JOURNAL OF ELECTRONIC TESTING: Theory and Applications 18, 529–538, 2002
© 2002 Kluwer Academic Publishers. Manufactured in The Netherlands.

Testing for Interconnect Crosstalk Defects Using On-Chip Embedded Processor Cores*,†

LI CHEN, XIAOLIANG BAI AND SUJIT DEY
Department ECE, University of California at San Diego, La Jolla, CA 92093-0407, USA
lichen@ece.ucsd.edu
xibai@ece.ucsd.edu
dey@ece.ucsd.edu

Received September 11, 2001; Revised January 15, 2002

Editor: Krishnendu Chakrabarty

Abstract. In deep-submicron technologies, long interconnects play an ever-important role in determining the performance and reliability of core-based system-on-chips (SoCs). Crosstalk effects degrade the integrity of signals traveling on long interconnects and must be addressed during manufacturing testing. External testing for crosstalk is expensive due to the need for high-speed testers. Built-in self-test, while eliminating the need for a high-speed tester, may lead to excessive test overhead as well as overly aggressive testing. To address this problem, we propose a new software-based self-test methodology for system-on-chips (SoC) based on embedded processors. It enables an on-chip embedded processor core to test for crosstalk in system-level interconnects by executing a self-test program in the normal operational mode of the SoC, thereby allowing at-speed testing of interconnect crosstalk defects, while eliminating the need for test overhead and the possibility of over-testing. We have demonstrated the feasibility of this method by applying it to test the interconnects of a processor-memory system. The defect coverage was evaluated using a system-level crosstalk defect simulation method.

Keywords: interconnect, crosstalk, self-test, processor

1. Introduction

The shrinking of feature sizes enables the integration of a large system comprising of multiple cores on a single chip. In core-based designs, a larger amount of core-to-core communications must be realized with long interconnects. As gate delay continues to decrease, the performance of interconnect is becoming increasingly important in achieving a high overall performance [15]. However, due to the increase of cross-coupling capacitance and mutual inductance, signals on neighboring wires may interfere with each other, causing excessive delay or loss of signal integrity. This effect, known as crosstalk, is more pronounced in deep-submicron technology [8, 13]. While many techniques have been proposed to reduce crosstalk [7, 10, 18], due to the limited design margin and unpredictable process variations, testing for crosstalk must be performed during manufacturing. In recent years, several crosstalk test generation techniques have been developed to generate tests for local interconnects in gate-level circuits [5, 6, 9, 11, 17]. Unlike these techniques, we focus on the testing of system-level interconnects, as they are most susceptible to crosstalk effects due to the large

*This work is supported in part by MARCO/DARPA Gigascale Silicon Research Center (GSRC) and the Semiconductor Research Corporation (SRC) through contract no. 98-TJ-648.
†A preliminary version of this paper appeared in *Proceedings of the 38th Design Automation Conference*, Las Vegas, NV, USA, June 2001.

coupling capacitance between long wires running in parallel.

Due to its timing nature, testing for crosstalk effect need to be conducted at the operational speed of the circuit-under-test. At-speed testing for GHz systems, however, is prohibitively expensive with external testers, as it requires testers with GHz performance. Moreover, with external testing, hardware access mechanisms are required for applying tests to interconnects deeply embedded in the system, which may lead to unacceptable area or performance overhead.

Compared with external testing, self-testing is a more feasible solution for at-speed crosstalk testing, as it does not impose any performance requirement on the external tester. A built-in self-test (BIST) technique has been proposed in [2], in which an SoC tests its own interconnects for crosstalk defects using on-chip hardware pattern generators and error detectors. Although the amount of area overhead may be amortized for large systems, for small systems, the amount of relative area overhead may be unacceptable. Moreover, hardware-based self-test approach like the one proposed in [2] may cause over-testing, as not all test patterns generated in the test mode are valid in the normal operational mode of the system.

In this paper, we address the problem of testing system-level interconnects in embedded processor-based SoCs, which are the most dominant type of SoCs. In such SoCs, most of the system-level interconnects, such as the on-chip busses, are accessible to the embedded processor core(s). Based on this fact, we propose a software-based methodology that enables an embedded processor core in the SoC to test for crosstalk effects in these interconnects by executing a software program. Unlike previous embedded processor core-based self-testing methodologies [3, 4, 14, 16], in which the embedded processor cores themselves are the subjects of testing, the focus of our methodology is on the testing of interconnects connecting the processor cores and other cores.

Compared with the hardware-based approaches for crosstalk testing, software-based self-test can be applied in the normal operational mode of the processor. Therefore, no extra hardware is needed. In addition, only the test patterns valid in the normal operational mode of the processor are applied. Thus, the system will not be over-tested. This is because crosstalk cases that cannot be excited in the normal operational mode do not affect the correct functionality of the system. Thus, the rejection of a chip due to a failure response in these cases causes unnecessary yield loss.

