Memory BIST Implementation and Validation

Major Project Report

Submitted in partial fulfillment of the requirements
For the degree of

Master of Technology
In
Electronics & Communication Engineering
(VLSI Design)
By

Javiya Mitali Harishbhai
(16MECV10)

Electronics & Communication Engineering Department
Institute of Technology
Nirma University
Ahmedabad - 382 481
May, 2018

Memory BIST Implementation and Validation

Major Project Report

Submitted in partial fulfillment of the requirements
For the degree of

Master of Technology
In
Electronics & Communication Engineering
(VLSI Design)

By
Javiya Mitali Harishbhai
(16MECV10)

Under the Guidance of

Internal Guide
Prof. Vaishali Dhare
Assistant Professor (EC, ITNU)
Nirma University

External Guide
Mr. Vasubabu Ravipati
Engineering Manager
Intel Technology India Pvt Ltd.

Electronics & Communication Engineering Department
Institute of Technology
Nirma University
Ahmedabad - 382 481
May, 2018

Declaration

1. I hereby declare that this thesis entitled **"Memory BIST Implementation and Validation"** was carried out by me for the degree of Master of Technology in VLSI Design under the guidance and supervision of Prof. Vaishali Dhare, Institute of Technology, Nirma University.

2. The interpretations put forth are based on my reading and understanding of the original texts and they are not published anywhere in the form of books, monographs or articles. The other books, articles and websites, which I have made use of are acknowledged at the respective place in the text.

3. For the present thesis, which I am submitting to the University, no degree or diploma or distinction has been conferred on me before, either in this or in any other University.

Javiya Mitali Harishbhai
(16MECV10)

Certificate

This is to certify that the Project entitled **"Memory BIST Implementation and Validation"** submitted by **Javiya Mitali Harishbhai (16MECV10)**, towards the submission of the Project for requirements for the degree of Master of Technology in VLSI Design, Nirma University, Ahmedabad is the record of work carried out by her under our supervision and guidance. In our opinion, the submitted work has reached a level required for being accepted for examination.

(External Guide)
Mr. Vasubabu Ravipati
Engineering Manager
Intel Technology India
Bangalore

Company Seal
Intel Technology India Pvt. Ltd.(Bangalore)

Date :

Place : Bangalore

Certificate

This is to certify that the Major Project entitled **"Memory BIST Implementation and Validation"** submitted by **Javiya Mitali Harishbhai (16MECV10)**, towards the partial fulfillment of the requirements for the degree of Master of Technology in VLSI Design, Nirma University, Ahmedabad is the record of work carried out by her under our supervision and guidance. The submitted work has reached a level required for being accepted for examination.The results embodied in this major project, to the best of our knowledge, haven't been submitted to any other university or institution for the award of any degree or diploma.

Prof. Vaishali Dhare	Prof. Dr N. M. Devashrayee
Internal Guide	PG Coordinator (VLSI Design)
Dr D. K. Kothari	Dr Alka Mahajan
Head, EC Dept.	Director, IT - NU
Date :	Place : Ahmedabad

Acknowledgement

First and foremost, sincere gratitude to my manager Mr. Vasubabu Ravipati. Also I want to thank Intel Technologies India Private Limited, Bangalore for assigning me such a project.

I would like to take this opportunity to thank my mentor Swapna Gundeboyina, at Intel Technologies India Private Limited, Bangalore who have supported and guided me throughout my project work.

I would also like to express my gratitude to internal guide, Prof. Vaishali Dhare and PG coordinator Dr N. M. Devashrayee, Professor, VLSI Design, Institute of Technology, Nirma University, Ahmedabad for encouraging, giving valuable advice and support throughout the semester.

I would also like to thank all the faculty members of Nirma University who have encouraged me during my post-graduate program.

- Javiya Mitali Harishbhai
(16MECV10)

Abstract

Testing is very important aspect of any VLSI product. Compromise in testing directly affects trust of people on product and company. As SoCs are becoming more complex day by day DFT has very important role in the design. In fact, DFT is expanding as the technology is shrinking. As the technology is scaling down (28nm, 22nm, 14nm, 10nm etc.), this introduces manufacturing challenges and also higher chances of failure and Crosstalk effects. Chips can fail after manufacturing due to Contamination causing open circuits, Extra metal causing short circuits, Insufficient doping, Open interconnect on the die caused by dust particles.

When manufacturing defects occurs, the physical defect has a logical effect on the circuit behavior and chip may not work as intended. Detecting faulty chips is more costly in the later stages of life cycle of the product. So, it is needed that the device which are shipped to the end customer are not defective one. So, before shipping to the end customers, the manufacturing defects must be tested, to filter out the bad device with the good one.

Memories are more vulnerable to physical defects than logic circuits because of their higher density and more complicated processing steps. Memory BIST is a digital logic, which is inserted in the design, to detect all the defects present in the memories arrays caused during manufacturing process. Memory BIST hardware incorporates various test algorithms to targets these defects.

This thesis covers MBIST architecture in brief where description for all the blocks is included,the flow of Memory BIST insertion at wrapper level and creation of the TAP controller RTL based on fullchip requirements. It also covers how to generate testbenches/patterns that can be used for pre and post silicon validation. Memory Rastering process is included which will help to find the loaction where exactly the memory is failing. This thesis also helps to understand the post MBIST simulation debug process.

