DOI 10.31114/2078-7707-2019-2-25-31

Ristretto: Random Test Generator for Multicore Microprocessor Cache Coherence Verification

A.V. Smirnov, P.A. Chibisov

Scientific Research Institute of System Analysis (SRISA RAS), chibisov@cs.niisi.ras.ru

Abstract — **Modern system-on-chip designs contain multiple computational cores with several levels of caches, as well as a sophisticated memory subsystem. Functional verification of multi-core microprocessor models is known to be a big challenge. There are different approaches for memory subsystem and cache coherence controllers verification but an automated functional test generation strategy is the most commonly used in the industry.**

In this paper, the technique of automated multi-core test generation is proposed. It can be applied for cache coherence and memory subsystem check in a top-level multi-core RTL-model simulation. Moreover, the presented test generator can be very effective in generating test scenarios for FPGA-prototypes of SoC being designed. In this paper, we also give a detailed description of the random test generator itself and the capabilities of generated test cases.

The proposed test generator got its name Ristretto due to the similarity of the word Ristretto with the abbreviation formed from the words "random instruction sequence" (RIS), and the word "threads" (and because ristretto is so concentrated and intense).

Some self-checking validation approaches are suggested to obtain correct responses in FPGA-based verification (post-silicon validation). In the paper, we also discuss the bug-masking problem in post-silicon random instruction tests that arises due to limited observability.

Keywords — **multicore microprocessor, pseudorandom tests generation, functional verification, RTL-model, cache coherence, false sharing, memory subsystem, post-silicon validation, self-checking, bug masking.**

I. INTRODUCTION

Nowadays, dynamic functional verification techniques (so-called simulation-based testing) for RTL models verification are widely used in the industry. Random automatic test generation is one of the critical parts in the traditional verification flow. There are special combinatorial random test generators focused on a given specific microarchitecture aspect. Such test generators solve user-defined constraint satisfaction problem to build a test suite. However, on the one hand, it is an extremely time-consuming task to develop above-mentioned generators, and, on the other hand, the errors which can be found by generated tests are too specific or possibly irrelevant. This happens because combinatorial random test generators are expected to cover the restricted class of problems.

It is well known that the combinatorial complexity of memory subsystem grows together with the number of computational cores per microprocessor chip. It is critical to prove both the proper functioning of the processor core stand-alone and the correct operation of the system controller with cache coherency modules, as well as to prove, by all means, the proper functioning for all the blocks as a whole.

In this paper, we propose the technique of automated multicore test generation for functional verification of cache coherence and memory subsystem. Moreover, we describe test generator (Ristretto) that can be very effective in generating test scenarios for FPGA-prototypes of SoC being designed. Test cases are created that consist of random combinations of load/store instructions to generate necessary stimulus. Adjustable load and store sequences are supposed to be sufficient memory access instructions for multicore microprocessors testing, for pre-silicon and post-silicon stages of verification. The generator tries to use all possible scenarios of interactions between the instruction streams in order to accomplish the required functional coverage. Furthermore, we analyze the concept of stress-functions (thread irritators).

Additionally, we are addressing some self-checking methods, their limitations and accuracy. At the same time, we would like to emphasize the importance of bug-masking problem for post-silicon stage of verification. The ability to evaluate the bug-masking rate of a test provides us some valuable information and, that is important, the opportunity to select effective tests for regression, as well as to find experimentally a well-balanced number of self-checks during the one test.

While designing the generator discussed in the publication, we have analyzed the existing approaches to the verification by means of random instruction test generators, such as [1], [2]. Moreover, we have especially examined experience of ARM Corporation [3] and IBM [4] in the research area.

II. RANDOM TEST GENERATOR DESCRIPTION

Ristretto random test generator is focused on test cases for memory subsystem (MMU, TLB, all levels of caches, cache coherency, data prefetch buffers and so on). It can create random tests for multicore KOMDIV architecture (which is MIPS-like). The generator source codes have been written on PERL or C++ language. User-defined scenario file can be read by the generator as an input configuration, the output of the generator is an assembly test program.

Test cases are created that consist of random combinations of load/store instructions to generate necessary stimulus. Every processor core executes its own distinct instruction stream ("thread" from the programmer's point of view). All test threads execute simultaneously. Moreover, threads can share some memory resources to initiate interactions on the system bus and coherency transactions on the bus. Memory areas can be "shared" (common) or "private". Memory configuration can include "read-only" and 'write only" areas as well to increase test complexity. A simplified memory configuration example for the two-core microprocessor is illustrated in Fig.1. The direction of the arrows shows the data flow direction.

