A BIST Implementation Framework for Supporting Field Testability and Configurability in an Automotive SOC

Amit Dutta, Srinivasulu Alampally, Arun Kumar and Rubin A. Parekhji
Texas Instruments, Bangalore, India.
Email:[amitd,srinu,arun.kumar,parekhji]@ti.com

Abstract
Built-in self-test techniques have been widely researched and adopted for reasons of improvements in test time and test cost, reduction in test resources required for test of large chips with embedded cores, and for field testability. While the adoption of these techniques is becoming prevalent, there continue to be challenges in making BIST solutions comprehensive to meet several design and application constraints. This paper describes the use of BIST implementations for self-test of logic and memories in an automotive SOC, (designed in Texas Instruments (India)), to support field testability. Novel aspects of this solution include (i) programmable coverage for logic, (ii) built-in self-analysis and self-repair for memories, and (iii) support for various system and application level interfaces for field test. It is shown how conventional BIST techniques must be augmented to provide test solutions in a system context.

Keywords: Built-in self-test, built-in self-repair, field test, system test, test interface control, software controlled test.

1 Introduction

The paradigm of one-time manufacturing test may be questioned for three possible reasons: (i) The increasing use of electronic components in safety critical systems where periodic testing, including in the application on the field, is recommended. (ii) The need to check the operating parameters of the device during its use for conformance to specifications, due to design variability in deep sub-micron technologies. (iii) The increasing inability to test for all defect models at time-zero manufacturing tests, and the need to monitor their influence during normal operation. These additional requirements are driven by the need for high dependability and low down-time in safety critical systems. Several design methodologies at the chip and system level have been proposed, leading to online testing, error correction and fault tolerance [1,2,3]. These are based on code-space redundancy, compute time redundancy, and module redundancy, and the associated check mechanisms. Corrective measures are based on this redundancy and check mechanisms, and control the latency period in which recovery is possible. In this paper, the design and implementation for field self-test for logic and memories for an automotive application SOC (system-on-chip), which has been designed in Texas Instruments (India), using built-in self-test (BIST) techniques, is described. While the application of such BIST techniques is well-known [4,5,6], the paper explains various chip, system and application level considerations which must be addressed to provide adequate controllability and observability during field test. The main contributions of this work include: (i) scaleable implementation solutions for obtaining programmable coverage, (ii) built-in self-analysis and self-repair for memories, and (iii) support for various system and application level interfaces for field test. Deterministic BIST with re-seeding is used for logic self-test [7] and programmable memory BIST [6] for memory self-test. It is shown how interfaces to conventional BIST techniques must be augmented, (together with the techniques themselves in some cases), to realize field testability. (Only an off-line periodic test solution with a fixed test schedule is considered in this paper).

The paper is organized into seven sections. Section 2 provides an overview of system level self-test. Section 3 describes the procedure for logic self-test, together with internal and external design considerations. The architecture, and its corresponding implementation, are explained in Section 4. Section 5 describes the procedure, considerations and architecture / implementation for memory self-test. Some device specific information is provided in Section 6. Section 7 concludes the paper.

2 System Level Overview of Self-test

The ability and ease of ascertaining the status of a device, whether it is functional or faulty, within an embedded application, are determined by a few device and system level parameters. These include:

(a) The type of tests performed, namely for static or parametric faults, and the portion of the device covered by the test.
(b) The test schedule, whether the test runs concurrently with normal operation, (interleaved with normal operation or self-checking [3]), or is run off-line non-concurrently, either completely or in parts.
(c) The ability of the external environment to trigger the test at different times, ascertain the duration of the test and time of test completion, and the status of test execution.

