Use test data to diagnose failed memory

Martin Keim - October 16, 2014

Here's a common situation for memory designers: your RAM or flash has been fabricated, but the yield is lower than you would like. Maybe there is an odd correlation between a particular failure mode and your manufacturing process, but you'll need solid data to track down the yield limiter. The only information you get from the tester is pass/fail, so it doesn't really help you to identify the root cause.

There are many different ways to get more details from your failing memories. As usual, there are pros and cons to each of them, but overall, you want meaningful data at the lowest cost. In the test world, costs can come in several forms: on-chip hardware dedicated to testing and data collection, the volume of test patterns that must be stored on a tester, and the time it takes to execute a test. For tests applied from a tester, execution time is usually determined by the number of test patterns and the speed at which they can be applied. However, when the testing is performed using BIST (built-in self-test), determining test execution time is not straightforward, especially when you are trying to collect data during test.

The most obvious method of collecting fail data from memories is to bring all memory ports to the top level of the design, create a physical interface to the tester and collect data from there. For many reasons, this method is just not practical for large, complex SoCs (system on chips) with many deeply embedded memories. Just to name a few reasons, the high number of required paths consumes chip real estate and causes routing congestion; it also requires many I/O pins, which drives up chip cost and may even exceed the number pins available on the tester.

Another approach is to employ BIST circuitry on the chip. Memory BIST (MBIST) is, however, usually designed to provide pass/fail data only. Unless an MBIST controller is specifically designed for data collection, you won’t get much useful information from it. With a purpose-built MBIST controller, both test application and data collection can be done on-chip at full operational clock speeds in the GHz range. Of course, at the end of a test pass, the data collected by the MBIST controller will be scanned back into the tester for subsequent processing.

To provide this type of functionality, the on-chip MBIST controller must be programmed to stop on the occurrence of a failure, and then scan out the failure data, while running any of the supported test algorithms. The commands for the MBIST controller are typically sent from the tester through unique command patterns. First, the external tester scans in an initialization sequence, then tells the MBIST controller (through the scanned in command pattern) to stop on the first failure. Once the controller has reached the first failure, it stops and the captured failure data is scanned out. The tester then applies a new command pattern, which tells the controller to stop at the second failure. Testing commences; failure data is scanned out; and so on. This works great, since you can now collect any number of failure data sets, up to a maximum number you had defined when generating the MBIST controller hardware.
There is one drawback to this approach—telling the controller to stop at the Nth failure involves embedding the desired failure number (N) into the command pattern being scanned in from the tester to the MBIST controller. That is, the pattern must be changed for each iteration. Because of this, the chip initialization sequence (e.g. clock selection, PLL locking) is also be repeated for each pass, and it requires you to either store a large number of individual command patterns on the tester (one for each different failure number, N). Alternatively you can generate the new command patterns on-the-fly by having the tester software call a special routine to modify the current command pattern by incrementing the value of N. Test engineers often view this as a very cumbersome procedure to setup and maintain.

To improve on this, you can use a technique called ESOE (Enhanced Stop on Error) in which the controller circuitry itself is enhanced so it not only stores the targeted failure number internally, like in the previous case, it automatically increments the stored failure number N when so instructed by a command pattern. In the ESOE case, it is the same pattern that is applied for each next pass. The controller will automatically halt at the next failure and scans out the collected data. The ESOE controller stops the sequence once the test algorithm has run to completion with no new failure detected.

This technique not only simplifies the pattern application (it is always the same pattern, with no special tester software needed), it also eliminates the need to re-initialize the MBIST controller for each iteration, so it significantly reduces the time needed for data collection, as Table 1 shows.

Table 1. Failure Data Collection Times

<table>
<thead>
<tr>
<th>Failure Data Collection Time [ms]</th>
<th>Algorithm Complexity</th>
<th>Pattern Patching</th>
<th>ESOE</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Failure Assumption A</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8N operations 2N compares</td>
<td></td>
<td>26,272</td>
<td>2,897</td>
</tr>
<tr>
<td>44N operations 18N compares</td>
<td></td>
<td>263,629</td>
<td>39,562</td>
</tr>
<tr>
<td><strong>Failure Assumption B</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8N operations 2N compares</td>
<td></td>
<td>26,272</td>
<td>616</td>
</tr>
<tr>
<td>44N operations 18N compares</td>
<td></td>
<td>263,629</td>
<td>8,428</td>
</tr>
</tbody>
</table>

(Experiments based on an ESOE MBIST controller generated by the Mentor Graphics Tessent DFT platform.)

The experiments in this table compare the average failure data collection time for different algorithms and failure assumptions. The underlying memory and tester model is the following: 32kx32 memory, a column has 256 cells, 200 MHz system (test) clock, 50 MHz scan clock, maximum number of tester cycles with compare errors that can be logged is 512, pattern patching time 40 ms. Failure assumption A is defined by all chips having column failures. Failure assumption B is that
80% of all chips have four individual cell failures and 20% have column failures. The table shows the time required for data collected under several different scenarios. In all cases there is a substantial reduction in collection time, ranging from a 6x to a 42X reduction factor.

The on-chip hardware required to implement this solution is one additional counter to keep track of the iteration number of the diagnosis loop. For example, for 4095 failures, this would be a 12-bit counter per controller, plus two control bits shared by all controllers. The comparison above demonstrates the advantage of ESOE over other data collection methods, given the very small investment in chip real estate.

Also see
- Noise Insensitive Digital BIST
- Logic BIST Applications and Usage
- Perform thorough memory testing