Scientia Iranica, Vol. 10, No. 4, pp 486–494

SCIENTIA
IRANICA

Research Note

Embedded Test for Processor and Memory Cores in System-on-Chips

M.H. Tehranipour[1], S.M. Fakhraie*, M. Nourani[2], M.R. Movahedin[1] and Z. Navabi[1]

Embedded processors are now widely used in system-on-chips. The computational power of such processors and their ease of access to/from other embedded cores can be utilized to test SoCs (System-on-Chips). The first step, naturally, is to test the processor itself and its memory partner, as all other SoC test activities require both. In this paper, an efficient test architecture is presented to achieve high quality testing of embedded processor and memory cores. Especially, in testing the memory core, a test algorithm is presented for bit-oriented memories and its enhanced version for word-oriented memories. The method requires low overhead but provides significant flexibility in terms of fault model, test mechanism and future reuse.

INTRODUCTION

SoC Design and Test Paradigm

Current design and manufacturing technologies of integrated circuits allow for the integration of complete systems onto a single chip, commonly referred to as system-on-a-chip (SoC) or system chips. System chips are increasingly designed by embedding large pre-designed and pre-verified modules [1]. The most important advantage of using these so-called cores is that they speed up the design cycle. In recent years, cores have captured the attention of designers who understand the potential of using these cells. Examples of cores are CPUs, DSP engines, memories and communication modules of various kinds. Being able to rapidly develop, manufacture, test, debug and verify complex SoCs is crucial for the continued success of the electronics industry [2].

Core-based SoCs have significant advantages. Due to the fact that almost all of the components are in the same chip, SoCs can operate faster with less power. The SoCs reduce the number of discrete components used, thereby reducing the total size and cost of the end product. Furthermore, using embedded cores in SoCs has the potential of greatly reducing the time-to-market because of the design reuse involved.

Testing core-based systems is a major challenge. The main reason is that the accessibility of the cores and blocks is greatly reduced. Additionally, the system designer might have a limited knowledge of the core internals due to the protection of the Intellectual Property (IP) of the cores [3]. Several factors, such as fault coverage, test time, performance and area overhead, need to be considered to determine an effective test strategy when integrating cores into a SoC. To obtain high fault coverage, all of the system blocks should be thoroughly tested. It is very important that in the chosen test methodology the test time be kept as small as possible. For reducing cost, the amount of additional silicon area needed to implement the test scheme should also be low. Finally, performance penalties should be carefully evaluated.

Embedded Test: BIST Generalization for Embedded Cores

The problem with strategies using external test patterns is that they typically require a large test volume

1. *VLSI Circuits and Systems Laboratory, Department of Electrical and Computer Engineering, The University of Tehran, P.O. Box 14395-515, Tehran, I.R. Iran.*
*. *Corresponding Author, Department of Electrical and Computer Engineering, The University of Tehran, P.O. Box 14395-515, Tehran, I.R. Iran.*
2. *Center for Integrated Circuits and Systems, The University of Texas at Dallas, Richardson, TX 75083-0688, USA.*

and need complex dynamic test control protocols to get stimulus and receive responses from the cores and set up the necessary control to execute the test on a core. A promising strategy that minimizes such requirements is BIST. All known benefits of BIST come from the fact that BIST embeds a test into the circuit under testing. It generates and evaluates the test patterns on-chip. Hence, it does not require porting of test vectors from the core creator to the integrator and then to the fabricator. Since BIST implements test reuse, it offers advantages, similar to in-design reuse, to reduce the product development cycle. Additionally, it reduces the cost of manufacturing testing and system maintenance by providing the same test capability throughout all levels of system integration [4].

In addition to providing on-chip test resources, BIST simplifies peripheral access complications, since it requires a simple interface and allows reuse of the core as a self-contained block. Today's BIST schemes are mature enough in terms of providing very high fault coverage. They also enhance the diagnosis capability, while allowing IP protection. BIST is an autonomous testing method and is considered efficient for core-based systems [5].

A self-testing methodology in a system-chip, by running test programs using a programmable core, has several potential benefits, including at-speed testing, low design for test (DFT) overhead due to elimination of dedicated test circuitry and better power and thermal management during testing. This self-test strategy is referred to as functional self-test or embedded-software-based self-test or embedded test for short [6,7]. For high-speed circuits, self-testing has clear advantages over testing which relies on external testers. On-chip clock speed is increasing dramatically, while the tester's Overall Timing Accuracy (OTA) is not. Self-testing offers the ability to apply and analyze at-speed test signals on chip. This, in general, provides greater accuracy and shorter test time.

