ISSN 0361-7688, Programming and Computer Software, 2010, Vol. 36, No. 1, pp. 28–35. © Pleiades Publishing, Ltd., 2010.
Original Russian Text © E.V. Kornykhin, 2010, published in Programmirovanie, 2010, Vol. 36, No. 1.

Generation of Test Data for Verification of Caching Mechanisms and Address Translation in Microprocessors

E. V. Kornykhin

Institute for System Programming, Russian Academy of Sciences,
ul. Solzhenitsyna 25, Moscow, 109004 Russia
e-mail: kornevgen@ispras.ru

Received April 10, 2009

Abstract—This paper considers the problem of test data generation for the core-level verification of microprocessors; namely, the problem of constructing a test program on the basis of its abstract form (test template). To solve this problem, we propose an algorithm reducing it to a problem of resolving constraints. This paper addresses the verification of memory-handling instructions (taking into account such microprocessor features as caching and address translation).

DOI: 10.1134/S0361768810010056

1. INTRODUCTION

A key component of many computing systems is microprocessors with control functions. The verification of microprocessors is an important problem, which is addressed in this study.

Like software systems, microprocessors also can be subjected to functional verification (i.e., verification aimed at checking whether the functional requirements are met by a microprocessor. By a functional requirement at the system level (as opposed to the module level), we mean that the microprocessor instructions accomplish the semantics declared in the microprocessor architecture [1]. Here, the semantics under verification should be free of the features of the specific implementation in a given microprocessor and must be irredundant.

Within system verification, the microprocessor is regarded as a single system with its input data being machine programs loaded into the memory (hereafter, they are called test programs). Within functional verification, these programs are executed, the execution process is logged, and then analyzed to check whether the functional requirements are met when the verification program is executed.

Test programs can be constructed on the basis of a microprocessor model incorporating a model of microprocessor instructions—their signature (name and parameters) and functionality branches—and the microprocessor state. First, the test programs are constructed in an abstract form (as a test template), with a given sequence of instructions but without specific parameters and without the initialization of the microprocessor's state before the execution of these instructions. Next, the test templates are completed to form test programs. For that purpose, the instruction parameters are chosen and the instructions for initialization of the microprocessor state are constructed. The solution of this problem for instructions only deal with registers is well known [2–4]. However, the model of the microprocessor state becomes highly complicated if a test template contains instructions operating with cache memory [5], which also complicates the construction of verification programs. This problem is considered in the present study.

2. A REVIEW OF STUDIES IN CORE-LEVEL VERIFICATION OF MICROPROCESSORS

At present, the practice of core-level verification of microprocessors includes the following approaches to constructing test programs:

(•) *Manual development of test programs.* Even though this approach practically inapplicable for complete testing of microprocessors, it still can be used for testing corner cases.

(•) *Verification with cross-compilation.* This approach is frequently used due to its low complexity: after the microprocessor specification has been established, one can start developing a cross-compiler while the code designed for cross-compilation is ready. However, this verification approach cannot guarantee completeness.

(•) *Random generation of test programs.* This approach is frequently used due to the simplicity of its automation. The test programs generated in this way make it possible to quickly detect simple errors but do not guarantee the completeness of testing. More complex variants of random generation are also available (see [6]).

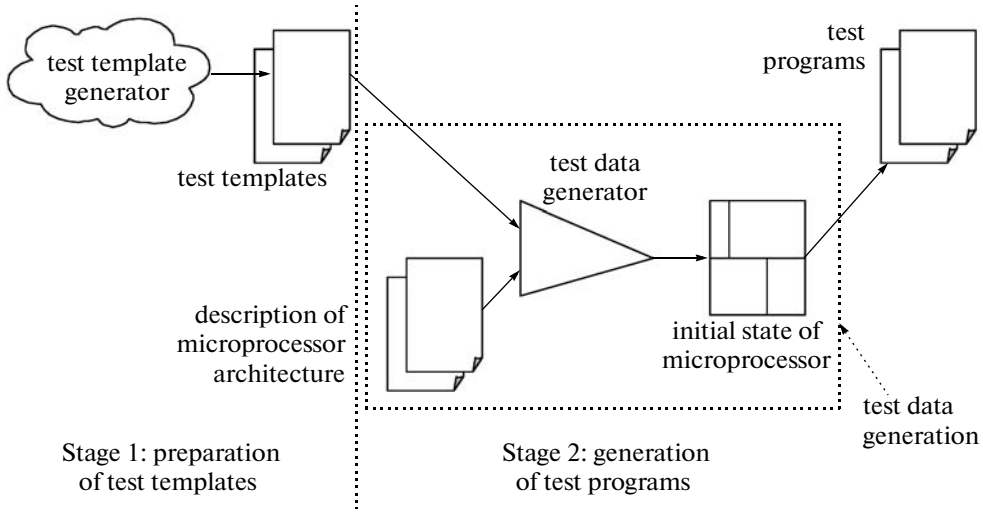

Fig. 1. Generation of test programs on the basis of test templates.

