Tricks of the Test Trade: ATPG Methods that Improve Fault Coverage of SoC Devices

Michael Lewis and Leah Clark

Cypress Semiconductor

uml@cypress.com
xlc@cypress.com

ABSTRACT

As system-on-a-chip (SoC) design becomes more prevalent, test-for-manufacturability becomes essential. Automatic Test Pattern Generation (ATPG) has become a critical factor in the success of SoC components. However, these types of devices often contain large amounts of embedded memory, and it can be problematic to achieve acceptable fault coverage of the shadow logic immediately surrounding RAM and ROM blocks with ATPG alone. Many creative techniques have been applied to resolve this issue, but most require extensive on-chip infrastructure. This paper describes a method to obtain a high degree of fault coverage on embedded memories and their test structures with little additional logic.

1.0 Introduction

Test tools have a difficult time fault-grading combinatorial logic surrounding embedded memory cells. It is true that test tools are improving in their capability of testing shadow logic through the use of behavioral memory modeling. However, vendor support of such models is not ubiquitous for this technology yet. This leaves the burden of fault-grading logic immediately surrounding embedded memory cells up to the designer.

The problem of covering shadow logic through means of scan-based test is so prevalent that many clever techniques have been defined to work around this problem, but most make tradeoffs of one kind or another. Scan-Through-RAM[1] (STR), an ATPG methodology described in this paper, is a novel and simple concept which works around the tool problem without making the same level of sacrifice.

This paper will describe a few of the more popular solutions to handling embedded memories in scan-based designs. It will also demonstrate the advantages of using STR over these methods.

Note that this flow was implemented on a design that was started almost two years ago, and so used only Test Compiler as our DFT and ATPG tool. Although some of the implementation details presented here are based on Synopsys' Test Compiler tool (which is being phased out), the methodology applies to designs with embedded RAM using Design for Test Compiler (DFTC) and TetraMax, or any other ATPG tool.

2.0 Motivation

Why is shadow logic so problematic for ATPG tools to fault-grade properly? Let's first define the concepts of *observability* and *controllability* as they relate to ATPG testing.

- *Observability:* A node is observable if you can predict the response on it and propagate the fault effect to the primary outputs where you can measure the response. A primary output is an output that can be directly observed in the test environment.

- *Controllability:* A node is controllable if you can drive it to a specified logic value by setting the primary inputs to specific values. A primary input is an input that can be directly controlled in the test environment.

In Figure 2-1, we see that the input shadow logic of a memory cell is not *observable* since it cannot be "captured by a scan chain or a primary output," but rather by the memory cell only. Further, the output shadow logic of a memory cell is not *controllable* since it cannot be "driven by a scan chain or a primary input," but instead by the memory outputs.

[1] Cypress has filed a patent application related to this idea, entitled, "A METHOD AND SYSTEM FOR TESTING THE LOGIC OF A COMPLEX DIGITAL CIRCUIT CONTAINING EMBEDDED MEMORY ARRAYS" on December 6, 2000.

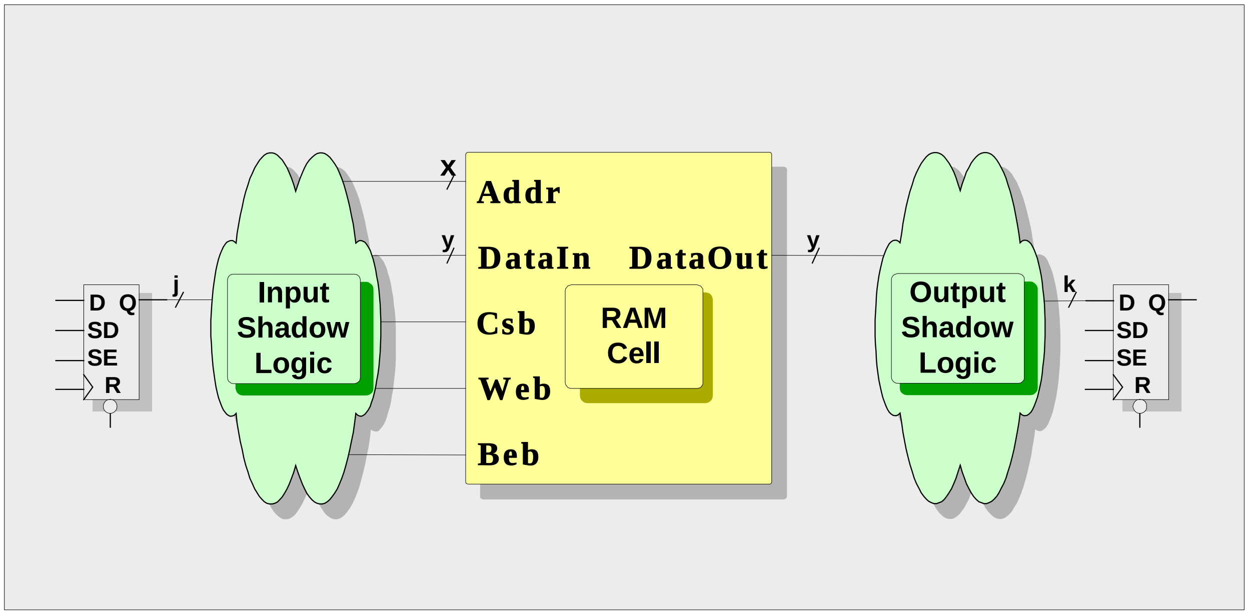

Figure 2-1. Embedded Memory Shadow Logic

As SoC devices start to incorporate *more* memory components of *deeper* arrays and *wider* data busses, the portion of the fault universe associated with shadow logic becomes greater. If the Design-for-Test (DFT) engineer does not handle this, the fault coverage of such devices becomes unacceptable.

3.0 Standard Approaches

Let's first review some of the standard approaches that have become popular in managing the shadow logic surrounding embedded memories.

3.1 Mux Bypass

With this strategy, a bypass mux is placed on the RAM output data, which allows the RAM's data-in bus to be tied directly to its data-out bus as shown in Figure 3-1. This technique is very easy to implement in logic, does a fine job of allowing the ATPG tool to cover both the input and output shadow logic, and requires no special handling by the ATPG tool or the tester program. However, it introduces a large number of gates and requires a lot of additional routing resources. It also introduces a not-insignificant static delay on the read data bus.

Note that the "scan_mode" signal is active during both the scan update and scan shift portions of ATPG. In a design with no other test modes, this is identical to the TESTMODE primary input of the device.