Table of Contents

Declaration i

Internship Certificate iii

Certificate **v**

Acknowledgement vii

Abstract ix

1 Introduction **1**

1.1 Motivation 2

1.2 Organization of Thesis 3

2 Literature Survey **5**

2.1 Why Memory Testing? 5

2.1.1 Memory Faults 5

2.2 How to test Memories? 8

2.3 Why Memory BIST? 8

3 Memory BIST Architecture **11**

3.1 MBIST Controller 13

3.2 Memory Interface (Collar) 13

3.3 BIST Wrapper 14

3.4 WTAP 16

3.5 MBIST-TAP (LVTAP) 17

4 MBIST Implementation **19**

4.1 MBIST Insertion Flow 19

4.1.1 General Steps Involved in the Flow 19

4.1.2 Dependencies 20

4.1.3 Inputs 20

4.1.4 Outputs 20

4.1.5 Flow Script Descriptions(Tessent Tool) 20

5 MBIST-TAP (LVTAP) Insertion **23**

5.1 Flow Prerequisites 24

5.2 Generating BIST TAP level testbenches and patterns 27

6 Memory Repair **29**

7 Post MBIST Simulation Debug **33**

7.1 Common Issues Encountered During Simulation 33

8 Conclusion **37**

References **39**

List of Figures

2.1.1 Stuck-At-0 Fault 6
2.1.2 Stuck-At-1 Fault 6
2.1.3 Down Transition Fault 6
2.1.4 Up Transition Fault 7
2.1.5 Inversion Coupling Fault 7
2.1.6 Idempotent Coupling Fault 8

3.0.1 Block Level 12
3.0.2 SOC Level 12
3.2.1 Memory Interface 14
3.3.1 BIST Wrapper 15
3.4.1 WTAP 17
3.5.1 LVTAP 18

5.0.1 Fullchip model for TAP insertion and pattern generation 24

6.0.1 Bit Mapping 29
6.0.2 Forces Applied 30
6.0.3 Fuse Added 30
6.0.4 Simulation Waveform 30
6.0.5 Simulation Waveform 31
6.0.6 Simulation Waveform 31
6.0.7 Simulation Waveform 31

7.1.1 Unit Level Simulation 34

Chapter 1

Introduction

Testing means to screen out defective chips so that only good chips are shipped to customer. But as the size of chip reduces and design complexity increases, testing of device via external tester becomes difficult. So to support the testing of chip, extra logic is added in the design other than functional logic. This makes testing of chip easier and faster. This addition of extra logic is nothing but DFT (Design for Test).

Testing which is post silicon process checks for manufacturing defects. Functionality is verified before sending it to fabrication. So after fabrication we need to check for manufacturing defects. Other reason of doing manufacturing test is it requires less number of patterns to test than functional test. The purpose of manufacturing test is to screen for manufacturing defects so that chips can be discarded before they reach to customer.

Defect screening efficiency of manufacturing test depends on how good the tests are in exposing large proportion of manufacturing defects. This requires a good understanding of various types of manufacturing defects and way of modeling or abstracting their behavior so that tools can be developed to generate and grade manufacturing tests. The impact of defects on design behavior is usually abstracted in terms of fault models.

1.1 Motivation

While purchasing any product customer will always look forward for two aspects: Cost and Quality. Every organization will aim to maintaining its brand name, so they need to ship good quality of products by which they can gain customers trust also. Components which do not perform per specification are considered defective by customer and result in customer returns. The outgoing quality is usually measured in Defects per Million (DPM) parts shipped and the quality is considered satisfactory as long as the outgoing DPM meets customer requirements. DPM means defective product out of 1 million product which are shipped. To achieve low DPM, product needs to be tested thoroughly before shipping and if the product is found defective, then it could not be used further. Testing for todays modern SoC is very difficult as size is becoming small and complexity is increasing. So to make testing possible, some extra design is added. This is where DFT (Design for Test) comes into picture. DFT includes many concepts like scan chain, JTAG, MBIST, LBIST etc.

Memories are the most universal component today. Almost all system chips contain some type of embedded memory, such as ROM, SRAM, DRAM, and flash memory. The ever increasing size and number of memories in the Systems on Chip has presented the designers and test engineers with a challenge for huge number of functional or ATPG patterns for verification of memory functionality. So, to test the memory functionality either functionally or through ATPG requires huge test time, and hence, huge test cost. It is almost impossible in such scenario to verify memory functionality fully. Thus, the designers are left with only one way; i.e. to verify memory functionality through BIST (Built-In Self Test) functionality.

BIST is an inbuilt testing hardware inside a product/equipment module. It just needs to trigger the hardware from outside. This hardware, at that point, runs the inbuilt patterns/algorithms and returns if the module is working appropriately. This, being inbuilt does not should be provided with designs from outside. Likewise, since, this is inside a module, consequently, the measured strategy for testing can be adopted which diminishes run time essentially.

The inherent individual test utilized for recollections is known as MBIST (Memory Built-In Self Test). Like other BIST rationale, MBIST rationale is inbuilt inside memory as it were. The MBIST rationale might be fit for running a few algorithms to check memory usefulness and test for memory faults particularly outlined and advanced for these.

So MBIST for memories is one of the structural test methods. MBIST architecture provide adequate test coverage to meet the DPM goals for SoC products with a large number of Register Files and at the same time provide a scalable and modular solution that could be readily be re-used by various IPs in a SOC.