Any memory area which is allowed to be overwritten by more than one core can contain unpredictable values because the ordering of writes from different cores cannot be identified. As a consequence, data from "write-only" memory regions cannot be verified. The following method is suggested to solve the problem of checking data in common memory areas.

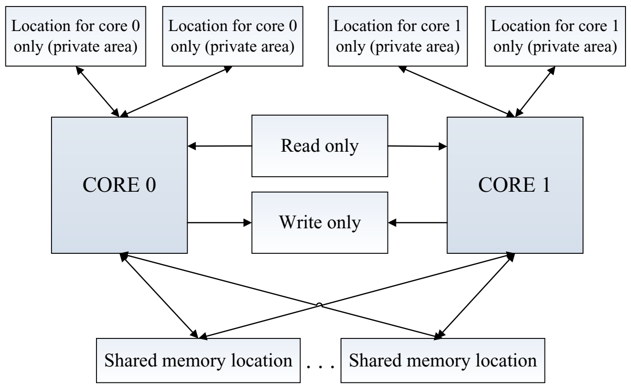
Location for core 0 only (private area)
Location for core 0 only (private area)
Location for core 1 only (private area)
Location for core 1 only (private area)
CORE 0
Read only
Write only
CORE 1
Shared memory location
. . .
Shared memory location

Fig. 1. Example of memory regions configuration for the dual-core processor

A test program consists of predefined instruction subsets and is executed on every microprocessor's core (from a programmer's point of view test code can be referred as a number of execution threads). Most of the instructions are memory access instructions, which can cause cache hits or cache misses. Several memory regions are reserved for every processor core available. The size of every region is usually equal from 3 to 5 cache line sizes. We will call this set of memory areas "global" memory map. During the initialization procedure, every memory area is filled with random values by its "own" (designated) core. Minimal and maximal numbers of regions are pointed out in the configuration file.

Any test consists of the given number of the independent test sections (subtests or test cases, each of them is being executed in the same time interval). One time interval corresponds to one test case, which consists of concurrent test threads. To ensure that every test case begins simultaneously on all of the processor cores, it is necessary to synchronize their instruction stream execution.

For every time interval (test section) a random subset of memory regions from the "global" memory map is chosen during test generation. Let's call these selected memory areas "local" memory map. The test generator tries to create memory access instructions in these selected areas. An example of memory allocation is illustrated in Fig. 2.

The other areas (which are marked grey or white) can be called "memory elements".

The main idea of such memory maps is to deal with memory areas configuration to create false sharing patterns in the test cases. False sharing is a well-known issue on SMP systems. It occurs when more than one processor core writes to the same cache line, however, not at the same location. Each simultaneous data update of individual elements located in the same cache line by different cores leads to entire cache line invalidation. If an updated element is used by only one core, all other cores have to update their cache lines, even though these updates are logically independent of each other. Such contention among shared resources is called false sharing.

As it was said before, every memory region is divided into several memory elements. Memory element size is chosen randomly, but in such a way that each region should have more than two elements. Each memory element corresponds to a subset of read and write instructions with the maximum possible data size. Moreover, for the selected instructions some random "equivalent" instructions are generated with less data width (for example, 1 store double = 2 store word = 4 store half = 8 store byte; 1 load double = 2 load word = 4 load half = 8 load byte). Thus, the test generator fills every memory region with different memory access instructions.

