Combinatorial Model-Based Test Program Generation for Microprocessors

Alexander Kamkin
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 of automated test program generation intended for systematic functional verification of microprocessors. The method supplements such widely-spread practical approaches as software-based verification and random generation. In our method, construction of test programs is based on microprocessor model, which includes structural model and instruction set model. The goal of generation is defined by means of instruction-level test coverage. Test programs are constructed by combining test situations for different sequences of instructions.

1. Introduction

Increasing complexity of microprocessors and shrinking time-to-market cause functional verification to be a key component of the microprocessor design cycle. The need of automated verification methods is widely recognized by the semiconductor industry. While significant progress has been made through the use of formal methods, these methods are still limited to the verification of relatively small blocks, or to very focused verification goals [1]. In core-level verification of microprocessors simulation-based techniques still play a major role.

Verification of microprocessors on a core level is usually performed with the help of test programs. Test programs can be produced by different ways. For example, they can be made by cross-compilation of the existing software. The other widely-distributed technique of test program construction is random generation. However, such kind of methods are not systematic and do not guarantee the high quality of verification.

Analysis of errors in MIPS R4000 PC/SC microprocessor shows that most of the bugs (56.5%) are related to multiple interactions between subsystems of microprocessor [2, 3]. To uncover such bugs it is necessary to fulfill multiple constraints on different instructions of test program simultaneously. This type of error is hard to find using hand-written tests because there is a huge number of multiple interaction cases. Random programs might find such bugs, but each of the

constraints is so improbable that finding an error that occurs at the conjunction of these constraints requires a prohibitively large number of simulation cycles [3].

The method suggested in this paper is directed against multiple interaction bugs. It should be emphasized that our method does not replace any other approach. It has its own niche and can be considered as an addition to existing techniques. In compliance with the method we have developed the test program generator, called TestFusion 4M. It is a model-based test program generator which uses instruction-level test coverage and combinatorial techniques to construct varied sequences of instructions. The generator has been successfully used in a number of projects. This paper describes two of them.

The rest of the paper is organized as follows. The second section reviews existing approaches to test program generation. In the third section the suggested method is described. This section briefly considers organization of microprocessor model and structure of test coverage. In the forth section directions of future research are outlined. They touch upon the issues of automatic derivation of test coverage from microprocessor model, paying attention to the model-based construction of test templates. The fifth section comprises two case studies. Finally, the sixth section concludes the paper.

2. Methods of Test Program Generation

Nowadays there are several widespread methods of test program construction:

- manual development of test programs;
- cross-compilation of existing software;
- random generation of test programs;
- template-driven generation of test programs.

Manual development of test programs is a popular approach for testing corner case situations. To create such kind of tests verification engineer should know details of microprocessor under verification. Evidently, this method is very low-productive. The other big problem of the approach is that verification engineer might overlook situation that is important for verification [4].

The other method in general use is software-based verification. It is always performed for verification of general purpose microprocessors. At least, microprocessors are verified on one or several operating systems. It should be noticed that tests based on existing software do not guarantee the high quality of verification. This type of tests extensively covers functionality of microprocessors, but not deep enough.

The simplest method of automated test program construction is random generation. This method allows to quickly discover relatively simple bugs. The other advantage of random generation is that it may create situations which are difficult to be imagined, but are interesting for testing [4]. Fully random generation is almost useless for producing corner cases and discovering multiple interaction errors.

In our opinion, the most perspective way of automated test program construction is template-driven generation. Test template is an abstract representation of test program or its fragment, which, first, fixes or restricts sequence of instructions, and second, constrains values of operands. To construct test program generator finds a random solution of the corresponding set of constraints. The use of test templates reduces labor costs of verification. However, if test templates are developed manually, there is a possibility to miss important test cases.

2.1. Related Work

The IBM company has been using automated test program generators since 1980s [5]. The need in a general method, which is suitable for wide class of microprocessor architectures, has led the company to a model-based approach. In this approach generator is split into two main components: engine (which is independent from microprocessor) and model (which describes target architecture). By now IBM has developed template-driven generator Genesys-Pro [5]. The advantages of Genesys-Pro are expressive language for test template description and convenient framework for microprocessor modeling.

