A Memory Debug Methodology Using BIST

Luis Basto
Analog Devices

Sankaran Menon
Intel Corp.

1501 S. Mopac Expressway
MS AN1-4A
Austin, TX 78746

Key Words: BIST, Manufacturing Test, Latency, Debug, Design for Test

Abstract

Built-in self test (BIST) is commonly used as a technique to test embedded arrays such as RAMs and ROMs. The primary use of BIST is for manufacturing or production testing but additional features can be added for diagnostics and debug. Conventional memory BIST implementations provide Pass/Fail information, which is usually sufficient for manufacturing test. This paper describes a novel feature of BIST using "Resume Mode" to start and stop BIST to capture memory failure information so it can be bitmapped. It also covers some issues of memory latency that causes complexity in implementation. This technique is useful both for engineering debug as well as for production testing.

Introduction

Rapidly increasing circuit complexity along with extremely stringent product quality requirements have made built in self test (BIST) a viable test solution for most processors. Most System-on-a-Chip (SOC) implementations use a variety of embedded cores and memory arrays, such as blocks of random logic, caches, SRAMs, etc. Fault free operation of the embedded logic within an SOC drives the need for implementing efficient test methodologies.

A variety of embedded memory arrays, such as caches, SRAMs, register files, branch prediction tables etc., exist in current microprocessors [1]. With the growing demand for storage capacity in palm-size PDAs (Personal Digital Assistants) and cellular phones with web-browsing capability, the size of memories is increasing drastically as opposed to the increase in size of the random logic. In addition, advances in memory technology and shrinking geometries are causing memory testing to be a constant challenge. With limited accessibility of embedded cores, conventional test methods using external testing do not provide adequate results.

Embedded memories are best tested using a Built In Self Test (BIST) technique which not only help achieve manufacturing test, but also for power-up testing. Several BIST methodologies have been proposed for memory testing [2-9]. The BIST methodologies presented in the above work offer solutions for testability of embedded memories that go into the present day System-on-a-Chip (SOC). Implementing memory BIST minimizes the embedded memory tester's requirements and greatly reduces memory test time within the test flow. Total test time is also reduced with BIST, since parallel testing can be carried out on the various embedded memories that exist within the SOC. Memory BIST provides a good way to protect the intellectual property contained within the core. In case of embedded cores, the core provider only delivers the BIST stimulus and response sequences for testing and diagnosis without divulging any details of the design.

Memory failures can be single or multiple bits. For all types of failures, but especially for multiple bits, it would be desirable to be able to diagnose and track all single and multiple bit failures. This would be useful in a debug or product test environment where there is a need to map out all the failures of the memories to detect pattern or process contributions.

Details of methodology

Memory BIST alleviates long test times by incorporating a self testing logic within the memory itself. Another major advantage of memory BIST is that the test can be run at-speed, which detect errors that generally occur

at the functional operating speed of the device.

Basic memory BIST implementations provide Pass/Fail information which is usually sufficient in production testing. However, in designs where failed locations are required to be known, then bit failure information corresponding to the failed locations need to be logged.

The memory BIST under discussion uses a March algorithm[10] to test all addresses, data locations, and address decoders for multiple failure mechanisms. These include memory bit stuck-at faults, shorts between bit lines, shorts between word lines, and coupling faults. When a failure is encountered, the test would stop and the failing address and data patterns captured into a failure buffer. This data can be scanned out and analyzed.

Figure 1 shows a memory BIST implementation with the top half of the Figure showing the traditional memory BIST. The address counter provides the address to the collared RAM and the March algorithm data is provided as Data-in. The data read out from the memory corresponding to the word written is compared within the comparator logic and a Pass/Fail flag is issued.

The bottom half of the Figure shows the BIST resume logic to log the failed addresses. When a BIST failure occurs, the control logic will stop the BIST and with the fail signal, the corresponding address is latched into a register, shown as “Address of Miscompare”. Similarly, the data corresponding to the failed location is also latched into a register, shown as “Miscompared Data”.

