



and costs up to
40% less than other solutions.

Agilent Technologies

Click here
to find out more.

[<<< Back](#) | [Print](#)



This article appeared on the Test & Measurement World web site at www.tmworld.com. © 2005

New methods test small memory arrays

Jeff Boyer, Intel, Austin Design Center, Austin, TX; and Ron Press, Mentor Graphics, Wilsonville, OR -- 2/1/2003

Test & Measurement World

[Forecasting test times](#)

As IC geometries shrink, the large, consolidated memory blocks within ICs are giving way to tens or even hundreds of smaller memory arrays distributed throughout each chip. These arrays serve as register files, FIFOs, and small, performance-critical memories in memory-management systems (such as tag arrays). Finding ways to test these small arrays for speed-related defects and stuck-at faults proves challenging.

Traditionally, you test a digital chip using functional patterns and patterns created by automatic test-pattern-generation (ATPG) tools. Such tools primarily generate scan-based test patterns for random logic and do not excel at grading the memory portions of a device under test (DUT). To grade memory, you could employ memory built-in-self-test (BIST) capability.

Memory arrays, especially those fabricated in new processes, suffer from a variety of potential defects whose occurrences are hard to predict. Traditional memory BIST approaches implement March (Ref. 1) and other algorithms that repeat simple test sequences designed to detect most memory faults. In one memory BIST approach, state machines within the DUT itself apply and analyze the test patterns needed to test each address of the memory, off-loading memory-test chores from external ATE. Alternatively, designers can assign memory-test tasks to an on-chip processor (Ref. 2), but it's often difficult to judge the effectiveness of this approach until the DUT design is nearly complete.

Unfortunately, small memory arrays don't often justify adding memory BIST logic because of its impact on chip area and performance. Memories that have few addresses but many ports are particularly poor candidates for BIST. The gates required to implement the BIST controller could be as large as the memory itself because the number of memory ports has a bigger impact on the size of the BIST controller circuitry than the memory array size does. Memory BIST also requires that all memory input pins have a multiplexer to select between BIST and system signals. The mux circuitry for multiple-port

memories can lead to significant routing congestion and can unacceptably degrade performance.

Embedded memory-test options

Given the overhead that BIST entails for small memory arrays, one option is to simply not test them—a poor career choice if customers receive defective parts. Or, if you would enjoy spending more time at the office, you could manually create patterns that implement the necessary test algorithms at each memory array.

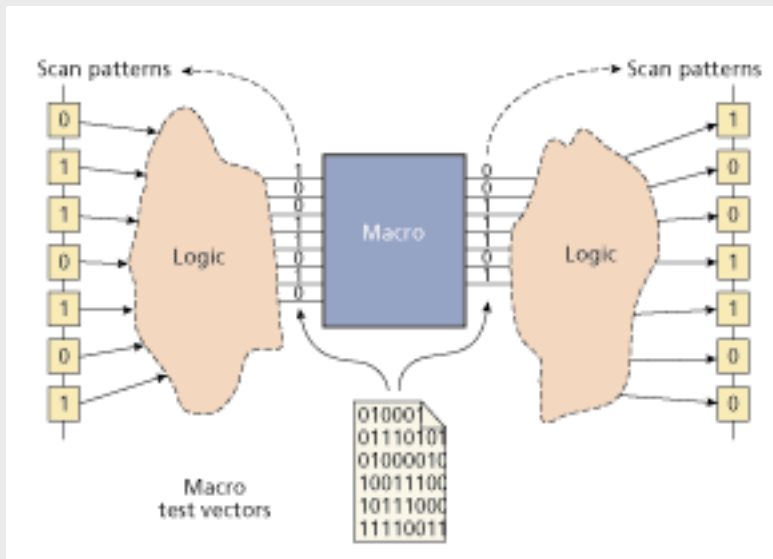


Figure 1 Macro testing employs vector-translation techniques to propagate test vectors between scan cells and an internal macro, such as an embedded memory array.

Fortunately, a more effective option exists—you can take advantage of ATPG engines and your DUT's scan access to apply test vectors at each embedded-memory input and then read the response from the output. This relatively recent capability—sometimes referred to as vector translation or macro testing (**Figure 1**)—allows you to apply a macro vector sequence necessary to test a stand-alone embedded block (also referred to as a macro). A few EDA companies offer tools with such functionality; such tools automatically convert the macro vector designed for a stand-alone memory (macro) into a chip-level scan pattern, and they propagate the expected outputs to scan cells for verification.

As a result, macro testing performs the desired test at the embedded block without any additional test logic. The resulting embedded-memory scan pattern can take on the same simple test protocol as a standard stuck-at scan pattern, thereby reducing vector debug time on production testers. Some companies are applying macro-test techniques to over 100 memories in parallel; the resulting macro-test scan patterns are only as long as the macro with the longest pattern set. This technique can be used to test any embedded macro—even a black box—as long as the patterns are defined at the block I/O and such a pattern can be propagated through the surrounding logic.