(•) *Generation of test programs on the basis of test templates.* This approach implies that the process of the test program generation is divided into two stages (see Fig. 1): at the first stage, test templates (abstract representations of test programs) are prepared (in test templates, the instruction parameters include constraints on the values rather than the values themselves); at the second stage, the test templates are used to generate test programs. The second stage involves generation of test data; i.e., the generation of instruction parameters (parameters–constants) and initial values of registers, cache memory cells, translation lookaside buffer (TLB) lines, etc. The individual registers–parameters of instructions are either specified in the test template (in this case, they must be included in the test program) [13] or not specified in the test template (in this case, the registers are chosen by the test data generator) [2].

Test templates describe the sequence of instructions and parameters of instructions with the indication of the occurring events (for example, overflow and cache misses or hits). The instruction parameters can be explicitly represented by registers and constants or, alternatively, the generator can be allowed to choose the registers itself. The sequence of instructions is most frequently specified explicitly. Normally, the need for a test template arises with a targeted testing when this target is expressed by a sequence of instructions of which each must be executed in a given way. An example of a test template is

```
REGISTER ax:32;
REGISTER bx:32;
REGISTER cx:32;
LW cx, ax, 0 @ l1Hit
SD ax, bx, 2 @ l1Miss
```

This test template includes two instructions: LW and SD. Each instruction has two registers and a constant as its parameters; after the symbol "`@`", there is information about the way the instruction must be executed (constraints on the values of the instruction parameters and on the microprocessor state): `l1Hit` denotes the cache hit (when the data are loaded from the specified memory location, they are already in the cache memory) and `l1Miss` denotes the cache miss (when the data are loaded from the specified memory location, they are not in the cache memory). To obtain a test program from this template, it suffices to specify the initial values of the registers `ax`, `bx`, and `cx` and the portion of the cache memory that is handled by the instructions: these are the test data for the given template. This can be performed by adding before the test template instructions initializing the processor state (the instructions that load data into registers, the sequence of instructions that load data into the memory that access the cache memory, TLB, etc.).

The resulting test program can be executed and checked whether the behavior of each instruction coincides with that declared in the test template.

Let us turn to the problem of test data generation. The following well-known methods of its solution can be mentioned:

(•) combinatorial techniques;

(•) automatic test pattern generation (ATPG);

(•) resolution of constraints.

The combinatorial techniques can be used in the case of simple test templates. Such test templates include only simple constraints (namely, an indication of the range of values of a variable). In this case, all values in the range in the test program are equivalent. This technique is simple but has a limited application

because the constraints on variables cannot always be reduced to such a simple form. The researchers in Fujitsu Lab [7] propose to describe test programs as expressions (test specification expressions, TSEs) and describe the processor instructions in ISDL. A special generator constructs test programs satisfying the TSE. Kohno and Matsumoto [8] consider the problem of verification of pipelined microprocessors using the generation of test programs based on test templates. The ranges of values of the variables in these templates are obtained from registers and numerical constants.

The researchers in Politecnico di Milano [9] proposed to generate test data on the basis of automatic test pattern generation (ATPG) techniques. ATPG is the problem of searching the values of the input signals ("vectors") of the scheme to detect its inadequate behavior. ATPG is more frequently applied for module testing if the RTL-model of the microprocessor is available. The ATPG problem has been around for a long time, and there are tools (including commercial ones) for its solution. To use ATPG in the generation of test programs, the RTL-model of the microprocessor should be ready by the time of test data generation. In addition, the use of this technique for functional verification is limited because the requirement of irredundancy of the instruction semantics is violated and the availability of an RTL-model is a redundant requirement and requires additional efforts for its preparation.