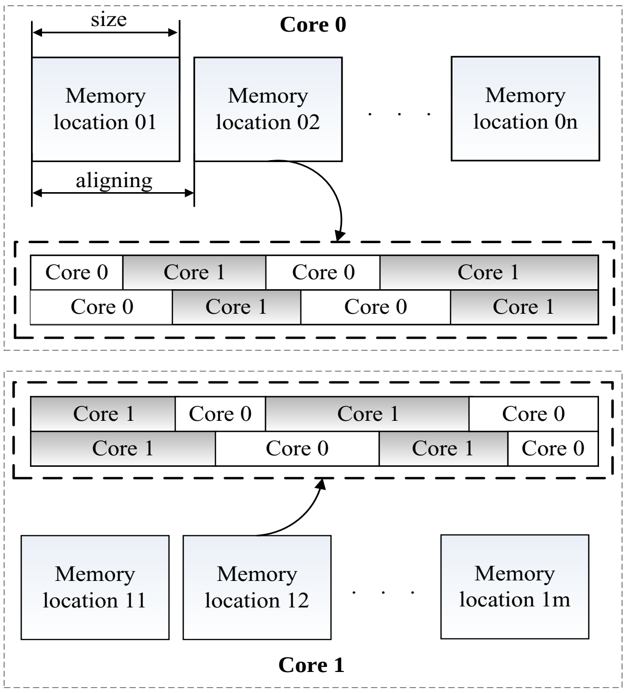

Fig. 2. Example of memory region splitting based on attribute: which core has to deal with that region

While the test is being generated, two (or more) memory regions are randomly selected (they can match on occasion). Some instructions can be purely random; some of them can be put in increment or decrement order. If equal start addresses of the memory areas are selected by the test generator, some addresses on different processor cores will point to the same cache line with some probability (this situation is considered as «false sharing»). The generator selects a virtual address to physical address translation, initializes general purpose registers, tables of data in memory and generates values for self-check for every memory region.

An example of test architecture for several time intervals with detailed memory regions mappings in the two-core configuration is shown in Fig.3. In the time interval "0" every processor core executes initialization procedures for their "own" memory regions independently. In the time interval "1" both cores interact with each other and produce read and write accesses to the only one memory region (however, in different bytes - according to the scheme described above, where the subtest has been generated for the given "local" memory map).

As a result of such memory representation inside any single cache line, any memory region will contain some deterministic values and, thus, memory content can be validated at the end of the time interval. In the time interval "2" an example of the simultaneous interaction of the test thread with a stress-function is introduced. In the time interval "3" every processor core accesses its own memory region in isolation, however, in these conditions all previous history and cache state accomplished before the time period are taken into account in a natural way. This code fragment will be executed twice. Self-checks of various types are performed at the end of each time period as well as at the end of the test (the number of checks can be configured by the user).