An interesting method for verification of pipelined microprocessors is proposed by P. Mishra (University of Florida) and N. Dutt (Center for Embedded Computer Systems, University of California) [6]. The authors use EXPRESSION language to describe microprocessor pipeline [7]. EXPRESSION description is translated into SMV model [8]. Verification engineer specifies a set of temporal properties which describe different situations in pipeline operational behavior (bypasses, control transfers, etc.). SMV tries to create counter-examples for the negations of the specified properties using model-checking techniques. Created counter-examples are mapped into test programs. The important feature of the approach is purposefulness (one test program covers one property). However, the suggested methodology does not scale well on complex industrial designs.

S. Ur and Y. Yadin (IBM Haifa Research Lab) propose a test generation method based on finite state machine (FSM) traversal [9]. The idea is in the following. Verification engineer manually develops SMV model of microprocessor. CFSM tool constructs set of paths (abstract tests) that cover all transitions of FSM derived from the model [10]. Abstract tests are translated into Genesys test templates. The method allows to achieve good coverage of control logic. There are two main drawbacks of the method. First, skilful expert is needed to develop SMV model of microprocessor. Second, to have the ability to map abstract tests into concrete ones, verification engineer should develop rather complex description in Genesys.

The other method based on FSM models is proposed by K. Kohno and N. Matsumoto (Toshiba) [11]. The researchers have implemented their method in mVpGen tool. The only input of the tool is pipeline specification. On the base of specification mVpGen automatically generates tests cases and FSM model of microprocessor. Test cases are states of FSM in which hazards between instructions are occurred. For each reachable test case a test template is constructed. Test

template is a path from the initial state of FSM to a state that corresponds to the test case. Finally, on the base of templates test programs are generated.

The completely different method, based on genetic algorithms, is proposed by F. Corno, G. Cumani, et al. (Politecnico di Torino) [12]. Generation uses a library of instructions, which describes assembler syntax of microprocessor under verification. Test program is represented as directed acyclic graph (DAG). Each DAG's node corresponds to program's instruction. It contains a reference to the instruction description in the library and, if it is necessary, values of operands. Test program is constructed by mutation of graph structure and values of operands inside the individual nodes. The method is rather flexible and multipurpose. It allows to achieve high level of test coverage for different metrics, but generation time can be considerable.

2.2. Niche of the Combinatorial Model-Based Methods

Summing up, many researchers come to consensus that model-based generation of test programs gives many advantages. The main question is what kind of models should be used. A number of papers are dedicated to test program generation on the base of cycle-accurate models. This type of methods are intended for increasing test coverage achieved by existing tests [13, 14] and for generation of tests directed to very specific corner cases [6].

Let us note some problems that appear when cycle-accurate models are used. There are two main ways to obtain a model – automatic derivation from register-transfer-level (RTL) model [2, 4, 13, 14] and manual development [9, 11]. Automatic derivation is a very complex task. Researchers recognize two approaches to do it – code annotation [2] and heuristics [13, 14]. Fully automatic derivation of model for complex microprocessors is next to impossible. Manual development confronts with the other problem – model should be debugged [9]. It should be also emphasized that it is very difficult to use cycle-accurate models at early stages of the microprocessor design cycle, because they are not clearly defined.

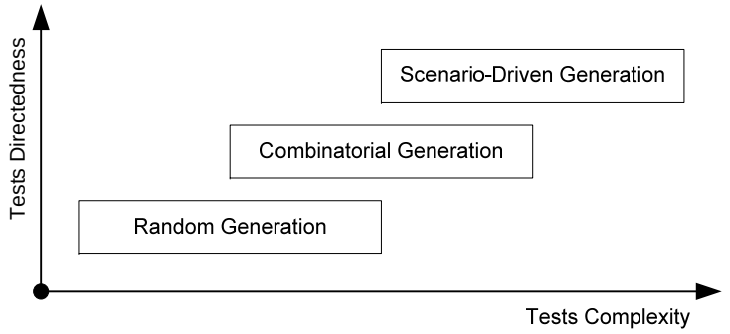

Fig. 1. Niche of Combinatorial Model-Based Methods.