The embedded test is an advanced solution for testing large SoCs. An embedded test is comprised of two distinct test approaches: external Automatic Test Equipment (ATE) and conventional DFT. Building on conventional DFT approaches, such as scan and BIST, an embedded test integrates high-speed and high bandwidth portions of the external ATE directly into the ICs. This integration facilitates the chip, board and system level test, diagnosis, debug and repair [5].

The embedded test integrates multiple disciplines: DFT features, BIST pattern sources and sinks (points in a system chip to generate test patterns and consume test responses), precision and high-speed timing for an at-speed test, test support for many different core types (logic, memory, processors and analog) and capabilities for diagnosis and debug. With an embedded test, the on-chip test data generation reduces the volume of external patterns and can be customized per core type. Also, the on-chip test and diagnostic data compression reduces ATE data logging requirements. More importantly, the on-chip timing generation achieves true at-speed testing that can scale to match process performance [8].

The use of BIST in some of the most common cores used in system-chips is briefly described next.

Random Logic Cores

Conventional BIST provides a typical, random logic embedded test. This self-test is performed using a pseudo-random pattern source as the stimuli generator and a Multiple Input Signature Register (MISR) for the output results compression. The random logic BIST IP must be capable of operating at full application speed [8].

Memory Cores

The nature of a core may also have an impact on the internal test strategy. For example, almost all memories today tend to use BIST. Hence, providers of embedded memory cores typically incorporate BIST wrappers in their memory core design [5,8]. Almost all memory test methodologies use deterministic test patterns to achieve a high quality testing of embedded memory cores.

Analog Cores

Embedded analog cores, such as RF and telecommunication circuits, may be tested with a similar embedded test approach to random logic and memories. The analog embedded test automatically generates synthesizable RTL codes, synthesis scripts and verification and test patterns. Analog BIST allows at-speed testing of analog cores using a standard digital ATE. For instance, in PLL BIST [8], the RTL design objects connect to only the inputs and outputs of the PLL under test. No changes or connections to the internal nodes of the PLL are necessary. A digital multiplexer drives the input of the PLL. Testing the PLL is fully synchronous, making it suitable for very high-speed tests. PLL BIST measures the loop gain, frequency lock range, lock time and jitter [8]. This is a characteristic of advanced analog embedded test techniques [8].

Processor Cores

Test designers today tend to use a self-testing methodology for processor cores. This will be a software-based test providing a test program made of deterministic and random instructions and data run by a processor core for testing itself. Testing the processor by running test programs using the processor itself has several advantages similar to at-speed testing, such as low cost testing, low DFT overhead and elimination of some additional test elements. In this case, test pattern generation/delivery and signature analysis are done on-

chip and applied at the speed of the processor. A GHz processor core can test itself without relying on expensive high-speed external testers [6,9-12].

Motivation and Contribution

Most approaches that use processors to test SoCs, such as [13-15], implicitly assume that the embedded processor and memory cores are already tested. In this work, a test mechanism for testing these two important cores is proposed to fill the void. The main contribution of this paper is twofold. First, a BIST-based test methodology is presented that utilizes an embedded processor to test a SoC, including the processor itself. Second, an efficient scheme is proposed to test embedded memories using the on-chip processor.

The major components already found in most SoCs are processors and memories. Testing processor and memory cores is the focus of this work. The tested processor and memory can be used efficiently for testing the entire core-based system chip [13,14]. In complete systems, embedded processors may be required to run software routines for test pattern generation and response evaluation [15]. The test scheme described here is based on using a small test RAM in the SoC, called test RAM (TRAM), to store the test program and data related to each core, such as the processor and memory. Practically, TRAM can be a dedicated RAM module holding only the test program(s) or an existing instruction memory sharing the space with normal mode programs. In either case, a direct downloading mechanism, e.g. DMA (Direct Memory Access), is required to allow test programs to be stored in TRAM before the test session begins. All of the test programs related to the processor under test and other cores that are tested by the processor, such as memories, are stored in TRAM.

In the presented approach, first the processor tests itself at-speed by reading its self-test program from TRAM. Then, the tested processor can test the embedded memory at-speed. Since testing is done inside the system and the test program is read from an embedded TRAM, only a low-speed device is required for transferring the test programs into TRAM. It is also assumed that a path is available from the system I/O pins to the TRAM, so that the test data and program can be downloaded into the memory. This reduces the need for high-speed expensive external testers. The size of the test controller is very small because the processor uses its normal-mode controller. Since most of the testing processor, memory and other cores are specified in software, the method is very flexible for: a) Future change and reuse and b) Incorporating various types of cores that may use different test methodologies (e.g. BIST, scan, etc.) or even different fault models (e.g. stuck-at-fault, functional fault, etc.).