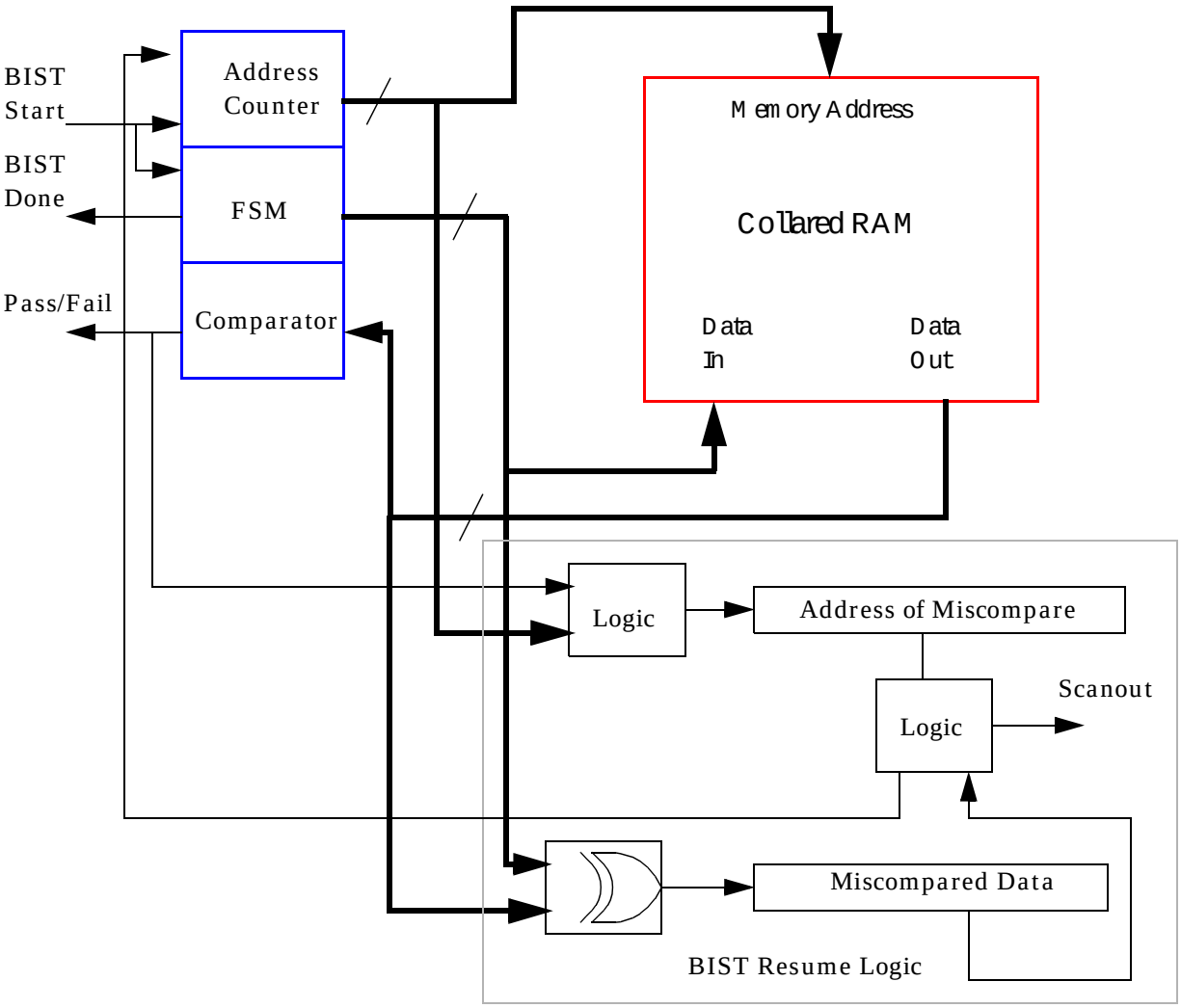

Figure 1: BIST Resume implementation

The BIST controller can be configurable to run in Pass/Fail mode, which is the typical manufacturing test mode, or it can be used for debugging memory failures. In Pass/Fail mode, if a failure occurs, its information is either not logged or only the first failure logged. The choice is left to the specific implementation. A flag indicates the status and scanned out at the end of the test.