Most impressive results are obtained by the test generation tools based on the resolution of constraints. From the logical point of view, a constraint is a predicate and the problem of constraint resolution is the satisfiability problem for a system of predicates; however, the solution of this problem requires special algorithms [10]. In [2], the researchers in the Chinese National University of Technological Safety describe the MAATG tool. The test template for this tool can involve only equality or inequality constrains on the values and an indication of the range for a variable values. The microprocessor architecture is specified using a description in the EXPRESSION language. Another tool (Genesys-Pro [11]) is positioned by IBM as a project that has incorporated the best designs in the last 20 years. The test templates make it possible to specify test programs of variable length. For each instruction in the template, one can indicate heuristics for choosing the values of the parameters [12]. Among these, one can find heuristics for events in the cache memory and for translation of addresses. These heuristics should be user-defined in the form of constraints on the arguments of the instruction and data arrays (memory, page tables). The parameters of the next instruction are generated by Genesys-Pro using the part of the test program already constructed and a model of the microprocessor state (which is completely known). On the basis of the model of the microprocessor state and instruction heuristics, Genesys-Pro constructs a system of constraints and generates the values of the instruction arguments that satisfy it. However, the construction of such a generator of instruction arguments (specifically, resolving the system of constraints) is a nontrivial problem. This approach allowed the tool to be scalable for large test templates and complex microprocessor architectures but lead to the need of using backtracking if it is impossible to choose the parameters for the next instruction.

In this study, we also use the constraint resolution technique to generate test data. This is carried out within the technological chain of constructing test programs on the basis of a microprocessor model [13]. However, for the test templates obtained within this technological chain, the MAATG tool cannot be applied because the templates can contain not only constraints on equality or inequality of registers, but also more complex constraints (for example, cache-miss). In this paper, as compared to Genesys-Pro, the test template is completely translated into constraints (the problem of constraint resolution, i.e., satisfiability, is known to be NP-complete; this means that, for large test templates, the method proposed in this paper can be not very efficient; however, in practice, the errors in control logic are detected using short test programs). The constraint construction method proposed here does not backtracking. Due to this, the resolving system of constraints is qualitatively different (Genesys-Pro reduces the general problem to a set of problems of much lower complexity). In addition, this paper proposes a more convenient method for constructing test data: the model of microprocessor instructions can be obtained from the microprocessor architecture standard.

The test templates obtained in [13] have the specific feature that registers–parameters are fixed for each instruction. For such templates with complex dependences of registers' values, the operation of Genesys-Pro can be inefficient because one is not able to choose parameters to fit the execution of the next instruction to events specified for it in the test template. On test templates in [13], Genesys-Pro operates in the following way: it chooses some initial state of the microprocessor, executes the test template, and, when it meets an instruction that is executed not as required in the template, it returns to the very beginning; that is, Genesys-Pro will have to choose another initial state of the microprocessor and run the entire process once again. Such a test data generation procedure is too inefficient. The scheme of address translation in Genesys-Pro is based on the Deep-Trans approach [14]. However, one cannot find out from the available literature how the scheme of address translation is translated into constraints. To describe the address translation method, the authors of the paper use the

elements of the Memory array with unknown indices. It is known that the construction of constraints describing the operation on the elements of an array with unknown indices leads to very complex constraints that can be hardly resolved in an acceptable amount of time.

If one naively tries to apply the ideas underlying the tools described in the review (encoding the changes in the state of each register and dependences between them as constraints) to the memory handling instructions, the resulting constraints become very complex and cannot be resolved in an acceptable amount of time. Even if address translation is neglected, the microprocessor state can be encoded by using the formula of the length of memory size ($mem_0 = var0 \wedge mem_1 = var1 \wedge ...$); each change is done with respect to an unknown index; therefore, when a new microprocessor state is written, one has to search through all the possible variants ($mem[i] := x$ leads to the formula $(i = 0 \wedge mem_0 = x \wedge mem_1 = var_1 \wedge ...) \vee mem_0 = var_0 \wedge mem_1 = x \wedge ...) \vee ...$), and, if there are more than one such changes, one has to consider all possible variants of the index values. The resulting formula is about $|L| \cdot 2^n$ long, where $|L|$ is the memory size, and n is the number of changes in the memory. In this paper, we propose a method for encoding the changes that gives a formula about $|L| + n$ long.