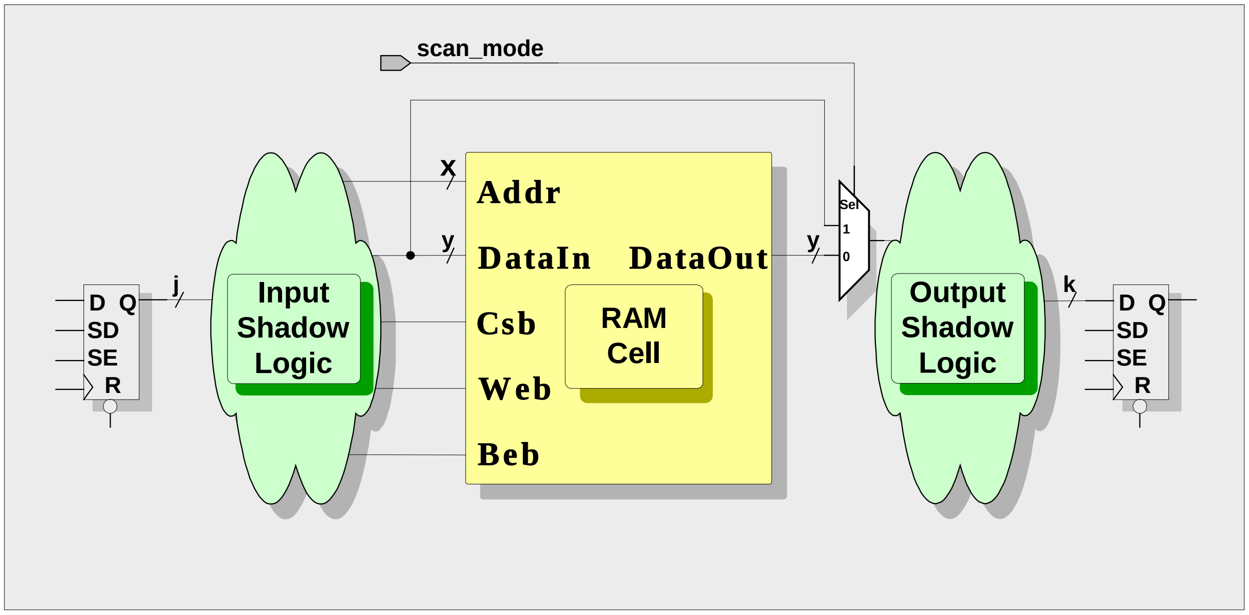

Figure 3-1. Mux Bypass Implementation

3.2 Forced Controllability

Another implementation worthy of consideration is simply to qualify the data out of the RAM with the scan mode signal such that during scan, the data-out bus always contains known values. The block diagram in Figure 3-2 details this concept. This concept is simple to implement, small in gate count, and has a lower routing impact than the mux bypass method. However, it does make the RAM data-out flip-flop uncontrollable for some stuck-at logic values so some fault coverage is lost through this technique. Further, it doesn't provide any observability to the input shadow logic. And again, a static delay component is added to the read data bus, although perhaps less than with the mux bypass method.

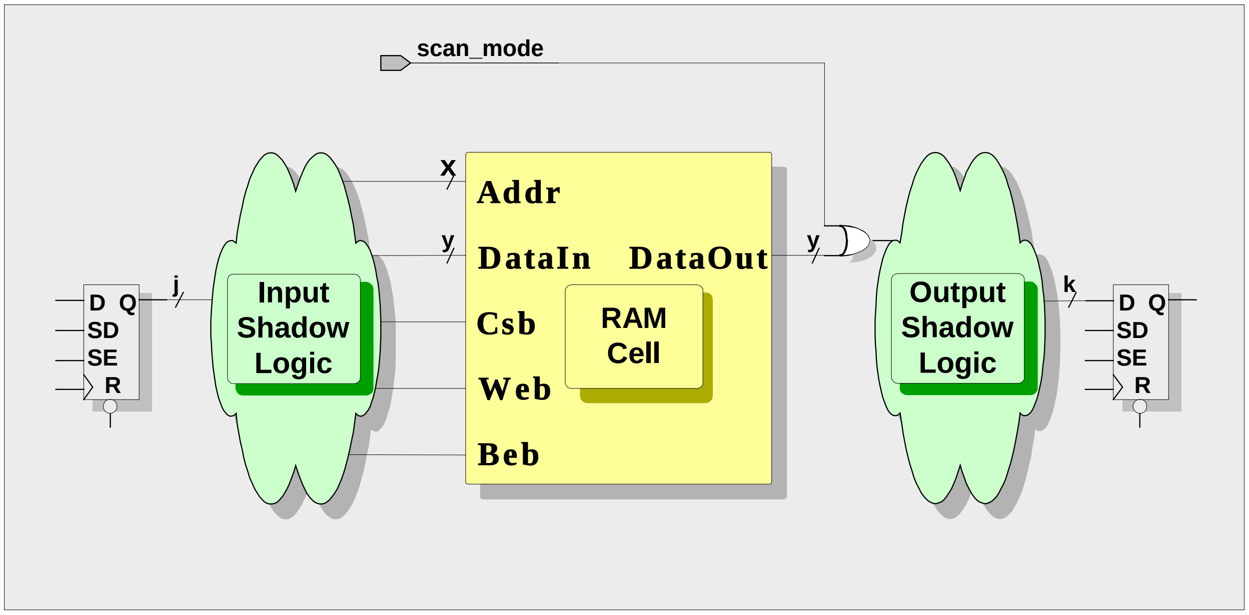

Figure 3-2. Forced Controllability Implementation

Figure 3-2 shows that during functional operation, the flip-flop after the data out of the RAM will receive the contents of RAM unmodified. The scan signal is de-asserted and thus doesn't affect the OR gate. However, during scan mode, the logic '1' is logically ORed with all of the bits on the data-out bus, allowing for limited stuck-at testing of the output shadow logic.

3.3 Wrapper or Register Collar

A register collar is the most obvious way to add full controllability and observability to an embedded RAM's shadow logic. With this method, a set of non-functional flip-flops is added to the design. On the input side of the RAM, these flops allow the ATPG tool to capture and shift out the input shadow logic. On the output side, they allow the tool to control the driving of the output shadow logic. These additional flops are shown in Figure 3-3 below.

DFTC can create and insert this wrapper automatically, assuming the proper DFTC options are available, or it can be created manually. This additional circuitry allows the input shadow logic to be observed, since a scan chain sinks the data out of this logic. It also allows the output shadow logic to be controlled, since a scan chain sources this combinatorial logic. However, it is clear to see in Figure 3-3 that this method adds a lot of overhead to the die and routing needs on the chip, along with the static delays inherent in the bypass mux method.