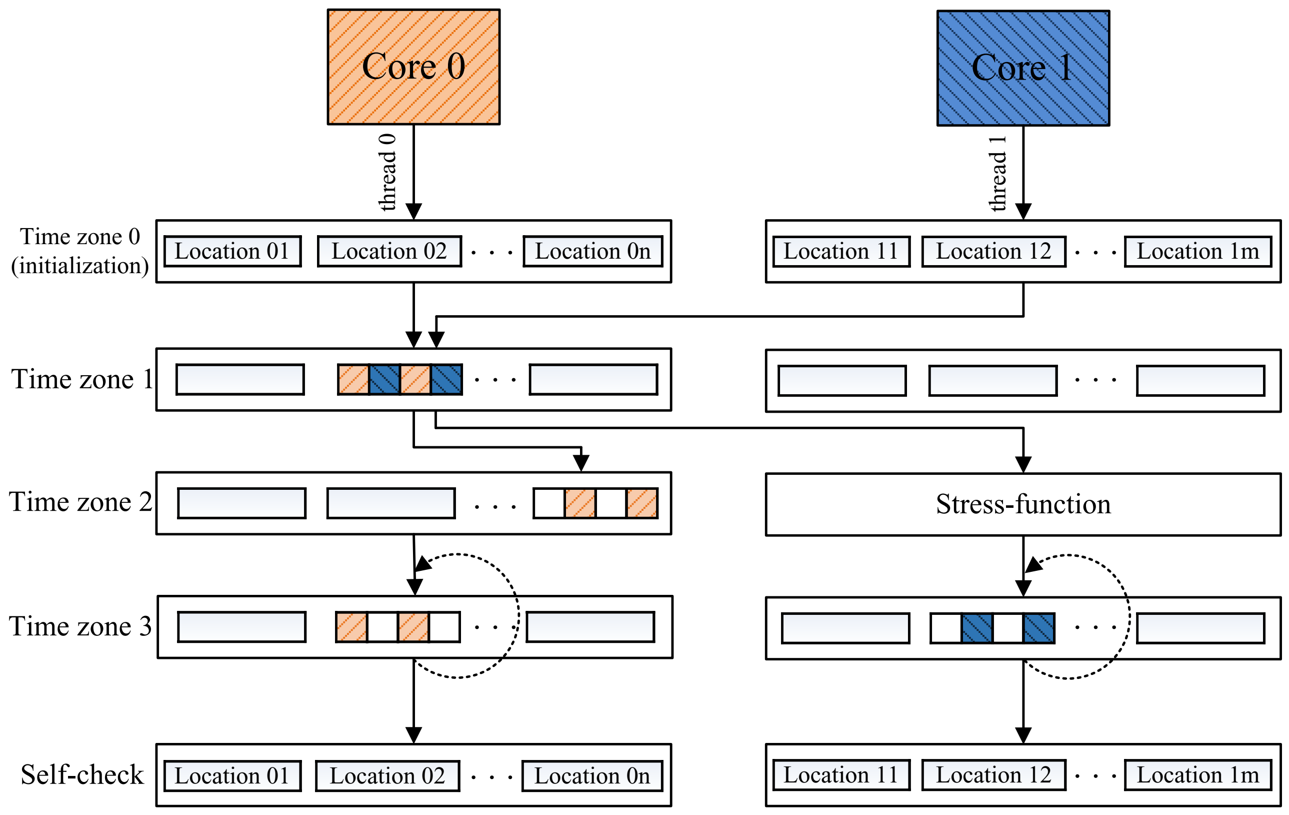

Fig. 3. Dual-core test architecture with detailed memory allocation description

To configure the test generator, the user can edit the configuration file, which contains the following parameters:

- the number of available processor cores in the system under test;
- the minimum and the maximum amount of memory regions for each core;
- cache size for all levels of cache, cache associativity;
- cache line size;
- the probability of memory regions alignment to the L1/L2 cache size;
- the number of subtests;
- memory region minimum and maximum sizes;
- arranged/random memory access offsets ratio;
- subtest re-execution probability;
- false sharing memory patterns probability;
- the number of instructions per subtest (min/max values);

- the frequency of stress functions occurrence per test;
- the number and types of self-checks during test execution.

The test is generated using the configuration file as a template. The purpose of the template for this type of test generators differs from one of the traditional template-based pseudorandom test generators. Here the template is not a task for test cases generation and it does not contain high-level test description or test specification written in pseudocode. It can only set weights of instruction groups, the correlation between user-defined functions, stress-functions and purely random code, the frequency of self-checks, memory allocation, processor modes and so on. Moreover, the user can define samples (or fragments) of stress-functions and acceptable degree of their randomization.

III. Test case description

Ristretto test generator creates random test sequences of memory access instructions for multicore microprocessors, for pre-silicon and post-silicon stages.

A stream of instructions for every microprocessor core is generated. Synchronization of the streams is achieved by macros. Every subtest (or all test) can be repeated with a given probability. Generator helps to improve inter-core interaction testing efficiency by a considerable diversity of test situations. In a first test run, most accesses have a large part of cache misses, while in the next run of the test they will be hits.

As the main purpose of the generator is the verification of interactions between the memory subsystem components, the following requirements are applied to the code generation.

The generator tries to use all possible scenarios of interactions between the instruction streams in order to accomplish the required functional coverage. This can be done by taking into consideration such shared resources as cache-lines and physical memory pages.

During one subtest execution, the following variants of shared memory dependencies are possible.

1) Shared memory usage. Unlinked memory locations are used in subtest when all pseudorandom memory accesses are treated as a stress test on the whole system;

2) False sharing. Randomly selected microprocessor cores (two or more) interact through access to multiple cache lines. In this case, the memory accesses are made in different, mismatched bytes in order to preserve the determinism of the values;

3) True sharing. Processor cores access the same memory addresses. The order of data changes is controlled by synchronization mechanisms, i.e., memory writes do not occur simultaneously, but in different time intervals. Between these time intervals the cores are synchronized;

4) Nondeterministic true sharing. The order of memory write operations from different cores does not need to be traced, because data in the write-only memory area is not subjected to validation.

In terms of one core the test situations are automatically created by the generator considering some microprocessor instruction dependencies. These dependencies between memory accesses or between instruction pipeline stages for one core can be distinguished in these test situations:

- RAR (read after reading)
- RAW (read after write)
- WAR (write after reading)
- WAW (write after write)

The stress test is created on translation lookaside buffer operations (TLB), on a variety of buffering data devices, prefetching data stream buffers, data buffers, all levels of the cache memory and also mechanisms of exceptions situations.