The rest of this paper is organized as follows. The next section describes the embedded test for processor cores and the main test architecture. Then, an embedded test for memory cores is described. Experimental results and implementation statistics for the proposed architecture are presented next, followed by concluding remarks.

EMBEDDED TEST FOR PROCESSOR CORES

Embedded processor cores are used in many systems. Core-based design offers several advantages, including design reuse and portability to different ASIC systems. This allows processors to be used in a variety of applications in a cost effective manner. However, designs that include processor cores present new challenges for testing, since access to these embedded processor cores becomes further restricted from the pins of the SoC [9].

For future high-speed (GHz) SoCs, it will be critical to test embedded processor cores at-speed. Therefore, there is a growing need for self-testing of embedded processor cores. By generating the required test patterns on-chip and applying the tests at the speed of a circuit, a high-speed processor core can test itself without relying on high-speed and prohibitively expensive external testers. Processors, due to their complex control structures, are highly random-pattern-resistant [6,16]. Acceptable fault coverage cannot be achieved by simply applying random test patterns to the entire processor. Structural techniques, like test point insertion and using deterministic test patterns, typically need to be performed to increase the fault coverage.

Proposed Architecture

Implementation of the presented embedded test architecture to test an embedded processor core is shown in Figure 1. This architecture can also be used for memory testing with the aid of the tested processor. TRAM can be an existing instruction memory or a dedicated memory for test programs. The instruction decoder and the processor controller execute the test

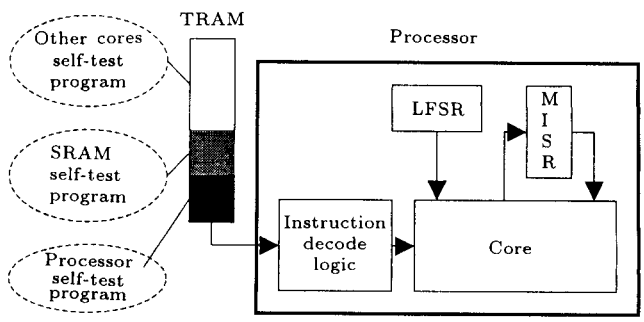

Figure 1. A general view of the test architecture.

programs similar to the normal mode. If needed, appropriate hardware modules (e.g. Linear Feedback Shift Register (LFSR)) provide random patterns to feed to cores/blocks under test.

Depending on the cores under test and the test budget, a hardware compressor/analyzer (e.g. Multiple Input Signature Register (MISR)) may be used to check the signatures or ask the processor to do the job. The latter requires more time but does not have the extra overhead for signature analysis. A hardware-software MISR has been used in this architecture where reading/writing from/into the MISR is done by newly defined instructions and signature generation is performed by the hardware of the MISR. This is similar to LFSR. Generating the test patterns is done with the hardware of the LFSR and reading the generated patterns is performed by a newly defined instruction. The most important advantage of this method is the at-speed test generation and signature analysis. The LFSR and MISR in this case are new elements for the processor and an inseparable part of the processor data path. Combined with other registers, LFSR/MISR can be controlled by instructions. Therefore, all blocks inside the processor core that can be controlled/observed by LFSR/MISR are controllable/observable by instructions directly. This removes the necessity of having additional multiplexors. Note carefully that no external test controller is needed, as all of the control operations are performed internally. Only a low speed ATE is used for transferring test programs into TRAM in a DMA fashion. In processor testing, the processor core can test itself by reading the processor self-test program stored in TRAM. For memory testing, the processor reads the memory test program and executes it for testing the targeted memory core on SoC. For other cores that are tested by the processor, their test programs are read by the processor and executed [16].

Developing Test Program

The stored test program into TRAM consists of: 1) A test program and some deterministic patterns for testing the processor. A random pattern generator provides random patterns when required (e.g. LFSR); 2) A program and deterministic patterns for testing the embedded memories; 3) Other test programs for testing the other cores on SoC, such as User Defined Logic (UDL), various data path and controller cores, etc. These programs are written in tested processor assembly language. The embedded microprocessor and memory pair will test the remaining components of the SoCs.