Item (a) above is device specific, item (b) depends upon how the application runs on the device, and item (c) depends upon the interface of the device to the rest of the system. Based on these parameters, a few design and test

considerations emerge to successfully implement and support field testability. These include:

(a) Support for power-on condition test through hardware configuration and periodic test through software configuration.
(b) Device specific test configuration, either through test mode hardware or through CPU (central processing unit – or processor) control through software.
(c) Choice of test interface, either through internal device functional or test bus, or through standard external test interface (e.g. JTAG interface), or standard external functional interface (e.g. USB or other host interfaces),
(d) Storage of the hardware or software controlled test configuration either as a fixed setup in a ROM or a variable and programmable setup in a RAM. (Various setup parameters are described together with the implementation in Sections 4 and 6. Examples include input seeds for stimuli generation for logic, algorithms for test of different memory cores and repair analysis, golden signatures for response comparison, etc.).

Figure 1 illustrates the high level interface of the DUT (device under test) to the system. It may be noted that since the CPU may be used for test control, the device configuration may or may not directly permit the CPU and its associated memory to be tested at the same time. Two different cases of self-test, therefore, emerge:

Non-destructive self-test (NDST): Here the test control portion, (e.g. CPU and associated memory, or dedicated test controller), of the DUT is not included in field test. This provides greater flexibility of test scheduling and sequencing, e.g. for periodic online testing, inter-leaved testing, resumption of normal operation during test and

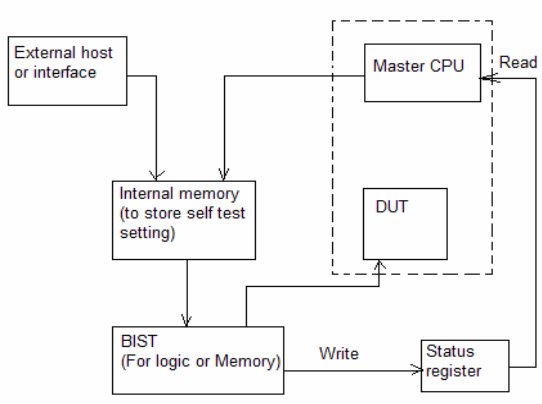

Figure 1: System level diagram for self test.

upon its completion. However, it is at the expense of the loss of coverage of this module. (Normal operation of power control or clock control logic is required for correct test and hence these blocks are almost always left out of the scope of such tests as well).

Destructive self-test (DST): Here the entire device is tested, and as such, the test process is destructive. Recovery to resume normal operation during test or at the end of it is not possible. A standard procedure, e.g. warm reset sequence, must be followed to read and analyze the results of the test and take corrective measures. The state of the DUT is lost, and a re-start of the application, (e.g. CPU boot), is required. (In this device, the self-test implementation is destructive).

3 Design for Logic Self-test

There are various ways to successfully implement logic self-test in a device. The generic self-test procedure, along with generic design and application level considerations, are presented here.

3.1 Self-test Procedure

The procedure for carrying out logic self-test in the field is based on a common software and architectural protocol. This is illustrated in Figure 2. The steps are explained below.

Reset control: Upon a successful power-on and reset sequence, the CPU operation is activated. Its bootstrap values are loaded by latching values on the device input pins upon an external reset, or through the default boot control register configuration. As part of this boot sequence, the self-test mode is checked. If enabled, self-test is executed internally as part of this power-on sequence. Self-test can also be executed through an external interface (e.g. JTAG, USB, etc.), after the CPU is halted. Both the internal or external modes of self-test operation can also be invoked through application code either from ROM or RAM. (In this design, a RAM based solution has been implemented).

Status register check: In the self-test routine, the completion of status register is first checked. If it is set, then the results of the previous (just completed) self-test operation are read. If not, it is an indication to begin a new self-test operation. This check prevents the device from entering into a self-loop, and de-couples the internal self-test operation and application code.

Self-test configuration: The association data and control for self-test are set, (fixed in ROM or loaded through CPU or external interface into RAM), in various configuration registers. These include different seed values, corresponding pattern count numbers and resulting golden signatures, and at-speed PLL clock control settings for different test modes. Such a variety in the configuration allows different coverage numbers to be

targeted through different tests (stuck-at, transition, etc.) through varying number of seeds and pattern counts.