1.2 Organization of Thesis

In the thesis

- Chapter 2 introduces the need of testing memories, the methodology to test memories and how Memory BIST logic is helpful to test memories
- Chapter 3 describes the whole architecture of Memory BIST and also gives brief description of all its blocks
- Chapter 4 explains how Memory BIST logic is inserted at unit (wrapper) level in any design, what are the pre-requirements for inserting that logic and what all steps are followed for insertion
- Chapter 5 explains MBIST-TAP insertion which will be done once the MBIST is inserted at unit level. It also explains the flow of generating patterns which are required for simulation purpose
- Chapter 6 explains the process of MBIST Rastering which means if any memory or controller is failing then how to find the failing location of memory and what steps are involved to repair those memories

- Chapter 7 covers the post simulation debug process which should be done if any test is failing and the root cause of failure can be found out through this process
- Chapter 8 includes conclusion of the entire thesis

Chapter 2

Literature Survey

2.1 Why Memory Testing?

- Memories are the most dense components with-in the SoC
- Contributes to most of the area as well
- As the technology node decreases they become more sensitive to process defects
- To overcome the Testability compromise
- To reduce DPM and get a good yield

2.1.1 Memory Faults

Can be categorized as follows:-

- Stuck-At Faults
- Transition Faults
- Coupling Faults
- Neighborhood Pattern Sensitive Faults
- Retention Faults

Stuck-At Fault:-

Memory Cell stuck to one particular value (either 0 or 1). Stuck-At-0:

Figure 2.1.1: Stuck-At-0 Fault

Suck-At-1:

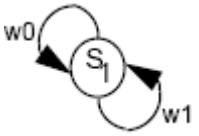

Figure 2.1.2: Stuck-At-1 Fault

Transition Faults:-

Differs from Regular ATPG Transition Fault. It occurs when one cell:-

- Can transition from 0 to 1 but not from 1 to 0(Down Transition Fault)

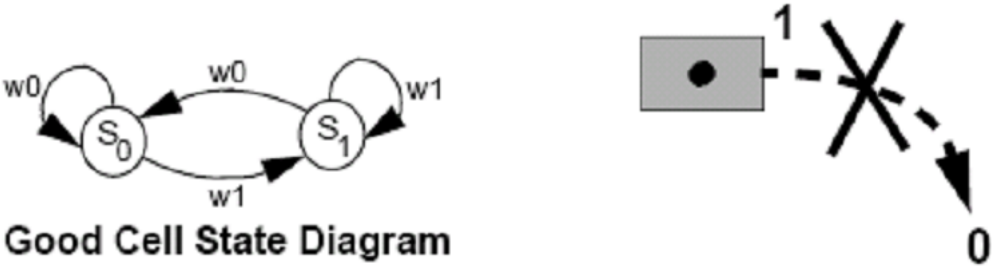

Figure 2.1.3: Down Transition Fault

- Can transition from 1 to 0 but not from 0 to 1 (Up Transition Fault)

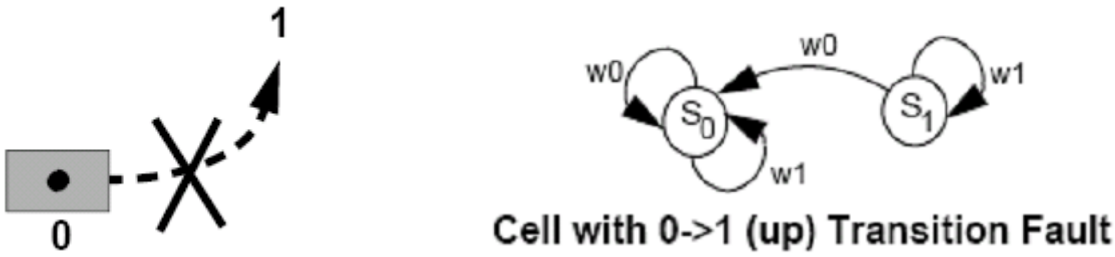

Figure 2.1.4: Up Transition Fault

Coupling Faults:- These faults occur when one cell affects the data on the other cell. This can be further divided as follows:-

- Inversion Coupling Faults (CFin):- When Transition on one cell's data inverts the other cell's data.

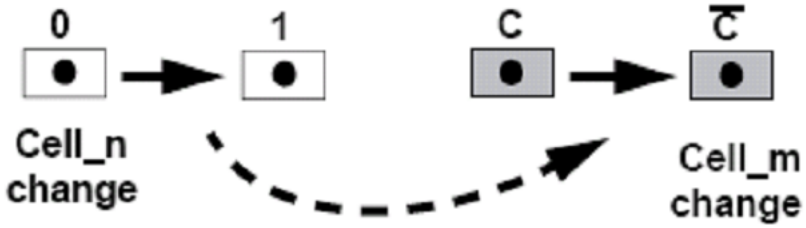

Figure 2.1.5: Inversion Coupling Fault

- Idempotent Coupling Faults (CFid):- Occurs when the transition on one cell constrains the other cell to a particular value (0 or 1).

- State Coupling Faults (CFst):- A coupled cell or line is forced to a certain value only if the coupling cell or line is in a given state.

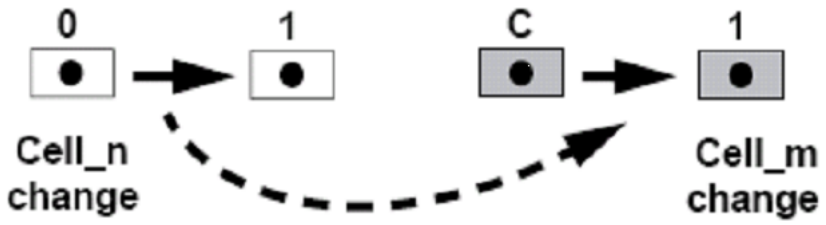

Figure 2.1.6: Idempotent Coupling Fault

2.2 How to test Memories?

The goal of array tests is to ensure each portion of the array logic is free of the modeled array defects and typically consist of tests which perform read and write operations through the entire address space of any array. This can be achieved by:-

- Direct access to the memory:- Memory will be accessed from the tester channels and Patterns will be applied through external ATE.
- Access from CPU through code
- MBIST (Memory Built-In Self-Test):- Some digital logic will be added in the design itself to test the Array structure.

