Software-Based Test Integration in a SoC Design Flow

Alexandre M. Amory, Leandro A. Oliveira, Fernando G. Moraes

Faculdade de Informática - Pontifícia Universidade Católica do Rio Grande do Sul (PUCRS)
Porto Alegre - Brazil - ZIP Code: 90619-900 - E-mail: {amory, laugusto, moraes}@inf.pucrs.br

Abstract

Hardware-based test imposes some constraints like area overhead and processing speed degradation. In addition, with the advance in hardware integration, system-on-a-chip (SoC) test activities using only automatic test equipments (ATEs) results in an expensive option. The main objective of this work is to investigate and evaluate pros and cons of software-based test. Techniques, such as pattern compaction based on LFSR reseeding are used to minimize some of the drawbacks. Another objective of this work is to introduce a Computer Aided Test (CAT) tool used to generate a C code for testing core-based SoCs. The benchmark c6288 are used to evaluate the proposed tool and test techniques.

1. Introduction

In recent SoC based systems the amount of test data transferred between automatic test equipments (ATEs) and devices under test is becoming too large. Even expensive state-of-the-art ATEs restrict the SoC test, as a result of limited memory resources, narrow channel bandwidth and low speed.

One known approach to overcome ATE limitations is to use built-in self-test (BIST [2]) to generate patterns and to analyze the results at speed. This approach reduces the ATE constraints and the test cost. However, BIST has some drawbacks [3]: (i) some circuits are resistant to random patterns, resulting into a low fault-coverage; (ii) since new modules are inserted into the system, the total area, operation frequency and power consumption are negatively affected.

Software-based test is an alternative approach to BIST and ATE. Currently, SoCs have at least one embedded processor, special-purpose cores, busses and memories. In this approach, embedded SoC processors are used to test the cores. Since there are no new test modules added to the system, figures as area usage, speed and power are not changed. A possible drawback of the software-based test, when compared to the hardware-based test is the longer time needed to apply the patterns and/or analyze the results [3][8][9]. A trade-off between hardware and software-based test is a function of the SoC requirements. For instance, in a situation where at-speed test is required, hardware-based test may be employed instead of the software-based one.

This work has two main objectives. The first one is to investigate the pros and cons of software-based against hardware-based test, and to analyze the trade-off between hardware/software test implementation. A case study is used to validate the investigation. The second objective is to introduce a Computer Aided Test (CAT) tool developed in order to generate the test code and the necessary IP core wrappers.

This paper is organized as follows. Section 2 presents the state-of-the art in software-based test. Section 3 presents the target SoC architecture. Section 4 details the CAT environment developed. Section 5 focuses on the results and the partition between hardware and software test components. Section 6 concludes this paper and presents some directions for future work.

2. Software-Based Test

Software-based test can minimize some of the BIST drawbacks discussed before. The following is a list of advantages of this approach:

- Ease to reuse and to modify the test strategy as it is implemented in software. This leads to a greater flexibility and programmability to the test engineer, which can easily increase the fault coverage of a device by running different test algorithms, for example;
- No specific test controller is required, since a generic embedded processor is responsible for the test control and execution;
- Reduced (or none) area overhead due to the test patterns generation and response compaction implemented in software;
- Reduced (or none) speed/power degradation, as there are no additional test modules;
- Can be applied to test processors, memories, general cores and interconnection (bus).

Possible drawbacks of the software-based test are:

- The SoC must have an embedded processor;
- Extra memory needed to store the test program and the deterministic test patterns;
- Increased test time when comparing to hardware-based test. BIST usually generates patterns/ compact responses in just one clock cycle. However, in software-based test patterns are provided by the processor to the cores, taking longer to execute the same task than BIST modules. On the other hand, software-based test may be faster than ATE based test due to the limited bandwidth.

2.1. Processor Test

The first works on software-based test for processors were conduct in the 70's [13]. Software-based self-test of processors is a challenging issue since: (i) the number of different hardware modules to be tested, such as ALU, registers, control, etc; (ii) the limited number of pins to access the processor; (iii) the existence of modules that can not be accessed directly, as interrupt controller, flag, exception handler; (iv) the huge variation of implementations and architectures.

