Selected Papers from: Communications & Information Technology 2008, Circuits, Systems and Signals 2008,
Applied Mathematics, Simulation, Modelling 2008, Marathon Beach, Attica, Greece, June 1-3, 2008

On one approach to random testing of MIPS microprocessors

Igor Gribkov, Alexey Zakharov, Petr Koltsov, Nikolay Kotovich,
Alexander Kravchenko, Alexander Koutsaev, Andrey Osipov, Ildar Khisambeev

***Abstract*—-** Stochastic testing is one of the most powerful tools for verification of hardware design. To simplify the generation of random tests and to create a tool for testing processors of the MIPS64 architecture, the system named INTEG has been developed. Here we give a description of the system's main components, as well as a review of the principles of random testing, implemented in INTEG. Finally, some information on the practical application of INTEG for verification of a new MIPS processor, is given.

***Keywords*—** verification, stochastic testing, MIPS64, RTL model, random test generation.

I. Introduction

The increasing complexity of recent microprocessor systems makes the verification process a significant part of the whole hardware design cycle. All available verification tools are usually used at any level of the project development process [1]-[2]. Among these tools, the stochastic testing is proved to be one of the most effective ones. During the last two decades a wide range of random test generators has been created, including special testers of processor subunits [3]-[5] and universal systems for comprehensive testing of the project's architecture [6]-[11]. Unlike the self-testing code methodology, the random testing requires at least two models of target device. For example, verification of processor model written in VHDL language involves running random tests on a Register Transfer Level (or RTL) model and on an Instruction Set Simulator (or reference model). The results of two runs are compared to detect and analyze possible differences.

Early test generators were based on a biased pseudorandom generation scheme. To increase the coverage of tests, the test generator should use a target device model of high consistency. In recent years technology has shifted towards generation schemes driven by solving constraint satisfaction problems (CSPs) [4], [12]. But constraint-based formulation of the generation task is still a difficult problem.

A more simple approach is to improve the biased scheme by the amendment of the hardware model, the enlargement of the biases, as well as enhancement of simulation data analysis. This approach is basic for the integrated system named INTEG. The system has been developed in NIISI RAN and used for testing MIPS64 architecture microprocessors.

The INTEG system comprises several components in combination. For creating and editing the test templates the graphic user interface (GUI) has been created. The random test generator Tergen forms the core of the system. It generates various tests out of templates, aiming to simulate the situations important to verification of the processor being tested. The integrated environment (control shell), also named Integ, runs the whole system and keeps the results. The comparison module analyses the results and detects the differences.

The paper is organized as follows. In the next section we discuss some common requirements for the random test generation. Then we describe the technique of test generation implemented into our system. Section 4 contains a thorough description of the system's components. Finally, in Section 5 we give some results of random testing a new microprocessor of the MIPS64 family.

II. Test GENERATION REQUIREMENTS

Random testing approach is based on design of test templates that cover certain functional features. The random portion in these templates may be augmented through the progress of testing, thus making the test area wider. The moment when the testing is complete is determined by coverage analysis, statistics of detected bugs and by the overall amount of successful tests.

The generated test code must meet some inherent requirements. First of all, the tests must be valid, that is, their behavior should be well defined by the specification of the system been verified. The test code must also be of high quality, in the sense that it should expand the coverage of the verified system and focus on potential bugs. For a consistent comparison of results, the code must not use arbitrary runtime data. Finally, the exceptions that are not prearranged are not allowed during the test execution. It is assumed that these conditions are assured by the test generator.

Manuscript received May 12, 2008.

Scientific Research Institute for System Studies, Russian Academy of Sciences (NIISI RAN), Nakhimovskii pr. 36/1, Moscow, 117218, Russia (e-mail: koltsov@ niisi.msk.ru, koutsaev@niisi.msk.ru).

In the process of test generation it is sometimes desired to know the state of the target device, i. e. to do the generation complemented by simultaneous simulation. This possibility is useful, but sometimes limited (e.g., for the tests containing loops), complicated for implementation and not needed permanently. In the INTEG system, a simplified approach to simulation is accepted, implying that the contents of a register may be either known or undefined (after an arithmetic operation or in the beginning of a loop). If needed, the generator updates the contents of registers and protects from writing to registers with valuable contents.

The test generator must ensure a maximal possibility for the free random selection. That is, random selection of a parameter covers its entire domain, unless restricted due to special requirements. The domain may be split into segments with given probabilities, i. e. the distribution of such parameter is piecewise-uniform.