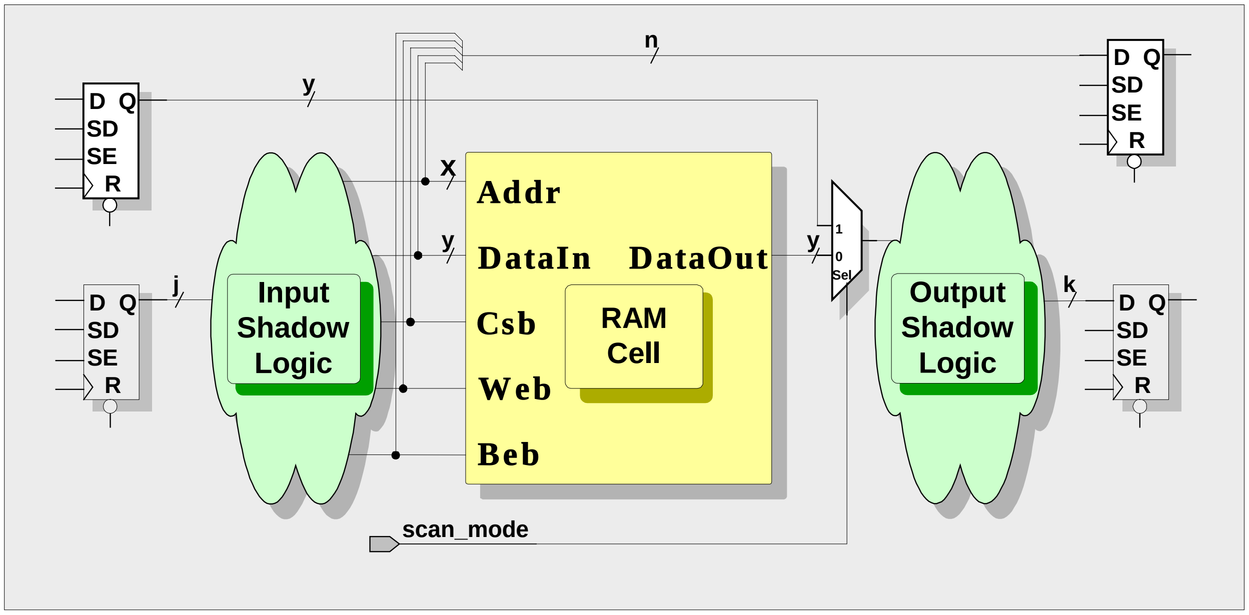

Figure 3-3. Register Collar Implementation

3.4 Smart Wrapper

In this implementation, a single shadow flop is used to create both the observability of the input shadow logic and the controllability of the output shadow logic. Using an XOR of the inputs to the RAM reduces the width of the bypass circuitry to the size of the read data. This method inserts some logic and some routing, although less than the full collar approach, and there is still the delay penalty due to the mux on the read data path.

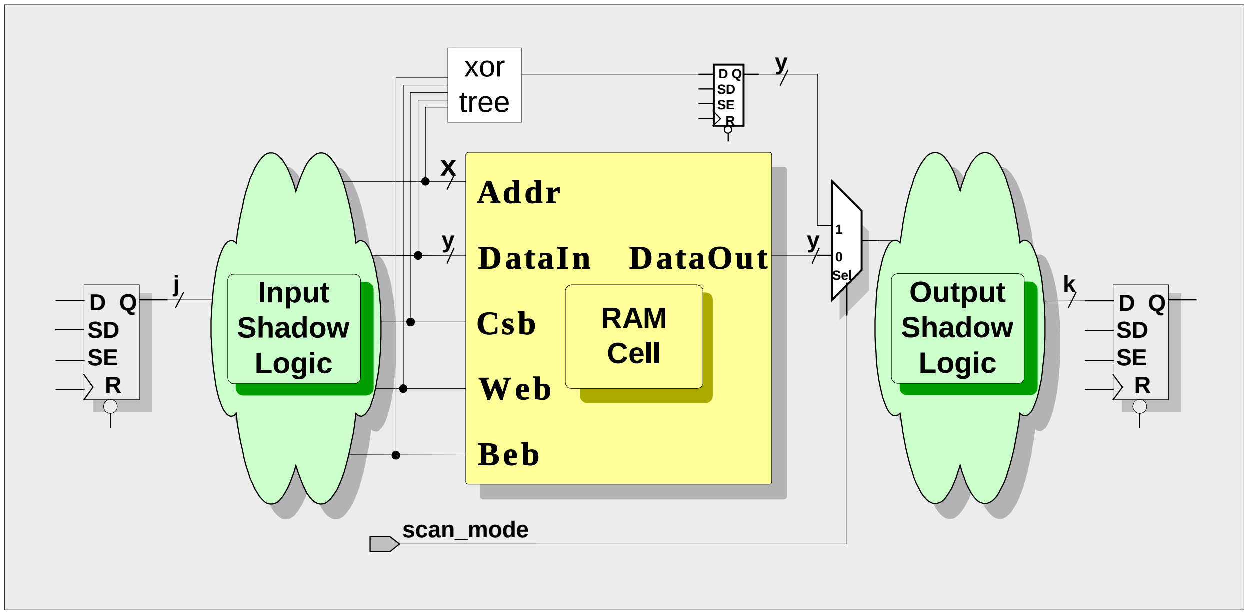

Figure 3-4. Smart Wrapper Implementation

3.5 Do Nothing

While this solution is clearly the least obtrusive to the size of the DUT, it also presents the worst possible fault coverage of the RAM shadow logic. This might be acceptable for parts with small amounts of embedded memory. However, before committing to this technique, it is strongly recommended that a detailed report of the fault coverage be created before making a final decision on test methodology.

4.0 Scan-Through-RAM Approach

4.1 STR Philosophy

Scan-Through-RAM (STR) methodology adds a small amount of DFT logic to the RAM control logic which disallows write operations to the memories during scan mode. Prior to performing scan, the memory is preloaded with known values via an initialization sequence. After the contents of the RAM are preloaded, scan mode will disable writing, such that the contents of the memories remain fixed and known throughout the entire scan test. This controlled knowledge of memory contents is provided to the ATPG tool via a functional model of the memories that contains the pre-loaded data. When the tool generates the scan chain vectors, the outputs of the memories are known values.

In order to read valid data out of the memory block, the contents of the memory pointed to by any given address are preloaded with the value of that address as shown in Figure 4-1. In this way, the address, which selects a particular location in memory, is the same value as the data being read out of memory. Such a technique is selected since it essentially reduces the model of the RAM to a wire as shown by the example in Figure 4-2. This is a very simple model to generate and a very efficient model for an ATPG tool to handle.