Embedded-memory timing faults

Like random logic within a design, embedded memories must be tested for both static faults and at-speed faults. Because memory BIST often runs from the system clock and is called "at-speed" memory BIST, it may seem like a logical choice for these tests. Yet, even though the BIST control logic often uses the system clock to set up test sequences, it requires several cycles to perform a single read or write operation. Therefore, even though "at-speed" memory BIST uses the system clock frequency, it may not perform read and write cycles as the chip would in normal operation. Pipelining the write and read operations results in a more effective test—called "full-speed" memory BIST (**Figure 2**)—performing back-to-back write or read operations in consecutive clock cycles, just as the memory would behave in normal system operation.

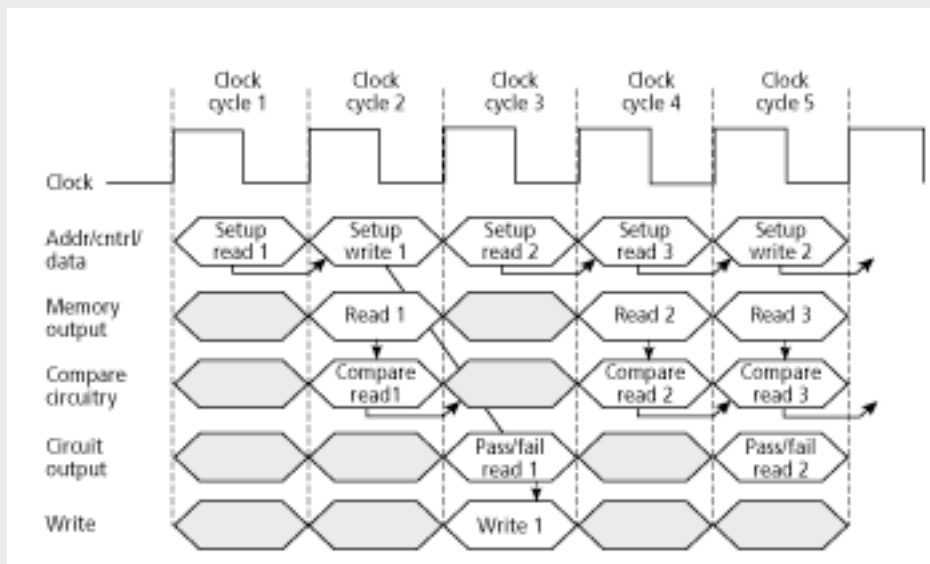


Figure 2 Pipelining enables back-to-back write or read operations on consecutive clock cycles, providing full-speed memory BIST capability.

Full-speed memory BIST presents a nice option for testing large memories for speed-related defects. But testing smaller memories or timing-critical memories for speed-related defects remains problematic. Macro testing tools convert each functional pattern into a scan pattern, so the application of each macro vector is slow. Furthermore, some embedded memories are fully synchronous and share the same clock with the scan chain. If the macro is clocked, then the scan cell values that were previously initialized are updated with new data, invalidating the scan cell values required to propagate macro outputs. Consequently, traditional macro testing cannot be applied to synchronous memory without gating the clock.

Synchronous macro testing solves this problem; it allows for several full-speed operations without reloading the scan chain. Synchronous macro testing determines the scan cell values needed to produce the first macro vector and loads them in the scan chain. At the same time, it determines the scan cell values for the second macro vector and pipelines those values at the scan cell inputs. After the scan chain is loaded, several clocks can be toggled at-speed, to apply several full-speed vectors

to the macro. Full-speed tests can be applied nonintrusively to almost any block within a design. Moreover, the full-speed test uses only the functional logic, so it can perform a truer full-speed test than is possible using test logic.

Figure 3 shows a portion of a macro-testing pattern file used to test the memory with a read/write/read sequence. The scan chain is loaded once; then three consecutive clock pulses result in a write, read, and capture of read values. The synchronous macro testing feature embedded in an ATPG tool predicts all values that must be loaded in the scan chain to ensure that the scan cells capture appropriate values with each scan and memory clock pulse. After the write-cycle clock pulse, the scan cells capture values resulting in address 0 and write-enable 0 at the memory.