Thatte and Abraham [13] propose processor functional models, abstracting the architecture to increase the flexibility of the approach. The most common approach to self-test processors is based on the execution of a sequence of random instructions [6][13]. These techniques present low fault coverage because processors have modules that resist to random pattern [3] and the test time is usually very high. An alternative approach proposed by different authors such as [3][4][8][11][12] is a detailed choice of the instructions to test the processors leading to reduced test time and higher fault coverage.

Lai and Cheng [8] propose to add test specific instructions to reduce the size of the test program, the test time and to increase the controllability and observability of some critical modules. Two processors were used to validate the approach. In both cases, it reaches 100% of fault coverage reducing the test program and test time in 20%. The area overhead induced by the test instructions was 1.6%. Shen and Abraham [11] present a tool to automatic generate functional test of processors. This tool can be used in manufacturing test and functional validation. The tool was validated on an 8051 processor with fault coverage of 90.2%. The test program was generated in 3.1 minutes and the simulation time was 4.1 hours. Corno et. al. [4] used a genetic algorithm to generate a test program. The fault coverage obtained was 85.19% against 80.19% of the random instruction approach.

2.2. Memory Test

Memory test is very simple to be implemented by a processor due to its regular structure. Well-know algorithms [14] using deterministic patterns, e.g. 55h and AAh, are used. Tehranipour *et. al*. [12] use a processor to run an assembly code of a March-like algorithm to test embedded RAM. Zorian and Ivanov [15] consider a ROM as a combinational circuit, which the address is the input and the memory content is the output. An exhaustive test is performed, reading all memory contents, compacting the output. A MISR and a 1's counter are used to reduce the alias probability.

2.3. Core Test

The embedded processor executes a specific program to test each core of the SoC. This program has the following functions: (i) generate the test patterns; (ii) send these patterns to the cores through the communication path (e.g. bus); (iii) read the test responses; (iv) compact these responses. Since it is a software-based test, it is possible to use different algorithms to test each core.

The access to the inputs and outputs of the core is a function of the SoC architecture. In a bus-based architecture, for example, the processor can access the core directly through the bus. In architectures that the processor does not have direct access to the core, there is a need for Test Access Mechanisms (TAM).

The test program can emulate the behavior of a LFSR and MISR in software, generating pseudo-random patterns and compacting responses. A disadvantage of emulating a LFSR or a MISR in software is the increasing test time, since it may need several clock cycles to produce one pattern. Others approaches using adders, subtractors and multipliers to generate patterns and/or to compact responses can be used. In [10] it is shown that these modules can be used to substitute LFSR, generating comparable fault coverage, with the advantage to generate a pattern for each instruction (e.g. add).

Hellebrand et al. [5] highlight the greater flexibility using software to test circuits. With this flexibility, it is possible to explore different LFSR configurations, such as multiple seeds and polynomials. This flexibility also enables to find the best pattern generator configuration to each core of the system. The LFSR configurations analyzed are the simple LFSR, the LFSR with multiple polynomials and the LFSR with multiple polynomials and multiples seeds. It is shown using ISCAS benchmarks that each module reaches higher fault coverage when different LFSR configurations are used. This higher fault coverage obtained from pseudo-random patterns reduces the number of deterministics patterns, reducing the necessary memory to store them. For example, the fault coverage of the c7552 module with a simple LFSR is 95.79%. Using a LFSR with multiple polynomials and seeds, the fault coverage attains 98.87%, reducing the number of deterministics patterns to be stored in the memory.

The architecture presented by Papachristou [9] has 3 modules: the processor, the test memory and a low-cost tester. The test process was divided in 2 phases: the memory load and the test. The tester uses direct memory access to load the memory. The processor performed the test reading the memory. In this way, these two phases can be executed in parallel, reducing the total test time. One interesting conclusion of this work is that the test time remains practically independent from the tester frequency. A simple 50 Mhz tester was used and it was observed that the test time depends only on the processor's frequency.