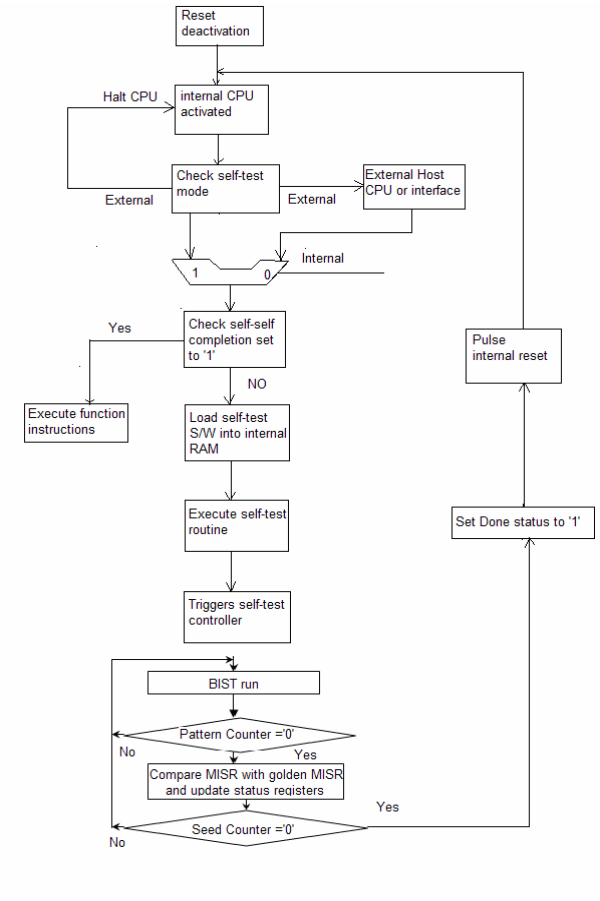

Figure 2: Procedure for logic self-test.

This is a significant advantage of re-seeding based BIST techniques and internal clocks [7,8].

Pass/Fail status generation: At the end of each phase of self-test, the resulting signature is stored in the status register as well as compared with the golden signature. At the end of all the phases, the *Pass/Fail* information is captured into the status register.

Resumption after self-test: This can be done through an external or internal reset. Care must be taken not to disturb both the status and configuration registers, as well as other modules which are not covered in non-destructive mode of operation. The action to be taken, (e.g. halt, re-configure, ignore, repeat, etc.), depends upon the *Pass/Fail* status, as well as the system / application code through which self-test is invoked.

3.2 Internal and External Control for Self-test

Various design considerations must be addressed for successful self-test operation.

3.2.1 Design Considerations

X intolerance: The BIST technique used requires the design to be free from functional as well as timing Xs. This is ensured through (i) appropriate bounding of X sources, including at the device inputs and outputs (I/Os), (ii) timing closure for all logic being BISTed, and (iii) skew balancing for clocks going to multiple modules in the design. The latter is important to avoid any asynchronous communication between modules which are being simultaneously BISTed. This also permits parallel operation of different modules, thereby saving on the test time, and providing greater flexibility of testing the inter-module and inter-clock domain logic [8].

Control of configuration and status registers: Updates should be controlled to prevent spurious over-writes due to reset or clock control changes during BIST operation and at the end of it. As examples, not using the internal reset nor internal clocks for the status register results in a more robust implementation. These registers must also be mapped onto the CPU addressable memory space for software control of the self-test operation. A protection mechanism in the form of *keys* is incorporated to prevent erroneous writes to the self-test configuration registers. It is necessary that these keys be unlocked first to get access to configurations registers. Two 32 bit keys were provided in this device to protect the unnecessary access to self-testing.

At-speed clock control: Additional support is provided in this device for at-speed test using internal PLL clocks for at-speed captures. Both launch-off-shift (LOS) and launch-off-captures (LOC) patterns can be applied. LOS patterns are applied with the capture clocks aligned to permit test of logic in different clock domains simultaneously. This is illustrated in Figure 3. The benefits of these techniques have been presented in [8,9].