Actually, the test program consists of normal instructions that are executed by the processor core, while the data are provided by using deterministic or random patterns generated by the LFSR. Structural faults are targeted during component test generation of the processor core. Component tests of the core can be provided by using deterministic or random patterns. For example, only deterministic patterns are used for the Shifter unit and for other blocks, such as ALU, in the processor, both deterministic and random tests are used. In ALU testing, a loop segment reads random patterns from LFSR, applies patterns to the ALU and writes the results of the operations into MISR for signature generation. The number of iterations of this loop is determined using a technique to tradeoff between fault coverage and test time. Clearly, the number of instructions in the test program(s) does not affect the test overhead, although it affects test time. Eventually, the signatures obtained will be compared to the fault-free signatures, pre-computed and pre-loaded into the memory.

In general, writing the test programs depends on the nature of the cores, the blocks within the cores, the test methodology employed for each core and the level of testability expected. In this work, it is assumed that such information is available for each core targeting the structural faults within the cores and the job of the processor is to deliver the patterns (random or deterministic) to the right ports at the right time and, eventually, analyze the signatures.

New Instructions

Minor modifications to the processor controller and the creation of new instructions are needed to accommodate test features. More specifically, the controller uses added new instructions in the test mode. These instructions are created for easy operation on test generation and signature analysis at the speed of the processor. In [17], a systematic approach has been proposed to modify the VHDL code of a processor to include test instructions and test controller systematically. Additional hardware components, e.g. LFSR and MISR and suitable test instructions can be added to allow test methodology applied to the selected cores (see Table 1). The new instructions read or write into/from the accumulator. These instructions are added in the execute stage of the pipeline. LFSR and MISR are not regular registers and new instructions for these two registers perform two jobs when these instructions are used. As shown in Table 1, since LFSR is only used for read operations, an instruction that reads the new content of the LFSR register, e.g. MOV Acc, LFSR is created. For this instruction, first, LFSR generates its new random data and, then, moves it to the accumulator for application. MISR is used for Read/Write operations. When the MISR is used as a signature analyzer, the result of execution of an instruction is loaded into MISR, e.g. MOV MISR, Acc. In this case, first, the content of the accumulator

Table 1. New instructions.

Instruction	Operations	Type of Operation
MOV Acc*, LFSR	1- Generates a new random data in LFSR, 2- Acc ← LFSR	Read from LFSR
MOV MISR, Acc	1- MISR ← Acc, 2- Generates a new signature in MISR	Write into MISR
MOV Acc, MISR	1- Acc ← MISR (Signature is loaded into Acc for comparison step)	Read from LFSR

* Acc: Accumulator

is loaded on the input lines of MISR, then, it is applied to MISR and the new signature is generated. When the value of MISR is needed to compare with the pre-computed results from a fault-free circuit, the content of MISR must be read, e.g. MOV Acc, MISR. Hence, the random pattern generation and signature analysis are performed, internally, at the speed of the processor and the LFSR and MISR are considered as new processor elements.

EMBEDDED TEST FOR MEMORY CORES

Other widely used cores today are the embedded memories. Future SoCs are expected to embed very dense memories of large sizes (256M bits or larger). These dense memories may include: SRAMs, DRAMs and/or flash memories. Almost all memory providers today tend to incorporate BIST in memory core design. They rely on embedded sources to generate input test data and sinks to evaluate the output results. Moreover, since the embedded sink evaluates the memory response data, the role of this sink could be slightly expanded in order to perform diagnosis of the failed bits [5,8].

Defects in the layout of memory are modeled as faults in the corresponding transistor(s) diagram. The electrical behavior of each defect is analyzed and classified, resulting in a fault model at SRAM cell level. Only spot defects are considered for memory testing. These defects result in breaks and shorts in the circuit [18,19].

In this section, the focus is on SRAM testing in system chips. The tested processor uses the data and address buses of memory for the testing process. The deterministic test patterns stored in the embedded TRAM use the data bus of the memory under test and the processor uses the address bus of the memory for read/write test data from/into each word of memory. In the test process of the system chip, once the testing of the processor core is completed, it is used to test the embedded memories. It is referred to as software-based BIST. The proposed architecture is also implemented for testing embedded memories but there is no need to use LFSR and MISR for the random pattern generator and signature analyzer. In memory testing, only deterministic patterns (e.g. marching 0/1) are used as test data.