2.3 Why Memory BIST?

With the increasing complexity, size and number of embedded memories:-

- It is difficult to get access to all the memories through external tester. Extra cost is involved to design such ATE. Also the speed from external ATE and available Pins are limited.
- With a little area overhead (equivalent to some flops and combinational logic), it provides an efficient method to access all the memories at any speed requiring only TAP connection.
- The area overhead becomes negligible if the memories are bigger and in huge number.

- Helps in Process debug by using different set of algorithms and diagnostic capabilities.
- Functional Debug capabilities (Array Freeze and Dump).

Features of Memory BIST:

- Controllable through IEEE 1149.1 TAP interface, and the BIST ”status” and ”result” are accessible via the TAP Interface.
- Simultaneous testing of multiple memories within the design, which may require sequential testing otherwise. A structured method which provides consistently high-quality test patterns as well as substantial test time saving, but with some cost such as area and routing resources.
- One controller can test multiple memories in parallel.
- Supports very comprehensive debug and diagnostic features.
- Isolate failing memory locations (”raster” capability).
- Requires very simple initialization and control to operate.

Advantages of Memory BIST:

- Reduces routing signals needed at chip level.
- Reduces the amount of test data that needs to be stored as the on chip circuity generates the test stimulus.
- Shortest test-time due to in-built test circuit [Test Cost Reduction].
- Can be programmed to test new scenario during Root Cause Analysis of a failing chip or on customer returns part [Diagnosis].
- Reparability [Reusability].

Disadvantages of Memory BIST:

- Extra Area Overhead.
- Extra PD effort for timing closer on MBIST area.

Chapter 3

Memory BIST Architecture

The built-in self test employed for memories is known as MBIST (Memory Built-In Self Test). Like other BIST logic, MBIST logic is inbuilt within memory only. Memory BIST is an extra logic, which is inserted in the design, to detect all the defects present in the memories arrays caused during manufacturing process. MBIST logic may be capable of running several algorithms to verify memory functionality and test for memory faults specifically designed and optimized for these. No extra pins are required at top level to execute the patterns. Since memories have very regular structure in a dense manner, not much change is required in MBIST logic. All the changes in MBIST are mainly based on the memory size and type. Other factors that affect the MBIST logic are the built-in redundancy and the number of algorithms to be hard coded.

Memories can be tested in serial or parallel manner. Multiple memories can be tested simultaneously through one single MBIST Controller provided the memories belong to the same clock domain. Multiple algorithms can be hard coded in the design. Debug on memory failure is easy as the failure data can be dumped out at the specific time. Default algorithm can be hard coded as per the test program.

At Block Level:

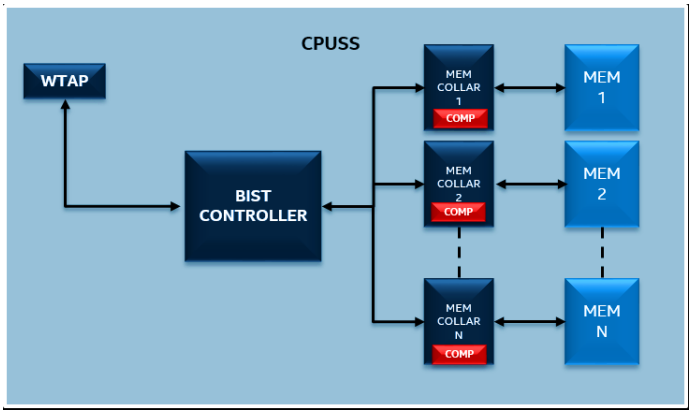

Figure 3.0.1: Block Level

At SOC Level:

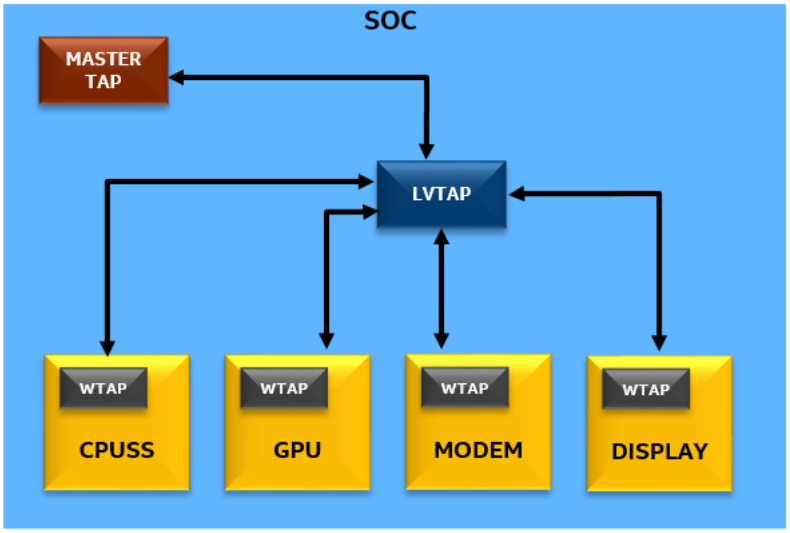

Figure 3.0.2: SOC Level

BUILDING BLOCKS OF MBIST HARDWARE:

- MBIST CONTROLLER
- MEMORY INTERFACE (COLLAR)
- WTAP
- MBIST-TAP (LVTAP)

3.1 MBIST Controller

The FSM is used to control the overall sequence of events. The MBIST Controller typically requires very simple initialization and control to operate. The initialization could be to the order of which memory test algorithm to apply. All the read comparison happens in the MBIST Collar logic, so no compare logic is included in the MBIST Controller. The MBIST controller RTL is an IP (blackbox) generated by the Mentor Graphics tool.

MBIST Controller are of 3 types:
Non-Programmable Controller: A memory BIST controller that utilizes a subset of the Mentor Graphics library algorithms. Such a controller does not support user-defined algorithms.