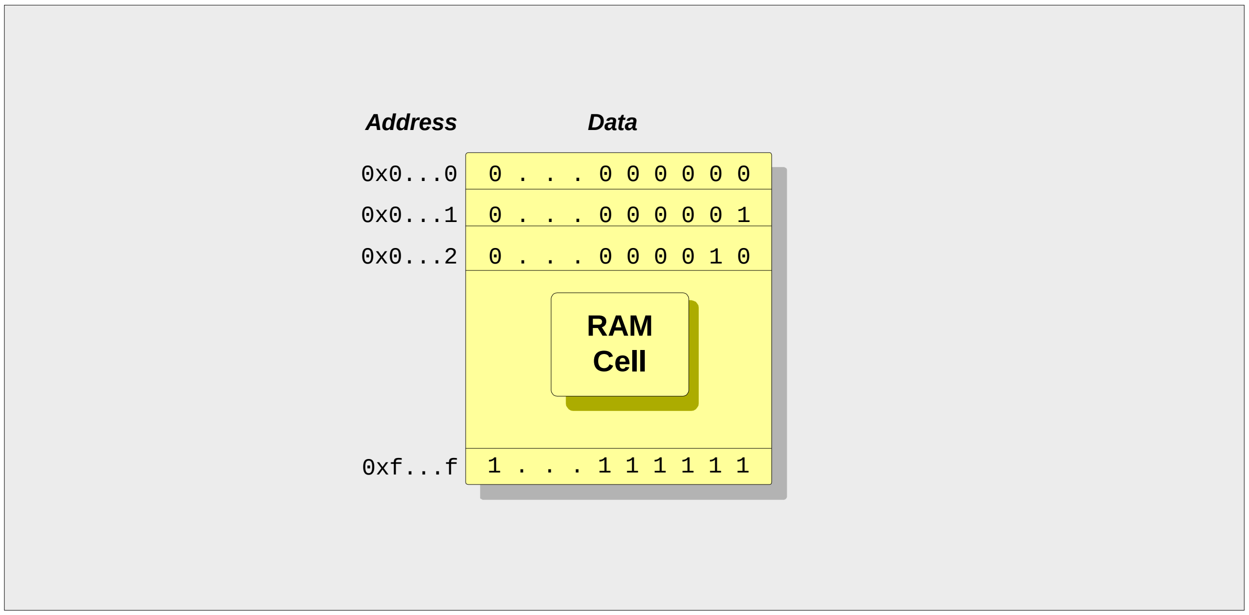

Figure 4-1. Preload Step of STR

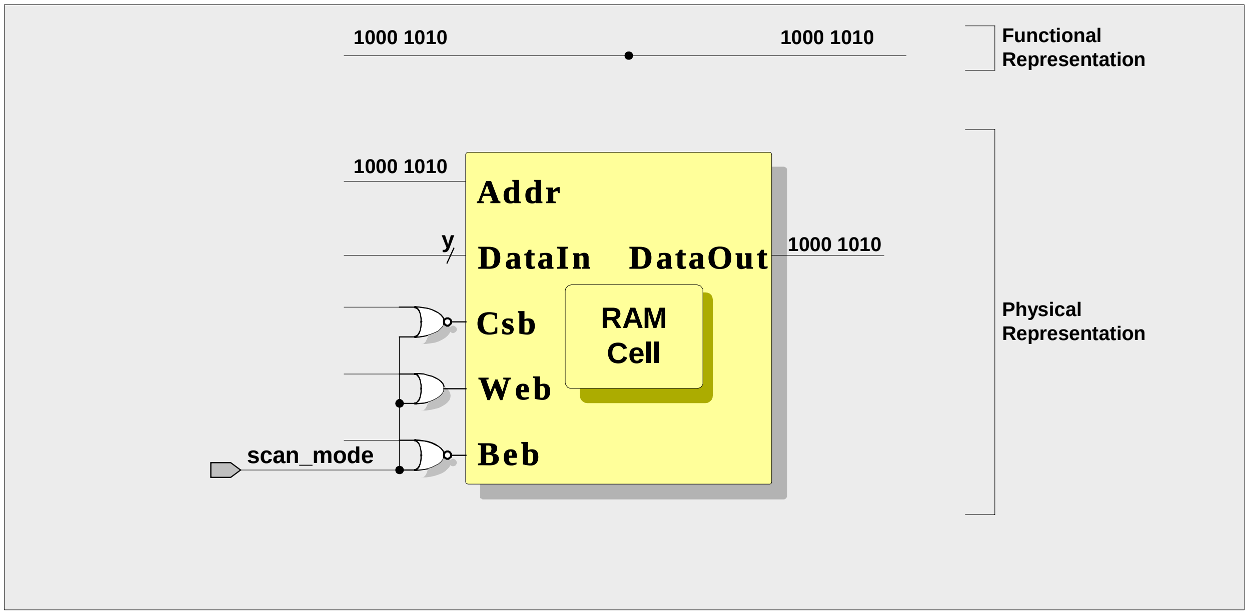

Figure 4-2. Physical and Functional Examples of STR ATPG Model

4.2 STR Implementation – DFT Logic

Figure 4-3 shows the additional logic required to implement this test mode. Simple 2-input OR/NOR gates are added to the RAM control logic in order to put the memory into read mode during scan test. In other words, the write enable (WEB) is de-asserted and the chip select (CSB) is asserted, allowing the contents of RAM to be readable but not writable. This gives the ATPG tool significant observability of the input shadow logic and full controllability of the output shadow logic. It provides this additional coverage with very little die-size overhead, which is now a simple function of the number of RAM instances and not related to their sizes. Also, no increase in scan chain length is incurred, and little additional delay is added to the timing critical read data path due to the additional logic required on the control ports of the RAM. At any rate, this delay is small when compared to the other techniques described herein.