3. SoC Arquitecture and Tools

This work targets SoCs employing a bus to connect a processor to memory and cores. The proposed approach is prototyped in FPGAs using the Excalibur™ environment [1]. Excalibur has been chosen due to the prototyping CAD tools available in this environment. The set of tools includes C compiler, core integration

facilities, co-simulation, temporal validation with a functional model of the Nios™ processor and real timing verification. These tools allow easily investigation of different test methods.

Figure 1 presents the target SoC architecture. A PC is used as the external tester, responsible for loading the test program and deterministic patterns into the memory, internal or external to the FPGA, according to the number of test patterns. Nios32 [1] is the embedded processor, and the 32-bit Avalon™ bus is used to connect the SoC components.

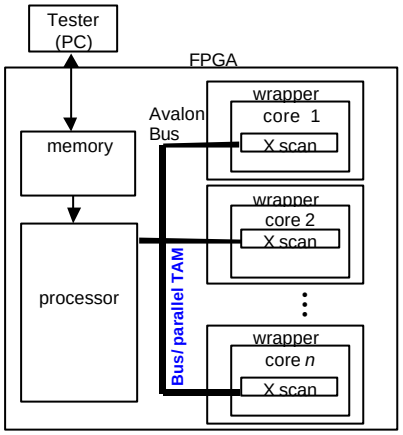

Figure 1 – Target SoC architecture.

Each core connected to the bus is wrapped with a P1500 interface. Using this approach, the test procedures implemented in the processor can be applied to all cores under test. Cores with multiple scan chains are also supported.

To evaluate the proposed test method and the CAT tool, the c6288 benchmark is prototyped in this environment.

4. CAT Tool Environment

The CAT environment has four main functions: (i) generates the test program in C language; (ii) parameterizes the wrappers; (iii) inserts scan chains when necessary; (iv) generates the SoC interface integrating the cores.

Figure 2 and Figure 6 present a design flow divided in two parts: the core integration into the SoC and the test code generation.

The core integration flow starts with a tool called *core generator* - Figure 2(a). This tool does not generate the core itself, but extracts the core interface (input/outputs) and may insert scan chains when required. Scan chains are inserted using the DFTAdvisor™ tool (Mentor). The interface extraction phase generates a file like the one presented in Figure 3. Some of the extracted data are: (i) the name of the core; (ii) the type of the core - *core*, *processor* or *memory*; (iii) the files composing the core; (iv) the top entity; (v) the ports of the top entity and generics, if they exist. Each port of the top entity is defined in one line and the last information inserted in this line is the port usage (e.g. *data*, *clock*, *reset*, *enable*, *external*). This is a key piece of information, since it enables the *SoC generator* tool to automatically generate the wrapper of this core and the SoC interface.

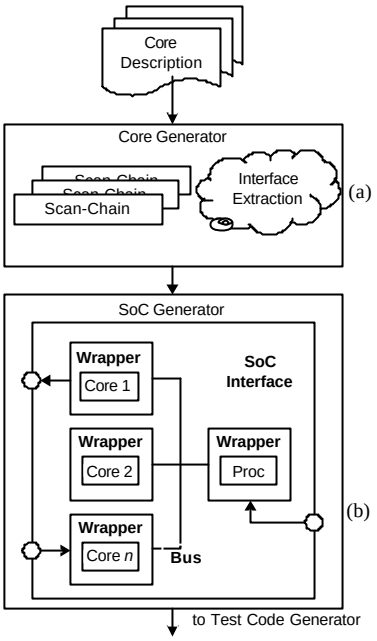

Figure 2 – SoC generation CAD flow.

```
NAME s1196
TYPE Core
FILES 1                          port
s1196.vhd                        usage
TOP s1196                          ↓
PORTS 4
PINP in std_logic_vector 0 to 13 data
OUTP out std_logic_vector 0 to 13 data
CLK in std_logic clock
reset in std_logic reset
```

Figure 3 – Example of core project file.