BIST resume mode makes use of JTAG to set up and control its functionality. Testing is initiated similar to Pass/Fail mode. When a failure occurs, the BIST controller is halted and the failed address and data shifted out. The user now has the option of entering BIST resume mode. On re-entry, the address counter is incremented by one and BIST restarted at the point where it stopped. Testing continues until the next failure. This sequence is repeated until there are no more failures or test completion. With this technique, the user can have a bitmap of all failures in the memory which will help greatly in manufacturing or process debug.

Effects of Timing Latency

In the ideal case with no memory timing latency the method describe above would work perfectly. However, owing to architecture requirements, an embedded memory in a processor design usually have some timing latency from applying a valid address to loading or storing valid data into the array.

Figure 2 shows a memory array with pipeline stages at both the entry and exit ports of the memory. For general discussion, let the number of input stages be N and the number of output stages be M. In an ideal situation, N and M are both zeroes. Now let us look at the cases where N and M are non-zero. In this situation, after a valid address is issued, data will take N clock cycles to be written into the memory array for a write cycle and M clock cycles to be read out for a read cycle.

In order to diagnose failures in this memory implementation, the BIST logic must take the timing latency into account. There are three potential ways to solve this problem.

1. Reuse existing logic
2. Mimic logic
3. Altering the March sequence by inserting redundant writes

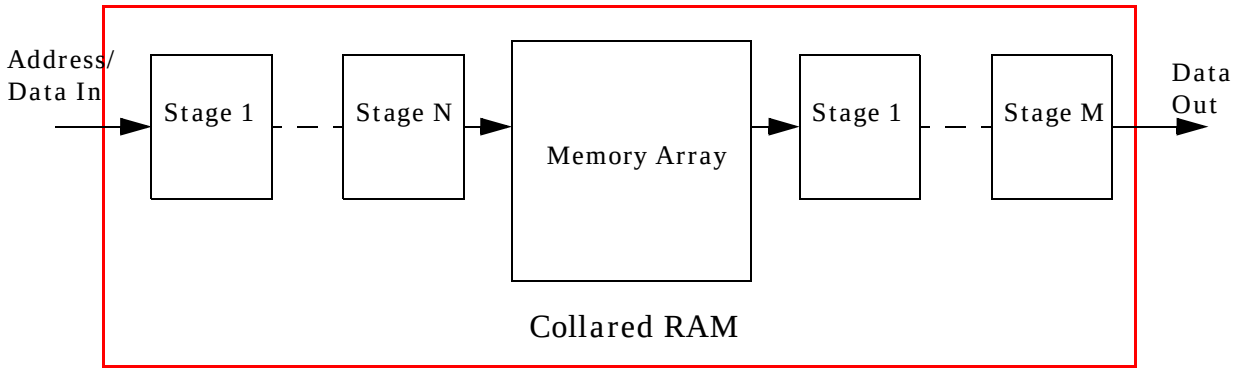

Figure 2: Memory with pipeline latency

The first solution requires multiplexing test values into the address and data path. This is not acceptable since these are usually timing critical paths and designers will not allow additional test logic to be added to the path.

The second solution requires the BIST logic to replicate the timing pipeline of the system logic. This is feasible if the total number of pipeline stages is very small, such as one or two. Anything deeper would be too costly to implement.

