Methodology and Experience of Simulation-Based Verification of Microprocessor Units Based on Cycle-Accurate Contract Specifications

Mikhail Chupilko, Alexander Kamkin, and Dmitry Vorobyev
Software Engineering Department
Institute for System Programming of Russian Academy of Sciences
25, B. Kommunisticheskaya, Moscow, 109004, Russia
E-mail: {chupilko, kamkin, vorobyev}@ispras.ru

Abstract— **In this paper we describe a methodology and experience 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. We have successfully applied the methodology to several units of the industrial microprocessor. The experience shows that cycle-accurate contract specifications are very suitable for simulation-based verification, since, first, they represent functional requirements on a unit in comprehensible declarative form, and second, they make it possible to automatically construct test oracles which verify unit correctness.**

I. Introduction

Microprocessors underlie all digital computer systems, therefore reliability of a whole system is strongly depends on the correctness of a microprocessor. 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], [2]. High cost of a bug and increasing design complexity 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 [3]. It also should be emphasized that unit-level verification provides significant return in terms of quantity and quality of bugs removed [4].

The main task of functional verification is to check correspondence between design under verification (DUV) behavior and functional requirements. To have the ability to do it automatically requirements should be represented in machine-readable form. Such form of requirements representation is usually called *formal specifications* or *specifications* for short.

In this work we consider specifications of a specific form, so-called *cycle-accurate contract specifications* or *contract specifications of pipeline* [5], [6]. Such specifications describe behavior of a microprocessor unit in the form of *preconditions* and *postconditions* of *microoperations* (parts of operations which are executed over one clock cycle). Summarily, precondition of microoperation formally determines when microoperation is not interlocked; postcondition defines microoperation functionality. Pair of precondition and postcondition is called *contract of microoperation*.

In our opinion, cycle-accurate contract specifications are very suitable for simulation-based verification. First, they allow to represent functional requirements on a unit in comprehensible declarative form. Second, they make it possible to automatically construct test oracles, which check conformance of the unit behavior to the requirements described in the specifications.

The rest of the paper is organized as follows. The second section contains a description of the suggested methodology of specification and verification of microprocessor units. In the third section short review of the UniTESK technology and the CTESK test development tool is given. This section also comprises two specification examples. The next two sections describes our experience. In the sixth section related work is outlined. Finally, the seventh section concludes the paper.

II. Cycle-Accurate Contract Specifications

In this section we define cycle-accurate contract specifications and consider how they can be used for representation of requirements and simulation-based verification of microprocessor units. It should be noticed that we consider only *synchronous units*, in which components are synchronized by a *clock signal*.

A. Specification of Requirements

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

The suggested methodology is based on contract specifications in the form of preconditions and postconditions. We propose to define contracts for separate microoperations and to obtain the contract for the entire operation by means of *temporal composition of contracts*. Here, under temporal composition of contracts we mean a special mechanism that in each cycle 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 microoperation a postcondition describing requirements to it is defined. Then, the postcondition of each microoperation is associated with the cycle at the end of which 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 consider such operations, that their microoperations are executed sequentially, i.e., $\tau_1 = 1, ..., \tau_n = n$. This assumption does not restrict the generality. In this case contract of the operation consisting of n microoperations is given by the formula:

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

For the contract C we introduce the following notation. The precondition of the operation is denoted as pre_C; the postcondition of the i^{th} microoperation is denoted as $post_C^i$.

B. Verification of Requirements

Suppose that at some moment of time the unit under verification performs m operations $f_1, ..., f_m$ that have been supplied for execution $\tau_1, ..., \tau_m$ cycles earlier, respectively. Let $C_1, ..., C_m$ be contracts of the operations $f_1, ..., f_m$, respectively, and let, at the moments when the operations $f_1, ..., f_m$ were supplied, the preconditions $pre_{C_1}, ..., pre_{C_m}$ have been fulfilled. Then, to verify correctness of the unit behavior at the given moment of time it is required to check satisfiability of the predicate called *test oracle*:

$$Oracle(C_1, \tau_1; ...; C_m, \tau_m) = \bigwedge_{i=1}^{m} post_{C_i}^{\tau_i}.$$

Test oracle organization for pipelined operations 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.