If any errors are found in the template, the test generator has to output a corresponding message and correct the error, if possible (usually by canceling an invalid setting).

III. THE TECHNIQUE OF RANDOM TEST GENERATION

A. Templates and Random Selection

A test template is a specification for a random test code. The test generator produces a large number of distinct well-distributed code instances that comply with the user's specification. The variation among different instances is achieved through a large number of random decisions made during the generation process. This results in increasing the test coverage and the probability of detecting bugs.

A template includes a generation scheme and assignments of test parameters. It is created by user, just like an application program. But while an application program has to do certain actions over input data, the template has to result in a test code that can achieve the desired processor states when running. Thereby, the properties of test code have to be embedded into the template. For the test code, the template describes memory distribution, initial data and the code generation scheme.

Random decision is a basic operation of test generation. The test generator uses a sequence of pseudo-random values that is defined with only one initial value, the "seed". By default, a new seed value is selected automatically in each run of the test generator. An explicit assignment of the seed allows one to generate an exact copy of the test code. This can save space, keeping the templates rather than the test code itself.

In random decision, the following typical situations can occur: (a) To select an integer value within given boundaries one has to get a uniformly distributed pseudo-random value, and to normalize it to the boundaries. (b) To select an item from a given list, it is necessary to assign biases to the elements. The sum of the biases is considered to be the bound for a non-negative random integer value, and the problem is reduced to the previous one. (c) To select an item from a tree structure, one has to assign biases to all tree nodes. An independent selection is made as before, on all levels, from the top node, until the end node is met.

In the rest of the section we enter into the details of program description language, the settings and other components of the test template.

B. Program Description Language

When building a test code the test generator interprets the operators which comprise the test scheme. This scheme defines some general properties of the test code, such as sequence of code blocks, content of instructions, and the arguments' values. In fact, the test scheme allows one to form a sequence of instructions with given probabilistic features. To achieve the desired states of the verified system, the user has to arrange random selection of the instructions' arguments, i. e. to set probabilities for all types of arguments. This task is described below.

In the INTEG system, the operators of the program description language include insertion of instructions, assignment of selected arguments, and control constructions. Further we describe in detail the operators and the features they provide.

Inserting instructions. Each of these operators inserts instructions into the test code. The number of instructions inserted by one operator is selected randomly within given limits. Depending on the operator, the instructions may be either directly specified or selected (from list or tree), as shown in the next table.

Name	Source of instructions
Code	A 32-bit code is specified directly
Random	Global instruction tree structure
InsItem	List of instructions - see *ForIns* below
Group	Local instruction tree structure
Instruction	Instruction name is specified directly

Each instruction of the target processor is accompanied with an *Instruction* operator that inserts it. This operator is often used to generate non-random code.

The *Code* operator is used to insert a given operation code, i. e. any 32-bit number. This operator allows one to bypass validity checking for testing the error handling .

The *InsItem* operator inserts a current instruction from the list of the corresponding *ForIns* iterator. Both operators are described below.

Instructions' arguments assignment. All or some arguments of instructions may be assigned in the test scheme. The user can specify the arguments by position (operator *Args*) or by name (operator *Xargs*). Both cases use the assembler syntax of an instruction. An argument value may be set with a constant, a register name or a symbolic name.

For example, the operator *XArgs (rs=$reg=0x100)* assigns the value of symbolic name *$reg* to the register argument *rs*. As the result, this register will be preloaded with the value 0x100 just before the execution of an essential instruction.

This assignment is applied to all instructions inserted by the next operator.

Control constructions. These operators provide iterative generation, runtime loops and macro calls.

Each iterator includes a nested sequence of operators, or its body. The *Repeat* iterator simply repeats its content N times, where N is a given constant. The *ForVal* and *ForReg* iterators specify a variable name and a list of values or register names, respectively. The variable name is replaced by a current value or name from the list. It is used for assignment of instructions' arguments.

The iterator *ForIns* specifies a variable name and a list of instructions. The variable name labels a nested InsItem operator. The latter inserts a current instruction from the iterator's list. A random shuffling of the list can also increase the coverage of the test.