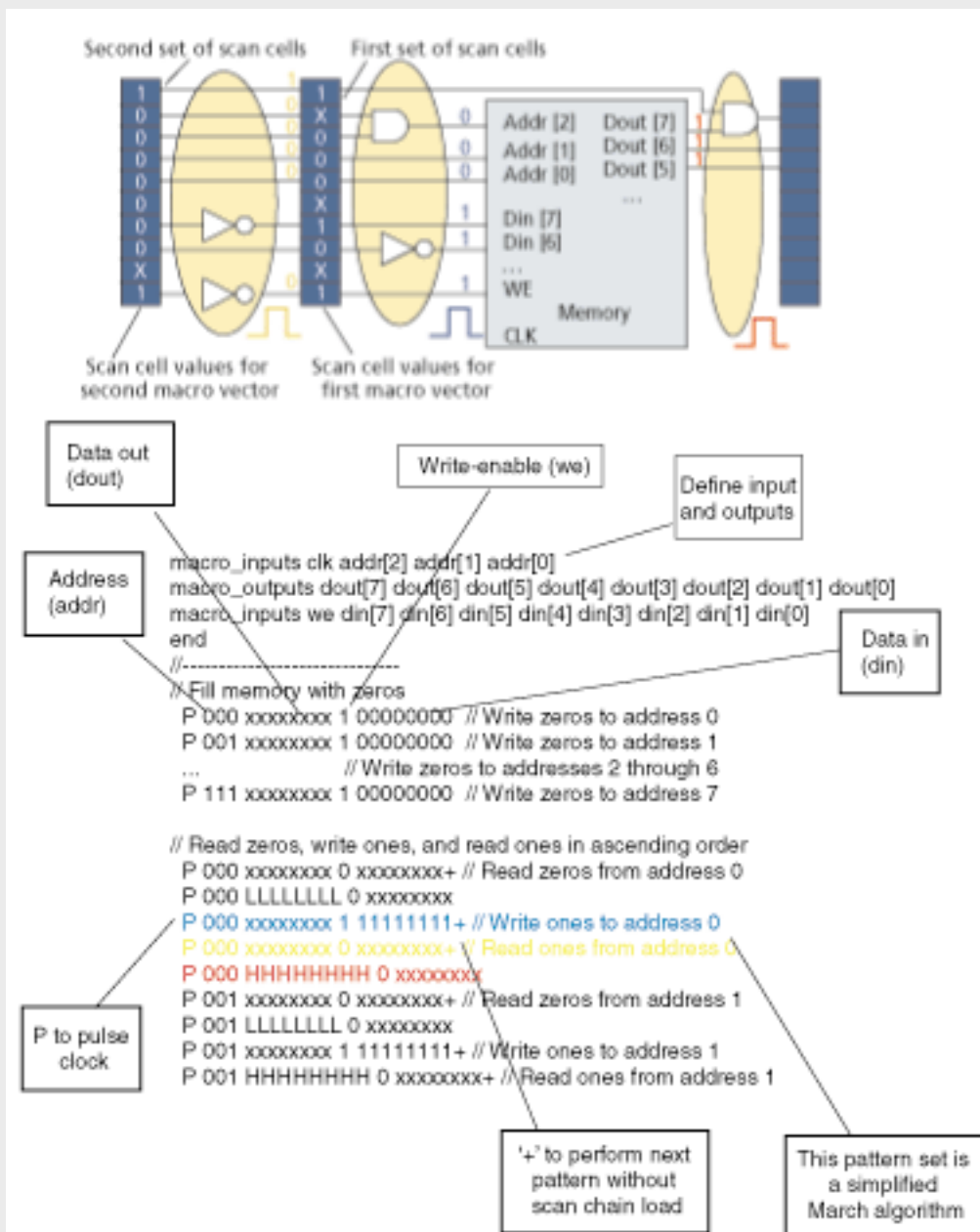


Figure 3 A macro-testing pattern file applies a read/write/read sequence that occurs in three consecutive clock cycles.

This form of macro testing allows any type and number of memory tests to be applied as long as they are possible with the scan-inserted logic surrounding the memory (or other block). As the number of consecutive patterns desired using one scan load increases, however, so does the complexity of processing the appropriate values to load into the scan chain. We recommend you perform no more than four consecutive macro-testing patterns before performing a new scan-chain load.

Test-time examples

We applied a commercial DFT tool's at-speed macro testing features to test two Intel Xscale silicon-based products. (See "[Forecasting test times](#)," below, for a description of vector-depth and test-time calculations.)

First, we tested nine identical 512-byte SRAMs using macro testing with synchronous capability, eight in parallel and one stand alone. The eight parallel memories have a 33-MHz test frequency requirement, and the stand-alone memory has a 66-MHz test frequency requirement. **Table 1** summarizes the test time and vector depth results for at-speed fault detection for each SRAM.

Second, we tested 13 register file memories (RFM) with varying address depths, scan-chain lengths, and frequency targets. **Table 2** summarizes the test time and vector depth results for at-speed fault detection for each RFM. Total vector depth is the sum of the largest vector depths at each test frequency point. Total test time is the sum of the longest RFM test time at each test frequency point.