In this paper we consider test program generation based on instruction-level models. Such kind of models are a basis for generation of a great bulk of tests for microprocessors. Instruction-level generation of test programs has two poles – fully random generation and scenario-driven generation. The first of them is a low-budget method that allows to generate ample quantity of undirected tests. The second one is a more complicated method. It is a flavor of template-driven generation where complex purposeful templates, called scenarios, are used. This type of generation is supported in industrial tools, like Genesys-Pro [5].

Communication with verification engineers points that there is a gap between random generation and scenario-driven generation of test programs. We think that this gap can be filled by combinatorial methods (see Fig. 1). Such methods use instruction-level test coverage and combinatorial techniques for automatic construction of test templates. Combinatorially generated test templates are not as complex and sensible as manually developed scenarios, but they are generated automatically in a systematic way. Our experience has shown that this make it possible to discover additional bugs which can be omitted by random generation and scenario-driven generation.

3. Description of the Method

The idea of the suggested method is based on the assumption that operational behavior of microprocessor depends on set of executing instructions (pipeline state), dependencies between them (via registers or memory), and situations (events) appearing when instructions are executed (exceptions, cache hits/misses, etc.).

3.1. Method Conceptions

In this section we consider the main conceptions of the suggested method: *test template*, *dependency*, *test situation*, and *test action*. First of all, let us consider a structure of generated test programs. It can be described by formula $\pi = \pi_{start} \cdot \{\langle \pi_i, x_i[s_i, d_i]\rangle\}_{i=1,n} \cdot \pi_{stop}$, where:

- π_{start} – *initialization program*
  is a prefix of test program that consists of instructions aimed for microprocessor initialization;
- $\langle \pi_i, x_i[s_i, d_i]\rangle$ – *test case* (i = 1, …, n):
  - π_i – *program of test action preparation*
    is a sequence of instructions that initializes registers and memory of microprocessor;
  - $x_i[s_i, d_i]$ – *test action*
    is a specially prepared sequence of instructions to be applied on microprocessor, where s_i is a *set of test situations*, and d_i is a *set of dependencies*;
- π_{stop} – *finalization program*

  is a postfix of test program that consists of instructions aimed for microprocessor finalization;
- n – *size of test program*
  is a number of test actions within test program.

The key notion of the method is *test action*. Test action is described by *test template* (which is a sequence of instructions to be applied on microprocessor without concrete values of operands), *set of dependencies* (which define how operands of different instructions are connected to each other), and *test situations* (which constrain values of operands and state of microprocessor). The goal of generation is systematic enumeration of test templates, dependencies and test situations.

Test Template. Test templates are sequences of instructions without concrete values of operands. They are intended for creation of different states of microprocessor pipeline. For each test template the generator creates a set of test actions, which are distinguished by dependencies and test situations.

Analysis of errata shows that a lot of bugs (of course not all of them) can be discovered by short sequences of instructions (2–5 instructions). Therefore we suggest using relatively short test templates for combinatorial generation of test programs. It is obvious that even if short test templates are used, their total number can be significant. To reduce number of test templates special heuristics should be used. For example, similar instructions can be unified into equivalence classes. If two test templates contain equivalent instructions at the same positions, they are considered to be equivalent. Equivalence class of test templates is called *generalized test template*.

We use enumeration of all generalized test templates of bounded length to perform combinatorial generation of test programs. For instance, we often use pairs or triples of instructions. However, it should be emphasized that TestFusion 4M supports other methods of combining instructions, called *combinators*. It also implements decomposition of test templates into weakly connected sections, which are enumerated independently by their own combinators.

Dependency. The usage of different test templates is not sufficient for thorough verification of microprocessor. Microprocessor can execute instructions in different ways depending on dependencies between them. In the suggested approach two types of dependencies are used – *register dependencies* and *address dependencies*. Register dependencies are expressed by equalities and non-equalities of registers in different instructions of test action. Address dependencies are closely connected to a memory hierarchy of microprocessor. Here are some examples of address dependencies: equality of virtual addresses, equality of physical addresses, equality of virtual page numbers, equality of cache rows, etc.

Let us illustrate address dependency by the example of RM7000 microprocessor [15]. Consider dependency which is equality of L1 rows accessed by two load/store instructions. We denote this dependency by `L1RowEqual`. Physical address of RM7000 consists of 36 bits. Bits [11:5] are used for indexing one of the 128 rows.