3.2.2 System and Application Considerations

I/O pad control: Device I/Os are tri-stated to prevent random toggles with BIST patterns, thereby preventing spurious activity within it as well as in the other board level components. Additionally, quiescent values must be set through appropriate pull-up and pull-down control to prevent triggering of any activity with these patterns.

Self-test indication: While the result of self-test is interpreted by the device software, it is important to provide adequate indication to the rest of the system about the availability of this device to accept functional requests, interrupts, etc. Also, a system level test function can be defined based on the indication obtain from different such devices.

Self-test control: Important considerations here include the ability (i) to break self-loops during self-test, using reset control (as explained in Section 3.1), or through the use of a dedicated watchdog timer, (the latter forces a

hard reset), (ii) to recover control at the end of self-test operation, and (iii) to take corrective measures based on the result of the self-test operation.

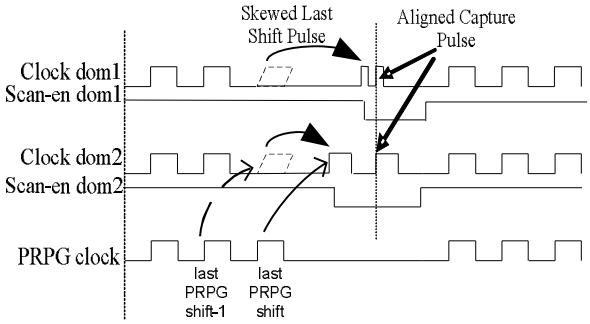

Figure 3: At-speed clocking - Aligned capture clocks.

4 Architecture and Implementation for Logic Self-Test

Logic self-test has been implemented using deterministic BIST (DBIST) with re-seeding, from Synopsys, Inc. [10]. The specific implementation of the DBIST controller is driven by various considerations in the design and the overall objectives of maximizing the coverage with minimal test time and minimal test data volume. The block diagram for DBIST is shown in Figure 4.

4.1 Enhancements to DBIST Architecture for Self-test

Several enhancements have been made to the DBIST architecture to support self-test through software. These include re-seeding through the device internal interface to the DBIST controller, user programmable pattern counter for each seed, internal clock selection and control for at-speed, internal signature storage and comparison, self-test control through 1149.1 JTAG interface, etc. [8].

4.2 Descriptions of Self test Controller Modules

Figure 5 shows the additional modules required to support self-test. These modules are explained.

BIST protocol driver module drives the BIST protocol to the DUT.

Read-write controller module reads seeds and golden MISR signatures, and other configurations information from the internal RAM, and writes into the BIST CoDecs.

Test mode controller module drives all hard macros and analog macros in ExTest to avoid MISR corruption.

BIST progress tracker module keeps track of BIST execution in terms of number of seeds, etc.

Watchdog timer module generates a timeout interrupt to master CPU, if the self-test operation does not complete within a user defined number of cycles.

Configuration register file contains the BIST configuration information

Status register file stores the status of self-test operation, e.g. Pass/Fail status, signature, etc., for each seed.

Memory interface unit serves as the RAM interface to load seeds and signatures, and write them into the BIST CoDec.

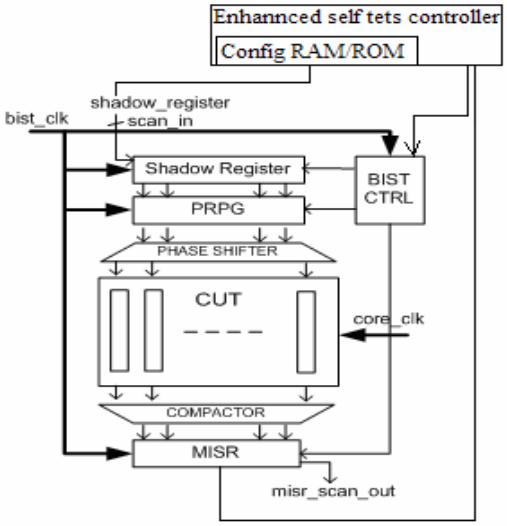

Figure 4: DBIST architecture with self-test support.