In the rest of the paper, we first briefly describe the crosstalk fault model used during the development of the proposed method. The software-based self-test methodology for crosstalk defects is introduced in Section 3. In Section 4 we illustrate the proposed method in detail by constructing a self-test program for testing interconnects in a particular CPU-memory system. In Section 5, we validate the proposed method by extensive defect simulation using an HDL-level simulation framework consisting of a high-level crosstalk error model. Section 6 concludes the paper.

2. Fault Model

During the development of the proposed self-test method, we used the Maximum Aggressor Fault (MAF) model [8] for modeling crosstalk effects. MAF model is a high-level abstraction of all physical defects and process variations that lead to crosstalk errors. Defining crosstalk faults based on the actual physical defects can lead to the explosion of the fault space, as the number of possible process variations and defects that need to be considered can be extensive even for a bus with a small number of interconnects. Instead, the MAF model defines faults based on the resulting crosstalk error effects, including positive glitch (g_p), negative glitch (g_n), rising delay (d_r), and falling delay (d_f). The interconnect on which the error effect takes place is defined as the victim. All the other wires are designated aggressors, acting collectively to generate the glitch or delay error on the victim.

Fig. 1 shows the signal transitions needed on the victim/aggressors to produce the strongest error effects on a victim wire. For example, to produce a positive glitch fault on Victim Y_i, a stable "0" is assigned to Y_i, whereas rising transitions are assigned to all aggressors. Each of these signal transitions, consisting of two consecutive test vectors, is defined as the Maximum Aggressor (MA) test for the corresponding MAF. For

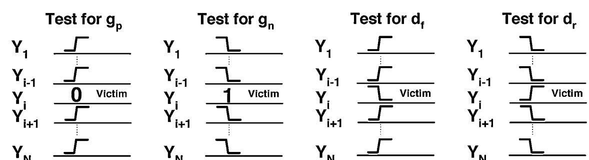

Fig. 1. Maximum aggressor tests for victim Y_i.

a set of N-interconnects, a total of $4N$ faults need to be tested, requiring $4N$ unique MA tests. It has been proven in [8] that in an RC network, the MA tests are necessary and sufficient for covering all possible physical defects and process variations that can lead to any cross-coupling induced crosstalk error effect on any of the N interconnects. The limitation of the MAF model is that the MA tests do not necessarily excite the worst-case crosstalk scenarios in an RLC network.

In the next section, we describe the general strategy for applying MA tests to system-level interconnects by executing a self-test program using an embedded processor.

3. Test Methodology

In a core-based SoC consisting of embedded processor, memory, and other cores, the address, data, and control busses are the main types of global interconnects, with which the embedded processors communicate with memory and other cores of the SoC via memory-mapped I/O (Fig. 2). Besides the single address/data busses shown in Fig. 2, there can be multiple busses or hierarchical bus structures. The most common mechanism for a CPU to communicate with a core is via memory-mapped I/O, in which certain addresses in the memory address space of the CPU are reserved for addressing the cores. Hence, the way the CPU communicates with memory cores is similar to the way it communicates with non-memory cores. In this paper, we focus on the testing of interconnects connecting the CPU and memory cores. Among such interconnects, we focus on the testing of data and address busses, as they are by far dominating and most susceptible to crosstalk defects due to a large number of long interconnects running in parallel. The testing of interconnects between the CPU and non-memory cores and the testing of control busses are subjects of future study.

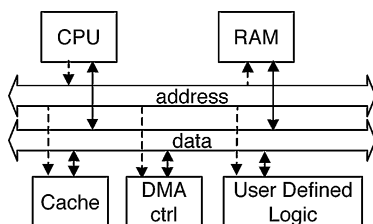

Fig. 2. An SoC containing embedded processor cores.

To test the address bus and the data bus with an embedded processor, our strategy is to let the processor execute a self-test program, with which the test vector pairs (as described in Section 2) can be applied to the appropriate bus in the normal functional mode of the system. In the presence of crosstalk-induced glitch or delay effects, the second vector in the vector pair becomes distorted at the receiver end of the bus. The processor can then store this error effect to the memory as a test response, which can later be unloaded by an external tester for off-chip analysis.

In this self-test approach, the loading of the self-test program and the unloading of the test response can be done with a low-speed tester. This prevents the corruption of the test program data or test response data due to crosstalk errors on the data or address busses, which can be activated at high speed. The self-test program itself is executed at-speed in the normal functional mode of the system without the monitoring of any external testers. Thus, with this approach, at-speed testing for crosstalk effects can be applied without a high-performance tester.

We next describe the general method for testing the data bus and the address bus.

3.1. *Testing Data Bus*

For a bi-directional bus such as a data bus, crosstalk effects vary as the bus is driven from different directions. Thus, crosstalk tests need to be conducted in both directions [2]. Note that, however, to apply a test vector pair ($v1$, $v2$) in a particular bus direction, the direction of $v1$ is irrelevant. Only $v2$ needs to be applied in the specified direction. This is because the signal transition triggering the crosstalk effect takes place only when $v2$ is being applied to the bus.

To apply a test vector pair ($v1$, $v2$) to the data bus from an SoC core to the CPU, the CPU first exchanges data $v1$ with the core. The direction of the data exchange is irrelevant. For example, if the core is the memory, the CPU may either read $v1$ from the memory or write $v1$ to the memory. The CPU then requests data $v2$ from the core (a memory-read if the core is memory). Upon the arrival of $v2$, the CPU writes $v2$ to memory for later analysis.