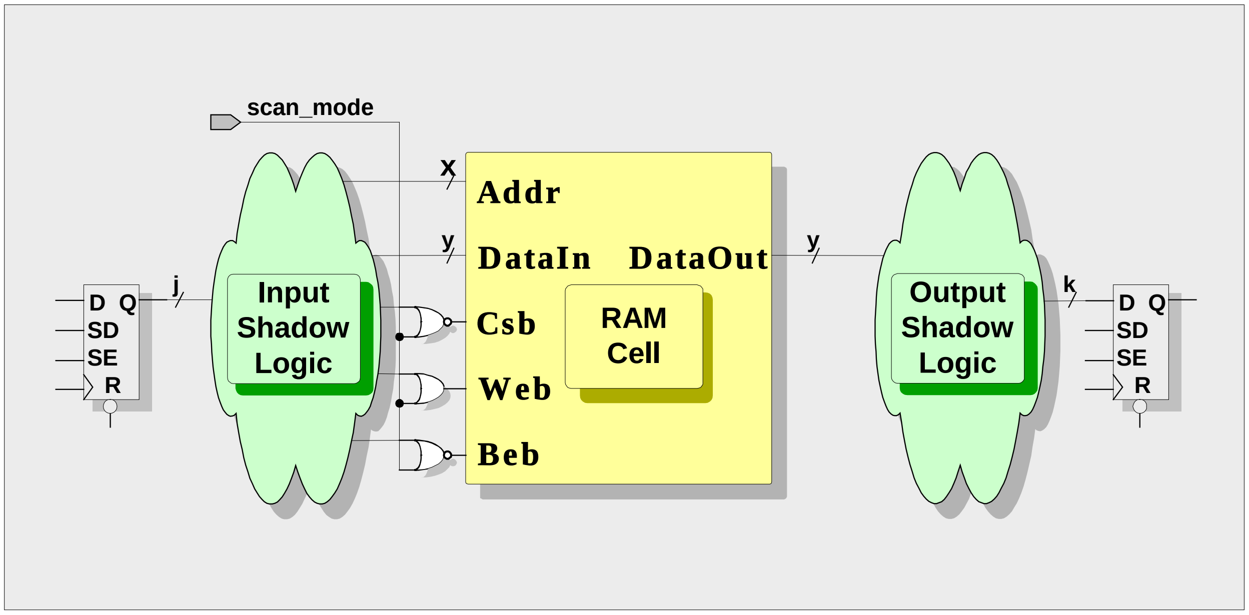

Figure 4-3. STR Implementation

4.3 STR Implementation – ATPG Model for RAMs

Test Compiler (Legacy) Flow

In order to use this methodology with Test Compiler, ASCII .lib models of the RAM cells used in the design are required. The easiest way to create these models is to start from the vendor's .lib models if they are available. If not, the .lib models will need to be created (see Appendix 1, section 11.1, for a template). Then the .lib models must be modified to represent the function of the "wire" described in section 4.0.

Referring to the template in Appendix 1, the bold, italicized, capitalized letters represent fields unique to a RAM cell. These are defined as:

AAA:	RAM Cell Name (e.g. "dpram128x16")
BBB:	RAM Port (e.g. "l", "r".) The bus and pin statements containing BBB will need to be duplicated, one for each port.
CCC:	Address Bus Width (e.g. "7".)
DDD:	Data Bus Width (e.g. "16".)
EEE:	Data Bus Pin Number (e.g. "15".) The pin statements will need to be duplicated, one for each bit on the data bus.
FFF:	Address Bus Pin Number (e.g. "1".) This represents the desired address bit that maps to a given data bit.

The actual mapping of address pins to data bits is arbitrary, but care needs to be taken to duplicate the pattern chosen in the simulation and tester environments. Table 4-1 shows an example mapping for the EEE and FFF fields of the template in Appendix 1 (section 10.1) for several different types of RAM cells.

RAM Cell	Data Bus	Addr Bus	.lib Example
128 by 16 (Address Bus = 7 bits) (Data Bus = 16 bits)	D[15:14]	a[1:0]	pin (dl[15]) {function : al[1];…
	D[13: 7]	a[6:0]	pin (dl[13]) {function : al[6];…
	D[6: 0]	a[6:0]	pin (dl[6]) {function : al[6];…
128 by 8 (Address Bus = 7 bits) (Data Bus = 8 bits)	D[7]	a[0]	pin (dl[7]) {function : al[0];…
	D[6: 0]	a[6:0]	pin (dl[6]) {function : al[6];…
1k by 16 (Address Bus = 10 bits) (Data Bus = 16 bits)	D[15:10]	a[5:0]	pin (dl[15]) {function : al[5];…
	D[9: 0]	a[9:0]	pin (dl[9]) {function : al[9];…
256 by 16 (Address Bus = 8 bits) (Data Bus = 16 bits)	D[15: 8]	a[7:0]	pin (dl[15]) {function : al[7];…
	D[7: 0]	a[7:0]	pin (dl[7]) {function : al[7];…
64 by 8 (Address Bus = 6 bits) (Data Bus = 8 bits)	D[7: 6]	a[1:0]	pin (dl[7]) {function : al[1];…
	D[5: 0]	a[5:0]	pin (dl[5]) {function : al[5];…

Table 4-1. Sample Data Bus Function

In Table 4-1, we essentially replicate the address bus as many times as needed across the data bus starting with the least significant bit of both.

<u>TetraMax Flow</u>

TetraMax uses special Verilog models of library cells in its ATPG algorithm instead of .lib models. Be sure to refer to the TetraMax manual for details on how to create these models for use during ATPG.

4.4 STR Implementation – ATPG Model for ROMs

The wire function model falls apart for ROM cells, since their array is pre-defined and typically contains data of a random nature. When using Test Compiler, it is inefficient to attempt to build a .lib model that represents the data out of the ROM as a function of the address. Instead, a different trick can be applied, where a synthesizable verilog model can be generated and compiled with Design Compiler. The code can be a large case statement that maps the address of the ROM to the data out of the ROM. It passes the function of the ROM on to Design Compiler, which in turn provides a .db file to describe the function for ATPG. The verilog code in Appendix 3 (section 10.3) provides an example of what the synthesizable verilog might look like, as created using a script to translate from the actual ROM image. Note that in a flow using TetraMax, the Verilog model of the ROM would not need to be synthesizable.