Within each row there are four 64-bit doublewords of data. Bits [4:3] are used to index one of these four doublewords. Bits [2:0] are used for indexing one of the eight bytes within each doubleword. Bits [35:12] contain tag. So, two pieces of data are mapped into the same L1 row if and only if bits [11:5] of their addresses are equal.

In general case dependency (family of homogeneous dependencies) between instructions is described by set of attributes. Concrete values of attributes fix dependency of the given type. Apart from attributes description of dependency includes the following components:

- *iterator* – enumerates all feasible combinations of attribute values;
- *precondition* – checks admissibility of using dependencies of the given type for given pair of operands;
- *constructor* – constructs (totally or partially) value of dependent operand basing on values of operands from which it depends via dependencies of the given type.

Consider dependency `L1RowEqual`. This dependency is described by one Boolean attribute. Iterator enumerates values `{true, false}` in some order. Precondition of the dependency checks if virtual addresses can be translated into physical ones (otherwise, when, for example, invalid addresses are used, cache memory is not accessed). If there is a dependency which attribute has value `true`, then constructor copies bits [11:5] from the determinant operand into the dependent one; if all dependencies are `false`, then constructor generates random value of the bits which is different from the used ones.

Test Situation. Generally instructions behave differently depending on values of operands and state of microprocessor. For example, an instruction that may cause an exception has two alternative ways of execution: normal execution and exceptional execution. In this article we use term *test situation* to refer to one of a number of variants of instruction execution. Formally, test situation is a constraint on values of operands and state of microprocessor. Test situations are derived from functional description of instruction set. Consider the description of instruction `add` from MIPS64 manual [16]:

```
if NotWordValue(GPR[rs]) or NotWordValue(GPR[rt]) then
    UNPREDICTABLE
endif
temp ← GPR[rs]31||GPR[rs]31..0)+(GPR[rt]31||GPR[rt]31..0)
if temp32 ≠ temp31 then
    SignalException(IntegerOverflow)
else
    GPR[rd] ← sign_extend(temp31..0)
endif
```

The first branch corresponds to the negation of the instruction's precondition. In this example both registers `rs` and `rt` must be initialized by sign-extended 32-bit

words. Predicate $\texttt{temp}_{32} \neq \texttt{temp}_{31}$ determines the test situation corresponding to the integer overflow exception; the complementary situation corresponds to the normal execution of the instruction.

Like dependency, test situation (family of homogeneous test situations) is described by set of attributes. Concrete values of attributes fix test situation of the given type. Apart from attributes description of test situation includes the following components:

- *iterator* – enumerates all feasible combinations of attribute values;
- *constructor* – constructs values of the operands;
- *preparator* – builds program of test situation preparation, which initializes instruction operands and state of microprocessor.

Consider instruction `add`. Test situations for this instruction (on some level of abstraction) are parameterized by Boolean attribute `IntegerOverflow`. If the attribute is `true`, it means that values of the operands should cause the integer overflow exception; otherwise the exception should not be caused. Constructor generates random values of independent operands, such that overflow condition is satisfied if and only if `IntegerOverflow` is `true`. Preparator for each independent operand appends to preparation program instructions which load the constructed value into the corresponding register.

Test Action. Test action is a sequence of instructions with given values of operands, and which execution is started in a given state of microprocessor. Test actions are said to be equivalent if their test templates are equivalent and they have equivalent dependencies and test situations. An equivalence class of test actions is called *generalized test action.*

The goal of generation is construction of all feasible generalized test actions. To achieve this goal the generator solves two tasks: enumeration of generalized test actions and construction of test actions. On each step of enumeration the generator formulates a set of constraints on a test action. These constraints are solved on the construction phase. Enumeration of generalized test actions is done with the help of iterators. Construction of test actions is implemented by constructors and preparators.

3.2. Model of Microprocessor

The suggested method of test program generation is based on microprocessor model. Model contains static and dynamic information on microprocessor under verification. Static constituent includes structure of microprocessor's subsystems, descriptions of instructions, and other information which is usually known as architecture. Dynamic part of the model is an abstract representation of microprocessor state. The generator interprets instructions appended to a test program by changing the model state. This allows to control instructions' preconditions and to correctly prepare test situations. It should be emphasized that

model-based generation allows to create self-checking test programs, which contain build-in checks of the microprocessor state.