To apply a test vector pair ($v1$, $v2$) to the data bus from the CPU to an SoC core, the CPU first exchanges data $v1$ with the core. The CPU then sends data $v2$ to the core (a memory-write if the core is memory). If the core is memory, $v2$ can be directly stored to an

appropriate address for later analysis. Otherwise, the CPU must execute additional instructions to retrieve $v2$ from the core and store it to memory.

3.2. *Testing Address Bus*

To apply a test vector pair ($v1$, $v2$) to the address bus, which is a unidirectional bus from the CPU to an SoC core, the CPU first requests data from two addresses ($v1$ and $v2$) in consecutive cycles. In the case of a non-memory core, since the CPU addresses the cores via memory-mapped I/O, $v2$ must be the address corresponding to the core. If $v2$ is distorted by crosstalk, the CPU would be receiving data from a wrong address, $v2'$, which may be a physical memory address or an address corresponding to a different core. By keeping different data at $v2$ and $v2'$ (i.e., mem[$v2$] $\neq$ mem[$v2'$]), the CPU is able to observe the error and store it to memory for analysis. In rare cases, the value stored in $v2'$ cannot be easily controlled. For example, $v2'$ may correspond to unused address space, read-only locations, or address space mapped to special registers whose values may change during operation. Furthermore, the value of $v2'$ may be unpredictable due to an unpredicted crosstalk error effect. For example, a signal transition on an aggressor can be corrupted by a strong victim. To address these problems, a relative unique value should be stored in $v2$ to maximize the probability of detecting the error.

Fig. 3 illustrates the process of testing the address bus. For example, in the case where the CPU is communicating with a memory core, to apply test (0001, 1110) in the address bus from the CPU to the memory core, the CPU first reads data from address 0001. The CPU then reads data from address 1110. In the system with the faulty address bus, this address becomes 1111. If different data are stored at address 1110 and 1111 (mem[1110] = 0100, mem[1111] = 1001), the CPU would receive a faulty value from memory (1001 instead of 0100). This error response can later be stored to memory for analysis.

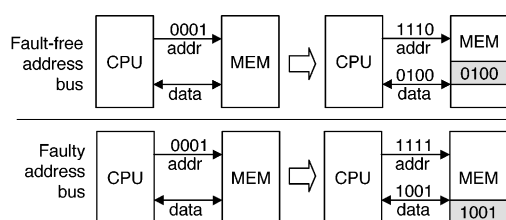

Fig. 3. Testing the address bus.

Although the generation of test programs is instruction-set-specific, a systematic approach can be followed. To summarize, we first analyze instructions with which the processor communicates with the memory core (e.g., load and store instructions). We then look for instructions or instruction sequences that can cause transitions on the data or address bus. Transitions with maximum flexibility are selected for applying crosstalk tests (e.g., MA test vector pairs). These are transitions with minimum constraints on signal values. Finally, for observing error responses, additional instructions may be used if necessary. The end goal is such that if the second vector in the test vector pair is corrupted, either an incorrect value is stored to the response location in memory (due to crosstalk error on data bus) or a correct value is stored to an incorrect location in memory (due to crosstalk error on address bus).

4. Application to a CPU-Memory System

In this section, we illustrate the proposed method in detail by applying the guidelines introduced in Section 3 to a CPU-memory system. Since memory-mapped I/O is a common mechanism for CPU to communicate with other cores, the same methodology can be extended for testing interconnects between the CPU and non-memory cores.

The particular CPU-memory system we used here consists of an 8-bit accumulator-based multi-cycle processor core with 23 instructions [12] and a 4K instruction/data memory. The CPU communicates with the memory via a 12-bit unidirectional address bus and an 8-bit bi-directional data bus.

4.1. *Testing Data Bus*

We chose to use the load instruction (LDA) of the processor to apply tests to the data bus. The load instruction takes a 12-bit address (A_x) as operand and loads the content of this address ($M[A_x]$) to the accumulator. As shown in Fig. 4, the load instruction is stored as two bytes in the memory. The addresses of these two bytes are A_i and $A_i + 1$, respectively. In the first byte, the first 4 bits contain the opcode of the instruction. The last 4 bits contain the *page number* of A_x, which is the first 4 bits of A_x. The second byte in the load instruction contains the *offset* of A_x, which is the last 8 bits of A_x.

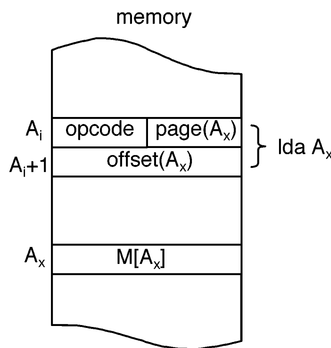

Fig. 4. Load instruction (LDA).