4.5 STR Implementation – ATPG Simulation

During simulation, the memories must be preloaded with the desired data before the vectors can execute successfully. To do this we can define a verilog model which performs the initialization of the memory cell using the $readmemh system task. (Note that VHDL issues are not addressed here, as we assume the use of a serial Verilog Test Compiler-generated testbench for the purposes of simulating the ATPG patterns.) The behavioral Verilog in Figure 4-4 shows one way the initial RAM data can be included in the ATPG simulation to verify the ATPG patterns.

```
module str_init;

  initial
  begin


     $readmemh("./init256x16.dat",TOP_ctl.TOP_inst.core.aram0.memory);
     $readmemh("./init256x16.dat",TOP_ctl.TOP_inst.core.aram1.memory);
     $readmemh("./init256x16.dat",TOP_ctl.TOP_inst.core.aram2.memory);
     $readmemh("./init256x16.dat",TOP_ctl.TOP_inst.core.aram3.memory);
     $readmemh("./init256x16.dat",TOP_ctl.TOP_inst.core.aram4.memory);
     $readmemh("./init256x16.dat",TOP_ctl.TOP_inst.core.aram5.memory);
     $readmemh("./init256x16.dat",TOP_ctl.TOP_inst.core.aram6.memory);
     $readmemh("./init256x16.dat",TOP_ctl.TOP_inst.core.aram7.memory);
     $readmemh("./init1kx16.dat",TOP_ctl.TOP_inst.core.bram0.memory);
     $readmemh("./init1kx16.dat",TOP_ctl.TOP_inst.core.bram1.memory);
     $readmemh("./init1kx16.dat",TOP_ctl.TOP_inst.core.bram2.memory);
     $readmemh("./init1kx16.dat",TOP_ctl.TOP_inst.core.bram3.memory);
     $readmemh("./init256x16.dat",TOP_ctl.TOP_inst.core.cram.memory);
     $readmemh("./init128x16.dat",TOP_ctl.TOP_inst.core.dram.memory);
     $readmemh("./init64x8.dat",TOP_ctl.TOP_inst.core.eram.memory);
     $readmemh("./init64x8.dat",TOP_ctl.TOP_inst.core.fram.memory);
     $readmemh("./init128x8.dat",TOP_ctl.TOP_inst.core.gram0.memory);
     $readmemh("./init128x8.dat",TOP_ctl.TOP_inst.core.gram1.memory);

  end

endmodule
```

Figure 4-4. STR Memory Initialization Verilog Code

In our design, a script was developed to generate .dat files in `$readmemh`-compatible format. The script takes the width of the address bus and data bus as arguments, and is given in Appendix 2 (section 10.2) for reference. As with any test flow, we urge you to run ATPG early and to simulate the resulting patterns in your sign-off simulation environment well before tapeout.

5.0 Empirical Results

Cypress Semiconductor's EZ-USB FX2 consists of approximately 100k gates and incorporates 12k of 16-bit-wide dual-port RAM instances in a variety of sizes. FX2 was the pilot project for the STR concept using Synopsys' Test Compiler v2000.05-1 and Cadence's Verilog-XL simulator on a SUN Solaris platform. Table 5-1 compares die size of the DFT solution used as a percentage of overall logic and fault coverage for the digital logic core.

ATPG Method	Die Size	Fault Coverage
Nothing	0.0%	84%
Partial RAM Collar	4.6%	93%
STR	**0.5%**	**93%**

Table 5-1. Comparison of Die Size vs. Fault Coverage

The "Partial RAM Collar" shown here is an effort to include the shadow logic containing the most combinatorial and least timing-sensitive logic. For the 18 RAM instances in FX2, this consisted of collar logic for fourteen 16-bit wide RAM instances, 178 address bits, 32 write data bits, and 512 read data bits. Only the address and data busses were considered for wrapper logic, since control logic was considered too narrow and small to impact fault coverage significantly.

Table 5-1 clearly indicates that STR provides the best solution compared to two other well-known solutions.

6.0 Gotchas

The STR solution for testing embedded memories did not come without its share of developmental headaches. Two problems that needed to be overcome are outlined here.

6.1 Synopsys Model Generation

We discovered that working with the .lib model was not a simple task. Quirks in the syntax and structure of .lib made the development of a model difficult. For example, when we tried to define the data function, we initially defined the function of the entire data bus as a function of the entire address bus. This, however, failed to perform as we expected. In the end, we had to declare the function on a bit-by-bit basis. The template provided in Appendix 1 (section 10.1) is the final working model. This headache will be avoided in the future by using behavioral Verilog models with TetraMax.

6.2 Unknown Values on Address Bus During Pattern Simulation (Gates)

At first, we discovered mismatches between Synopsys expected vector outputs and those we were seeing when simulating the Verilog serial testbench from Test Compiler. This was traced back to observing that in some cases, unknown values were allowed to propagate from the pins of the chip to the address bus. In our vendor's RAM simulation models (written in behavioral Verilog), if one bit of the address becomes unknown, the entire output data bus is forced to unknown. This behavior was not modeled in our vendor's Synopsys .lib format RAM models. Figure 6-1 demonstrates the behavior of the verilog under these circumstances.

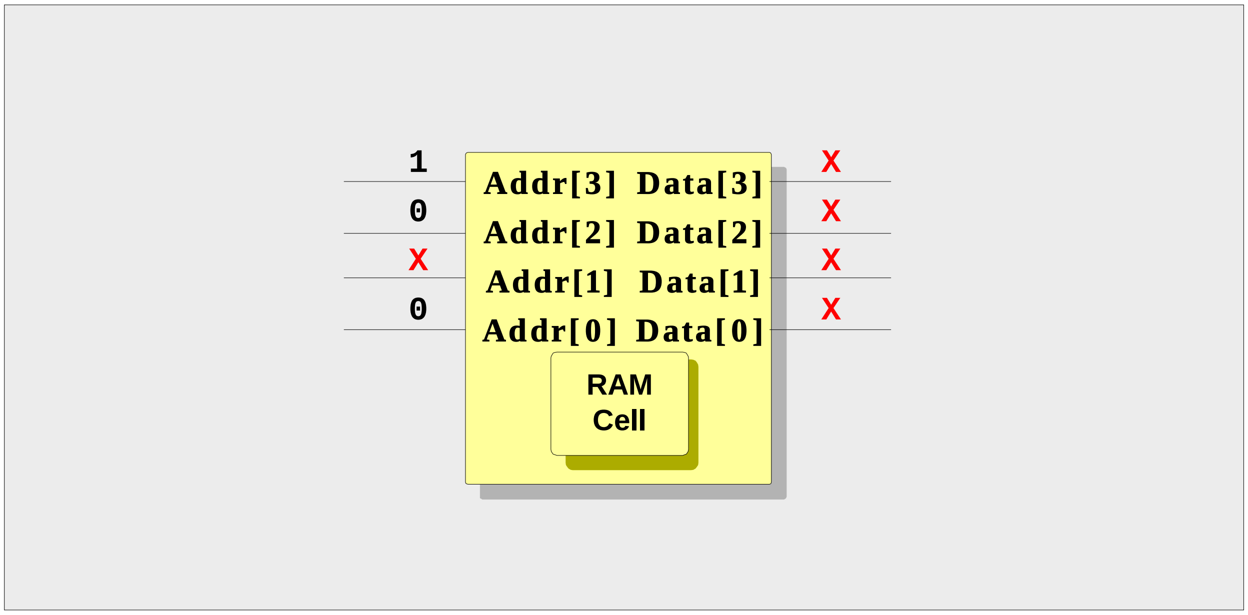

Figure 6-1. Verilog Behavior with Unknown Address Bits

This of course defeats the purpose of controlling the output shadow logic, but more catastrophic is the fact that the gate-level simulations fail since now the vendor supplied RAM Verilog models are behaving differently than the "wire" model used for ATPG. Figure 6-2 shows how the Synopsys model differs from the behavior of the verilog model in this case.