Hard Programmable Controller: A memory BIST controller that supports user defined algorithms. The selected algorithms are hard-coded and cannot be changed the controller is generated.

Soft Programmable Controller: An extension of the Hard Programmable controller. Such a controller supports both hard-coded and soft-coded algorithms.

3.2 Memory Interface (Collar)

There is usually a wrapper around memory, known as memory collar that is used to select between functional inputs and test inputs based upon MBIST/functional mode selection bit. It interfaces the memory with on-chip logic and MBIST controller. The MBIST controller indicates the start of MBIST with a select input. The memory, then, starts the BIST algorithms and provides the test output to the controller. The controller compares this output with the reference output and indicates if the MBIST has passed or failed. There can be one controller for several memories. Also, memories can share the collar depending upon the test time requirement and type of memories. The memory collar RTL is an IP (blackbox) generated by the MG tool.

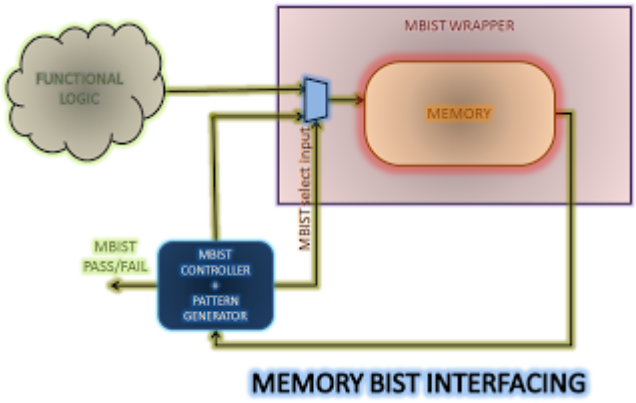

Figure 3.2.1: Memory Interface

3.3 BIST Wrapper

The BIST wrapper is the module where the MBIST logic is inserted by the Mentor MBIST tool. Its role is to keep the MBIST logic completely separate from the functional RTL. By segregating the MBIST logic, the bist wrapper enables a modular DFX strategy, allowing development of functional RTL to continue without any dependency on the MBIST RTL. It also facilitates the reuse of the MBIST logic for a particular IP block across multiple product derivatives.

BIST wrapper which is supported by the MBIST flow is just a verilog module with the MBIST logic (WTAP, controllers, and interfaces). The memories being tested reside outside of the bist wrapper within the functional RTL.

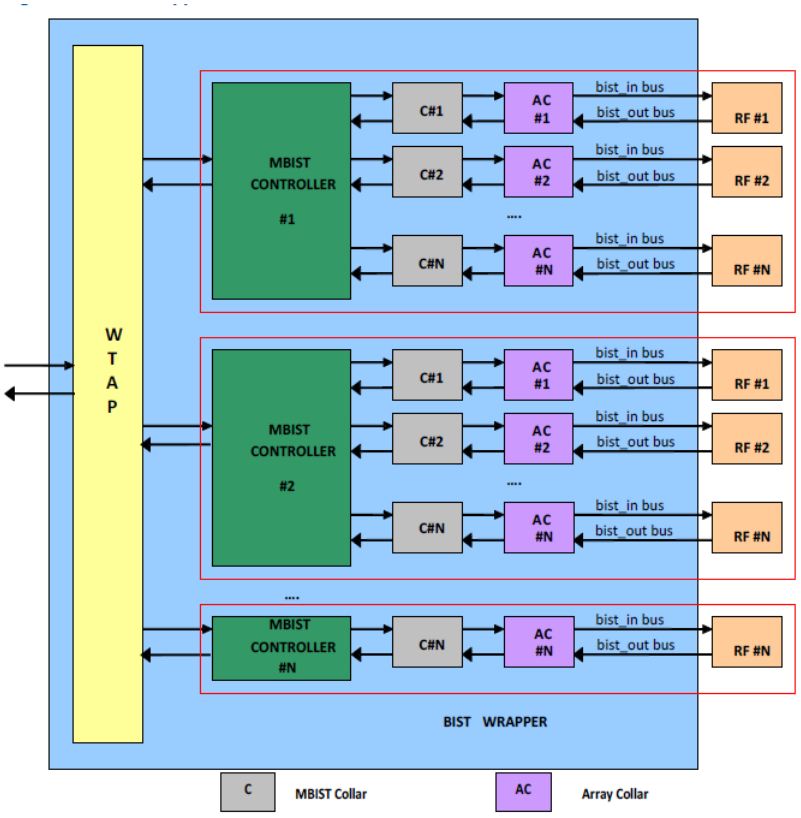

Figure 3.3.1: BIST Wrapper

Figure illustrates WTAP to RF (or ROMs/SRAM) connectivity through the bist wrapper. In order to facilitate MBIST insertion into this wrapper, which does not have memories instantiated, array collars are inserted into the wrapper to model the behavior of memories, as shown by the purple blocks in the diagram. After insertion, the array collars are removed and all that remains is the MBIST logic and the bist_in and bist_out buses from the MBIST logic to the boundary of the bist wrapper. The bist wrapper should then by integrated into the functional design and the bist_in/bist_out signals connected to the memories. This type of bist wrapper, as well as the array collars, are generated by the MBIST flow so these items do not need to be provided as input to the flow.

3.4 WTAP

WTAP is assigned one per array wrapper and generated by tool itself. The WTAP circuit is similar to the MBIST TAP except that it has no Finite State Machine (FSM). WTAP includes a Instruction register (includes status register) and BIST Setup Logic. Each WTAP is connected as a separate Data Register of the MBIST TAP. The MBIST TAP has seven global control signals (tdi, testLogicResetInv, updateDREnable, shiftDR_2EDGE, captureDR_2EDGE, tck, setupMode0) that is common to all WTAPs and fan-out from MBIST TAP to WTAP. Each WTAP is connected to a unique bistEn (Input to WTAP) and fromBist (Output to WTAP) from MBIST TAP The WTAP controller connected directly to the MBIST TAP (star topology) instead of connecting them in a daisy-chain configuration (ring topology). This allows connecting a unit with WTAP to the BIST TAP as soon as one is ready, and its verification with the testbench generation can be completed without waiting for the other units to be ready.

