not restrict the generality. In this case contract of an operation consisting of n microoperations is given by the formula:

$$Contract = \langle pre, \{post_i\}_{i=1,n}\rangle.$$

Previously we have considered operations in which cycles when microoperations are executed were statically fixed. Some units of microprocessors are more complicated. In general case there are dependencies between operations. So, execution of an operation is suspended until all necessary data is prepared and all required resources are deallocated by the previous operations. Requirements on such operations are specified with the help of the following contracts:

$$Contract = \langle pre, \{(pre_i, post_i)\}_{i=1,n}\rangle,$$

where pre is the precondition of operation, pre_i is the *guard condition* of the i^{th} microoperation, and $post_i$ is the postcondition of the i^{th} microoperation. *Guard condition* is a negation of interlock condition, i.e., microoperation is interlocked, if and only if the corresponding guard condition is false.

3. Verification of Microprocessor Units

In this section we consider how cycle-accurate contract specifications can be applied for simulation-based verification of microprocessor units.

3.1. Checking Correspondence between Specification and Implementation

For operation x we introduce the following notation. The precondition of the operation is denoted as $pre(x)$; the guard condition of the i^{th} microoperation is denoted as $pre(x, i)$; the postcondition of the i^{th} microoperation is denoted as $post(x, i)$; and, finally, the number of microoperations in the operation is denoted as $L(x)$.

Suppose that at some moment of time the unit under verification performs m operations $x_1, \ldots, x_m$ (without interlocks) that have been supplied for execution $\tau_1, \ldots, \tau_m$ cycles earlier, respectively. Let at the moments when the operations were supplied, the preconditions $pre(x_1), \ldots, pre(x_m)$ have been fulfilled. Then, to verify correctness of behavior at the given moment of time it is required to check satisfiability of the predicate called *test oracle*:

$$Oracle(\{(x_i, \tau_i)\}_{i=1,m}) = \bigwedge post(x_i, \tau_i).$$

Test oracle organization is illustrated in **Fig. 1**. In the first cycle of simulation operation A is started. Then, one cycle later operation B is supplied for execution. At the end of the second cycle both postcondition of microoperation A_2 and postcondition of microoperation B_1 should be fulfilled.

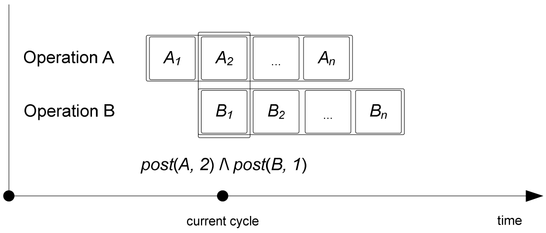

Fig 1. Execution of two operations.

To verify operations with interlocks, testbench should keep track which microoperations are finished and check corresponding postconditions. As it was said before, a special mechanism, called temporal composition of contracts, is used for this purpose. Formal description of the mechanism can be found in paper [4]. General idea is the following. Contract specifications of a unit are interpreted as extended finite state machine (EFSM) [4]. State of the EFSM is a set of pairs (x, l), where x is an identifier of operation (stimulus), and l is the number of microoperation (stage). Suppose that at some moment of time state of EFSM is $\pi = \{(x_i, l_i)\}_{i=1,m}$ and testbench is starting execution of operation x (guard condition and postcondition of the operation are fulfilled). Then, the state is changed in the following way:

$$\pi' = \begin{array}{ll} \{(x, 1)\} \cup & (1) \\ \{(x_i, l_i) \mid pre(x_i, l_i) = false\} \cup & (2) \\ \{(x_i, l_i + 1) \mid pre(x_i, l_i) = true \wedge l_i < L(x_i)\} & (3). \end{array}$$