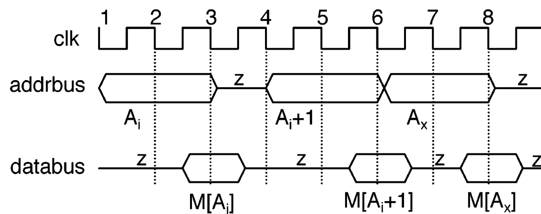

Fig. 5. Load instruction: timing diagram.

Fig. 5 shows the timing diagram of the load instruction. In the system under consideration, access to busses is controlled by tri-state buffers. When all tri-state buffers are disabled, the signal on the bus becomes high impedance ("z"). When "z" appears, we assume the bus holds the last defined value before "z".

To fetch the instruction from the memory, the CPU first requests the first byte of the instruction by placing the address of the instruction, A_i, on the address bus. The memory responds by sending the content of $A_i(M[A_i])$ to the CPU through the data bus. After decoding the first byte of the instruction, the CPU recognizes that the current instruction is a load instruction, which contains two bytes. Thus the CPU fetches the second byte of the instruction from memory ($M[A_i + 1]$). After receiving the complete instruction, which contains the data address A_x, the CPU fetches the content of $A_x(M[A_x])$ from memory and places it into the accumulator.

During the execution of the load instruction, there are two signal transitions on the data bus, which can be used to apply the vector pair for crosstalk testing. The first transition is from $M[A_i]$ to $M[A_i + 1]$. As $M[A_i]$ contains the opcode of the instruction, we are constrained when applying vector pairs using this transition. The second transition is from $M[A_i + 1]$ to $M[A_x]$, where $M[A_i + 1]$ contains the 8-bit address offset of the data to be loaded and $M[A_x]$ contains the actual data to be loaded. We chose this transition over the first transition, as it does not come with any constraints and thus can be used to apply crosstalk tests with maximum flexibility.

To apply an arbitrary vector pair ($v1$, $v2$) to the data bus, we need to have $M[A_i + 1] = v1$ and $M[A_x] = v2$. Hence, we need to load from an address with a specific offset ($v1$) containing a specific data ($v2$). For example, to apply (00000000, 11110111), one of the tests for positive glitch faults, we may load from address 1110 : 00000000 (offset = 00000000), which contains data 11110111). If the test passes, $v2$ (11110111) is loaded to the accumulator. If the test fails due to a positive glitch, $v2$ becomes 11111111. Thus a wrong value is loaded into the accumulator. The error response can be collected by storing the accumulator content to a specific memory address, which can be checked by an external tester upon the completion of the tests. Thus, a two-instruction sequence is needed for applying this test: (`lda` 1110 : 00000000, `sta resp`), where the content of memory address 1110 : 00000000 must be set to 11110111, and `resp` is the memory address where the test response will be stored.

Similarly, the same strategy can be used to apply tests for other types of crosstalk faults (negative glitch, falling delay, and rising delay) on the data bus.

4.2. *Testing Address Bus*

During the execution of the load instruction, there are two transitions on the address bus. The first transition is the increment from A_i to $A_i + 1$. The second transition is from $A_i + 1$ to A_x. Since the first transition comes with a much more strict constraint than the second one, we choose to use the second transition to apply the vector pair.

To apply an arbitrary vector pair ($v1$, $v2$) to the address bus, $A_i + 1$ must be $v1$ and A_x must be $v2$. Thus, the second byte of the instruction must be located at memory address $v1$, which implies that the instruction itself must be located at memory address $v1 - 1$. In addition, the data address accessed by the instruction must be $v2$.

4.2.1. Testing for Rising/Falling Delay Faults. For example, to apply (0000 : 00010000, 1111 : 11101111), one of the tests for falling delay faults, we place the load instruction at address 0000 : 00001111 ($v1 - 1$), and load from address 1111 : 11101111 ($v2$). If the test fails due to a falling delay defect, $v2$ becomes

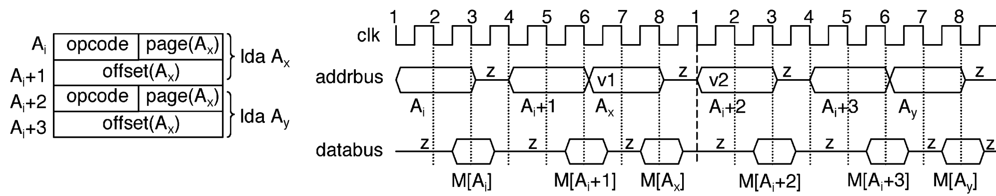

Fig. 6. Testing for positive/negative glitches on the address bus using two instructions.

1111 : 11111111 and we would be loading from address 1111 : 11111111 instead of address 1111 : 11101111. To observe the error, we store different values at address 1111 : 11101111 and 1111 : 11111111 (e.g., 00000001 at 1111 : 11101111 and 00000000 at 1111 : 11111111). If the test passes, 00000001 is loaded to the accumulator. If the test fails, 00000000 is loaded instead. Again, the error response can be collected by storing the accumulator content to memory. Thus, a two-instruction sequence is needed for applying this test: (`lda` 1111 : 11101111, `sta resp`), where the `lda` instruction is placed at memory address 0000 : 00001111, and the contents of memory address 1111 : 11101111 and 1111 : 11111111 are set to 00000001 and 11111111, respectively.