Thread synchronization is required in order for every test iteration to start at the same time. The synchronization procedure can be organized in any available way, software or hardware. Among the threads synchronization methods which are commonly used in multi-core systems, there are such methods as inter-processor interrupts, message passing through special registers of processor interconnect (mailboxes). The mechanism for transferring control between the cores, implemented through atomic operations such as "read-modify-write" can also be provided.

Ristretto test generator supports several synchronization mechanisms, which are programmatically a set of library primitives. In particular, the library includes a function to organize the critical sections in the test code that are necessary for the timing separation of different threads accesses to shared resources which can be used by only one core in a given time interval, for example, the memory area with the true data sharing

IV. Stress-functions

In publication [4] the concept of stress-functions (thread irritators) was introduced for simultaneous multithreading testing in IBM processors. Three methods of improving the test program generation were proposed: stress-function (thread), merged threads and replication of threads. This approach was adapted and successfully applied in the test generator discussed in this article. The generator can reach some rare test situations that can find errors on prefetching intelligent data buffers and mechanisms for rearranging and merging writebacks in memory. Since randomized instructions are not enough, it is necessary to insert special stress functions into the test from time to time instead of a purely random flow of memory access instructions.

The idea of the approach is associated with the introduction of the stress function concept as follows. Let us assume that there is some memory access instructions thread (original, primary thread) running on the first microprocessor core. A given number of stress functions are run on one or more of the remaining cores instead of other original threads. The stress functions consist of repetitive memory accesses. Their purpose is to create memory traffic and make stress on the memory controller.

A stress function can be a short cycle or a sequence of identical memory access instructions. The stress function interacts with the main thread in order to increase the degree of interactions between microprocessor cores at the microarchitectural level. The result of the approach is that the probability of finding errors in the pre-silicon and post-silicon stages of the designed microprocessor increases. Examples of errors are a hang, livelocks and deadlocks, logical errors in the memory subsystem blocks, as well as errors associated with the incorrect or untimely updates of the cache or memory states.

Also in publication [4], an approach of merging threads is proposed, which is based on the fact that the test program generator creates fragments of single-threaded tests for each core and then connects them to a multithreaded test. The process of test fragment building must ensure that a single-threaded test does not change the shared memory areas when it will be parallelized across multiple threads. Then, multithreaded tests are repeatedly created by random selection of single-threaded fragments from a ready-made set with their subsequent merging. In this approach, due to merging threads, memory access instructions from each single-threaded test fragment can be executed multiple times, each time in combination with different test actions from other threads. Random combinations of single-threaded tests reproduce unique situations, significantly changing the testing model, as well as increasing the probability of finding hard-to-detect microarchitectural errors.

In the thread replication approach, the test generator creates single-threaded tests and then replicates multiple copies of the test to multiple threads, simulating a multithreaded scenario. The user configures the generator so that the test can be run in multiple threads. This is achieved by the restriction that no two memory access instructions can have access to the same memory cell. In this case, when threads are replicated, you can get a deterministic result at a run time coinciding with the predicted value which has been obtained during test generation.

Interesting multithreaded scenarios can be created as a result of thread replication because two or more threads execute the same sequence of instructions and the same resources are used.

The automated generation of stress functions is also possible by the proposed Ristretto test generator, however, further research on the effectiveness and feasibility of the approaches on merging and replication of the threads is required.

V. SELF-CHECKING METHODS

There are two variants of the test generator implementation. First one was developed for the automatic generation of the tests targeted at memory subsystem and cache coherency verification while debugging RTL-model of the designed SOC. Concurrent test simulation on the RTL under test and on the instruction set simulator (ISS) is one of the known methods of RTL-model verification. After the test has finished, the results obtained on the RTL-model and ISS are compared. If a mismatch is found, the block that caused incorrect behaviour which has led to the bug is tracked down. An additional advantage of this approach is that it makes possible to detect the error immediately, exactly at the position where it occurred with an accuracy of processor instruction.

The second variant of the generator implementation is intended to be run on FPGA-boards (post-silicon validation stage) and it creates tests with embedded self-checks. This implementation does not need to have an external instruction set simulator. The execution time of one test decreases by several orders of magnitude compared to the RTL modelling time [5]. However, an internal state of the microprocessor (the same is true for FPGA-prototype) cannot be accessed, therefore diagnostic information about the origin of the bug is extremely difficult to obtain [6].