5 BIST and BISR for Memories

Field test for memories has been implemented using a run-time programmable memory BIST solution (PBIST), (proprietary to Texas Instruments – Refer to Figure 6), has been used. In addition, a built-in self-analysis / self-repair engine have also been designed to provide for memory repair. The repair solution described here nevertheless offers the added advantage of re-configurability in case of failures in the field in an embedded application [11]. Self-test for memories can also be performed through standard test interface like JTAG or software control using CPU [12]. The repair solution is soft, since the redundancy allocation information has to be regenerated upon power on.

5.1 Procedure for Memory Test

The flow diagram for self-test and self-repair is shown in Figure 7. The steps include:

***Memory BIST activation by software*:** The application must set the BIST configuration registers for various algorithms which must be applied to individual memory cores. This information can be obtained from internal memory or can be supplied through an external interface. The memory BIST operation can also be destructive or non-destructive.

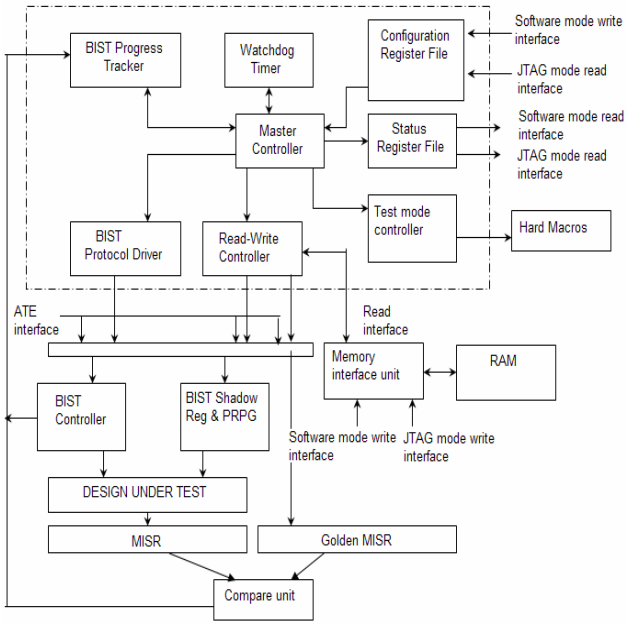

Figure 5: Enhancements for self-test controller.

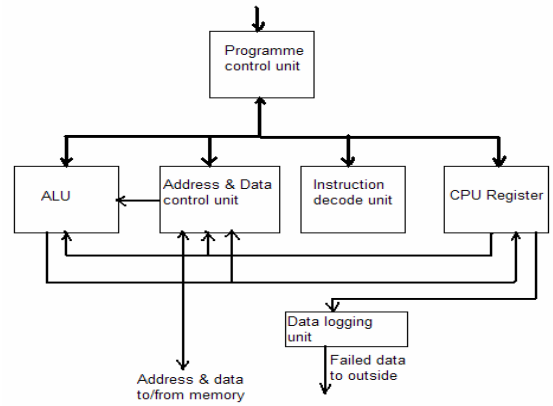

Figure 6: Programmable BIST for memories.

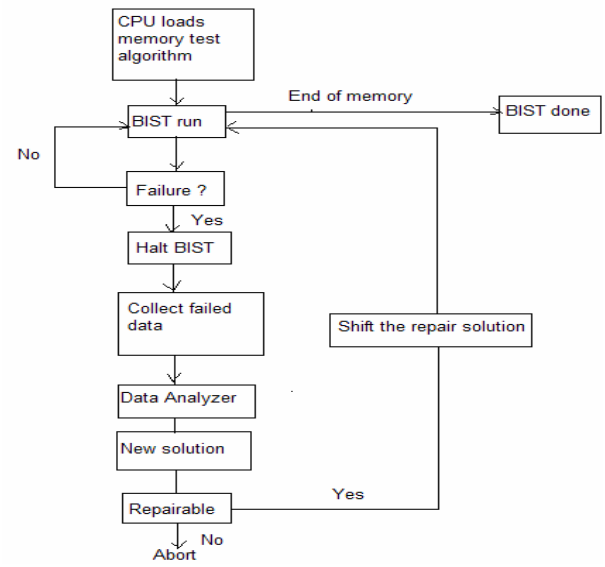

Figure 7: Flow for memory self-test and self-repair.