Our tests illustrate three points. First, if at-speed fault detection is a requirement, then for a given algorithm, you should test all memories running at a given high frequency in parallel. Second, if at-speed fault detection is not a requirement, you should test all memories in parallel at a slower test frequency for static fault detection for a given algorithm. In this case, total vector depth reduces to that of the largest individual vector depth. Similarly, total test time becomes the product of the largest individual vector depth and the slower test frequency. Third, you can see how scan chain length heavily influences test time and vector depth, so keep scan-chain length to a minimum when using macro testing techniques via scan to fault-grade only small embedded memories. (This suggestion doesn't apply if you perform random fault detection via scan concurrently with fault detection on the embedded memory via scan macro testing.)

What should you do?

In the real world of test, you'll always face tradeoffs among the costs of design-for-test tools and overhead, test-development time, tester time, vector-debug time, and test coverage. Macro testing is a tool you can add to your repertoire to effectively test the growing number of small embedded blocks for both static and at-speed defects. You can even apply it to larger memories to perform several full-speed tests automatically through the IC's functional logic.

Table 1. At-speed macro testing for SRAM example

Memory Size (Bits)	Address Depth	Test Frequency (MHz)	Scan-Chain Length	MarchC-Vector Depth	MarchC-Test Time (ms)	MarchC+Vector Depth	MarchC+Test Time (ms)
1 x 512	512	66	128	393216	5.96	589824	8.94
8 x 512	512	33	128	393216	11.92	589824	17.88

Table 2. At-speed macro testing for RFM example

Memory Size (Bits)	Address Depth	Test Frequency (MHz)	Scan-Chain Length	MarchC-Vector Depth	MarchC-Test Time (ms)	MarchC+Vector Depth	MarchC+Test Time (ms)
3 x 896	128	66	102	78336	1.19	117504	1.78
2 x 896	64	66	102	39168	0.59	58752	0.89
1 x 192	64	66	66	25344	0.38	38016	0.58
4 x 128	32	66	107	20544	0.31	30816	0.47
2 x 560	112	48	105	70560	1.47	105840	2.21
1 x 1024	128	26	71	54528	2.10	81792	3.15

Author Information

Jeff Boyer holds a BS and MS in electrical engineering from Texas A&M University. He is currently a DFT engineer for Intel in Austin, TX.

Ron Press holds a bachelor's degree in electronic and electrical engineering from the University of Massachusetts and is currently a technical marketing manager for design-for-test products at Mentor Graphics, Wilsonville, OR.

References

1. van der Goor, A.J., *Testing Semiconductor Memories: Theory and Practice*, John Wiley & Sons, 1991.
2. Rajsuman, R., "Testing a System-On-a-Chip with Embedded Microprocessor," *Proceedings of the 1999 International Test Conference*, pp. 449-508.

Forecasting test times

To forecast vector depth and test time for macro testing patterns, first compute vector depth as follows:

num_scan_load_operations

**memory_address_depth*scan_chain_length*

where *num_scan_load_operations* depends on the memory algorithm chosen, *memory_address_depth* depends on the memory under test, and *scan_chain_length* depends on the scan chain partitioning for the device under test (DUT).

To compute *num_scan_load_operations*, consider this typical 10n MarchC- memory algorithm:

$$\begin{matrix} (w0)^{\wedge} & + & (r0 + w1)^{\wedge} & + & (r1 + w0)^{\wedge} & + & (r0 + w1)_{\vee} & + & (r1 + w0)_{\vee} & + & (r0)_{\vee} \\ 1 & & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 \end{matrix}$$

where *w0* is write data, *w1* is write inverse data, *r0* is read data, *r1* is read inverse data, the caret indicates ascending order, and the inverse caret indicates descending order.

To optimally detect speed-related faults, back-to-back read + write operations 2 and 3, 4 and 5, 6 and 7, and 8 and 9 should all occur synchronously at-speed. Each of these represents one scan load/unload operation. Adding to this the first ascending *w0* operation and the final descending *r0* operation yields a *num_scan_load_operations* value of 6.

You can similarly compute the *num_scan_load_operations* for a 13n MarchC+ algorithm:

$$\begin{matrix} (w0)^{\wedge} & + & (r0 + w1 + r1)^{\wedge} & + & (r1 + w0 + r0)^{\wedge} & + & (r0 + w1 + r1)_{\vee} & + & (r1 + w0 + r0)_{\vee} \\ 1 & & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 & 13 \end{matrix}$$

To optimally detect speed-related faults, back-to-back write-plus-read operations 3 and 4, 6 and 7, 9 and 10, and 12 and 13 should occur synchronously at-speed. Adding these and the remaining operations yields a *num_scan_load_operations* value of 9. Once you know the vector depth, calculate test time as the product of vector depth and the test-clock period for the memory under test.--*Jeff Boyer and Ron Press*

© 2005, Reed Business Information, a division of Reed Elsevier Inc. All Rights Reserved.