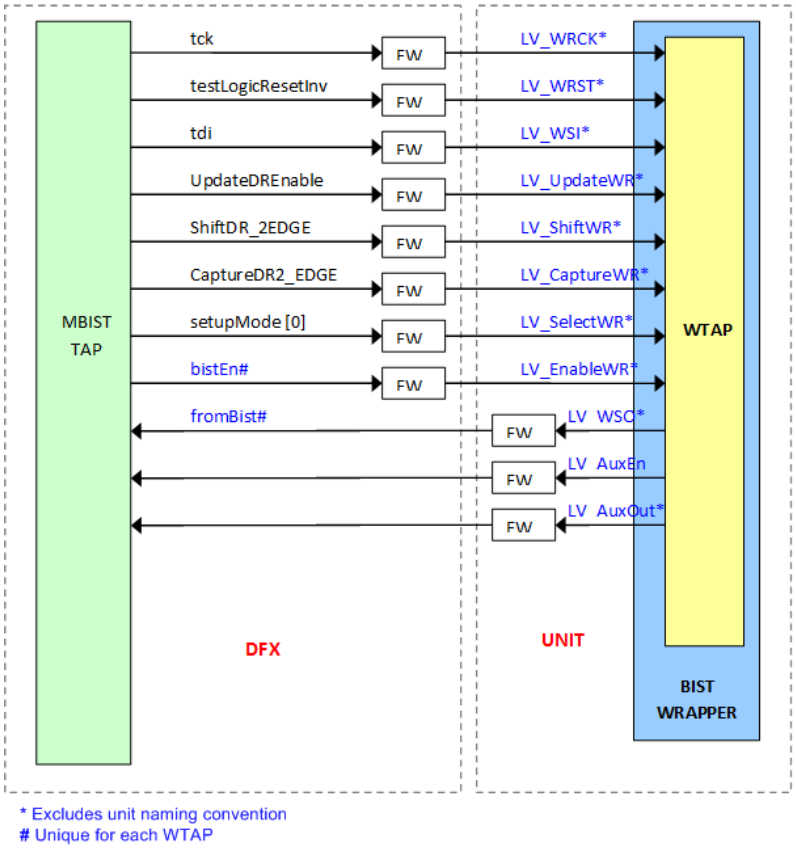

Figure 3.4.1: WTAP

3.5 MBIST-TAP (LVTAP)

The MBIST-TAPs reside in partitions cdus. Each unit has its owns WTAP(s) that communicate with the MBIST-TAP. The WTAPs communicate to the MBIST Controllers which in turn control MBIST Collar/MBIST Interface to each RF/ROM. The MBIST Controller and Collar reside in the Bist Wrapper. The final MBIST patterns are generated by Mentor Graphics tool with a post script to fit it into the testbench structure which inserts the preamble for reset sequence as well.

MBIST TAP is IEEE 1149.1 compliant. The interface to/from the BIST TAP are TDI, TRST, TMS, TCK and TDO. The TAP controller contains a finite state machine (FSM) that manages

access to all the instruction and data registers within the TAP and within the chip.

This is nothing but an interface b/w WTAP and MasterTap (CLTAP). This block provides the connections to n number of WTAPs based on the design requirements. i.e. with single MBIST-TAP multiple array wrappers or bist wrappers can be accessed. Also this is the reference point for all the patterns (algorithms), because this is where the selection of array wrapper takes place. Separate interface to all the WTAPs. Number of MBIST TAPs depend on the area of SoC as well as the number of MBIST Controller it is fed to. To minimize the routing and to ease the timing, output from this module are routed through WTAP.