The *ForIns* iterator can be linked with any nested *InsItem* operator. This can provide sequences like "each instruction after each other". Here is an example of a template fragment that generates such a sequence.

```
ForIns ($i1, 0, ADD, ADDU, SUB, SUBU) {
  ForIns ($i2, 0, SLL, SRL, SRA, SLLV, SRLV) {
    InsItem ($i3)
    InsItem ($i2)
  }
}
```

The *Loop* operator adds a runtime loop. It includes a nested sequence of operators that generate the loop body instructions. The loop count is selected randomly. The loops may be nested.

Macros are the templates which are designed to perform a certain action or to reach a given state of the target system. The *Mcall* operator is used for calling a macro from a given file. Macro calls may be nested.

An important feature of a macro is its own context. Like other templates, a macro can arrange probabilities for random selection and other generation parameters. Thus macros provide means for dynamic context changes. For example, one macro can fill the queue of used registers or addresses with certain data, while another macro can use the contents of the queue in the further actions.

In some situations special macros may be called or used automatically. For example, the unused instructions between branch instruction and target address are inserted from the bnt (branch-not-taken) macro. This macro name is specified in the template.

C. Template Settings

The behavior of the test generator is defined by template settings. The collection of settings is large enough, because it includes biases and boundaries for random selection. For convenience the settings are organized in a tree structure.

The template settings specify the probability distribution for instruction arguments of all types. Together with direct setting of arguments, it can be used for transitions to a desired state of the verified system. The details for the selection of arguments are given below.

Integer numbers generation. The cases for random selection include uniform distribution, end points, some characteristic values, and some bit patterns.

Floating point numbers generation. Different characteristic values are set for single and double precision, while the bias values are usually the same. The fraction is selected first. In most cases the exponent is also selected. The cases for selection include zeroes, infinities, NaNs, end points, normalized and denormalized numbers.

Addresses and conditions for branches. A new target address must point to a vacant part of a code memory area. The target address is represented as N*B+D, where the block byte size $B = 2^n$ and the offset $D < B$ can be selected randomly. For conditional branches the condition may be either selected randomly, or accepted as is. The selected condition is forced through the register's contents.

Data addresses. In MIPS64, the effective data address results as a sum of two values defined by the instruction arguments. The data address has to be selected if it is unknown or invalid. A valid address must point to a data memory area that is compatible with the read or write operation and data type of the instruction. Besides, a correspondence of data types is desired for the instruction and the memory areas.

Unlike code addresses, the data addresses are reusable. Such reuse of data addresses is suitable for testing memory paging and cache. The test generator can randomly select between new and used data addresses. The new data address is selected similar to the code address.

When a data address is selected, it has to be mapped on the instruction arguments, i. e. base register and offset, or base and index registers. The task for immediate offset is simple, but often base and index registers must be preloaded. This preload breaks a continuous sequence of load/store instructions. However, it is sometimes desired to keep the continuous sequence. To avoid breaking, the test generator has a special mode, which protects the base registers from random writes.

Registers. For a register argument value, its random selection depends on the register's type and operation (read or load). For example, read-only registers cannot be selected randomly as destination ones, though this case can be set explicitly. Reuse of registers is also of interest, so the generator can randomly select between new and used registers.

Many instructions accept one register for all arguments. Usually this case is of special interest and can be selected randomly after the first register argument is defined.

Special macros. To assign some actions to a certain situation, it is convenient to link it with a special macro name. The special macro is called after the linked state is reached. The examples are the above-mentioned macro *bnt*, and macros *Start* and *Final* that produce the beginning and the end of test code. Each of these names corresponds to a setting that defines the macro name.

Control registers. For each control register, the random selection for read or write operations can be enabled or disabled. By default all reads are enabled, all writes are disabled.

D. Memory and Registers

Memory distribution. The template may specify the memory areas, intended for code or data. The user can set permissions to allow read and/or write operations in the data memory area. A preferred data type is defined for any data area.

A macro can define its own memory areas for code and/or data. During a macro call the macro memory replaces or complements the memory of the template.