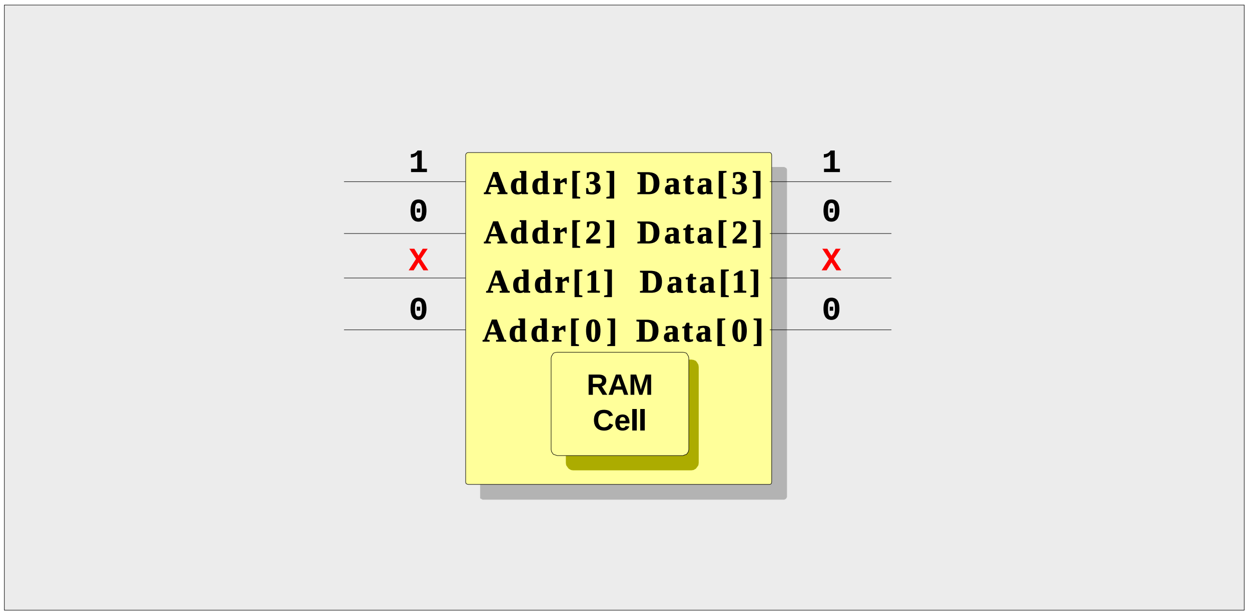

Figure 6-2. Synopsys Model Behavior with Unknown Address Bits

In order to get the simulation environment to match the expected values, it is important to ensure that no unknown values propagate to the address bus. If these unknown values come from the pins, then the following Synopsys command will force Test Compiler to put a known value of zero onto the input pin:

```
write_test_input_dont_care_value = "0" (or “1”)
```

This will cause Synopsys to place a defined value on the inputs to the chip in place of the logic "X" and the entire address bus will remain known. The logic value is arbitrary, since any known value will generate a known output value.

6.3 Vector Depth Limitations

The requirement of pre-loading all RAM instances on the die can cause your test pattern suite to exceed your target tester memory depth. In our case, despite having over 12k bytes of RAM (mostly word addressable), we had no problem with this. We used the special scan feature of our Versatest VT3300 (with 1 Meg of memory) to store vector data. In all cases, you should check with your test engineer ahead of time to confirm that your target tester will support your test methodology.

7.0 Extensions

Dual Port RAMs

The FX2 project used only full dual port RAMs that were physically 8 or 16 bits wide. In our case, we had to apply STR techniques to each independent port of the RAM, using the same preload value on each side, of course.

Other RAM Architectures

The STR philosophy can be applied to single port RAMs with separate data in and data out lines, as is shown in the examples throughout this paper. In the case of a RAM with a shared data in/data out bus, control logic can be implemented to force the data bus to be an output during scan_mode. At that point, STR can be implemented as is shown here.

In the case where multiple RAM instances are used to create a larger memory space, the test coverage attained using scan through RAM techniques can be dependent on the initial values loaded into the RAMs. Typically the different RAM instances will share most of an address bus and have their data outputs muxed onto a single read data bus. The upper bits of the address bus are used for chip select generation and read mux control. If the same data is loaded into each RAM instance, faults on the upper address bits (RAM chip select and read mux select lines) can be hidden. Simply loading a different data pattern into each RAM instance will reveal these hidden faults. Note that these same potentially hidden faults, along with the same solution, can be applied to RAM instances whose outputs drive a shared tri-state bus, where the mux select faults are replaced by tri-state enable faults.

8.0 Summary and Recommendations

In summary, the advantages and disadvantages of the various RAM testing techniques are summarized in table 8 – 1. The categories are labeled “Y” or “N” for yes and no, or with a “0”,“1”, “2”, or “3”, where “0” represents none, “1” very little, “2” some, and “3” a lot.

Method	Static delay on read data?	Static delay on input signals?	Increased scan depth	Increased gat count?	Increased routing?	Increased Coverage?
Mux Bypass	Y	N	0	2	2	2
Forced Control	Y	N	0	1	1	1
Full Wrapper	Y	N*	3	3	3	3
Smart Wrapper	Y	N*	2	2	2	3
Do Nothing	N	N	0	0	0	0
STR	N	Y	0	1	0	3
N*: In these cases, there is no static delay per se, but there is additional loading placed on these signals.						

Table 8-1. Comparison of Shadow Logic Coverage Strategies

We were highly pleased with the results we achieved using this methodology over others to test the shadow logic associated with embedded RAMs in our FX2 SoC. It gave us the largest increase in test coverage with the lowest overhead of the methods discussed here. Our largest hurdle in implementing STR was struggling with the .lib language to model the STR function of the RAMs properly, but with the introduction of TetraMax and its capability of using behavioral Verilog models for ATPG, this hurdle is (re)moved.

9.0 References

Jaramillo, Ken, and Subbu Meiyappan. *Ten Commandments of Scan Design.* SNUG 2000 User Session MB2

Pathak, Jaydeep. *DFT Compiler with Physical Synthesis and TetraMax ATPG.* SNUG 2001 Tutorial Session WB2

Synopsys, Inc. *DFT Compiler Overview*. v2001.08.