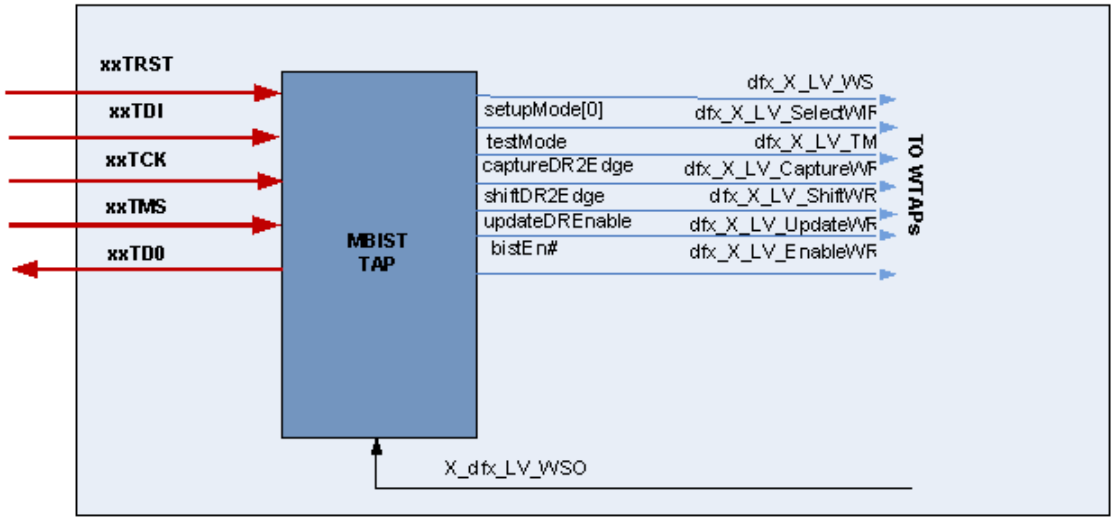

Figure 3.5.1: LVTAP

Chapter 4

MBIST Implementation

MBIST implementation must be performed in the RTL phase. This includes all Mentor controller blocks such as TAP, WTAP, MBIST controller, MBIST interfaces, etc. It is possible to insert MBIST into a gate-level netlist. All memories should be partitioned to a bist wrapper that is instantiated in the top level of the design. If the number of memories in the bist wrapper is controllable through parameters, the parameters must be set within the bist wrapper during BIST insertion and not outside of it. All clocks must be controllable and accessible from a top level port. The clocks used for MBIST should be the same clocks connected to the memories. If a memory has more than one clock port, one of the clocks will be chosen by the MBIST tool for MBIST. Usually the clock with the highest frequency is chosen.

4.1 MBIST Insertion Flow

4.1.1 General Steps Involved in the Flow

- MBIST compatibility rule checking and clock identification
- Generation of MBIST insertion/validation environment
- Generation and insertion of MBIST control hardware into design
- Test bench generation

- MBIST validation
- MBIST RTL and SDC post processing

4.1.2 Dependencies

- RTL should have completed some functional validation prior to MBIST.
- Validated mentor memory (lvlib) views and verilog files available for all memory types used in the design.

4.1.3 Inputs

- RTL that meets quality and memory partitioning requirements
- ROM contents file if ROMs are being MBISTed

4.1.4 Outputs

- MBIST inserted RTL files
- MBIST constraints for downstream tools
- SDC constraints for synthesis
- LVDB (Mentor Database) for downstream pattern generation
- Verilog tests for validation of MBIST hardware

4.1.5 Flow Script Descriptions(Tessent Tool)

- run_etchecker_clocks

Script which runs ETChecker tool to perform clock identification and design information extraction for the input netlist. Extracted clock information will be fed forward to downstream

MBIST tools to create clock control logic and clocking constraints. Extracted design information will be used to populate downstream tool configuration files and reduce the amount of input required by the user.

- run_etchecker_rules

Script re-runs ETChecker tool on input netlist and performs MBIST compatibility checks.

- run_etplan_gen

Script that generates a starting configuration file for the ETPlanner tool. The ETPlanner uses a configuration file and the design information extracted by ETChecker to build an environment where MBIST insertion can be performed. This environment is populated with pre-filled configuration files and Makefile automation scripts to make MBIST insertion easier and repeatable.

- run_fixETPlan

Script which modifies the auto-generated ETPlan configuration file to fill in design and SEG design kit information which is already known. This script is designed to simplify and reduce the amount of steps a user must perform.

- make checkPlan

This make target will run the ETPlanner tool to create a summary of the MBIST architecture which will be created by the MBIST insertion environment. If the user is not satisfied with the MBIST architecture they can adjust the options in the ETPlanner configuration file and rerun the make checkPlan target until the MBIST summary meets their expectations.

- make genLVWS

This make target will create an environment which automates the MBIST insertion process.

- make embedded_test

This Makefile target generates and inserts RTL for MBIST control logic. This step will also generate constraint files for downstream flow steps such as SDC files, PV constraints, FEV constraints, etc.

- make designe

Runs a Mentor analysis tool which checks the connections to/from the MBIST control logic created by the make embedded_test step of the flow. This is done to ensure the correctness of the MBIST connections and create a port mapping file which is used by downstream validation tools to create MBIST patterns and testbenches.

- run_tb

Generate testbenches for each algorithm.

- run_sim

Automates the simulation of each algorithm using the VCS simulator. Simple pass/fail result is reported at the end of each simulation.

- Final post processing

Post processing of the MBIST inserted RTL (if needed). Post processing of the SDC file to insert the correct hierarchy for the WTAP and MBIST controllers

Chapter 5

MBIST-TAP (LVTAP) Insertion

The goal of this part of the flow is to create the TAP controller RTL based on fullchip requirements and to generate testbenches/patterns that can be used for pre and post silicon validation. Normally, it is needed to have MBIST inserted into all of lower level IP blocks, those IP blocks integrated into fullchip, and a top level verilog file of available for TAP insertion. Since many of these items arent ready in the early stages of design, the flow works independent of the real fullchip design.

This flow generates a model of fullchip, which is basically just a top level verilog file that instantiates a dfx module and all of the lower level bist wrappers specified as input, and inserts the TAP controller into the dfx module.

TAP controller, and the dfx module created can be taken and it can be integrated into real design. In addition, the model which is created can be used to generate any of the MBIST testbenches and patterns needed in order to run validation.

The steps below cover creation of the input files, generation of the fake top level model, creation of the TAP controller, and pattern generation.

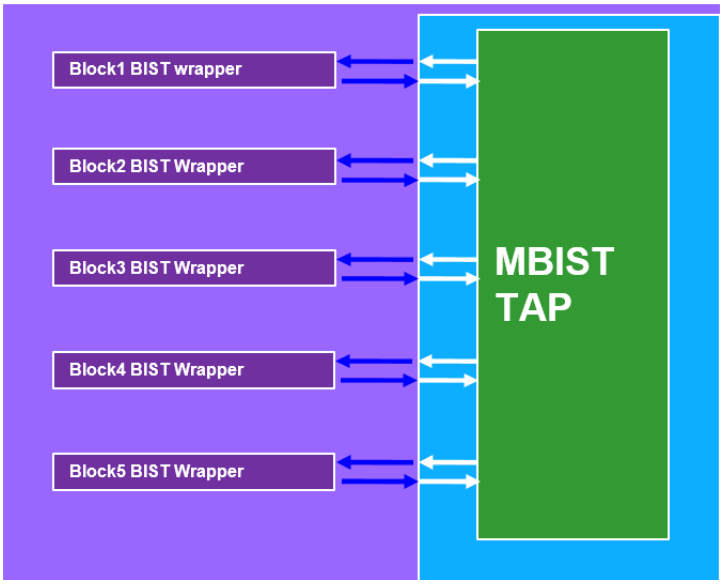

Figure 5.0.1: Fullchip model for TAP insertion and pattern generation

5.1 Flow Prerequisites

- Completed MBIST workspace areas for all bist wrappers to be covered by the TAP being inserted