The reason for this situation is that an effect caused by error can be detected only a million cycles after it has occurred due to low observability. The more important thing is the considerable time consumption for engineers to debug such erroneous behaviour. The main question in this variant of generator implementation is the selection of criteria for a proper test. Method of test validation for such tests should be selected as well [7], [8].

There exists an approach, called "multi-pass consistency check" [7]. According to this method, each test case runs several times. The results of the first pass are considered as reference results. Each next pass is compared with the reference results. The check includes a comparison of general purpose registers and some memory regions for every pass. A number of restrictions and special conditions while generating test guarantee the consistency:

1) instructions with probable unpredictable result are not included in the test instruction stream;

2) any "write-after-write" conflicts in known memory areas are forbidden because it is impossible to predict the order of conflicting store operations of that kind;

3) any data written in permitted local resource must be the same in every test pass.

Despite these limitations, nevertheless, it is useful to generate these mentioned unverifiable events, because they increase the stress on memory subsystem and help to find bugs which cannot be found by other methods [5]. Also in [7], it is suggested to generate a slightly modified test code for the second and subsequent passes; these implemented modifications should not affect the test results. Moreover, it is recommended to change threads priority in order to change resource or instruction pipeline dependencies while the test is being re-executed.

Additionally, an alternative approach, so-called Reversi, has been examined [8]. The Reversi test generation system creates pseudo-random test programs in such a way that the reference final outcome is known during the test generation process. There is no need to run the test on the golden model to obtain valid results. A key observation which provides the basis for the Reversi development is that the majority of instructions in a processor's ISA have counterparts, i.e., operations whose functionality has a corresponding inverse instruction. For example, let us consider addition and subtraction, load and store and so on. Accordingly, an error can be detected if initial and final microprocessor states do not coincide after execution of the direct and the reverse test fragments. In [9], it is suggested to duplicate every processor instruction and compare the results of all such pairs to control the correctness of the test.

One more approach proposing a self-checking technique (so-called "ISA diversity") for pseudo-random tests is presented in [10]. The idea is to construct an enhanced random instruction test in addition to unmodified random instruction test and compare their execution results. According to the authors, in major contemporary ISAs, more than 75% of instructions can be replaced with equivalent instruction or instruction sequence. The equivalent instruction sequence is a sequence that produces the same response as the original processor instruction. The most time-consuming task in this technique is to analyze the instruction set architecture (ISA) and identify the extent of ISA in the microprocessor under test. All processor instructions can be classified into three categories.

1) Full equivalence: instructions for which there are one or more equivalent ways to realize their operation. This category includes most of the arithmetic and logic instructions, data transfer instructions, and a large number of control flow instructions.

2) Partial equivalence: instructions for which there are partially equivalent ways to execute their operation. For example, floating point instructions can lose accuracy.

3) No equivalence: instructions with no equivalences. This category of instructions cannot be replaced. This category includes mainly the privileged instructions that access system resources and some others.

VI. Estimation of bug masking probability

The approaches to self-checking tests generation described above do not allow to indicate precisely the place where an error has occurred. Besides, post-silicon validation is inherently limited by internal signal observability, which impacts the ability to diagnose and detect bugs. The error can be masked and remain undetected during the test's execution. The example of bug masking which occurs when the bug becomes undetectable after some other processor instructions have been executed since its manifestation is shown on Fig.4.

An analysis of bug masking probability provides an opportunity to select effective tests for regression, as well as to find experimentally a well-balanced number of self-checks during the one test. This analysis should be done in the way to avoid oversimplification of test cases and to maintain the nonzero probability of finding hard-to-find bugs.

In [11], a software tool is proposed which makes possible to assess the pool of potentially unrevealed bugs. Moreover, it allows the user to select effective tests with the smallest masking effect for the high coverage regression. Additionally, one more approach is proposed which offers the possibility to make modifications in test cases code in the way of minimization for buggy values propagation. This tool is expected to increase the bug rate due to mask-preventing code instrumentation.

```
LW    R2,0x45(R3)
>>> R2=0x12E5
/* Loading Incorrect Value - ERROR! */
. . . . . . . . .
LB    R2,0x3F(R5)
>>> R2=0xB7
/* Bug Masking */
. . . . . . . . .

Registers checking, including R2:
(R2 = = 0xB7) ? If true – error has been masked.
```

Fig. 4. Example of bug masking during the test with self-check

Bug masking probability analysis is still actual if self-checking is performed using the results obtained from the external instruction set simulator or even the results from the internal reference model. The source code of the generator with the internal reference model can be written on C++ for the purpose of running the generator itself and generated tests on FPGA-platform or post-silicon platform.