The 9N Test Algorithm

The pseudo code of the 9N test algorithm for a Bit-Oriented Memory (BOM) has been presented in Figure 2, where N is the number of addresses. It is a march-based algorithm that uses marching '0' and '1' values 9 times. In the initialization and March element 1 and 2 steps, the all stuck-at 1/0 and transition faults are checked. In March element 3 and 4, state coupling faults (coupling faults between two adjacent cells at one address) are covered. Because read and write operations are done for each cell of memory, decoder address faults are also detected. A data retention test is added to this algorithm. The proposed wait-time in the data retention test depends on the nodal capacitance and the leakage current in a memory cell. For example, in the Philips 8 K × 8, a wait-time of 100 msec was estimated. The 9N test algorithm is a march-based test algorithm for memory testing [18,19]. The second column in the figure shows the type of operation. The third column shows the increasing of the required addresses for each step of the algorithm and, finally, the algorithm needs 9 times read or write operations from or into the N address of the memory [18,19].

The 9N test algorithm is written in the assembly

// Begin Marching // N: The number of bit addresses	Type of Operation	
// Initialization Write '0' in addresses 1 to N;	Wr0	1N
// March element 1 Read '0' from addresses 1 to N; Write '1' in addresses 1 to N;	Rd0 Wr1	2N 3N
// March element 2 Read '1' from addresses 1 to N; Write '0' in addresses 1 to N;	Rd1 Wr0	4N 5N
// March element 3 Read '0' from addresses N down to 1; Write '1' in addresses N down to 1;	Rd0 Wr1	6N 7N
// March element 4 Read '1' from addresses N down to 1; Write '0' in addresses N down to 1;	Rd1 Wr0	8N 9N
// End Marching		

// Data retention test
// Wait
Pause RAM for 100 msec;		

// March element 5 Read '0' from addresses 1 to N;	Rd0	
// End of the 9N Algorithm		

Figure 2. The pseudo code of the 9N algorithm for a bit-oriented memory.

language program of the embedded processor core and stored into the TRAM. Although the $9N$ algorithm was used, any memory test algorithm can be adopted and used as the test program. When the processor enters the test mode for testing the embedded memory cores, it simply runs the test program corresponding to the $9N$ algorithm. It can be proven that the $9N$ test algorithm detects all faults of the fault model [20]. The test algorithm detects all stuck-at, transition, state coupling, multiple accesses and stuck-open faults.

Word-Oriented Memory (WOM) Test

Word-oriented memories contain more than one bit per word; i.e., $B \geq 2$, where B represents the number of bits per word and usually is a power of two. Read operation reads the B bits simultaneously and write operation writes data into the B bits of memory. Many different data backgrounds are used for testing the word-oriented memories [21]. The fault model for word-oriented memories can be divided into two classes: 1. Single-cell faults: this class includes stuck-at, stuck-open and transition faults and 2. Fault between memory cells: this class of faults consists of coupling faults.

The $9N$ algorithm reads and writes '0' and '1' to each bit cell in bit-oriented memories, while it needs to be converted for word-oriented memories. March tests for bit-oriented memories can be converted to march tests for word-oriented memories by taking into account that in the bit-oriented memory tests, the 'Rd0', 'Rd1', 'Wr0' and 'Wr1' operations are applied to a single bit. In the case of word-oriented memories, an entire word of B bits has to be read or written; the data value of this word is called the Data Background (DB). Figure 2 shows that the data backgrounds in a bit-oriented memory for consecutive read and write operations are '0' and '1'. In bit-oriented memory, if '0' is data background (DB), '1' will be considered as the inverse data background (IDB).

Word-oriented SRAMs introduce the problem of state coupling faults between two cells at one address. To detect these faults, all states of two arbitrary cells at the same address must be checked. This is only possible if several data backgrounds are used. A minimum of K data backgrounds will be needed where $K = [\log_2 B] + 1$. In many memories, B is the power of two, then, the formula is simplified to: $K = \log_2 B + 1$. For example, for a memory with $B = 4$, then $K = 3$ and the DBs are: "0000", "0101", "0011" and IDBs are: "1111", "1010" and "1100". In the converted $9N$ algorithm for word-oriented memories, for read operation, 'RdDB' and 'RdIDB' are used instead of 'Rd0' and 'Rd1'. Similar generalization is needed for write operations in this algorithm. It can be easily verified that for any two cells, all four possible cases (i.e. '00', '10', '01' and '11') occur. Therefore, the coupling between any pair of cells in the same address is checked [21].

$K(9M)$ Algorithm

The pseudo code of the $K(9M)$ algorithm for the WOM test has been shown in Figure 3, where M is the number of word addresses and K is the number of data backgrounds. The $K(9M)$ is a converted version of the $9N$ algorithm in WOM test mode.