Tests for rising delay faults can be applied in a similar manner. However, tests for positive glitch/negative glitch faults are considerably different. This is because all tests for positive glitch faults start with vector 0000 : 00000000 (Fig. 1). To apply such a test using one instruction, the second byte of the instruction must be placed at memory address 0000 : 00000000. Moreover, to apply two vector pairs starting with the same vector (0000 : 00000000 in this case), two instructions need to be placed at the same memory location, causing an address conflict. This problem can be solved by utilizing the signal transition between *two* instructions.

4.2.2. Testing for Positive/Negative Glitch Faults. As shown in Fig. 6, we use two load instructions, "lda A_x" and "lda A_y", where A_x and A_y are the addresses to load from. The two instructions reside from memory address A_i to $A_i + 3$. To apply vector pair $(v1, v2)$ to the address bus, we use the transition between two instructions, from A_x to $A_i + 2$. Therefore, A_x, the address to be loaded from in the first instruction, has to be set to $v1$. $A_i + 2$, the address of the second instruction, has to be set to $v2$, putting the first instruction at address $v2 - 2$. As a result, we are able to avoid the problem of address conflicts when applying two vector pairs starting from the same vector. For example, to apply (0000 : 00000000, 1111 : 11111110), the vector pair needed for testing the positive glitch fault on bus line 1, we place the first instruction at address 1111 : 11111100 and use it to access address 0000 : 00000000. To apply (0000 : 00000000, 1111 : 11110111), the vector pair needed for testing the positive glitch fault on bus line 4, we place the first instruction at address 1111 : 11110101 and use it to access address 0000 : 00000000. There are no address conflicts between the two tests.

Fig. 7 illustrates the collection of test response for a positive glitch test, (0000 : 00000000, 1111 : 11101111). To apply this test, we place the first instruction at address 1111 : 11101101 and use it to access address 0000 : 00000000 (Fig. 7(a)). In the presence of a positive glitch fault, the execution of the second instruction is affected. Instead of executing the instruction located at addresses 1111 : 11101111 and 1111 : 11110000, the CPU now executes the instruction located at addresses 1111 : 11111111 and 1111 : 11110000 (Fig. 7(b)). Note that the first byte is changed while the second byte is not. To observe this error, we store different values at addresses 1111 : 11101111 and 1111 : 11111111 such that the content at 1111 : 11101111 corresponds to loading

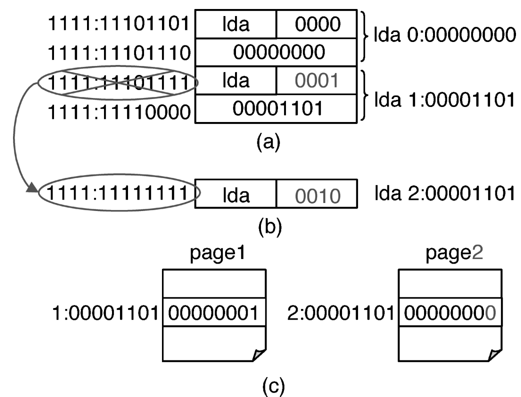

Fig. 7. Response collection.

from page 1 and the content of 1111 : 11111111 corresponds to loading from page 2. An arbitrary address offset, for example, 00001101, is stored at 1111 : 11110000. If the test passes, the CPU loads from address 1 : 00001101. If the test fails, the CPU loads from address 2 : 00001101 (Fig. 7(b)). We store different values at these two addresses such that the CPU loads a wrong value into the accumulator in the presence of the fault (Fig. 7(c)). The error response can be collected by storing the accumulator content to memory.

Thus, as shown in Fig. 7, to apply test vector pair (0000 : 00000000, 1111 : 11101111), a three-instruction sequence is needed: (`lda` 0000 : 00000000, `lda` 0001 : 00001101, `sta resp`), where the first `lda` is placed at address 1111 : 11101101, with the content of address 1111 : 11111111 set to "`lda` 0010" (the first byte of a `lda` instruction loading from page 2) and the content of address 0001 : 00001101 and 0010 : 00001101 set to 00000001 and 00000000, respectively.

4.3. Test Response Compaction

After each test, the test response is stored in the memory so that it can be unloaded and checked by an external tester. To reduce the data exchanged between the memory and the tester, we compact the test responses into as few bytes as possible without losing any diagnostic information.

Our basic idea for test response compaction is to sum up individual test responses into a collective test response. If the individual tests are properly constructed, the collective test response can show the pass/fail status of each test. Fig. 8 illustrate the test response compaction process, when tests for rising delay faults on the data bus (Section 4.1) are applied.