C. Specification of Interlocks

In the previous section 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:

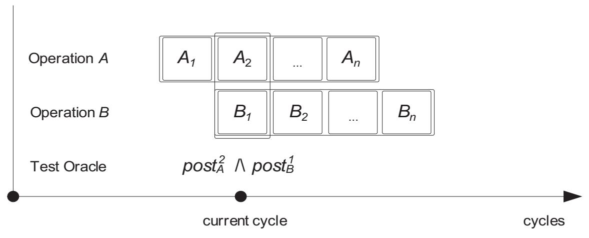

Fig. 1. Test oracle for pipelined operations.

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

where pre is the precondition of operation, pre_i is the *guard condition*[1] of the i^{th} microoperation, and $post_i$ is the postcondition of the i^{th} microoperation.

To verify such requirements testbench should keep track which microoperations are finished and check corresponding postconditions. As it was said before, a special mechanism that in each cycle of simulation calculates a set of postconditions to be checked is called temporal composition of contracts. Formal description of temporal composition mechanism can be found in papers [5] and [6].

D. Formalization of Requirements

Development of formal specifications can not be done if requirements are not formulated clear enough. Development of cycle-accurate specifications demands additional clearness and exactness from the requirements. They should specify not only constraints on input and output signals of microoperations, but temporal constraints restricting cycles in which microoperations are executed as well. Such requirements are called *cycle-accurate requirements*.

Experience shows that original documentation that customers give along with RTL model of a unit should be supplemented, refined, and structured before it becomes suitable for formalization. Usually such descriptions say how test system can act upon a unit, but they do not distinctly say how test system can verify unit correctness. Typical documentation of a unit for each operation specifies a set of actions performed by the unit. This set is generally incomplete, and it is not clear how many cycles are required for execution of the actions.

As it was said before, original documentation should be supplemented and altered. To do it, we suggest using a special form of requirements representation, so-called *catalogue of requirements* or *catalogue* for short. We distinguish four types of requirements to an operation:

- *pre requirements* – requirements on operation precondition;
- *guard requirements* – requirements on microoperation guard condition;

[1]*Guard condition* is a negation of interlock condition, i.e. microoperation is interlocked, if and only if the corresponding guard condition is false.

TABLE I
TABLE OF REQUIREMENTS.

Operation	**Microoperation**$_1$	...	**Microoperation**$_n$
Pre	Guard	...	Guard
...	...	...	...
...	Update	...	Update
...	...	...	...
...	Post	...	Post
...	...	...	...

- *update requirements* – implicit requirements on microoperation functionality;
- *post requirements* – explicit requirements on microoperation functionality.

For each operation catalogue contains a table combining requirements on this operation. Operation consisting of n microoperations is represented by the table with $n+1$ columns (see **Table I**). The first column corresponds to operation precondition, while the others are connected with microoperations. Each cell of the table contains a requirement. It is implied that one requirement defines constraints on input and output signals within one clock cycle. All requirements are divided into four groups depending on their types.

The process of catalogue compiling is as follows. At the first step requirements on functionality (update requirements and post requirements) are defined. The difference between update and post requirements is that update requirements imperatively define how microoperation updates the state of the unit, while post requirements declaratively describe the expected result of the microoperation. Pre requirements and guard requirements are usually refined when test experiments have begun.

It should be emphasized that compiling of catalogue is generally done in active co-operation with customer. Compiling takes additional labor time, but it shrinks specification development and considerably simplifies support of specifications and tests. Catalogue is also very useful for developers, because it gives them high-quality documentation.

III. UNITESK TECHNOLOGY

The UniTESK technology [7], [8] was developed at the Institute for System Programming of Russian Academy of Sciences (ISPRAS) [9]. The UniTESK technology and supporting tools have been successfully applied for functional testing of different kinds of software (operating systems, telecommunication protocols, real-time systems, etc.). A key moment in the successful use of the UniTESK technology is the flexible and scalable test system architecture, which allows to adapt the technology to various classes of systems [8].

A. UniTESK Test System Architecture

UniTESK test system architecture has been developed as a result of many years experiments on specification-based testing of the industrial software from different fields and of different levels of complexity [7], [8]. These experiments have allowed to create the flexible and scalable test system architecture. Interactions of UniTESK test system components are shown in **Fig. 2**.