A key part of the model is a description of the microprocessor instruction set. Description of an individual instruction includes the following components:

- *instruction interface* – describes the operands of the instruction. Definition of each operand contains name, type (immediate or register), data type (word, floating-point number, etc.), and data flow direction (input, output, or inout);
- *instruction precondition* – defines situations when execution of the instruction is predictable;
- *function of instruction execution* – calculates values of the output operands of the instruction and updates model state of the microprocessor;
- *assembler format* – specifies assembler format of the instruction.

3.3. Generator TestFusion 4M

The TestFusion 4M generator takes microprocessor model, description of test coverage (test situations and dependencies), and generation parameters as input. Some instructions are marked as being under verification – these instructions are used for making test actions.

Test program generation is carried out as follows. Test templates, dependencies, and test situations are enumerated. First of all, the generator allocates registers according to the register dependencies. Then, for each instruction of the test action it constructs address dependencies and test situations. After that, it creates a preparation program for the instruction. From preparation programs of all instructions of the test action the generator constructs an aggregate preparation program. Process continues until all generalized test actions are enumerated. Here is the simplistic scheme of the generation:

- get the next test template
    - get the next set of register dependencies
    - construct the register dependencies
    - get the next set of test situations*
        - get the next set of address dependencies
        - for each instruction of the test action:
            - check precondition:
              if the precondition is failed, then go to *
            - check existence of address dependencies:
              if address dependencies exist, then construct them
            - construct test situation of the instruction
            - get preparation program of the instruction
        - construct preparation program of the test action
        - interpret the preparation program
        - interpret the test action

At first sight, construction of aggregate preparation program is a trivial task – it is sufficient to concatenate preparation programs of all instructions of test action. Commonly this method works, but not always. There are situations when preparation program of an instruction has influence on previous instructions. In such situations, verification engineer ought to divide preparation programs of instructions into several fragments. Each fragment is responsible for initialization of a certain subsystem of microprocessor. Verification engineer sorts fragments in such an order that each successive fragment does not affect subsystems which are initialized by the previous ones (this work can be automated). To have the ability to perform such ranking, graph which reflects influence that initialization of one subsystem has on states of the others should be acyclic.

The TestFusion 4M generator has the flexible architecture which is compliant with the method concepts. The main components of the generator are responsible for enumeration of test templates, dependencies, and test situations. These components form a generator core. User can tune the generator core by selecting proper values of parameters. Apart from the core components the generator has a number of libraries which simplify development of microprocessor models and also include many ready-to-use components, like combinators, test data generators, etc. To increase the ease of using TestFusion 4M it has graphical user interface (see Fig. 2).

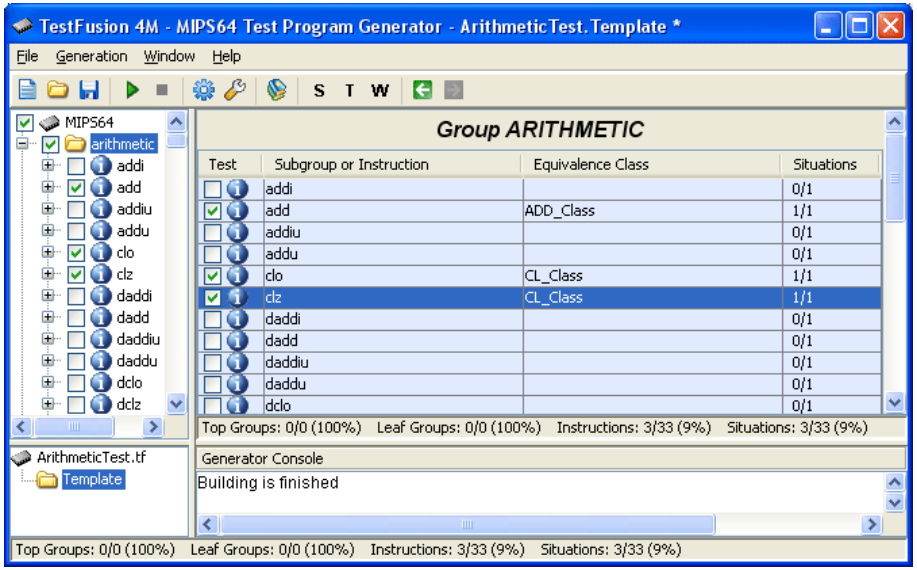

Fig. 2. Graphical User Interface of TestFusion 4M.