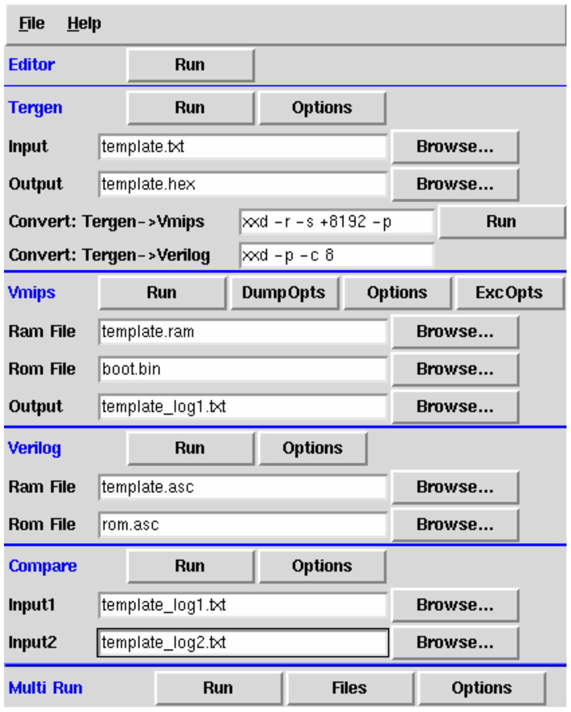
Registers. The initial contents of any register may be specified in the template. A register may be reserved in template or macro. Reserving a register inside a macro is not seen outside it.

IV. INTEG COMPONENTS

For practical implementation of the above facilities an integrated system INTEG was developed in NIISI RAN. It includes an integrated environment and a visual template editor. The system allows one to prepare templates, generate test code and run it on available simulators (Vmips and RTL model). The system includes the following components.

- Integrated environment (or control shell) Integ;
- Visual template editor GUI INTEG;
- Random test generator Tergen;
- A behavioral processor model Vmips, written in C language;
- Register transfer level (RTL) representation of the tested processor, written in Verilog language;
- A program for analyzing the results.

Any of the system components can be run separately, but the control shell ensures some level of correctness for them. The control shell provides a simple interface using menus, buttons, dialog boxes and file selection windows. The main window of the shell [Fig 1] displays general arguments for the components, while additional parameters are specified in the pop-up windows. Running the system requires only a few actions, because the system handles all routine actions itself.



Input template.txt
Output template.hex
Convert: Tergen->Vmips xxd -r -s +8192 -p
Convert: Tergen->Verilog xxd -p -c 8
Ram File template.ram
Rom File boot.bin
Output template_log1.txt
Ram File template.asc
Rom File rom.asc
Input1 template_log1.txt
Input2 template_log2.txt

Fig. 1. Main window of the control shell

In addition to the interactive use, the control shell allows the user to create a batch task for a sequence of tests in background mode. Also the control shell can save the testing environment and reload it in later sessions.

The visual template editor GUI INTEG provides a visual toolset to create and edit the templates through the drag-and-drop technique. The editor provides access to all facilities of the template, as well as to apparent hierarchic representation of the data.

As mentioned above, the random test generator Tergen transforms the template into the test code, i. e. into an image of target processor's memory. The test generation has been described in detail in the previous sections.

The system uses a simulator of target processor Vmips (reference model) based upon the model for MIPS32 processors described in [13]. The RTL-model of the target processor, written in Verilog is a part of the processor project.

Finally, the program for analyzing the results compares log files of the simulation (the outputs of Vmips and RTL models). In case of a mismatch, it tries to find its reason.

V. RESULTS

The concept of random testing implemented in INTEG has found its application in verification of a new microprocessor of MIPS64 architecture family. Our system is developing in parallel with the design of the microprocessor itself. During this period the tests have become far more complicated. From one side, this fact reflects the growth of the potential of INTEG, from the other side it is driven by the requirements of the microprocessor design process (on the later stages of microprocessor's project more complicated tests are needed to verify it). In this section we give some statistics about the testing and show some examples of detected bugs.

To get a better idea about our testing procedure, consider a certain template which we run on a regular basis, and after each detected error we correct the project. Our experience shows, that the dependence of the number of errors detected by this template as a function of time can be represented as on Fig. 2.

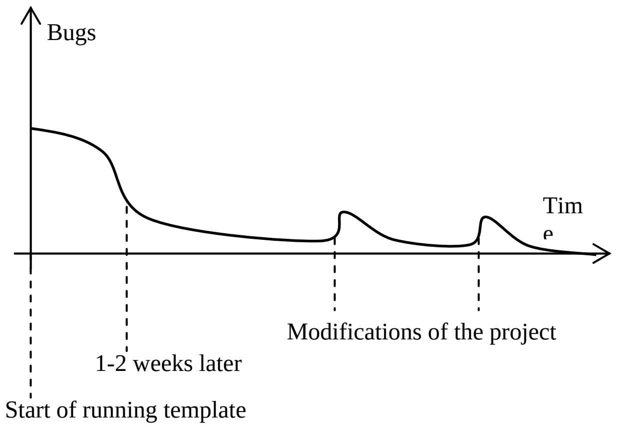


Fig. 2. One template: bugs in time (qualitatively)

The overall amount of bugs equals the sum of bugs detected by each template. This sum is expected to decrease slowly (or even to be maintained at the same level) by the following three reasons. First, new templates for regular use appear. Second, the project is permanently modified and each functional modification implies new potential bugs. Third, the amount of machine resources for testing is growing.