Intuitive significance of this formula is quite simple: new pair $(x, 1)$ is added to the state (1); if microoperation is interlocked, then its stage is not changed (2); if microoperation is not interlocked (3), then its stage is increased. Test oracle for the transition is calculated as the conjunction of postconditions associated with executed microoperations:

$$Oracle(\pi) = \bigwedge \{post(x_i, l_i) \mid pre(x_i, l_i) = true\}.$$

3.2. Test Coverage Definition

In the suggested method test coverage for a unit is described with the help of *test situations* and *dependencies*. Test situation is a predicate that constrains arguments of an operation and a state of a unit. Test situation describes event that is interesting for testing (exception, cache miss, etc.). Dependency restricts arguments for a pair of operations. Usually dependencies have an influence on operation interlocks.

We use constructive definition of test coverage that allows testbench to automatically construct arguments of operations during test sequence generation: test situations and dependencies are supplied with special functions called *constructors* that totally or partially construct arguments of the corresponding operations. When applying operation for a given test situation and a given set of dependencies, testbench calculates arguments of the operation as a composition of values generated by corresponding constructors (test situation and dependencies should not contradict each other).

We develop constructors manually, but they can be generated automatically using constraint solving techniques. Currently we are doing research and development in this field and planning to implements a prototype of the test generation tool in the short run.

3.3. Test Sequence Generation

When test coverage of a unit is specified, the goal of the unit verification can be defined formally as covering all feasible *generalized states* of the unit EFSM. Generalized state is a set of elements $(x[s, d], l)$, where x is a stimulus, s is a test situation, d is a set of dependencies that connect x with previously launched operations (which are also presented in this state), and l is a stage. Stimulus x labeled by s and d is called *generalized stimulus*.

To achieve the goal of verification we use *irredundant algorithms* for traversing directed graphs [5]. Usually generalized state graphs are *deterministic* (assume that dependencies determine interlocks) and *strongly connected*, therefore we can apply such kind of algorithms. The important feature of irredundant algorithms is that they operate with graphs defined implicitly by function that for each node calculates a set of possible stimuli. Such approach goes very well with contract specifications (see **Fig. 2**).

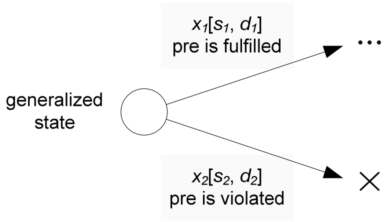

Fig 2. Generalized state graph.

It should be noticed that precondition satisfiability must be determined on the base of generalized state. To meet this requirement verification engineer ought to manually supplement state with additional information, if it is needed.

4. Related Work

There are a lot of articles dedicated to methods of microprocessor verification. Many researchers come to a consensus that model-based test generation is the right direction for simulation-based verification of microprocessors. The main question is which models and notations should be used.

Existing approaches utilize explicit cycle-accurate models to generate test sequences, e.g., Ur et al. [6] and Mishra et al. [7-10] use SMV models; Ho et al. [11] utilize Synchronous Murφ. The main differences between approaches are concentrated in the following methods:

- method of constructing a model:
  - manual development [6];
  - automatic derivation from RTL [11];
  - automatic derivation from specifications [8,*[1]].
- method of test sequence generation:
  - state graph traversal techniques [6,11];
  - model checking techniques [7-10,*].

It should be emphasized that manual development of a model for test sequence generation is error-prone, while automatic derivation from RTL description does

[1] * = this paper.

not scale well on complex designs. We think that the most perspective method of model construction is automatic extraction from formal specifications and test coverage definition.

Model checking techniques are not intended for full-scale functional verification. They are aimed to verify relatively small number of properties and to generate counter-examples if properties are violated. The most usable way of test sequence generation based on state graph traversal.

In the suggested method model for test sequence generation (generalized state graph) is automatically derived from specifications and test coverage definition; test sequence generation is based on state graph traversal techniques. The distinction feature of the approach is that it utilizes implicit (declarative) specifications for description of behavior and irredundant algorithms for state graph traversal. We believe that use of implicit models increases the scalability of the approach.