The next step is the SoC generation, Figure 2(b). The following actions are executed during this step: (i) selection of the cores and the processor; (ii) selection of the communication media; (iii) wrapper synthesis for each core; (iv) SoC interface generation.

The cores are selected from a library created in the first step (*core generator*). After that, the user defines the communication media, defining in this way the communication protocol. Different communication media can be easily integrated to the environment, but presently only bus-based is allowed.

Once the communication protocol is defined, the wrapper is automatically synthesized for each core. Wrappers are generated after the communication media selection because the communication protocol and the wrapper interface are associated to the media.

Figure 4 presents an example of wrapper. The wrapper has shift-registers adapting the bus width (32-bit) to the core input/output widths. In this example, the input width of the wrapped core is 68, and the output width is 42. The test is executed in two steps: patterns loading and response reading. The six least significant bits of the *addr* port are used to set the test mode, i.e. pattern loading or response reading.

When the wrapper is in test mode and a write operation is required (*wr* port), the internal *clockEnb* is asserted during one clock cycle to process the incoming test patterns. Note that the *clockEnb* assertion occurs only when the last part of the incoming pattern is written into the input shift-register.

The responses are stored into the output shift-registers one clock cycle after *clockEnb* assertion. The processor reads the responses when the *rd* port is asserted.

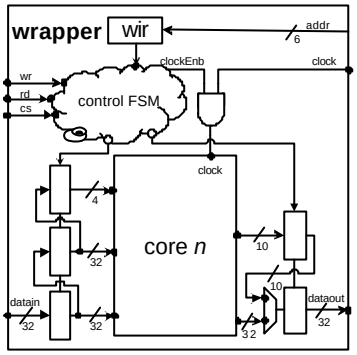

Figure 4 – Wrapper scheme.

At this step of the flow, two files are created for each core: *wrapper* and *testbench*. This *testbench* is used to validate the *wrapper*. The test patterns for this *testbench* can be internally generated from a parameterizable LFSR or read from patterns stored in an external file. When read from external files, it can be used to validate the test patterns and generate the signature.

The last step is the SoC interface generation, which is obtained from the data contained in each *core file*. When the port usage is specified as external, it is connected to the SoC interface. The wrapped core, illustrated in Figure 5, has ports connected to the communication interface and external ports connected to the SoC interface. These external ports impose modifications (addition of ports) in the wrapper and in the SoC interface.

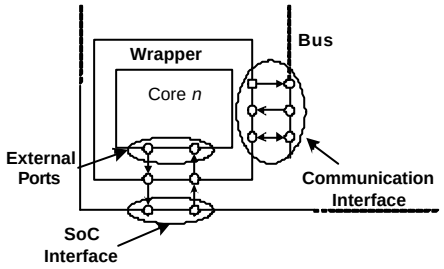

Figure 5 – Types of ports.

Once the SoC is built, the next step is to create the *test code*. The test code generates the test vectors, compacts the responses and evaluates each core. This approach is based on the Hellebrand's approach [5], which uses different LFSR configurations to increase the fault coverage obtained from pseudo-random patterns. Therefore, fewer deterministic patterns are stored in the memory. The main advantage of this approach is the minimization of technological requirements of the external tester, such as bandwidth, memory and frequency. However, the challenge is to find a tradeoff between run-time for pseudo-random test and the memory requirements for deterministic test.

The first action of the *test code generator* (Figure 6(a)) is to select the polynomials and seeds to generate patterns to each core. Multiple polynomials can be selected and each polynomial may have several seeds. The environment supports generic modular LFSR and MISR descriptions.