The data backgrounds and inverse data backgrounds (DBi and IDBi) for a 32K-16 bit memory are shown in Table 2.

In the $9N$ algorithm, N is: $N = V.B$, where V represents the size of memory and B is the number of bits per word. In the BOM test, the total number of read and write (R/W) operations is: $N_{BOM} = 8.V.B$. In the $K(9M)$ algorithm, M is: $M = V$, $N_{WOM} = K.8.V$ and, then, $(N_{WOM}/N_{BOM}) = K/B$. As seen, the number of R/W operations in the WOM test is much lower than the BOM test. It shows that in the WOM test, the total number of R/W is reduced dramatically when the number of bits (B) is increased. For example, in a 32K-16 bit RAM, $B = 16$, $K = 5$ and, then, $K/B = 5/16$. Converting the BOM test algorithm to

Table 2. DBi and IDBi for a 32K-16 bit memory, $B = 16$, $K = 5$.

i	1	2	3	4	5
DBi	0000	5555	3333	0F0F	00FF
IDBi	FFFF	AAAA	CCCC	F0F0	FF00

```
// Begin Marching                                   Type of
// M: The number of word addresses                  Operation
 For i=1 to K Loop;
     // K = The minimum of data backgrounds
   // Initialization
     Write 'DBi' in addresses 1 to M;                Wr DBi
   // March element 1                                          1M
     Read 'DBi' from addresses 1 to M;               Rd DBi
     Write 'IDBi' in addresses 1 to M;               Wr IDBi  2M
   // March element 2                                          3M
     Read 'IDBi' from addresses 1 to M;              Rd IDBi
     Write 'DBi' in addresses 1 to M;                Wr DBi   4M
   // March element 3                                          5M
     Read 'DBi' from addresses M down to 1;          Rd DBi
     Write 'IDBi' in addresses M down to 1;          Wr IDBi  6M
   // March element 4                                          7M
     Read 'IDBi' from addresses M down to 1;         Rd IDBi
     Write 'DBi' in addresses M down to 1;           Rd DBi   8M
// End Marching                                               9M
// Data retention test
   // Wait
     Pause RAM for 100 msec;
   // March element 5
     Read 'DBi' from addresses 1 to M;
   End Loop;                                         Rd DBi
// End of the K(9M) Algorithm
```

Figure 3. The pseudo code of the $K(9M)$ algorithm for WOMs.

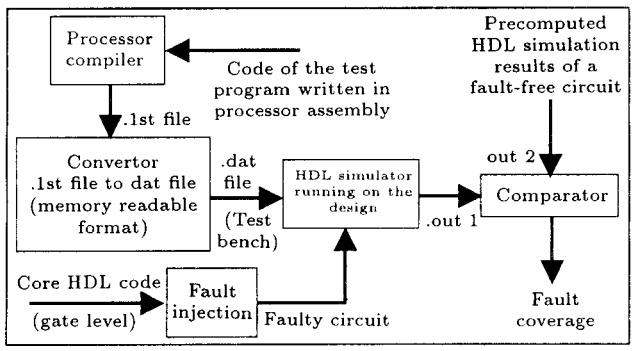

Figure 4. A test evaluation framework for processors.

a WOM test algorithm can be performed for all of the memory test algorithms.

EXPERIMENTAL RESULTS

Case Study: UTS-DSP

A UTS-DSP processor is the case study in this paper and its structure is briefly described in this subsection. The UTS is compatible at the instruction set level with TI's TMS320C54x DSP processor family, which has CISC architecture. This is a fixed-point digital signal processor designed in the VLSI Circuits and Systems Laboratory, Department of Electrical and Computer Engineering of the University of Tehran. The UTS-DSP Central Processing Unit (CPU), with its modified Harvard architecture, features minimized power consumption and a high degree of parallelism. Also, the versatile addressing modes and instruction set improve the overall system performance. In this system, there are the processor cores, SRAM and ROM and some interface units, such as Serial Port Interface (SPI) and Host Port Interface (HPI) [22,23]. Testing of the two main embedded cores in SoCs, i.e., processor and memory cores, is discussed here. The processor core is comprised of some functional blocks, such as ALU, Shifter and Multiply/Accumulate (MAC) units. The testing of each functional block has been done in the processor-testing phase.

