INTEG: A Stochastic Testing System for Microprocessor Verification

I. V. GRIBKOV, A. V. ZAKHAROV , P. P. KOLTSOV, N. V. KOTOVICH,
A. A. KRAVCHENKO, A. S. KOUTSAEV, A. S. OSIPOV, I. S. KHISAMBEEV
Scientific Research Institute for System Studies, Russian Academy of Sciences
(NIISI RAN)
Nakhimovskii pr. 36-1, Moscow, 117218
RUSSIA

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. The system has demonstrated its merits in verification of a new microprocessor. 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.

Key-Words: - Hardware verification, stochastic testing, biases, MIPS64, random test generation.

1 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]. 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 [2] and universal systems for comprehensive testing of the project's architecture [3-5]. 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 a 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) [6]. 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. Finally, we focus on the system's testing tools.

2 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.

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 with 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 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 uses all its 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 in the template, the test generator has to output a corresponding message and correct the error, if possible (usually by canceling an invalid setting).

3 The Technique of Random Test Generation

3.1 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 tested states when running. Thereby, the attributes of test code are to be put into the template. The descriptive means for the templates are designed to reach this purpose. 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 element 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 element 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 template components.

3.2 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 connected 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.

3.3 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.

3.4 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.

4 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.

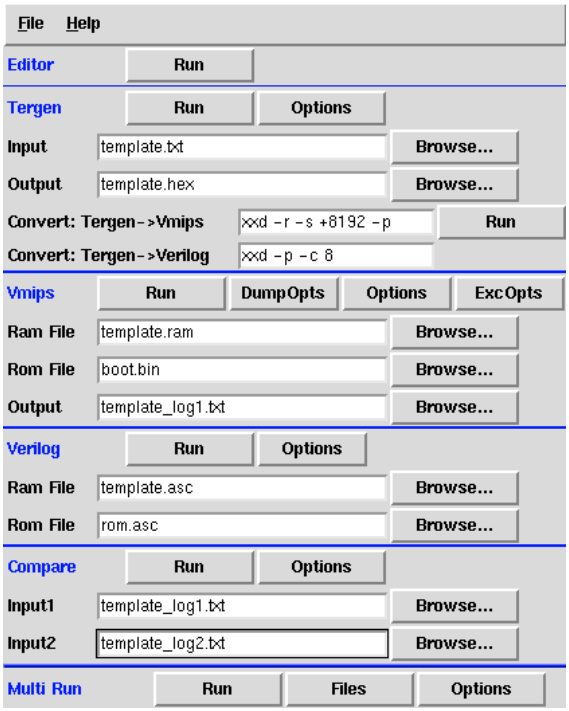

Fig. 1. Main window of the control shell

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 gives access to all template's facilities, 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 based upon the model for MIPS32 processors described in [7]. 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 differences, it tries to find the reason.

5 Conclusion

The integrated system INTEG has been developed and used in NIISI RAN for verification of a new MIPS64 microprocessor. The system has demonstrated its power 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.

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 to the methodology of testing, the concept of test coverage metric has proved its significance for the whole software testing process [1]. 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] D. Wood, G. Gibson, and R. Katz, "Verifying a Multiprocessor Cache Controller Using Random Test Generation," *IEEE Design and Test of Computers,* August 1990, pp. 13-25.

[3] 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.

[4] 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.

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

[6] 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.

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