***Fail data analysis*:** Upon a failure, the BIST operation is halted. In the non-destructive mode, an interrupt is sent to the CPU or the external host to retrieve the fail data in the data logger unit and forward it to the data analyzer block for computing the repair solution. In the destructive mode, this information must be automatically latched into fail storage unit, which should also be accessible from the external host.

To obtain repair solutions, Various analysis engines have been proposed, the popular ones being CRESTA and its derivatives [13,14]. (The memory considered for repair in this device had only redundant rows, and the basic CRESTA algorithm was used). The final solution is a bitmap which depicts the allocation of the spare resources (redundant rows / columns) to failing memory locations.

Shifting new repair solution: The bitmap corresponding to the repair solution is shifted into the address mapping electrical fuse farm (and thereby over-riding the initial settings), which is programmed to hold the correct address map. Once the new repair solution is shifted, the BIST operation is resumed. In the case where all spare resources have been exhausted, and a new failure occurs, the BIST operation is aborted and the status register is updated. At the end, an interrupt is sent to the CPU.

5.2 Design Considerations

The considerations for successful field test of memories are similar to those for logic, described in Section 3.2. More specifically, for memories, additional care to be taken includes:

(a) The entire BIST / BISR engine, including the CPU control and memories themselves must lie in one synchronous clock domain.

(b) For non-destructive test, L1 and L2 caches must be treated properly in the self-test mode. During the test of L1 cache, CPU accesses to L1 cache must be prevented. During test of L2 cache and other program space, CPU cannot execute any program from these RAMs. The CPU must be prevented from accessing these memories in one of the following ways: (i) The CPU is halted. (ii) The program counter is set to a *safe state*, i.e. not in the targeted memories. (iii) The application code can discretely avoid this address space.

5.3 Implementation of Memory Self-test

The specific implementation on this device is shown in Figure 8. It depicts the interaction between the CPU, PBIST controller and data analyzer inside BISR block to arrive at a new repair solution. The CPU reads the fail information from the *Fail Registers* and sends it through its bus interface to the BISR data analyzer.

6 Device overview

The automotive device in which field testability has been incorporated is shown in Figure 9. The device has two CPUs. One is a C64x DSP core from Texas Instruments and the other is the ARM927 processor core. The ARM processor is the master CPU and provides the primary control function. The DSP core is the slave CPU and provides the primary compute function. Device self-test is performed by software running on the ARM processor. The PLL and clock control modules control and supply clocks to various parts of the device. All the self-test features implemented have been successfully tested in silicon and in the debugging / emulation environment around it. The device finds applications in navigation and telematics in the automotive space. This has motivated the development of the techniques described in this paper.