Processor Testing Results

A testability analysis of each block was first performed within the processor. Then, a complete self-test program for testing the embedded processor core of the UTS-DSP has been written. To evaluate the fault coverage of a test program on the processor under test, a test evaluation framework has been established, as shown in Figure 4. As shown, the compiler takes the code of the test program written in processor assembly language and prepares a .lst file [24]. A converter program (written in C language) converts the .lst to a .dat file (memory readable format). This file contains

Table 3. Processor testing results.

Fault Coverage	**> 95%**
Clock cycle	162,500
Test time (For a processor with $f = 65$ MHz)	2.5 msec
Number of words of TRAM for a complete processor test program	350

the instructions and data that are used as a test bench for the VHDL simulator. In reality, the test bench is a program to be executed by the HDL simulator on the codes corresponding to the faulty and fault-free cores. This simulation is like executing an assembly program on a processor in terms of checking the functionalities and possible errors. The HDL simulator takes the design description (faulty/fault-free), runs the test bench and captures the input signals to the processor. A "C" program performs the fault injection, automatically, into the HDL code of the core.

A structural/net coverage is undertaken that is a good estimate of the fault coverage in practice. Accurate fault coverage cannot be obtained. The structural/net coverage can be captured much faster while it shows the same trend in terms of quality of testability. Test program development has been performed based on covering nets and structural components. As shown in Figure 4, the fault coverage is, eventually, determined by comparing the fault-free and faulty responses. Fault coverage and test time for one of the best programs is shown in Table 3. This program occupies 350 words of TRAM. The total area overhead for processor testing is negligible. It consists of one 16-bit LFSR, one 16-bit MISR, some control elements for executing the new instructions and 350 words of RAM. This area overhead, compared to the size of the processor core, is negligible.

Memory Testing Results

The same process is also performed for embedded SRAM testing with a test program and a defined fault model. The SRAM test program that is written in assembly language occupies 88 words of embedded TRAM. In this case study, there is a 32K 16-bit SRAM, then, $B = 16$, $K = 5$ and the algorithm will be $5(9M)$. The results of these experiments are shown in Table 4.

This table shows that the $K(9M)$ algorithm covers all of the faults under the defined fault model for memory, such as stuck-at, transition, decoder address and state coupling faults, while other methods cannot cover all faults. The Marching 1/0 method does not detect state coupling fault, but is performed faster. The Checker pattern method cannot cover state coupling fault and it covers 50% transition faults. In this case, this method only covers '0' to

Table 4. Comparison of several methods for a WOM (M = number of addresses).

	GALPAT Algorithm	Checker Pattern	Marching 1/0	$K(9M)$ Algorithm
Complexity	K($4M^2$)	K(4M)	K(6M)	K(9M)
Stuck-at Fault	Yes	Yes	Yes	Yes
State Transition Fault	Yes	50% Yes	Yes	Yes
Decode Address Fault	Yes	No	Yes	Yes
State Coupling Fault	Yes	No	No	Yes
32K SRAM (f=65MHz)	9hr	31msec	46msec	81 msec

'1' transition and cannot detect '1' to '0' transition fault. This algorithm is faster than other methods. The GALPAT algorithm covers all of the faults in the defined fault model, however, it is very low speed in comparison with the $K(9M)$ algorithm. The second row of Table 4 shows the complexity of each method, where M is the number of words [21,25]. In the seventh row, the test time of each method on a SRAM 32K is shown. In the proposed implementation, with increasing the size of RAMs, only the test time is increased, without any increasing in hardware overhead. The memory test program is a loop program. When the size of memory is increased, the number of loop counters is increased, while the size of the program is fixed. The key point in this architecture is to use the embedded processor (as opposed to using an external ATE) to apply, control, observe and manage the memory block test process. This scheme is much less costly and runs at a speed that is important in SoC testing. The proposed test algorithm shows excellent features in both test time and fault coverage and is independent of row, column and cell arrangement in the memory array. The given test algorithms are both suitable for bit- and word-oriented SRAMs.

CONCLUSION

A low-cost, at-speed test methodology has been presented for testing embedded processor and memory cores. This architecture reduces the need for high-speed expensive external testers. There is no need for additional test controllers. The embedded processor architecture runs the test programs to test itself, memory cores and, then, all other cores in SoC and controls all test activities. Specifically, for embedded SRAM testing, the test algorithm (program) covers 100% of faults under the defined fault model. The proposed test algorithm for memory cores shows excellent features in both test time and fault coverage. It is independent of row, column and cell arrangement in the memory array and suitable for both bit and word-oriented SRAMs with no additional hardware overhead.