VII. Conclusions

It is widely accepted that automatic test program generation is a fundamental approach to microprocessor functional verification, including, in particular, memory subsystem and cache coherency mechanisms. In the paper, we presented a technique of automated multi-core test generation. The proposed test generator Ristretto is expected to be very effective in generating test scenarios for RTL-models and FPGA-prototypes of SoC being designed, i.e. for both pre-silicon and post-silicon stages. In fact, there is no need to have reference responses from instruction set simulator, in contrast to the previous multicore test generation technique which has also been developed at SRISA RAS [12].

To reach a high level of test coverage in a reasonable time, Ristretto generator performs all different scenarios of the interaction between processor cores from the point of view of data dependencies.

Moreover, some self-checking validation approaches are suggested to obtain reference responses in FPGA-based verification (post-silicon validation). In the paper, we also discuss the bug-masking problem in a post-silicon random test that arises due to limited observability.

References

[1] Hudson J., Kurucheti G. A Configurable Random Instruction Sequence (RIS) Tool for Memory Coherence in Multi-processor Systems. Workshop on Microprocessor Test and Verification, 2014, pp. 98-101. DOI: 10.1109/MTV.2014.26

[2] Venkatesan D., Nagarajan P. A Case Study of Multiprocessor Bugs Found Using RIS Generators and Memory Usage Techniques. Workshop on Microprocessor Test and Verification, 2014, pp. 4-9. DOI: 10.1109/MTV.2014.28

[3] S. Thiruvathodi and D. Yeggina, "A Random Instruction Sequence Generator for ARM-Based Systems," 2014 15th International Microprocessor Test and Verification Workshop (MTV), Austin, TX, USA, 2014, pp. 73-77. doi:10.1109/MTV.2014.20

[4] Ludden J.M., Rimon M., Hickerson B.G., Adir A. (2011) Advances in Simultaneous Multithreading Testcase Generation Methods. In: Barner S., Harris I., Kroening D., Raz O. (eds) Hardware and Software: Verification and Testing. HVC 2010. Lecture Notes in Computer Science, vol 6504. Springer, Berlin, Heidelberg.

[5] Wisam, K., et al., “Improving Post-Silicon Validation Efficiency by Using Pre-Generated Data,” Proc. Intl. Haifa Verification Conf., pp. 166-181, 2013.

[6] Satish Kumar Sadasivam, Sangram Alapati, Varun Mallikarjunan: Test Generation Approach for Post-Silicon Validation of High-End Microprocessor. DSD 2012: 830-836.

[7] Allon Adir, Amir Nahir, Avi Ziv: Concurrent Generation of Concurrent Programs for Post-Silicon Validation. IEEE Trans. on CAD of Integrated Circuits and Systems 31(8): 1297-1302 (2012).

[8] Ilya Wagner, Valeria Bertacco Post-Silicon and Runtime Verification for Modern Processors, Springer, 2011, 224 p.

[9] Lin, David; Singh, Eshan; Barrett, Clark; Mitra, Subhasish. / A structured approach to post-silicon validation and debug using symbolic quick error detection. International Test Conference 2015, ITC 2015 - Proceedings. Vol. 2015-November Institute of Electrical and Electronics Engineers Inc., 2015.

[10] Nikos Foutris, Dimitris Gizopoulos, Mihalis Psarakis, Xavier Vera, and Antonio Gonzalez. 2011. Accelerating microprocessor silicon validation by exposing ISA diversity. In Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-44). ACM, New York, NY, USA, 386-397.

[11] Doowon Lee, Tom Kolan, Arkadiy Morgenshtein, Vitali Sokhin, Ronny Morad, Avi Ziv, Valeria Bertacco: Probabilistic bug-masking analysis for post-silicon tests in microprocessor verification. DAC 2016: 24:1-24:6.

[12] Grevcev N.A., CHibisov P.A. Podhod k stohasticheskomu testirovaniyu RTL-modelej mnogoyadernyh mikroprocessorov (A Practical Approach to Verification of multicore Microprocessor models) // Problemy razrabotki perspektivnyh mikro- i nanoehlektronnyh sistem. 2018. Vypusk 2. S. 52-58. (in Russian).