3. GENERATION OF TEST DATA FOR OPERATIONS WITH MEMORY

By the generation of test data, we mean the generation of the initial state of the microprocessor (registers, cells of the cache memory, TLB lines, etc.) for a given test template. When the test template is executed started from the generated initial processor state, all the instructions must be executed in accordance with the template requirements. The test situation for the instruction (constraints on the values of the parameters and processor state) is used form the requirements for the execution of this instruction.

Cache memory and TLB can be involved for execution of the memory loading and story instructions [5]. The instruction parameters form a virtual address that is handled by the instruction. Then, using the virtual address (possibly, with the help of TLB), the microprocessor calculates the physical address. Finally, using the physical address (possibly, with the help of cache memory), the required operation with main memory is executed. If the data from the corresponding address are present in the cache memory (this situation is called cache hit), they are used by the instruction, and the main memory is not accessed. If the data are not in the cache (this situation is called cache miss), the memory is accessed to retrieve or write these data, which are then placed in the cache.

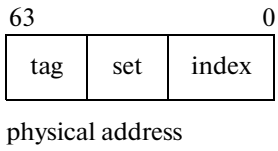

Fig. 2. Structure of physical 64-bit address.

Because the cache size is limited, some data are evicted from the cache in this case.

The TLB consists of a set of lines specifying the correspondence between the virtual address page numbers and the physical address page numbers. If the processor contains additional buffers accelerating the search for the required TLB, the test template may contain test situations for these buffers as well (cache hits or cache misses).

Therefore, the TLB may be characterized by such situations as cache hit in the TLB buffer and cache miss in the TLB buffer.

The test situations for memory operations may include cache miss, cache hit, constraint on the address that is handled by the instruction, and constraint on the TLB line. For example, the test template

```
REGISTER ax:32; REGISTER bx:32; CONST c:16;
LW ax, bx, c @ l1Hit, tlbMiss
SW ax, bx, c @ l2Miss, tlbHit
```

has `l1Hit` and `tlbMiss` as the test situation of the first instruction. The identifier `l1Hit` implies the L1 cache hit. The identifier `tlbMiss` implies a miss in the TLB buffer. The test situation of the second instruction assumes that there is an L2 cache miss (`l2Miss`) and a hit in the TLB buffer (`tlbHit`).

In this study, we propose a test data generation method for test templates with memory handling instructions. The test situations for these instructions can be divided into test situations concerning cache and test situations concerning address translation. Since the cache test situations for different instructions are interrelated[1], the proposed method first needs to generate constraints for the test situations in the cache (see Section 4 of this paper). Since the test situations concerning address translation are also interrelated (here, caching is used as well), the next step of the method is to generate the constraints for test situations in address translation (see Section 5 of this paper). As a result, we obtain a system of constraints on the initial values of the registers, physical addresses, and TLB lines; after resolving this system, it remains to supplement the test template by initializa-

[1] For example, a cache miss in one instruction means a special choice of the physical addresses for the preceding instructions with a cache hit, because, in the case of a cache miss, the evicted address is one of those addresses that had earlier involved cache hits.

tion instructions for the initial processor state on the basis of the calculated initial state of its subsystems.

4. SIMPLIFICATION OF TEST SITUATIONS IN THE CACHE

Each memory handling instruction operates with some physical address. A physical address at the specific cache memory level can be perceived as three bit fields: a tag (most significant bits of the address), a set, and an index in the cache line [5] (see Fig. 2).

The simplification of test situations in the cache memory starts with the distribution of a test template by sets; i.e., each of these test situations should be assigned the number of the set of its physical address. The assignement must be performed so that later it would yield a consistent system of constrains. For example, one can assign the same set to all the test situations or assign different sets to them.

Then, for each set, the following algorithm is used to obtain equality–inequality constraints for tags. The initial data for this algorithm is the sequence of test situations in the cache associated with the same set. For each test situation, one or two tags are specified (the tag of the physical address for the cache hit or the pair of evicting and evicted tags for the cache miss). The algorithm consists of two steps. At the first step, the constraints on the finite sets of tags are composed, and, at the second step, these constraints are simplified to obtain the desired form. The constraints can be simplified using any existing algorithm [10]. It is the use of symbolic simplifications and interpretation of the cache memory as a set[2] that made it possible to avoid coding changes of the cache state, which simplifies the system of constraints.