Since the ADD instruction has the same construct and timing characteristics as the load instruction (Fig. 8(a)), we use the add instruction to apply the tests. Fig. 8(b) shows the test program for the rising delay faults on all data bus lines. First, the accumulator is cleared. To apply the first test (01111111, 10000000), the test vector pair for the rising delay fault on bus line 8, we add to the accumulator the content of address 3 : 01111111, which is 10000000 (Fig. 8(c)). The page number, 3, is arbitrarily chosen. In the presence of a rising delay fault, the second vector in the vector pair, 10000000, becomes 00000000. Thus, 0 is added to the accumulator instead. The rest of the tests are

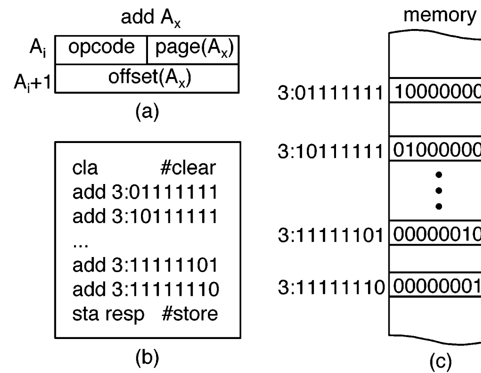

Fig. 8. Test response compaction.

constructed in similar manner. After the application of all tests, we store the accumulator content to a test response vector. If all tests pass, since we add 10000000, 01000000, . . . , 00000001 to the accumulator, the final test response is 11111111. Otherwise, at least one bit in the test response vector is 0. The position of the "0" bit tells which test failed.

Similarly, the test responses for the address bus can be compacted.

Depending on the actual instruction set of the processor core used in the SoC, the detailed constructs of the test programs may be different. Nonetheless, the general testing strategy we described here may be used to test the address/data busses between any CPU-memory pair. Moreover, since the cores in a SoC are often addressable by the CPU via memory-mapped I/O, the same test strategy can be extended to test address/data busses between any CPU-core pair.

With the software-based self-test approach, the proposed method has no area or delay overhead. Currently, the generation of test programs involves manual effort because it is instruction-set-specific. However, since we only need to consider the behavior of memory-accessing instructions when composing test programs, the test generation effort does not increase as the size of the instruction set increases. For each MA fault, a constant number of instructions are needed for applying the test. For a CPU-memory system with N interconnects, the number of MA faults is $4N$. Thus, the size of the test program is proportional to N. This corresponds to the size of the memory required for storing the test program, the tester time needed for loading the test program from the tester to the on-chip memory, as well as the test application time needed for executing the test program in the self-test mode.

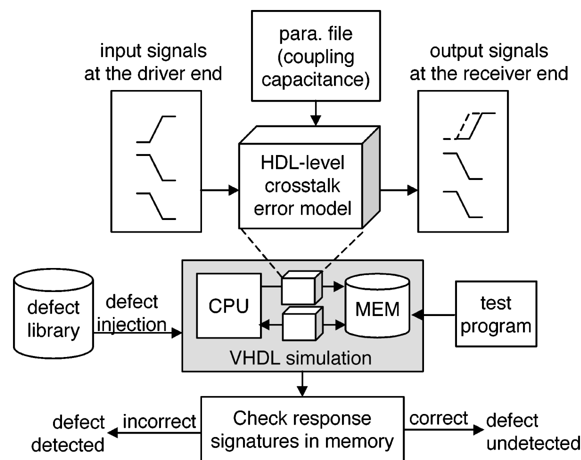

Fig. 9. HDL-level defect simulation environment.

5. Validation

To validate the proposed software-based self-test methodology, we composed a complete test program for the CPU-memory system described in Section 4 and evaluated its defect coverage under an HDL-level defect simulation environment (Fig. 9). During simulation, the CPU exercises the busses by executing the crosstalk test program stored in the memory. To simulate the effect of crosstalk *on HDL-level*, we used the high-level crosstalk error model proposed in [1]. Coded in HDL, the error model takes as input a parameter file containing the values of the coupling capacitance among interconnects. Given an input transition on the driver end of the bus, the error model determines whether a crosstalk error happens on the receiver end. Note that with this high-level crosstalk error model, we are able to take into account the effect of fault masking when evaluating defect coverage, since a crosstalk defect on the bus is indeed activated many times as the CPU executes the test program.

During the execution of the test program, the CPU stores test response signatures to the memory. Upon the completion of the simulation, we determine whether the defect has been detected by the test program by comparing the resulting response signature with its expected value. To estimate the defect coverage, the same defect simulation process is repeated on all defects from a pre-constructed defect library.

Fig. 10 illustrates the generation of the defect library. To generate the defect library, we first randomly perturb the nominal values of coupling capacitances among interconnects according to a given defect distribution. Given the resulting perturbation, we use the criteria in [8] to determine whether the perturbation is large enough to be detectable by *any* tests. In particular, if a perturbation in capacitance causes the net coupling capacitance (C) on any interconnect to be larger than a threshold value (C_{th}), it is recorded as a defect. The value of C_{th} depends on the value of acceptable delay length or glitch height. This is because in an RC network, given fixed resistance values, the amount of crosstalk-induced delays and the height of crosstalk-induced glitches on one interconnect increase monotonically as C increases. The process of defect generation is repeated until a satisfactory number of defects are generated. In this paper, we only consider crosstalk within the same bus when injecting defects. It is possible to inject defects causing crosstalk between two busses by treating them as one bus.