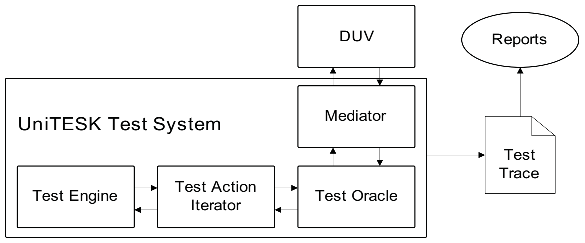

Fig. 2. UniTESK test system architecture.

Test engine is a library component of the UniTESK test system. Test engine and test action iterator are intended for test sequence generation. Test engine is based on an FSM traversal algorithm.

Test action iterator works under the test engine control. It calculates current FSM state, iterates corresponding stimuli, and applies them. Test action iterator is automatically generated from the high-level test scenario description.

Test oracle verifies the DUV behavior in response to a single stimulus. It is automatically constructed from the formal specifications.

Mediator connects formal specifications and DUV implementation. It makes some transformations of the stimuli and reactions and also synchronizes the specification state with the implementation one.

Test trace shows the events happening during the testing. It is used by the UniTESK supporting tools to automatically generate different reports that help in the test results analysis.

B. CTESK Test Development Tool

CTESK test development tool is an implementation of the UniTESK conception for the C programming language. It uses SeC (specification extension of C) language to develop test system components. SeC language provides test developers with special functions:

- *specification functions* – to specify DUV operations and to define functional coverage structure;
- *mediator functions* – to connect specification functions with corresponding stimuli;
- *function of FSM state calculation* – to calculate FSM state on the base of the specification state;
- *scenario functions* – to define a set of stimuli to be applied in each of the reachable states.

We have adapted the CTESK tool for functional verification of Verilog and SystemC designs [10]. Further we consider the use of CTESK for specification development by the examples of two simple units.

C. Specification Example: Floating-Point Adder

To illustrate basic ideas of the suggested approach let us consider an example of a 3-stages floating-point adder. The adder is intended for adding two normalized floating-point numbers (zero values are also permitted) [11]. The operation consists of three microoperations (stages): (1) alignment of exponents, (2) addition of fractions, and (3) normalization of result.

The first step of operation specification is the definition of so-called *operation context type* that describes the current state of the operation execution. This type usually contains operands of the operation and all kinds of temporal values calculated on one stage to be used on the successive stages.

```
// Context of ADD operation
specification typedef struct ADDContextT {
  // Operation operands
  bool op1_sign;
  uint8_t op1_exponent;
  uint32_t op1_fraction;
  ...
} ADDContextT;
```

The specification function of the operation contains operation precondition.

```
// Specification function of ADD operation
specification void ADD_spec(SingleT op1, SingleT op2) {
  // Operation precondition
  pre {
    return (isZero(op1) || isNormalized(op1))
       && (isZero(op2) || isNormalized(op2));
  }
}
```

For each stage of the operation a special function is developed to set forth stage requirements. Consider specification of the following requirement on the alignment stage: “If operands have different exponents, then fraction of the operand with the smaller exponent is shifted to the right (the number of positions that the bits in the fraction are to be shifted is the difference between exponents). If there is a unit among the shifted bits, then output *inexact_align* is set to high; otherwise, it is set to low”.

```
// Specification of the alignment stage
reaction ADDContextT* ADD_align_spec(void) {
  AdderUnitT *adder_unit = getAdderUnit();
  ADDContextT *add = ADD_align_spec;
  // Postcondition of the stage
  post {
    int shift = abs(add->op1_exponent -
      add->op2_exponent);
    if(add->op1_exponent > add->op2_exponent) {
      return adder_unit->inexact_align ==
        (add->op2_fraction & mask(shift)) != 0;
    }
    ...
  }
}
```

In the code above $getAdderUnit$ is a function that returns specification representation of the adder state.

D. Specification Example: Translation Lookaside Buffer

Hereafter we consider specification of a simplified version of translation lookaside buffer (TLB). TLB is a buffer in a microprocessor that is used for address translation. TLB has a fixed number of entries containing part of the page table, which translates virtual addresses into physical ones.

The memory of the TLB under verification comprises three buffers: an instruction micro TLB (ITLB), a data micro TLB (DTLB) and a joint TLB (JTLB). The purpose of the micro TLBs is to allow two address translation operations to be performed simultaneously – one for an instruction fetch address (via the ITLB) and one for a data load/store address (via the DTLB).