Figure 3 shows the pseudocode of the algorithm that constructs the constraints describing the sequence of test situations in the cache for the given test template. The eviction strategy (the rules for determining the tag to be evicted) in this algorithm is the Last Recently Used (LRU) because this is one of the most widely used algorithm in modern microprocessors. The current state of the set is modeled by the set L. The cache hit is described by the fact that the tag belongs to this set, and the cache miss is described by the fact that the evicting tag does not belong to this set. The LRU replacement policy is reformulated in the following form: after a tag is accessed, it can be evicted only after all the remaining tags in the set have been accessed. *xs* are the tag variables of the initial state of the set. The final address equality–inequality constraints are imposed on the contents of *xs* and on the evicting tags.

[2] In distinction from the list (vector) interpretation of the cache set; in essence, the order of elements given by the policy of replacement in the cache set is encoded into a sequence of test situations in the cache rather than into set state.

This algorithm can be optimized to reduce the size C taking into account the following remarks:

(1) Between the last access of the tag q and its eviction, there should be at least $N-1$ accesses of any tags, where N is the cache associativity (the set size); therefore, in the loop of the procedure `lru`, one can disregard the cache hits located nearer than $N-1$ misses from τ.

(2) The order of last tag accesses repeats the order of their eviction; i.e., in the loop of the procedure `lru`, one can disregard the cache hits from the start of *tt* until the cache hit of the tag evicted by the preceding cache miss in the loop of the procedure `A`.

(3) The sequence of the cache hits in the loop of `lru` should not pass through more than N evictions (otherwise, this sequence would have included the tag being evicted); therefore, one can disregard in the loop the cache hits from the start of *tt* to the cache miss located exactly N cache misses away from τ.

(4) In the procedure `lru`, one can generate the set C of constraints in an `lazy` manner; i.e., to obtain C, one can pass a small number of loop iterations, then return to algorithm `A`; if this C did not yield a solution, return and pass some additional number of iterations (for example, this mechanism was implemented in constraint logic programming [15]).

(5) If *tt* starts with a sequence of cache misses, one can easily calculate (without resolving the constraints) the tags evicted by them (for example, the first evicted tag is identical to the first tag added to the set, the second evicted tag is identical to the second added tag, etc.).

5. CONSTRUCTION OF CONSTRAINTS FOR ADDRESS TRANSLATION

The remaining steps of the test data generation algorithm include the generation of constraints for the mechanism of address translation, resolution of the resulting addresses, and formation of the initial state of the microprocessor subsystems. In this paper, we consider a TLB-based technique for address translation that is commonly used in the architecture of MIPS microprocessors [16], although similar ideas can be used for the address translation in other architectures. The process of address translation and main structures are shown in Fig. 4.

To generate the constraints describing the address translation, we use the test template and the information about the sets and tags obtained at the stage of the simplification of the test situations in the cache. The first stage of constraint generation is the simplification of test situations in the TLB buffer (this is done as described in Section 4: the set is the same and the indices of the TLB lines are used as tags). Then, for each pair of the instructions accessing a TLB line, constraints on their virtual addresses are formed (in terms of the instruction parameters):

```
procedure A(tt: test_template_for_set, xs: tag-list )
returns C : constraint-set
begin
   C := {};
   var L : tag-set := {}
   for each (tag t in xs)
   begin
      add constraint t ∉ L to C;
      L := L ∪ {t};
   end;
   for each (test_situation t in tt)
   begin
         if t is a cache hit of tag p,
            add constraint p ∈ L to C;
         else if t is a cache miss of tag p with eviction of tag q then
      begin
         add constraint q ∈ L to C;
         add constraint p ∈ L to C;
         add constraint lru(q, L, t, tt);
         L := L ∪ {p} \ {q};
      end;
   end;
   simplify C;
end,
procedure lru(q: tag, L: tag-set, t: test_situation,
tt: test_template_for_set ) returns C: constraint
begin
   C := ⊥;
   for each (τ'(p1): cache hit of tt from start[3] to τ)
   begin
           var T: tag-set := set of evicting tags and hit tags
                in tt between τ' and τ noninclusive
           C := C ∨ (q = p1) ∧ (L \ {q} = T);
   end;
end
```

Fig. 3. Algorithm of constraint construction for test situations in cache memory.