4. Future Research

This section outlines several ideas we are planning to work with in the nearest future. The ideas touch upon the issues of automatic derivation of test coverage (test situations and dependencies) from microprocessor model and also have to do with model-based construction of test templates. Microprocessor model comprises descriptions of instructions and descriptions of subsystems. Since the descriptions are formal they can be used for automatic extraction of test situations and dependencies between instructions.

Semantics of a particular instruction can be represented by an *execution tree*. Nodes of the execution tree correspond to certain subsystems of the microprocessor; edges describe transfers of control. Each edge contains a predicate which defines a condition for the corresponding transition (see Fig. 3). Execution trees can be used for factorization of instructions. For example, two instructions are considered to be equivalent, if they have the same execution tree or the same prefix of execution trees. Such factorization can reduce the total amount of generated test templates. What is even more important is that execution trees can be used for automatic extraction of test situations. Each situation corresponds to a path (branch) in the execution tree of the instruction. For example, fragment of the execution tree in Fig. 3 defines four test situations: `{TLB[Hit, Valid], L1[Hit]}`, `{TLB[Hit, Valid], L1[Miss]}`, `{TLB[Hit, Invalid]}` и `{TLB[Miss]}`.

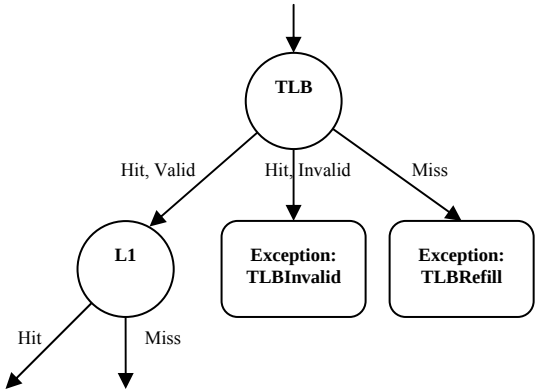

Fig. 3. Fragment of execution tree of load/store instructions.

To have the ability to extract dependencies between instructions, descriptions of subsystems should include all necessary information, like resource access type (direct-mapped, set-associative, associative, etc.), number of elements, structure of elements, function of tag calculation (for buffers), etc. For example, description of TLB might look like this:


```
associative buffer TLB<TLB_ENTRY, 64> {
VIRTUAL_PAGE_NUMBER tag(VIRTUAL_ADDRESS va) { ... }
}

structure TLB_ENTRY {
 VIRTUAL_PAGE_NUMBER  vpn;
  PHYSICAL_PAGE_NUMBER pfn;
}
```

From this description one can automatically derive dependency `TLB_ENTRY_EQUAL` which describes access to the same TLB entry from two different load/store instructions.

Consider test template construction. One of the possible scenarios can be the following. Verification engineer selects subsystems he or she wants to verify. The generator analyzes the execution trees of the instructions and determines which of them use the selected subsystems. Then, it factorizes these instructions, if necessary, and extracts test situations and dependencies for them. Finally, it starts the generation of test templates. The main question here is how to construct test templates for the given set of instructions. One of the promising approaches is based on FSM traversal.

Since we use high-level models it is impossible to extract cycle-accurate FSM of the microprocessor. However, we can construct *abstract FSM model* by adding special attributes to the execution trees' nodes (execution time, capability of concurrent processing of instructions, etc.). Using these attributes we can tune time aspects of instructions execution. FSM states can be of the type $\{(branch_i, node_i, time_i)\}_{i=1,n}$, where $branch_i$ is a branch in the execution tree, $node_i$ is a current node, and $time_i$ is a time of instruction processing in the current node (see Fig. 4).