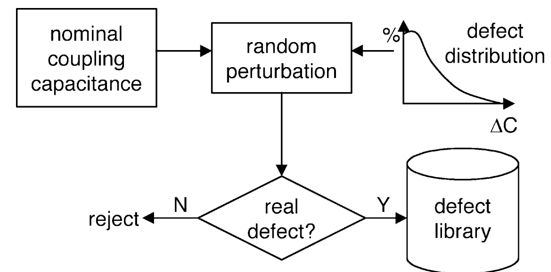

Fig. 10. The generation of defect library.

In our experiments, we used a Gaussian distribution to model the defect distribution in terms of the variation of capacitance values (in %). A 3δ point of 150% was chosen. A total number of 1000 defects were generated for each bus.

For the CPU-memory system described in Section 4, there are 64 MAFs on the 8-bit bi-directional data bus ($8 \times 4 \times 2$) and 48 MAFs on the 12-bit address bus (12×4). With the test program, we were able to apply 64 out of 64 MA tests for the databus and 41 out of 48 tests for the address bus. Some of the tests cannot be applied due to address conflicts—i.e., multiple tests compete for the same instruction address. This problem can be solved by separating conflicting tests into *multiple* test programs, which can be executed in different sessions. The total execution time of the programs is 1720 processor cycles. The size of the test program is proportional to the width of the busses, as a certain number of instructions are needed for testing for each MAF.

Fig. 11 shows the individual and cumulative defect coverage obtained by applying each of the MA test for

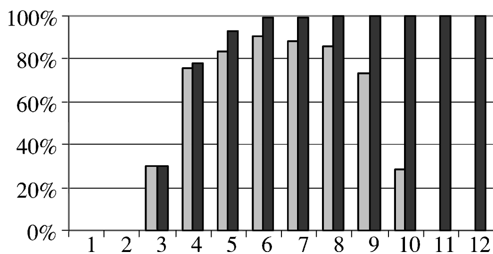

Fig. 11. Crosstalk defect coverage of MA test programs.

the address bus of the CPU-memory system. The horizontal axis indicates the MA test for each interconnect of the bus, with the ith test being the MA test for the ith interconnect. The individual defect coverages are shown in light gray, while the cumulative coverages are shown in dark gray. It can be seen that different MA tests have different levels of defect coverages, with the MA tests for the center interconnects having more coverage than the MA tests for the side interconnects. This is because the probability of a side interconnect being defective is small, as the net coupling capacitance on a side interconnect is smaller than the one on a center interconnect (i.e, a much larger perturbation is needed to render the side interconnect defective). In fact, the probability is so small such that in the particular defect library we have generated, no perturbation is large enough to cause Lines 1, 2, 11, and 12 to be defective. This is shown in Fig. 11, where the MA test programs for Lines 1, 2, 11, and 12 have no defect coverage. The cumulative defect coverage shows that the MA tests combined together provide a 100% coverage of the crosstalk defects on the address bus interconnects.

Since the MA tests are necessary for detecting all detectable defects [8], in theory, some of the defects can only be detected by the missing tests. However, using our defect library, the defect coverage of the test program is 100% on both address and data busses. This is because a large overlap exists among the defects set detected by different MA tests. Of all the defects detectable by one MA test, only a tiny fraction cannot be detected by any other MA tests.

6. Conclusions

At-speed testing for crosstalk effects is expensive with external testers. Built-in self-test for crosstalk may result in high test overhead or over aggressive testing. To address these issues, we proposed a cost-effective method for testing system-level interconnects using embedded processor cores. By executing a self-test program, a processor is able to test the address and data busses through which it communicates with memory components. The same method can be extended for testing the interconnects between the processor and non-memory cores, as these cores are typically addressed by the processor via memory-mapped I/O. We have constructed an HDL level defect simulation environment to validate the proposed method and evaluate the defect coverage of any given test program. Experimental results show that a self-test program written following the proposed method is able to achieve its projected defect coverage.