REFERENCES

1. Gupta, R.K. and Zorian, Y. "Introducing core-based system design", *IEEE Design & Test of Computers*, **14**(4), pp 15-25 (1997).
2. Zorian, Y., Marinissen, E. and Dey, S. "Testing embedded-core-based system chips", *Proceedings of the International Test Conference (ITC)*, pp 130-143 (1998).
3. De, K. "Test methodology for embedded cores which protects intellectual property", *VLSI Test Symposium (VTS)*, pp 2-9 (1997).
4. Rajski, J. and Tyszer, J. "Modular logic built-in self-test for IP cores", *Proceedings of the International Test Conference*, pp 313-321 (1998).
5. Zorian, Y. "System-chip test strategies", *Proceedings of the 35th Design Automation Conference*, pp 752-757 (1998).
6. Radeka, K., Rajski, J. and Tyszer, J. "Arithmetic built-in self-test for DSP cores", *IEEE Trans. On CAD/ICAS*, **16**(11), pp 1358- 1368 (1997).
7. Lai, W.C. and Cheng, K.T. "Instruction-level DFT for testing processor and IP cores in system-on-a-chip", *Proceedings of the 37th Design Automation Conference* (2001).
8. Zorian, Y., Dey, S. and Rodger, M.J. "Test of future system-on-chip", *Proceedings of the International Conference on Computer-Aided Design*, pp 392-398 (2000).
9. Chen, L., et al. "Embedded hardware and software self-testing methodologies for processor cores", *Proceedings of the 37th Design Automation Conference*, pp 625-630 (2000).
10. Shen, J. and Abraham, J.A. "Native mode functional test generation for processors with applications to self-test and design validation", *Proceedings of the International Test Conference*, pp 990-999 (1998).
11. Batcher, K. and Papachristou, C.A. "Instruction randomization self-test for processor cores", *Proceedings of the 17th IEEE VLSI Test Symposium*, pp 34-40 (1999).
12. Paschalis, A., et al. "Deterministic software-based self-testing of embedded processor cores", *Proceedings of the Design, Automation and Test in Europe*, pp 92-96 (2001).

13. Papachristou, C.A., Martin, F. and Nourani, M. "Microprocessor based testing for core-based system on chip", *Proceedings of the Design Automation Conference*, pp 586-591 (1999).

14. Rajsuman, R. "Testing a system-on-a-chip with embedded microprocessor", *Proceedings of the International Test Conference*, pp 499-508 (1999),

15. Hellebrand, S., Wunderlich, H.J. and Hertwig, A. "Mixed-mode BIST using embedded processors", *Proceedings of the International Test Conference*, pp 195-204 (1996).

16. Tehranipour, M.H., Navabi, Z. and Fakhraie, S.M. "A low cost BIST architecture for processor cores", *Proceedings of the Electronic Circuits and Systems Conference (ECS)*, pp 11-14 (2001).

17. Tehranipour, M.H. "Built-in self-test for DSP cores", M.Sc. Thesis, Department of Electrical and Computer Engineering, The University of Tehran (2000).

18. Dekker, R., et al. "A realistic fault model and test algorithm for static random access memories", *IEEE Trans. on CAD*, **9**(6), pp 567-572 (1990).

19. Dekker, R., et al. "Fault modeling and test algorithm development for SRAMs", *Proceedings of the International Test Conference (ITC)*, pp 343-351 (1988).

20. Tehranipour, M.H., Navabi, Z. and Fakhraie, S.M. "An efficient BIST method for testing embedded SRAMs", *Proceedings of the International Symposium on Circuits and Systems (ISCAS)*, **5**, pp 73-76 (2001).

21. Van de Goor, A.J. and Tlili, I.B.S. "March test for word-oriented memories", *Proceedings of the Design Automation and Test in Europe*, pp 501-508, (1998).

22. Riahi, P.A., et al. "UTS-DSP IC core", *Proceedings of the Iranian Conf. on Electrical Engineering (ICEE), Computer Proceedings*, pp 71-79 (1999).

23. *TMS320C54x DSP: CPU and Peripherals*, Texas Instruments (1996), available online at: http:// focus.ti.com/lit/ug/spru131g/spru131g.pdf.

24. *TMS320C54x Assembly Language Tools, Users' Guide*, Texas Instruments (1996), available online at: http:// focus.ti.com/lit/ug/spru102f/spru102f.pdf.

25. Abramovici, M., *Digital System Testing and Testable Design*, New York, Computer Science Press (1990).