Translation of address is performed by two microoperations (stages): (1) access to micro TLB (ITLB or DTLB), and (2) access to JTLB. Virtual address of data (instruction) is propagated via input d_va (i_va). The outputs of the unit are the following:

- d_jtlb_req (i_jtlb_req) – request to JTLB from data (instruction) address translation operation;
- d_pa (i_pa) – physical address of data (instruction);
- $jtlb\_hit$ – JTLB hit.

Consider the following requirements on the JTLB access stage of the instruction address translation operation: (1) “if an entry for the given virtual address is found in the JTLB, then $jtlb\_hit$ is set to high; otherwise, it is set to low”; (2) “if TLB hit (ITLB hit or JTLB hit) is occurred, then i_pa is set to the physical address corresponding to the virtual address”; (3) “when two address translation operations (one for an instruction fetch and one for a data load/store) try to access JTLB simultaneously, the operation of data address translation gets a priority; execution of the instruction address translation operation is interlocked”. These requirements can be formalized as follows.

```
// Specification of the JTLB access stage
reaction ITranContextT* ITRAN_jtlb_access_spec(void) {
  TLBUnitT *tlb_unit = getTLBUnit();
  ITranContextT *itran = ITRAN_jtlb_access_spec;
  // Precondition of the stage
  pre {
    Process *process = getProcess_StimulusID(
      tlb_unit->pipe, DTRAN, JTLB_ACCESS);
    // Specification of requirement (3)
    return process == NULL;
  }

  // Postcondition of the stage
  post {
    if(!isJTLBHit_TLB(tlb_unit, itran->i_va)) {
      // Specification of requirement (2)
      return tlb_unit->jtlb_hit && tlb_unit->i_pa ==
        translateAddress_TLB(tlb_unit, itran->i_va);
      } else {
         // Specification of requirement (1)
         return !tlb_unit->jtlb_hit;
      }
   }
   ...
}
```

In the code above $ITranContextT$ is a context of the instruction address translation, and $getProcess\_StimulusID$ is a CTESK library function that returns information on executing operations.

IV. Case Study: Translation Lookaside Buffer

The suggested approach was applied to TLB of the industrial microprocessor with MIPS64-compatible architecture [15]. This section outlines our experience and the achieved results.

A. Description of TLB

The memory of the TLB under verification comprises three buffers: a 4-entries instruction micro TLB (ITLB), a 4-entries data micro TLB (DTLB), and a large 64-entries joint TLB (JTLB). If an entry is not found in the corresponding micro TLB, then the JTLB is accessed. Once the entry is retrieved, it is written back to the micro TLB. To refill micro TLBs the least-recently-used (LRU) strategy is used – micro TLBs always replace the entry which has not been accessed for the longest amount of time. Thus, micro TLBs contain a subset of translations that are most-recently-used.

The TLB implements the following operations:

- *read entry* – reads entry from the buffer;
- *write entry* – writes entry to the buffer;
- *probe entry* – probes if the entry exists in the buffer.
- *translate data address* – translates virtual address of data;
- *translate instruction address* – translates virtual address of instruction;

Address translation operations are organized as multi-cycle pipelined operations. A micro TLB miss sequence has a penalty of one extra clock cycle. If we have simultaneous ITLB miss and DTLB miss, the DTLB gets first priority when accessing the JTLB, and the translation of instruction address stalls an additional cycle, giving a total penalty of two latency cycles.

The interface of the TLB under verification contains about 30 inputs and as many outputs. The RTL model of the TLB is implemented in Verilog. The main part of the source code consumes about 3.5 KLOC.

B. Specification and Verification of TLB

A total number of the requirements that we have specified is about 100. TLB requirements partitioning is shown in **Table II**. Requirements on each operation were represented in the form of cycle-accurate contract specifications. It should be emphasized that all requirements were cheaply formalized.

TABLE II
TLB requirements.

Operation	Pre	Guard	Update	Post	Total
Read	5	0	0	2	7
Write	5	0	2	2	9
Probe	5	0	0	3	8
Translate Data Address	5	0	3	30	38
Translate Instruction Address	5	3	2	27	37
Total	25	3	7	64	99

The volume of specifications consumes about 2.5 KLOC in SeC language.

We have found 9 errors in the TLB implementation including critical ones. It should be noticed that all errors were found in address translation operations and the majority of errors are connected with the control logic of the unit. The total labor costs of the testbench development make up to about 2.5 man-months.