5. Case Studies

The suggested approach was applied to translation lookaside buffer (TLB) and to L2 cache of the industrial MIPS64-compatible microprocessor [12].

The RTL model of the TLB is implemented in Verilog. The source code of the model (without libraries) makes up to 3.5 KLOC. All functional requirements (about 100) have been formalized by cycle-accurate contract specifications; tests have been developed according to the suggested method. The volume of specifications and tests is about 3.5 KLOC in SeC language, which is a specification extension of C. The total labor costs of testbench development are about 2.5 man-months. We have found more than 10 errors in the TLB implementation including very critical ones.

The RTL model of the L2 cache is also developed in Verilog. The volume of the unit source code is about 3.0 KLOC. In this project we have specified about 170 functional requirements. The volume of specifications and tests makes up to 4.7 KLOC in SeC. The labor costs of testbench development are about 4.0 man-months. Testbench has discovered 3 errors.

6. Conclusion

The need of automated testbench development for microprocessor units is widely recognized. The paper described the method of coverage-directed verification of microprocessor units that is based on cycle-accurate contract specifications. We have successfully applied our approach to several units of the industrial MIPS64-compatible microprocessor. Critical bugs were found in the implementation of the units that had not been discovered earlier at the chip level.

We consider the method to be very useful for functional verification of microprocessors at the unit-level. Now we are planning to generalize our approach for branching pipelines, pipelines with cycles, etc. The other thing that we are going to do is the development of tools that improve test result analysis and simplify design of specifications and tests.

7. References

[1] W. Lam. *Hardware Design Verification: Simulation and Formal Method-Based Approaches*. Prentice Hall, 2005.

[2] B. Bentley. “Validating the Intel Pentium 4 Microprocessor”. *In Proc. of Design Automation Conference*, 2001. pp. 244–248.

[3] J.M. Ludden et al. “Functional Verification of the POWER4 Microprocessor and POWER4 Multiprocessor Systems”. *IBM Journal of Research and Development*, Volume 46, Number 1, 2002. pp.53–76.

[4] A. Kamkin. “Testbench Automation for Pipelined Designs Based on Contract Specifications”. *In Proc. of East-West Design & Test Symposium*, 2007. pp. 348–353.

[5] I. Bourdonov, A. Kossatchev, V. Kuliamin. “Irredundant Algorithms for Traversing Directed Graphs: The Deterministic Case”. *Programming and Computer Software*, Volume 29, Number 5, 2003. pp. 245–258.

[6] S. Ur and Y. Yadin. “Micro Architecture Coverage Directed Generation of Test Programs”. *In Proc. of Design and Automation Conference*, 1999. pp. 175–180.

[7] P. Mishra and N. Dutt. “Automatic Functional Test Program Generation for Pipelined Processors using Model Checking”. *In Proc. of High-Level Design Validation and Test Workshop*, 2002. pp. 99–103.

[8] P. Mishra and N. Dutt. “Architecture Description Language Driven Functional Test Program Generation for Microprocessors using SMV”. *CECS Technical Report 02-26*, September 13, 2002. 18 p.

[9] P. Mishra and N. Dutt. “Graph-Based Functional Test Program Generation for Pipelined Processors”. *In Proc. of Design, Automation and Test in Europe*, 2004. pp. 182–187.

[10] P. Mishra and N. Dutt. “Functional Coverage Driven Test Generation for Validation of Pipelined Processors”. *In Proc of Design, Automation and Test in Europe*, 2005. Volume 2, pp. 678–683.

[11] R. Ho, C. Yang, M. Horowitz, and D. Dill. “Architecture Validation for Processors”. *In Proc. of International Symposium on Computer Architecture*, 1995. pp. 404–413.

[12] *MIPS64™ Architecture For Programmers*. Revision 2.0. MIPS Tecnologies Inc., June 9, 2003.