References

1. X. Bai and S. Dey, "High-Level Crosstalk Defect Simulation for System-on-Chip Interconnects," in *Proc. 19th VLSI Test Symp.*, Los Angeles, CA, April 2001.
2. X. Bai, S. Dey, and J. Rajski, "Self-Test Methodology for At-Speed Test of Crosstalk in Chip Interconnects," in *Proc. 37th Design Automation Conf.*, Los Angeles, CA, June 2000, pp. 619–624.
3. K. Batcher and C. Papachristou, "Instruction Randomization Self Test for Processor Cores," in *Proc. 17th VLSI Test Symp.*, Dana Point, CA, April 1999, pp. 34–40.
4. L. Chen and S. Dey, "Software-Based Self-Testing Methodology for Processor Cores," *IEEE Trans. Computer-Aided Designs*, vol. 20, no. 3, March 2001.
5. W. Chen, S.K. Gupta, and M.A. Breuer, "Test Generation in VLSI Circuits for Crosstalk Noise," in *Proceedings IEEE International Test Conference*, 1998, pp. 641–650.
6. W. Chen, S.K. Gupta, and M.A. Breuer, "Test Generation for Crosstalk-Induced Delay in Integrated Circuits," in *Proceedings IEEE International Test Conference*, Oct. 1999, pp. 191–200.
7. Z. Chen and I. Koren, "Crosstalk Minimization in Three-Layer HVH Channel Routing," in *Proceedings IEEE International Symposium on Defect and Fault Tolerance in VLSI System*, 1997, pp. 38–42.
8. M. Cuviello, S. Dey, X. Bai, and Y. Zhao, "Fault Modeling and Simulation for Crosstalk in System-on-Chip Interconnects," in *1999 Int. Conf. Computer-Aided Design*, San Jose, CA, Nov. 1999, pp. 297–303.
9. N. Itazaki, Y. Matsumoto, and K. Kinoshita, "An Algorithmic Test Generation Method for Crosstalk Faults in Synchronous Sequential Circuits," in *Proceedings Sixth Asian Test Symposium*, Nov. 1997, pp. 22–27.
10. A.B. Kahng, S. Muddu, E. Sarto, and R. Sharma, "Interconnect Tuning Strategies for High-Performance ICs," in *Proceedings Design, Automation and Test in Europe*, Paris, France, Feb. 1998, pp. 471–478.
11. K.T. Lee, C. Nordquist, and J. Abraham, "Automatic Test Pattern Generation for Crosstalk Glitches in Digital Circuits," in *Proceedings IEEE VLSI Test Symposium*, 1998, pp. 34–39.
12. Z. Navabi, *VHDL: Analysis and Modeling of Digital Systems*, New York: McGraw-Hill, 1993.
13. P. Nordholz, D. Treytnar, J. Otterstedt, H. Grabinski, D. Niggemeyer, and T.W. Williams, "Signal Integrity Problems in

Deep Submicron Arising from Interconnects Between Cores," in *Proc. 16th VLSI Test Symp.*, Monterey, CA, April 1998, pp. 28–33.
14. K. Radecka, J. Rajski, and J. Tyszer, "Arithmetic Built-in Self-Test for DSP Cores," *IEEE Trans. Computer-Aided Design*, vol. 16, no. 11, pp. 1358–1369, Nov. 1997.
15. Semiconductor Industry Association, *The International Technology Roadmap for Semiconductors*, 1999.
16. J. Shen and J.A. Abraham, "Native Mode Functional Test Generation for Processors with Applications to Self Test and Design Validation," in *Proc. Int. Test Conf.*, Washington DC, Oct. 1998, pp. 990–999.
17. A. Sinha, S.K. Gupta, and M.A. Breuer, "Validation and Test Generation for Oscillatory Noise in VLSI Interconnects," in *Proceedings of the International Conference on Computer-Aided Design*, Nov. 1999.
18. H. Zhou and D.F. Wang, "Global Routing with Crosstalk Constraints," in *Proceedings of the 35th Design Automation Conference*, San Francisco, CA, USA, June 1998, pp.374–377.

Li Chen (S'98) received the B.S. and M.S. degrees in electrical and computer engineering from Carnegie Mellon University, Pittsburgh, PA, in 1996 and 1998, respectively. She is currently a Ph.D. candidate in the Department of Electrical and Computer Engineering at the University of California, San Diego. From January 1997 to July 1997, she was with Advanced Micro Devices, Sunnyvale, CA. From June 2001 to September 2001, she worked at Intel Corporation, Santa Clara, CA. Her current research interests include the self-test and self-diagnosis of microprocessor cores.

Xiaoliang Bai (S'98) received the B.S. in electrical engineering from Xi'an Jiaotong University in 1995, and the M.S. degree in computer engineering from Peking University in 1998. He is working toward the Ph.D. degree in electrical and computer engineering at the University of California, San Diego. His research interests include signal integrity analysis and testing, timing analysis, circuit tuning and simulation.

Sujit Dey is a Professor in the Electrical and Computer Engineering Department at the University of California, San Diego. His research group at UCSD is developing configurable platforms, consisting of adaptive wireless protocols and algorithms, and deep submicron adaptive system-on-chips, for next-generation wireless appliances as well as network infrastructure devices. He is affiliated with the California Institute of Telecommunications and Information Technology, the UCSD Center for Wireless Communications, and the DARPA/MARCO Gigascale Silicon Research Center. Prior to joining UCSD in 1997, he was a Senior Research Staff Member at the NEC C&C Research Laboratories, Princeton, NJ. He obtained a Ph.D. in Computer Science from Duke University in 1991.

Dr. Dey has co-authored more than 100 publications, including journal and conference papers, a book on low-power design, and several book chapters. He received Best Paper awards at the Design Automation Conferences in 1994, 1999, and 2000, and the 11th VLSI Design Conference in 1998, and several best paper nominations. He is a co-inventor of 9 U.S. patents, and has 2 other pending. He has presented numerous tutorials and invited talks, and participated in panels, in the topics of low-power wireless systems design, hardware-software embedded systems, and deep sub-micron system-on-chip design and test. He has been the General Chair and Program Chair, and member of organizing and program committees, of several IEEE conferences and workshops.