C. Reuse of TLB Specifications

After we have verified TLB (v2.0) we started testbench development for the next version (v3.0). We tried to reuse specifications developed for TLB v2.0 for the verification of TLB v3.0. In this section we consider which modifications of specifications were done and which percentage of reuse was achieved.

The most changes were concentrated in read, write, and probe operations. In TLB v3.0 width of the output bus is twice smaller than in TLB v2.0, therefore reading is twice longer (two stages instead of one). Operations of writing and probing have been prolonged as well. Address translation operations have not been changed too much.

Modifications of specifications were of the following kinds: modification of precondition, modification of guard condition, addition of microoperation, addition of requirement, and redistribution of requirements between different microoperations. In case of read operation, one requirement was split into three constraints. One of them ought to be fulfilled at the end of the both stages; the others are distributed among the stages. So, one requirement on TLB v2.0 was split into the four requirements on TLB v3.0.

In sum, number of requirements on TLB v3.0 was increased by 14 (we do not take into account splitting of the requirement in the read operation). Besides, 16 requirements on the operation preconditions were changed. Percentage of code reuse consumes more than 50% (if even one line of the specification function has been changed, the whole function is considered as non-reusable). All modifications affect less than 20% of the code. Labor costs of modifications are about 1 man-week.

V. Case Study: L2 Cache

In this section we describe our up-to-the-minute experience that was obtained in the project of L2 cache verification. L2 cache is the most complex unit that we have specified and verified.

A. Description of L2 Cache

The L2 cache under verification is a direct-mapped 256 KB cache that consists of 8192 rows. Each row contains data (four 64-bit double words), tag (18 upper bits of physical address), and control bits (V – Valid and W – Writeback). The cache serves both data and instructions. The microprocessor supports direct memory access (DMA) to the cache via load/store instructions when operating in digital signal processing (DSP) mode.

The L2 cache implements the following operations:

- *load data* – reads data from the L2 cache;
- *load instruction* – reads instruction from the L2 cache;
- *store data* – stores data into the L2 cache;
- *cache operation* – modifies data, tags, and control bits;
- *load data*(*DSP mode*);
- *store data*(*DSP mode*).

If L2 miss occurs when executing a load data operation, then, if it is necessary (control bit W is set in the cache row), write-back is performed, after that required row is read from the memory. If hit occurs when executing store operation, then data are written into the cache row and control bit W is set. Processing of the miss for the store operation is the same as for load operations. We should mention the following particular feature of the store operation. It consists of two parts: preliminary request on data retention, which can be canceled by interrupt signal, and acknowledgement of data retention.

Cache operation unifies six completely different sub-operations: *index writeback invalidate*, *index load tag*, *index store tag*, *hit invalidate*, *hit writeback invalidate*, and *hit writeback* [15].

All operations implemented by the unit are multi-cycle pipelined operations. Pair of store operations can be executed sequentially one by one. The L2 cache can not start operation execution if it handles operation that causes L1 miss. Operations for data and operations for instructions can be supplied for execution simultaneously, but operations for data have a priority. The other operations (cache control operation, load/store in DMA mode, etc.) can not be started until the unit finishes the execution of the previous operations.

The interface of the L2 cache under verification contains about 70 inputs and 30 outputs. The RTL model of the cache is implemented in Verilog. The source code of the model consumes about 3 KLOC.

B. Specification and Verification of L2 Cache

A total number of the requirements that we have specified is about 180. Requirements partitioning is shown in **Table III**. The volume of specifications consumes about 3 KLOC in SeC language.

TABLE III
L2 CACHE REQUIREMENTS.

Operation	**Pre**	**Guard**	**Update**	**Post**	**Total**
Load Data	4	10	7	3	24
Load Instruction	2	5	0	2	9
Store Data	6	13	77	15	111
Cache Operation	5	3	20	6	34
Load Data (DSP)	1	0	0	1	2
Store Data (DSP)	1	0	1	0	2
Total	19	31	105	27	182

When we were developing specifications for L2 cache we have confronted with two particular features of the unit. The first one is that processing of operations depends on many factors (L2 hit/miss, interruptions, control bits, etc.). We have distinguished about 25 different functional branches. The other feature is the existence of nonatomic test actions. It means that testbench should provide the unit with additional data during the operation execution. It was simply implemented by the mediators of microoperations.