(•) coincidence of the bits of the virtual addresses corresponding to the field "r";

(•) coincidence of the bits of the virtual addresses corresponding to the field "vpn/2" with regard to the "mask" field (for example, if the field "vpn/2" occupies the bits from 40th to 13th, then $v_{40\ldots(13+m)} = w_{40\ldots(13+m)}$, where v and w are virtual addresses, and m is the integer form of the field "mask"; namely, the half of the number of zeros in the field "mask");

(•) constraint on the fields "g" and "asid;"

(•) constraints on their lower bit of the virtual page number; for example, if the translation of addresses of two instructions is accompanied with access to the same TLB line and the physical addresses are different (this can be conceived from equality–inequality of tags and sets), then their lower bits are different;

(•) the correspondence between the virtual and the physical addresses associated with the offset in the page; for example, if the field "vpn/2" occupies the bits from 40th to 13th, then $v_{(11+m)\ldots0} = phys_{(11+m)\ldots0}$, where v is the virtual address of the instruction, *phys* is

[3] "Start" means the addition of tags–variables of the initial state to the set added before the test template.

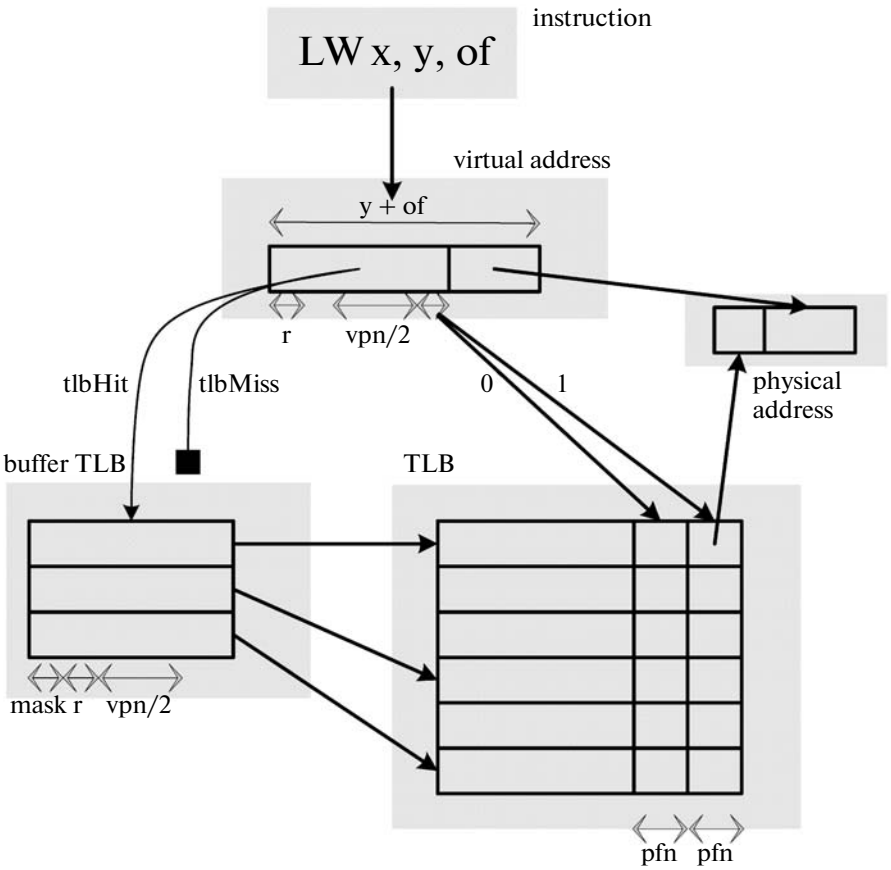

Fig. 4. Address translation in microprocessors of MIPS architecture.

the physical address of the same instruction, and *m* is the integer form of the field "mask".

The test template operates with pages (of the virtual memory) of the size specified by the operating system. Therefore, for all the TLB lines that are handled by the test template, the value of the field "mask" is known and specified before the test data generation. This simplifies the resulting constraints because there is no need to pass the ranges of bits with variable boundaries.

Then, for each TLB line involved, an arbitrary instruction is chosen from the test template. For each pair of such instructions (the pair is composed of instructions related to different TLB lines), a constraint is constructed to represent the fact that their TLB lines are different. For the address translation adopted in MIPS, this may be the disjunction of the difference in the virtual address bits corresponding to the field "r", the bits corresponding to the field "vpn/2" (with account of the mask), and the zero value of the bit "g" provided that the field "asid" is different from the value in the register EntryHi (this value, as well as the field "mask", is specified before the construction of the constraints because this is a functional parameter of the operating system that is invariable during the execution of the test template).