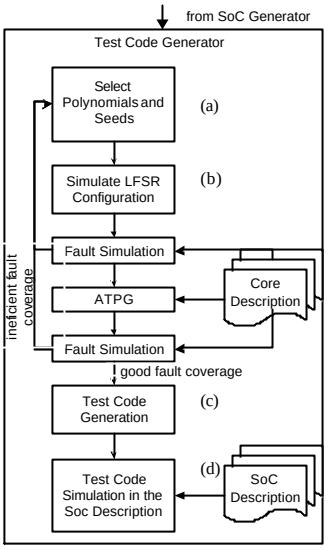

Figure 6 – Test code generation CAT flow.

Each LFSR configuration is simulated, creating the test patterns for each core - Figure 6(b). The generated patterns are translated to the fault simulator format (Flextest™/Fastscan™) and the fault coverage is evaluated. If the resulting fault coverage is not sufficient, the user may run the ATPG tool to generate the remaining test patterns or choose other LFSR configuration, restarting the process. At the end of this process, the user has all the necessary patterns to test each SoC core.

Finally, these patterns (pseudo-random and/or deterministic) are translated to the test program in C language - Figure 6(c). After the code generation, the user can evaluate the required test time using a hardware/software co-simulation tool - Figure 6 (d). The co-simulation tool is provided by the Excalibur environment.

Figure 7 presents an example of pseudo test code. The core under test is memory mapped. This can be observed by the **c6288* pointer, in the third line. The first lines define the expected signature, the deterministic test vectors generated by the ATPG and the LFSR configuration. The *ExecuteTest* procedure is the main test function. For all cores of the system, it applies the pseudo-random patterns and/or the deterministic patterns. Depending on the test time requirements, the test designer may implement the MISR in hardware to speed-up the test, as presented in this example. Each core has an embedded MISR, which is read by the software at the end of the core test. The returned signature is compared against the expected one.

```
#include "nios.h"
#define c6288ExpectedSig 0xABCDEF01
volatile np_usersocket *c6288 = na_c6288;
const vet_det[]={...};
const lfsr_cfg[]={...};
void lfsr( unsigned polynomial, int *state);

void ApplyPseudoRandomPattern(volatile np_usersocket *hwif){
    for each polynomial {
      poly = lfsr_cfg[ind];
      n_seed = lfsr_cfg[ind+1];
      ind+=2;
      for each seed {
        value = lfsr_cfg[ind];
        n_patterns = lfsr_cfg[ind+1];
        ind+=2;
        for each pattern {
             *hwif = value;      // send pattern to UUT
             lfsr(poly,&value);  // generate new pattern
}}}}
void ApplyDeterministicPattern(volatile np_usersocket *hwif){
   for each pattern {
     *hwif = vet_det[i]; // send deterministic pattern to UUT
}}
int ExecuteTest(void){
   ApplyPseudoRandomPattern(c6288);
   ApplyDeterministicPattern(c6288);
   c6288Sig = *c6288;  // read signature from the core
                          under test(UUT) - MISR in hardware
   if (c6288Sig != C6288ExpectedSig)
       abortTest();
}}
```

Figure 7 – Pseudo test code example. LFSR implemented in software (*lfsr* function) and MISR in hardware.

QT™ is used to build the graphical interface of this environment. QT enables to build a multiplatform GUI in C++. The CAT tool generates 'C' and 'VHDL' parameterizable code from pre-validated templates, integrates commercial CAD tools (Leonardo, ModelSim, FlexTest, FastScan, DFTAdvisor) and guides the user through the design and test flow.

5. Results

The goal of this section is to evaluate the software-based test method in the architecture presented in Section 3 using the tools presented in Section 4 (the results obtained depends on the target processor and its instruction-set). The benchmark c6288 is used to perform this evaluation. This benchmark is mapped to the tms035 library, using 1659 gates. The input and output are 32-bit wide. The features evaluated are fault coverage, test time and memory usage to store the test program and deterministic patterns.

Table 1 compares the fault coverage obtained with three different test approaches: deterministic patterns, pseudo-random patterns and mixed patterns. The deterministic approach needs 74 patterns to reach 99.90% of fault coverage, requiring 74 memory words (296 bytes). On the other hand, the mixed approach use just 4 memory words to reach 99.96% of fault coverage. This reduces the memory requirement of the external tester, reducing its cost [5][7].