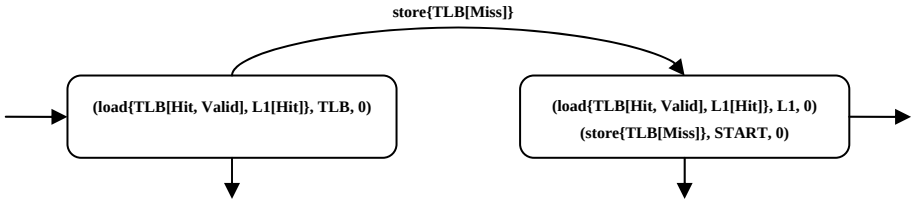

Fig. 4. Fragment of abstract FSM of pipeline.

The goal of generation is to traverse the abstract FSM. On each step of construction the generator chooses an instruction to be added into the test template, test situation (branch of the instruction execution), and dependencies between this instruction and currently executing instructions (which are presented in the state). To reduce the total size of constructed test templates the generator can use special heuristics. For example, it can restrict the number of dependencies between instructions.

The other promising direction of future work is integration into TestFusion 4M some facilities of OTK [17] and Pinery [18] generators, which are aimed for complex test data generation, and like TestFusion 4M, are developed at Institute for System Programming of RAS. Particularly, such facilities can be used for generation of various control flow and data flow graphs of test programs.

5. Case Studies

The TestFusion 4M generator has been used in two industrial projects: verification of memory management unit (MMU) of MIPS64-compatible microprocessor and core-level verification of the other MIPS64-compatible microprocessor. It should be emphasized that in both projects apparent from combinatorial generation we used different verification methods, like random generation, scenario-driven generation, and manual test development. Test programs generated by TestFusion 4M found additional bugs which were omitted by other verification techniques.

5.1. Verification of Memory Management Unit

Test actions on MMU were organized as pairs of load/store instructions: `lb` (load byte), `ld` (load double-word), `sb` (store byte), and `sd` (store double-word). Test situations for the instructions were parameterized by the following attributes:

- `isMapped` – mapped/unmapped virtual address space;
- `isCached` – cached/uncached virtual address space;
- `tlbHit` – TLB hit/miss;
- `DVG` – control bits of TLB section[1];
- `dtlbHit` – DTLB[2] hit/miss;
- `cachePolicy` – cache policy;
- `l1Hit` – L1 hit/miss;
- `l2Hit` – L2 hit/miss.

Dependencies between instructions were described with the help of the following attributes:

- `vaEqual` – equality/inequality of virtual addresses;
- `tlbEqual` – equality/inequality of TLB entries;
- `pageEqual` – equality/inequality of TLB sections;
- `paEqual` – equality/inequality of physical addresses;
- `l1RowEqual` – equality/inequality of L1 rows;
- `l2RowEqual` – equality/inequality of L2 rows;

[1] In MIPS64 microprocessors TLB entry consists of two sections – the first section is for even virtual page and the second one is for odd virtual page.
[2] DTLB (Data TLB) is a small buffer that caches TLB entries for operations of data address translation.

- `dtlbReplace` – equality/inequality of TLB entry used by the second instruction with the TLB entry replaced from DTLB by the first instruction;
- `l1Replace` – equality/inequality of L1 tag of physical address used by the second instruction with the tag replaced from the L1 cache by the first instruction;
- `l2Replace` – `equality`/inequality of L2 tag of physical address used by the second instruction with the tag replaced from the L2 cache by the first instruction.

We have found one critical bug in the MMU design that appears only if certain constraints on instructions and dependencies between them are satisfied.

5.2. Verification of MIPS64 Microprocessor

Core-level verification of the MIPS64-compatible microprocessor is the most large-scale application of the suggested method. In this project we used triples of instructions as test actions. Total number of instructions is 221[3]. All instructions were clustered into 13 groups:

- `arithmetic (33 instructions)`;
- `logic` (8);
- `move` (8);
- `shift` (15);
- `branch` (20);
- `nop` (2);
- `memory` (26);
- `interrupt` (14);
- `system` (13);
- `fpu.arithmetic` (24);
- `fpu.move` (26);
- `fpu.convert` (26);
- `fpu.branch` (6).