Finally, there exists the possibility of inserting redundant write cycles in the March algorithm to compensate for the effect of pipeline staging. As an example, let's look at a MATS++[10] implementation with two stages of pipelines. The standard pattern is:

M0: $w0^{\wedge}$

M1: $(r0w1)^{\wedge}$

M2: $(r1w0r0)_{v}$

The altered algorithm becomes:

M0: $w0^{\wedge}$

M1: $(w0r0w1)^{\wedge}$

M2: $(w1r1w0w0r0)_{v}$

Notation:

$w0^{\wedge}$ = write a 0 and increment address

$r1_{v}$ = read a 1 and decrement address

Since the pipeline has a depth of two in this example, every read must be preceded by two writes, therefore adding one redundant write cycle per read. The general form of this algorithm can be extended to $(wx^{n-1}rx)$ where n is the number of pipeline stages and x is the true or complement data being tested, or 1 and 0 respectively.

The result of this scheme is longer test time. The size of the memories to be tested will help justify whether this solution is feasible.

Besides longer test time, an unanswered question is whether modifying the March sequence in this manner invalidates the test. Further research is needed to find the answer.

Results and Conclusions

Rapidly increasing complexities coupled with extremely stringent product quality requirements have made Built-In Self Test (BIST) a viable test solution for most microprocessor and digital signal processors. This becomes particularly important with the use of embedded cores and memory arrays within the System-on-a-Chip (SOC). In conventional Memory Built-In Self Test schemes, the BIST is designed to provide Pass/Fail information on the memory module that is being tested.

The BIST Resume methodology presented here is useful for getting information on the failed locations and the data bits that have failed and allowing the user to continue until all failure data is mapped. This scheme is applicable both as a benchtop test as well as during manufacturing test.

Embedded arrays with timing latency have the additional complexity of replicating the data stored in the pipeline stages. Solutions proposed include adding test logic to mimic the pipeline or adding redundant write cycles to the test algorithm to preserve the data stored in the pipeline.

The technique presented in this paper is planned for implementation in future products by Analog Devices and Intel Corp.

References

[1] M. E. Levitt, "Designing UltraSparc for Testability," IEEE Design and Test of Computers, Vol. 14, No.1, pp. 10-17, January - March 1997.

[2] K. Kinoshita, K. K. Saluja, "Built-In Testing of Memory using an On-Chip Compact Testing Scheme," IEEE Transactions on Computers, Vol. C-35, No. 10, pp. 862-870, Oct. 1986.

[3] K. T. Le, K. K. Saluja, "A Novel Approach for Testing Memories using a Built-In Self Testing Scheme," Proc. IEEE International Test Conference, pp. 830-839, 1986.

[4] P. H. Bardell, W. H. McAnney, and J. Savir, Built-In Test for VLSI: Pseudorandom Techniques, John Wiley and Sons, 1987.

[5] R. David, A. Fuentes and B. Courtois, "Random Pattern Testing Versus Deterministic Testing of RAMs," IEEE Transactions on Computers, Vol C-38, No. 5, pp. 637-650, May 1989.

[6] R. Dekker, F. Beenker and L. Thijssen, "Realistic Built-In Self Test for Static RAMs," IEEE Design & Test of Computers, Vol. 6, No. 1, pp. 26-34, Feb. 1989.

[7] B. Nadeau-Dostie, A. Silburt and V. K. Agarwal, "Serial Interface for Embedded-Memory Testing," IEEE Design & Test of Computers, Vol. 7, No. 2, pp. 52-63, April 1990.

[8] M. Nicolaidis, "Transparent BIST for RAMs," Proc. IEEE International Test Conference, pp. 598-607, Oct. 1992.

[9] H. Cheung, S. K. Gupta: "A BIST Methodology for Comprehensive Testing of RAM with Reduced Heat Dissipation," Proc. IEEE International Test Conference, pp. 23-32, Oct. 1996.

[10] A. J. van de Goor, Testing Semiconductor Memories, John Wiley and Sons, 1995.