Table 1 – Fault coverage and number of patterns for three different test methods: deterministic, pseudo-random and mixed test.

Deterministic		*Pseudo-random*		*Mixed*	
Patterns	**Fault coverage**	**Patterns**	**Fault coverage**	**Patterns**	**Fault coverage**
74	99.90	96	99.91	96+4	99.96

The mixed approach was partitioned into four different hardware/software test configurations. In the first partition, the LFSR, the MISR and the control structure used to apply deterministic patterns are implemented in hardware (*hardware partition*). In the second partition, all test modules are implemented in software (*software partition*). The third partition implements the LFSR in software and the MISR in hardware. In the fourth partition, the test patterns are generated using arithmetic functions (*adders)* provided by the processor and the MISR is implemented in hardware [10].

Table 2 presents the total test time for the mixed test approach for the c6288 benchmark using the four partitions.

Table 2 – Total test time for the mixed test approach, in clock cycles;

Hardware	LFSR+MISR software	LFSR software MISR hardware	Adder software MISR hardware
96+4	8237	4739	1374

The hardware partition applies each pattern, pseudo-random (96) and deterministic (4), in one clock cycle.

The software partition takes 8237 clock cycles to complete the test. This difference is due to the overhead induced by the data transfer from the processor to the core over the bus, and the test serialization. Test serialization means that each pattern must be fully processed, i.e., generated, sent to the core, received and compacted. In the hardware approach, these steps are executed in parallel, as a pipeline structure.

The third and the fourth hardware/software test partitions reduce the total test time. The MISR implemented in hardware reduces the traffic in the bus, since the responses are compacted in hardware, sending back to the processor only the signature (see Figure 7). The test time in the last partition was reduced by using a processor specific instruction to generate the pseudo-random patterns. This has an advantage over the LFSR emulation because a new pattern is created with only one instruction, reducing the test time and the test code length [8][10].

Table 3 complements the test time information presented in Table 2. This table presents the number of clock cycles between the generation of two patterns. In the hardware partition, the patterns are generated cycle by cycle. The software partition takes 82 cycles to apply a pseudo-random pattern and 52 cycles to apply a deterministic pattern, due to the test serialization. In the next two partitions, the deterministic patterns take just 19 cycle to be applied since the MISR is implemented in hardware. The most important difference is observed in the pseudo-random test time. The pattern generator using adders is faster, confirming [8], which suggest the use of specific test instructions to reduce the test time and test code length.

Table 3 – Time in clock cycles to apply a new pattern to the UUT.

Pattern	LFSR+MISR software	LFSR software MISR hardware	Adder software MISR hardware
Pseudo	82	46	13
Det	52	19	19

Table 4 compares the total memory usage to store the patterns and/or test code. The first column presents the memory usage to store deterministic patterns, being the test implemented in hardware. The remaining three columns presents the memory usage to store the test program, deterministic patterns and LFSR configuration.

As presented in Table 1 the deterministic approach needs 74 patterns (296 bytes) to test the c6288, and the mixed approach 4 patterns. It is used 40 bytes to store the four deterministic patterns and the LFSR configuration. The values 276, 250 and 186 correspond to the test code length. The test code is in its major part re-usable by different cores (see the control functions *ApplyPseudoRandomPatterns* and *ApplyDeterministic Patterns* in Figure 7), reducing in this way the memory requirements when testing several cores. These data shows software-based test does not required large amount of memory space, as stated in the literature.

Table 4 – Memory usage, in bytes, for the hardware/software partitions of the test modules.

Hardware	LFSR+MISR software	LFSR software MISR hardware	Adder software MISR hardware
296	40+276	40+250	40+186

6. Conclusions

Software-based test minimizes the two main drawbacks related to BIST: performance degradation and additional hardware area. According to the literature, software-based test increases test time and the test memory requirements.