We have found 3 critical errors in the L2 cache implementation. Two errors relate to cache operation, and one error affects instruction loading operation. The total labor costs of the testbench development make up to about 4 man-months.

VI. Related Work

The means of specification used in modern hardware verification languages (HVLs), like OpenVera [12], SystemVerilog [13], etc., are based on *temporal logics* [14]. HVLs operate with limited sequences of events, for which it is possible to address both the past and the future. From simple sequences one can construct more complicated ones using logical operations AND and OR or by means of regular expressions.

In the approaches based on temporal logics the focus is placed on *temporal decomposition* of operations. For each operation its temporal structure (admissible sequences of events and delays between them) is first determined. Then, the predicates describing separate events are defined. In our approach, the focus is placed on *functional decomposition of operations*. First, functional structure of the operation, i.e., the set of microoperations, is determined. Then, each microoperation is specified, and temporal composition is carried out.

We assume that the functional structure of an operation is more stable compared to the temporal structure. Hence, the approaches based on the functional decomposition of operations make it possible to develop specifications that are more robust with respect to modifications of the implementation compared to the approaches based on the temporal logics. The advantageous features of the proposed approach are also its clarity and simplicity. The preconditions and postconditions are usually simpler than temporal logic formulas and do not require special knowledge from the test developer.

VII. Conclusion

The need of automated testbench development for microprocessor units is widely recognized. It is clear that high-level automation can not be achieved without using formal specifications, which describe unit behavior in machine-readable form. The paper described the methodology to formal specification of microprocessor units which is suitable for simulation-based verification. The approach is based on cycle-accurate contract specifications in the form of preconditions and postconditions of microoperations. The methodology is supported by the CTESK test development tool from the UniTESK toolkit.

We have successfully applied our approach to several units of the industrial MIPS64-compatible microprocessor. The approbation has demonstrated effectiveness and relatively low labor costs of testbench development with the aid of cycle-accurate contract specifications and the UniTESK technology. We have found a number of bugs in the implementation of the units that had not been discovered earlier at the chip level; there are some critical bugs among them.

We consider the methodology 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.

References

[1] J. Bergeron. *Writing Testbenches: Functional Verification of HDL Models*. Kluwer Academic Publishers, 2000.

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

[3] B. Bentley. *Validating the Intel Pentium 4 Microprocessor*. DAC'2001: Design Automation Conference, 2001.

[4] J.M. Ludden, W. Roesner, G.M. Heiling, J.R. Reysa, J.R. Jackson, B.-L. Hui, M.L. Behm, J.R. Baumgartner, R.D. Peterson, J. Abdulhafiz, W.E. Bucy, J.H. Klaus, D.J. Klema, T.N. Le, F.D. Lewis, P.E. Milling, L.A. McConville, B.S. Nelson, V. Paruthi, T.W. Pouarz, A.D. Romonosky, J. Stuecheli, K.D. Thompson, D.W. Victor, and B. Wile. *Functional Verification of the POWER4 Microprocessor and POWER4 Multiprocessor Systems*. Volume 46, Number 1, 2002.

[5] A. Kamkin. *Contract Specification of Pipelined Designs: Application to Testbench Automation*. SYRCoSE'2007: The 1^{st} Spring Young Researchers Colloquium on Software Engineering, 2007.

[6] A. Kamkin. *Testbench Automation for Pipelined Designs Based on Contract Specifications*. IEEE-EWDTS'2007: The 5^{th} East-West Design & Test Symposium, 2007.

[7] http://www.unitesk.com

[8] I. Bourdonov, A. Kossatchev, V. Kuliamin, and A. Petrenko. *UniTESK Test Suite Architecture*. FME'02: Formal Methods Europe. LNCS 2391, Springer-Verlag, 2002.

[9] http://www.ispras.ru

[10] A. Kamkin. *The UniTESK Approach to Specification-Based Validation of Hardware Designs*. ISoLA'06: The 2^{nd} International Symposium on Leveraging Applications of Formal Methods, Verification and Validation, November 2006.

[11] IEEE 754-1985. *IEEE Standard for Binary Floating-Point Arithmetic*. NY: IEEE, 1985.

[12] http://www.open-vera.org

[13] http://www.systemverilog.org

[14] S. Edwards. *Design and Verification Languages*. Technical Report, New York, Columbia University, 2004.

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