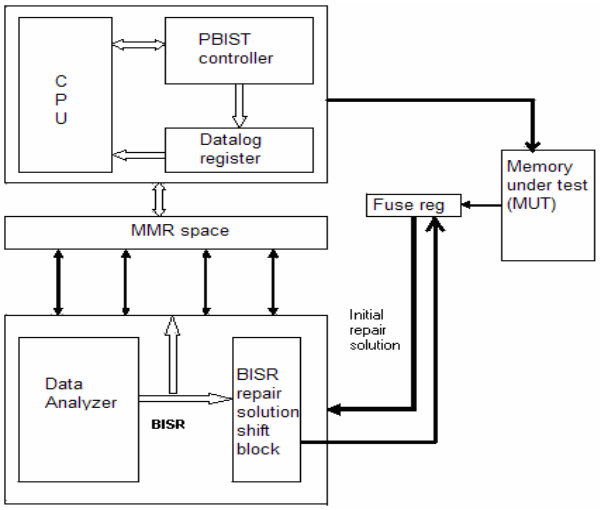

Figure 8: System architecture for self-test and self-repair

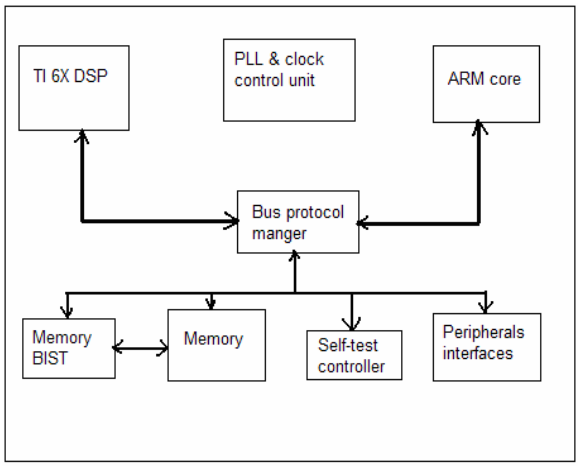

Figure 9: Device architecture.

7 Conclusion

This paper describes BIST implementation for logic and memories in an automotive SOC. Various design and test careabouts are explained, and it is shown how the implementation addresses them. It is shown how conventional BIST techniques have to be augmented to support the requirements of field test and repair. Other automotive designs in our design groups are also investigating the use of additional DFT and test control mechanisms to support online testing where the normal and test modes are interleaved.

References

[1] E.J. McCluskey and S. Mitra, "Fault-Tolerance", in Encyclopedia on Computer Science and Engineering, CRC Press, 2004.

[2] D.P.Siewiorek and R.S.Swarz, *Theory and Practice of Reliable System Design*, Digital Press, 1982.

[3] J.Wakerly, *Error Detecting Codes, Self-Checking Circuits and Applications*, Elsevier North Holland, Inc., 1978.

[4] I.Voyiatzis, A.Paschalis, D.Gizopoulos, H.Kranitis, C. Halatsis, "A concurrent built-in self-test architecture based on a self-testing RAM", IEEE Trans. on Reliability, March 2005.

[5] C.Stroud, *A Designer's Guide to Built-In Self-Test*, Kluwer Academic Publishers, 2002.

[6] R.D.Adams, *High Performance Memory Testing: Design Principles, Fault Modelling and Self-Test*, Kluwer Academic Publishers, 2002.

[7] P.Wohl, J.Waicukauski, S.Patel and M.Amin, "Efficient Compression and Application of Deterministic Patterns in a Logic BIST Architecture", Design Automation Conf., 2003, pp. 566-569.

[8] S.Jain, J.Abraham, S.Vooka, S.Kale, A.Dutta and R.Parekhji, "Enhancements in Deterministic BIST Implementations for Improving Test of Complex SOCs", Intl. Conf. VLSI Design, 2007.

[9] S.Goel and R.A.Parekhji, "Choosing the Right Mix of At-speed Structural Test Patterns: Comparisons in Pattern Volume Reduction and Fault Detection Efficiency", Asian Test Symp., 2005.

[10] Synopsys, Inc., SoCBIST Deterministic Logic BIST User Guide. Version V-2005.09, 2005.

[11] Y.Zorian, S.Shoukourian, "Embedded-Memory Test and Repair: Infrastructure IP for SoC Yield", IEEE Design and Test of Computers, Vol. 20, 2003, pp.58-66.

[12] J.Dreibelbis, J.Barth, H.Kalter, R.Kho, **"**Processor-based built-in self-test for embedded DRAM", Journal of Solid-State Circuits, 1998, Vol.21, pp. 71-89

[13] T. Kawagoe, J. Ohtani, M. Niiro, T. Ooishi, M. Hamada, Hidaka, "A Built-in Self-Repair Analyzer (CRESTA) for Embedded DRAMs", Intl. Test Conf. 2000,pp.567-574.

[14] S.Thakur, R.Parekhji, A.Chandorkar, "On-chip Test and Repair of Memories for Static and Dynamic Faults", Intl. Test Conf. 2006