One contribution of this work is the evaluation of these possible drawbacks. Actually, the test time is a drawback, since the test time using only software-based test is much longer compared to the hardware-based test. As shown, the test time is minimized when the test modules are partitioned into hardware and software modules. On the other hand, the test memory requirement for software-based test is not a drawback. A careful design of the test code can enable its reuse to test different cores. It is also possible to reduce the memory requirement using compression algorithms in the determinist patterns, an issue not explored in this work.

The second contribution of this work is the developed CAT environment, integrating software-based test to the SoC design flow. An absent feature, which is being investigated, is a technique to find good seeds for the reseeding test method [7].

As ongoing work, the CAT environment is been validated with a greater subset of ISCAS85/89, and ITC99 and ITC02 benchmarks. Others criteria, such as hardware overhead and power dissipation are also been evaluated.

More information about the tools and the experiments presented can be found in http://www.inf.pucrs.br/~amory/socgen/.

Acknowledgements - Fernando Moraes gratefully acknowledges the support of the CNPq through research grant number 522939/96-1. The authors acknowledge to Marcelo Lubaszewski and Érika Cota to the helpful suggestions during this work.

7. Referências

[1] Altera Inc. “Nios Embedded Processor: 32 bits Programmer’s Reference Manual”. ver 2.1, 2002, 124 p.

[2] Bushnell, M. L.; Agrawal, V. D. “Essentials of Electronic Testing for Digital, Memory, and Mixed-Signal VLSI Circuits”. Kluwer Academic, 2000, 690p.

[3] Chen, L.; Dey, S. "Software-Based Self-Testing Methodology for Processor Cores". *IEEE Transactions on Computer-Aided Designs*, vol. 20-3, 2001, pp. 369-380.

[4] Corno, F.; Reorda, M.S.; Squillero, G.; Violante, M. “On the Test of Microprocessor IP Cores”. In: *Design, Automation and Test in Europe Conference*, 2001, pp. 209-214.

[5] Hellebrand, S.; Wunderlich, H. J.; Hertwig, A. “Mixed-Mode BIST Using Embedded Processors”. In: *Proc. International Test Conference*, 1996, pp. 195-204.

[6] Klug, H. P. “Microprocessor Testing by Instruction Sequences Derived from Random Patterns”. In: *International Test Conference*, 1988, pp. 73-80.

[7] Krishna, C.V.; Touba, N.A. "Reducing Test Data Volume Using LFSR Reseeding with Seed Compression”. In: *Proc. of IEEE International Test Conference*, 2002, pp.

[8] Lai, W. C.; Cheng, K. T. “Instruction-Level DFT for Testing Processor and IP Cores in System-on-a-Chip”. In: *Design Automation Conference*, 2001, pp. 59 –64.

[9] Papachristou, C.A.; Martin, F.; Nourani, M. “Microprocessor based testing for core-based system on chip”. In: *Proceedings of Design Automation Conference*, 1999, pp. 586-591.

[10] Rajski, J.; Tyszer, J. “Arithmetic Built-In Self-Test for Embedded Systems”. Prentice Hall, 1998, 268 p.

[11] Shen, J.; Abraham, J. A. “Native Mode Functional Test Generation for Processors with Applications to Self-Test and Design Validation”. In: Proceedings International Test Conference, 1998, pp. 990-999.

[12] Tehranipour, M. H.; Nourani, M.; Fakhraie S. M. “Testing Embedded Processor and Memory Cores in SOCs”. In: *VLSI Test Symposium*, 2002, pp.

[13] Thatte, S. M.; Abraham, J. A. “A Methodology for Functional Level Testing of Microprocessors”. In: Proc. *International Symposium on Fault-Tolerant Computing*, 1978, pp. 90-95.

[14] Van de Goor, A.J. “Using March Tests to Test SRAMs”. *IEEE Design and Test of Computers*, vol.10-1, 1993, pp. 8–14.

[15] Zorian, Y.; Ivanov, A. “An Effective BIST Scheme for ROM’s”. *IEEE Transaction on Computers*, vol. 41-5, 1992, pp. 646-653.