The overall number of bugs detected for the last three years is presented on Fig. 3. The errors are divided into four categories:

1) arithmetic calculations (ALU+FPU);
2) memory management (load/store instructions, cache, TLB);
3) exceptions, interrupts, system coprocessor;
4) instruction execution sequence (instruction buffer);

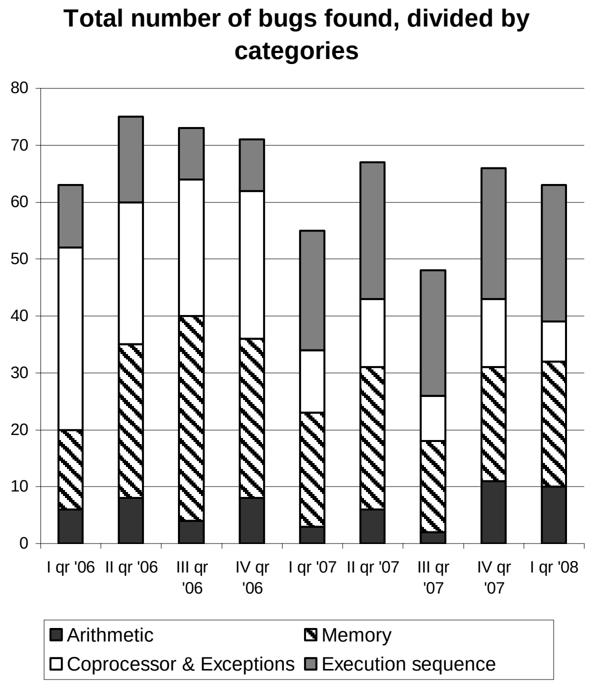


Fig. 3. Total number of bugs detected from 2006

This picture shows the shift of accent in our testing. Namely, from the tests studying the exceptions we shifted towards the memory tests and during the last year we have been focused on the instruction execution sequence tests.

Finally, we give two examples of errors detected by random tests.

1. In one of our tests the following combination of instructions has been generated:

```
lw    r1,0(r2)
div   r3,r1
```

The hung up happened during its execution. The error happened because of a conflict between two consecutive requests to the **r1** register, for writing and for reading.

2. In one later test the following situation occurred:

A request to the level 2 cache memory for the next instruction has been generated. At the same time, an exception from one of the previous instructions has happened. In such a situation the instruction request should be cancelled until the exception is handled. Nevertheless, these events were synchronous, so request was not cancelled correctly. As a result, wrong instruction has been fetched.

Note that the first of the above situations can be easily tested manually. But the second situation happened as a result of coincidence of several factors which cannot be generated directly in the program. Creating a similar test manually requires a profound knowledge of the details of the project's design and takes a lot of effort and time.

VI. DISCUSSION AND CONCLUSIONS

The integrated system INTEG has been developed and used in NIISI RAN for verification of a new MIPS64 microprocessor. The system has demonstrated its value for the microprocessor verification, with such advantages as:

- High performance of test code generation;
- Insuring the validity of test code and the reproducibility of results;
- Utilizing the known target processor state data;
- Managing the probabilities of corner cases by using queues of registers and addresses;
- A convenient interface for templates creation and for testing.

Note that on the later stages of the RTL model development process, the errors become more complicated. By “complicated errors” we mean situations when the model shows erroneous behavior under the influence of combination of several factors. Some of these factors can be given directly in the test program: instruction combinations, values of operands of instructions, contents of memory blocks, etc. But far more often, the error factors cannot be given directly in the test code (for example, cache memory and TLB states, RTL signal values). Besides, it is often difficult to understand the mechanism of influence of such factors on an error, and take them into account in advance, in writing non-random tests.