6. CONCLUSIONS

This paper considers the generation of test data for the core-level verification of microprocessors; more precisely, we consider the problem of constructing a test program on the basis of test template specified for it. To solve this problem, we propose an algorithm that uses constraint resolution approach [10]. This algorithm is implemented on the basis of the ECLiPSe logic programming system with constraints [15] as a solver of constraints on integer numbers and finite sets of integer numbers. The algorithm is applied in the project of testing of an industrial microprocessor of MIPS64 architecture [1]. In future, we plan to consider the generation of test data with account of some given initial state of the microprocessor. In this case, the size of the additional initialization of the microprocessor state (generated by the proposed algorithm) can be minimized. In addition, we plan to extend the spectrum of the caching strategies in use by developing general techniques for their description in the form of constraints (for the FIFO eviction strategy, we have already obtained the corresponding result [17]).

REFERENCES

1. *MIPS64: Architecture for Programmers, vol. II: The MIPS64 Instruction Set.*
2. Li, T., Zhu, D., Guo, Y., Liu, G., and Li, S., MAATG: A Functional Test Program Generator for Microprocessor Verification, *Proc. of the 8th Euromicro Conf. on Digital System Design (DSD'05)*, 2005.
3. Kornykhin, E.V., Generation of Test Data for Verification of Arithmetic Instructions of CPUs, *Trudy ISP RAN*, 2008, vol. 15, pp. 107–117.
4. Kornykhin, E., Test Data Generation for Arithmetic Subsystem of CPUs MIPS64, *Proc. of the Second Spring Young Researchers' Colloquium on Software Engineering*, 2008, vol. 2, pp. 43–46.
5. Patterson, D. and Hennessy, J., *Computer Organization and Design: The Hardware/Software Interface. The Morgan–Kaufmann Series in Computer Architecture and Design*, Morgan and Kaufmann, 2006.
6. Corno, F., Sanches, E., Reorda, M.S., and Squillero, G., Automatic Test Program Generation—a Case Study, *IEEE Design & Test. Special Issue on Functional Verification and Testbench Generation,* 2001, vol. 21, no. 2, pp. 102–109.
7. Fallah, F. and Takayama, K., A New Functional Test Program Generation Methodology, *Proc. of the IEEE 2001 Int. Conf. on Computer Design: VLSI in Computers and Processors*, 2001, pp. 76–81.
8. Kohno, K. and Matsumoto, N., A New Verification Methodology for Complex Pipeline Behavior, *Proc. of the 38th Design Automation Conf. (DAC'01)*, 2001.
9. Beardo, M., Bruschi, F., Ferrandi, F., and Sciuto, D., An Approach to Functional Testing of VLIW Architectures, *Proc. of the IEEE Int. High-Level Validation and Teat Workshop (HLDVT'00),* pp. 29–33.
10. Semenov, A.L., Methods of Constraint Propagation: Main Concepts, *PSI'03/Interval Mathematics and Methods of Constraint Propagation*, 2003.
11. Behm, M., Ludden, J., Lichtenstein, Y., Rimon, M., and Vinov, M., Industrial Experience with Test Generation Languages for Processor Verification, *Proc. of the 41st Design Automation Conference (DAC'04)*, 2004.
12. Adir, A., Almog, E., Fournier, L., Marcus, E., Rimon, M., Vinov, M., and Ziv, A., Genesys-pro: Innovations in Test Program Generation for Functional Processor Verification, *IEEE Design and Test of Computers*, 2004, vol. 21, no. 2, pp. 84–93.
13. Kamkin, A.S., Generation of Test Programs for Microprocessors, *Trudy ISP RAN,* 2008, vol. 14, no. 2, pp. 23–64.
14. Adir, A., Emek, R., Katz, Y., and Koyfman, A., Deeptrans—a Model-Based Approach to Functional Verification of Address Translation Mechanisms, *Proc. of the 4th Int. Workshop on Microprocessor Test and Verification: Common Challenges and Solutions*, 2003, pp. 3–6.
15. Apt, K.R. and Wallace, M., *Constraint Logic Programming Using Eclipse*, Cambridge Univ. Press, 2007.
16. *MIPS64: Architecture for Programmers, vol. III: The MIPS64 Privileged Resource Architecture.*
17. Kornykhin, E.V., Generation of Test Data for Core-level Functional Verification targeted to FIFO Cache Memory of Microprocessors, *Vychislitel'nye metody i programmirovanie*, 2009, vol. 10, pp. 107–116.