To reduce size of test programs we used the following heuristic – test actions were composed by instructions from at most two different groups. Test situations and dependencies used for load/store instructions were very similar to the described in the first case study. Test data for all kinds of arithmetic instructions were directed to exceptional cases and boundary values. Branch instructions were described by condition value (for conditional jumps) and jump direction (forward or backward). We have found 9 errors in the RTL model of the microprocessor and 6 errors in the microprocessor simulator.

[3] Instructions differing by format of operands, for example, `add.s` (addition of single-precision numbers) and `add.d` (addition of double-precision numbers), were considered as different instructions.

6. Conclusion

In this paper we have described the method of automated test program generation for microprocessors. In contrast to widely-spread practical methods, like software-based verification and random generation, combinatorial model-based approach is much more systematic and technological. Our experience shows that the suggested method allows to find bugs which are usually omitted by other verification techniques. We position this method as an essential supplementation to existing approaches. Since the description of combinatorial tests does not require a lot of labor costs, such kind of testing can be done prior to advanced scenario-driven generation. In compliance with the method we have developed test program generator TestFusion 4M which has been successfully used in a number of projects. Further, we are planning to implement facilities of automatic test coverage derivation and construction of test templates.

References

[1] M. Behm, J. Ludden, Y. Lichtenstein, M. Rimon, M. Vinov. *Industrial Experience with Test Generation Languages for Processor Verification*. Design Automation Conference, 2004.

[2] MIPS R4000PC/SC Errata, Processor Revision 2.2 and 3.0. MIPS Technologies Inc., May 10, 1994.

[3] R. Ho, C. Han Yang, M. Horowitz, D.L. Dill. *Architecture Validation for Processors*. International Symposium on Computer Architecture, 1995.

[4] R. Ho. Validation Tools for Complex Digital Designs. PhD Thesis. November, 1996.

[5] A. Adir, E. Almog, L. Fournier, E. Marcus, M. Rimon, M. Vinov, A. Ziv. *Genesys-Pro: Innovations in Test Program Generation for Functional Processor Verification.* Design and Test, 2004.

[6] P. Mishra, N. Dutt. *Automatic Functional Test Program Generation for Pipelined Processors Using Model Checking*. IEEE International High-Level Design Validation and Test Workshop, 2002.

[7] P. Grun, A. Halambi, A. Khare, V. Ganesh, N. Dutt, A. Nicolau. *EXPRESSION: An ADL for System Level Design Exploration*. Technical Report 1998-29, University of California, Irvine, 1998.

[8] www.cs.cmu.edu/~modelcheck/smv.html.

[9] S. Ur and Y. Yadin. Micro Architecture Coverage Directed Generation of Test Programs. Design Automation Conference, 1999.

[10] D. Geist, M. Farkas, A. Landver, Y. Lichtenstein, S. Ur, Y. Wolfsthal. *Coverage Directed Test Generation Using Symbolic Techniques*. Formal Methods in Computer Aided Design, 1996.

[11] K. Kohno, N. Matsumoto. *A New Verification Methodology for Complex Pipeline Behavior*. Design Automation Conference, 2001.

[12] F. Corno, M. Sonza Reorda, G. Squillero, M. Violante. *A Genetic Algorithm-Based System for Generating Test Programs for Microprocessor IP Cores*. IEEE International Conference on Tools with Artificial Intelligence, 2000.

[13] D. Moundanos, J. Abraham, Y. Hoskote. *A Unified Framework for Design Validation and Manufacturing Test*. International Test Conference, 1996.

[14] D. Moundanos, J. Abraham, Y. Hoskote. *Abstraction Techniques for Validation Coverage Analysis and Test Generation*. IEEE Transactions on Computers, Vol. 47, 1998.

[15] RM7000 Family User Manual. Issue 1, May 2001.

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

[17] A.S. Kossatchev, A.K. Petrenko, S.V. Zelenov, S.A. Zelenova. *Application of Model-Based Approach for Automated Testing of Optimizing Compilers*. International Workshop on Program Understanding, 2003.

[18] A.V. Demakov, S.V. Zelenov, S.A. Zelenova. *Generation of Structurally Complex Test Data with Respect to Context Constraints.* Proceedings of Institute for System Programming of RAS, 2006 (In Russian).