The approach implemented in INTEG allows one to reproduce these combinations of factors by randomly choosing the parameters of the test template. Thus, the key feature of our testing system is that instead of reproducing the concrete situation fraught with error, we prepare the conditions under which this situation, alongside with other ones, can occur. This feature can be considered as both the source of advantages and disadvantages of our approach. For example, while preparing the test template, some technical details can be skipped (the test generator will take care of them), thus making the test creation process more simple. However, the understanding of the reason for the detected error can be often difficult, because it occurs as a result of influence of unpredictable factors. This understanding requires a deep knowledge of the designed project. Another disadvantage of our approach is that in the process of testing the uncertain situations may occur. For example, suppose that the test template is prepared to study some potential bug, and after a certain number of runs no error has been detected. In this case, three choices are possible: to continue running the template until the error occur, to modify the template and launch it again, or to stop the testing and assume that the error does not exist in the current project. Making the right choice completely depends on the user’s experience and intuition.

Further development of the system should focus on model-based schemes and improvement of interface. In particular, we plan to develop a database of testing knowledge which handles core verification task and can be extended to include specific implementation testing knowledge. As for the methodology of testing, the concept of test coverage metric has proved its significance for the whole software testing process (see e.g. [1]-[2], [9], [14]-[15]). We plan to realize this concept in future versions of our system.

REFERENCES

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

[2] J. Bergeron, *Writing Testbenches using SystemVerilog*. Springer, 2006.

[3] D. Wood, G. Gibson, and R. Katz, "Verifying a Multiprocessor Cache Controller Using Random Test Generation," *IEEE Design & Test of Computers*, vol. 7, Aug. 1990, pp. 13-25.

[4] M. Aharoni, S. Asaf, L. Fournier, A. Koifman, R. Nagel, " FPgen – A Test Generation Framework for Datapath Floating-Point Verification," in *Proc. IEEE International High Level Design Validation and Test Workshop (HLDVT03)*, 2003, pp. 17-22.

[5] C. Jacobi, C. Berg, "Formal Verification of the VAMP Floating Point Unit," *Formal Methods in System Design*, vol. 26, no. 3, 2005, pp. 227-266.

[6] A. Aharon, A. Bar-David, B. Dorfman, E. Gofman, M. Leibowitz, and V. Schwartzburd, "Verification of the IBM RISC System/6000 by a Dynamic Biased Pseudo-random Test Program Generator," *IBM Systems Journal*, vol. 30, no. 4, 1991, pp. 527-538.

[7] R. Velazco, A. Assoum, H. Ziade, "SYNAM : A software tool for microprocessor random testing," in *Proc. of the 6th Int. Conference on Microelectronics (ICM' 94)*, September 1994, pp. 144-147.

[8] M. Kantrowitz and L. Noack, "Functional Verification of a Multiple-issue, Pipelined, Superscalar Alpha Processor — the Alpha 21164 CPU Chip," *Digital Technical Journal*, vol. 7, no. 1, 1995, pp. 136-143.

[9] Michael Kantrowitz, Lisa M. Noack, "I'm Done Simulating; Now What? Verification Coverage Analysis and Correctness Checking of the DECchip 21164 Alpha microprocessor," in *Proc. of the 33rd Annual Conference on Design Automation (DAC'96)*, 1996, pp. 325-330.

[10] W. Kruijtzer, V. Reyes, W. Gehrke. "Design, synthesis and verification of a smart imaging core using SystemC" *Design Automation for Embedded Systems*, vol. 10, no. 2-3, 2005, pp. 127-155.

[11] *RAVEN Software User's Manual*, Obsidan Software, 2003. Available: http://www.obsidiansoft.com/files/manual.pdf

[12] E. Bin R. Emek G. Shurek A. Ziv. "Using a constraint satisfaction formulation and solution techniques for random test program generation," *IBM Systems Journal*, vol. 41, no. 3, 2002, pp. 386-402.

[13] Brian R. Gaeke. "VMIPS Programmer's Manual,"Fourth Edition, for version 1.3, 2004. Available: http://www.dgate.org/vmips/doc/vmips.pdf

[14] A. Gargantini, E. Riccobene., "ASM-Based Testing: Coverage Criteria and Automatic Test Sequence Generation," *Journal of Universal Computer Science*, vol. 7, no. 11, 2001, pp. 1051-1068.

[15] S. Devadas, A. Ghosh, K. Keutzer, "An observability-based code coverage metric for functional simulation," in *Proc. of the 1996 IEEE/ACM Int. Conference on Computer-aided design*, 1997, pp. 418-425.