10.0 Appendices

10.1 Appendix 1—Synopsys .lib Model STR Template

```
cell (AAA) {
        area : 0.0;
        dont_touch : true;
        dont_use : true;

        bus (aBBB) {
                bus_type : "BUSCCC";
                direction : input;
                capacitance : 0.075;
                fanout_load : 0;
        }

        bus (iBBB) {
                bus_type : "BUSDDD";
                direction : input;
                capacitance : 0.025;
                fanout_load : 0;
        }

        bus (dBBB) {
                bus_type : "BUSDDD";
                direction : output;
                capacitance : 0.054;
                max_capacitance : 1.0;

        pin (dl[EEE]) {
                        function : aBBB[FFF];
                        timing () {
                                timing_type : combinational;
                                intrinsic_rise : 5.5;
                                intrinsic_fall : 5.5;
                                rise_resistance : 0.65;
                                fall_resistance : 0.65;
                                related_bus_pins : "aBBB[FFF]";
                        }
                }

        pin (clkBBB) {
                direction : input;
                capacitance : 0.0825;
                fanout_load : 0;
                clock : true;
        }

        pin (csbBBB) {
                direction : input;
                capacitance : 0.0825;
                fanout_load : 0;
        }

        pin (webBBB) {
                direction : input;
                capacitance : 0.0825;
                fanout_load : 0;
        }

        bus (bebBBB) {
                bus_type : "BUS";
                direction : input;
                capacitance : 0.0825;
                fanout_load : 0;
        }

        pin (vpwr) {
                direction : input;
                capacitance : 1.0;
                fanout_load : 0;
        }

        pin (vgnd) {
                direction : input;
                capacitance : 1.0;
                fanout_load : 0;
        }

        pin (i_25u) {
                direction : input;
                capacitance : 1.0;
                fanout_load : 0;
        }
}
```

10.2 Appendix 2—Data File Generator Script for STR Simulations

```
#!/usr/local/bin/perl

$addr_width = $ARGV[0];
$data_width = $ARGV[1];

for ($addr_loop = 0; $addr_loop < (2 ** $addr_width); $addr_loop++)
{

    # 1, Print the address column heading
    printf("@");

    # 2, Pad extra zeros to be pretty
    $addr_pad = $addr_loop * 16;

    # Gotta start with some non-zero base, since 16*0 will
    #equal zero forever and hang this loop
    if ($addr_pad == 0)
    {
      $addr_pad = 16;
    }

    # There's certainly an easier way, but what the heck...
    #
    #                  |----Don't roll over from all Fs-------------------------------|
    #                    |-----There are 16 decimal values per hex character----|
    #                            |--------round up to nearest integer----------|
    #                                 |-----addr_width-base-16--------|
    while ($addr_pad < ( ( 16 ** (int((log(2 ** $addr_width) / log(16)) + 0.99))) - 1) )
    {
      printf("0");
      $addr_pad = $addr_pad * 16;
    }

    # 3, Finally Print the address
    printf("%X ",$addr_loop);
    $addr_div = $addr_loop;
    $data_val = 0;
    $addr_index = 0;
    $data_index = 0;

    # Let's create a binary array for use in concatenating
    # the data field
    for ($addr_bit_loop = 0; $addr_bit_loop < $addr_width; $addr_bit_loop++)
    {
      $addr_bit = $addr_div % 2;
      $addr_bin[$addr_index] = $addr_bit;
      $addr_div = $addr_div / 2;
      $addr_index++;
    }

    # Time to iteratively concatenate the addr_bin array into the data_bin array
    for ($data_bit_loop = 0; $data_bit_loop < $data_width; $data_bit_loop++)
    {
      $data_bin[$data_index] = $addr_bin[$data_index % ($addr_index)];
      $data_index++;
    }

    #
    for ($data_loop = $data_width; $data_loop >= 0; $data_loop--)
    {
      $data_val = (2 * $data_val) + $data_bin[$data_loop];
    }

    # 1, Pad extra zeros on data column to be pretty
    $data_pad = $data_val * 16;

    # Gotta start with some non-zero base, since 16*0
    # will equal zero forever and hang this loop
    if ($data_pad == 0)
    {
      $data_pad = 16;
    }

    # There's certainly an easier way, but what the heck...
    #
    #                  |----Don't roll over from all Fs-----------------------------|
    #                    |-----There are 16 decimal values per hex character----|
    #                            |--------round up to nearest integer----------|
    #                                 |-----addr_width-base-16--------|
    while ($data_pad < ( ( 16 ** (int((log(2 ** $data_width) / log(16)) + 0.99))) - 1) )
    {
      printf("0");
      $data_pad = $data_pad * 16;
   }

    # 3, Finally Print the address
    printf("%X\n",$data_val)
}
```

10.3 Appendix 3—Synthesizable Verilog for ROM STR Functional Model

```
// Synthesizable ATPG Verilog model for ROM-based STR implementation

`timescale 1ns / 10ps
module rom_model

(al,
 il,
 dl,
 csbl,
 webl,
 bebl,
 ar,
 ir,
 dr,
 csbr,
 webr,
 bebr
);

input [9:0] al;
input [9:0] ar;
input [15:0] il;
input [15:0] ir;
output [15:0] dl;
output [15:0] dr;
input csbl;
input csbr;
input webl;
input webr;
input [1:0] bebl;
input [1:0] bebr;

// Set the outputs as register also for assignment purposes.
wire [15:0] dl;
wire [15:0] dr;

// Define the registers for the left and right data storage.
reg  [15:0] left_data;
reg  [15:0] right_data;

// Define the bits that enable reads for left and right.
wire lread;
wire rread;

// The following statements have the case statement for every possible
// address line and the contents of the ROM. First the data is defined
// for the left port and the exact same data is defined for the right
// port.

always @ (al) begin
  case (al)
    10'h000 : left_data = 16'hFD2F;
    10'h001 : left_data = 16'hC3FC;
    10'h002 : left_data = 16'h95ED;
    10'h003 : left_data = 16'hEC1E;
    //------------------------------------------------------------
    //    MORE ADDRESS-TO-DATA MAPPING HERE
    //------------------------------------------------------------
    10'h3fe : left_data = 16'h90F0;
    10'h3ff : left_data = 16'h14E6;
    default : right_data = 16'bx;
  endcase
end

always @ (ar) begin
  case (ar)
    10'h000 : right_data = 16'hFD2F;
    10'h001 : right_data = 16'hC3FC;
    10'h002 : right_data = 16'h95ED;
    10'h003 : right_data = 16'hEC1E;
    //------------------------------------------------------------
    //    MORE ADDRESS-TO-DATA MAPPING HERE
    //------------------------------------------------------------
    10'h3fe : right_data = 16'h90F0;
    10'h3ff : right_data = 16'h14E6;
    default : right_data = 16'bx;
  endcase
end

assign dl = left_data;
assign dr = right_data;

endmodule // rom_model
```