1. Setup the MENTOR tool by sourcing the lv_setup script in the design kit

source /p/hdk/rtl/proj_tools/mbist/kit version/10nm/scripts/lv_setup

2. Create a file named ”unitPaths” and edit it to include the following information for each bist wrapper to be controlled by the TAP being inserted (one bist wrapper per line):

-unit unit name -module bist wrapper name -clk1 list of clocks separated by spaces in the following format: unit clock name:top level reference clock name:unit clock to reference clock ratio -path path to the to the MBIST run for this unit, ”none” if the MBIST run isn’t complete yet

Example:

-unit mcu_top_inst -module mcu_top_rf_roms -clk1 ”mcu_clk:VLV_REF_CLK_1:0.5 mcu_ch0_clk:VLV_REF
mcu_ch1_clk:VLV_REF_CLK_1:1” -path path to the MBIST run for this unit

A starter template for the unitPaths file is located at:

DFX_KIT_PATH/scripts/unitPaths_template

3. Copy the mbist_tap_mapping.txt file from the design kit and edit the Map2UnitAbbr, Map2UnitTag, and MapUnitOutPost sections to customize how signals to/from each of the bist wrappers are named. Map2UnitAbbr maps each unit name to an abbreviated name to be added to each signal related to that unit. Map2UnitTag adds an rf, sram, or rom suffix to each signal. MapUnitOut-Post can be used to add an additional identifier to the signal, ie. firewall identifiers. For any of these sections, if a signal identifier is not desired, it can be mapped to ””.

cp DFX_KIT_PATH/scripts/mbist_tap_mapping.txt .

4. Copy the mbist_tap_setup.txt file from the design kit and edit as needed for your design.

cp DFX_KIT_PATH/scripts/mbist_tap_setup.txt .

The following are the required edits to mbist_tap_setup.txt needed to enable this particular flow:

set MEM_LVLIBS_DIR ”full paths to all directories containing the memory .lvlib files used by the unit bist wrappers, separated by spaces” set REF_CLOCKS ”top level reference clock:frequency”

The following options are for generating asynchronized mode pattern at TAP level, which allow pattern generation at true TAP to IP frequencies instead of fixed TckRatio 8 :

USE_ASYNC_CLOCKS: Async mode pattern generation is turned on by default. set this option to 0 to turn off async mode pattern generation.

TCK_PERIOD: set proper TCK clock period here, for example if one IP clock frequency is 133MHz (7.518797ns), set TCK period to 40ns (25MHz) will set TCKRatio at about 5.3

5. Copy the flow run template from the design kit and edit with your design specific information

cp DFX_KIT_PATH/scripts/run_genMBIST_TAP_template
./run_MBistTap

The options available in this script are:

-path: Path to your work area
-ws: Name of your work area (name of directory created in the Work Area Setup step)
-tap: Name of top level DFX unit where TAP will be inserted, ie. dfx
-unitInfo: unitPaths
-project: Name of top level name for chip, ie. pnw

Then the functioning of commands from ”run_etchecker_clocks” upto ”make designe” command which are explained in MBIST insertion flow will be similar.

5.2 Generating BIST TAP level testbenches and patterns

1. Execute the run_mbist_tap_tb script to generate the initial etSignOff file and generate the preLayout_lvdb

./run_mbist_tap_tb

2. Edit the modulepaths.file.RF file in the current directory. Comment out any modules that don't belong, ie. if this is an RF run, comment out the SRAM modules and vice versa.

3. Run the run_Combine_etSignOff step to merge the etSignOff files of all modules specified in the modulepaths file into one file that will be used for pattern generation.

./run_Combine_etSignOff

4. Edit run_genMBISTPatterns file to select the patterns you want generated. By default all non-ROM patterns are generated. Comment out the patterns you do not want generated.

5. For each pattern, edit the corresponding options.* file to configure how the pattern is generated.

6. After editing the options files, run the pattern generation script.

./run_genMBISTPatterns

7. To generate patterns for SRAMs, ROMs, the steps above are repeated for the most part.

Chapter 6

Memory Repair

After the patterns are generated, its simulation is needed to verify if any memory/controller is failing or not. If a memory is failing, then it is necessary to find at which location it is failing. Suppose in a memory, one row is failing where failure is at 2 bits. But if we replace one row,then whole memory will be replaced. Then we need to enable that. Enable will be normal enable signal which will always be 1 if repair needs to be done. We need to know which row to be enabled. Suppose 3rd row needs to be enabled, then program RedRowAddr to 3 then only RedRowEn will work. If RedRowEn is 0 then repair won't work. Here there are 4 row redundancies that's why 4 enables and 4 addresses are needed. This fuses will be given through simv_args, this simv_args will map to svh file and in svh file, it will be mapped to sb_fuse_data and it will be reflected in simulation.

Below is the snippet of svh file where sb_fuse_data mapping will be given:

```
function void set_default();
    super.set_size(1);
     `ifdef GLS  `ovm_warning("tbt_fuse_tbt_fuse_RedRowEn0", " no GLS Signal specified.
     super.set_fuse_hdl_path({"tbt_tim_ip.tbt_tim_wrap.sb_fuse_data[41]"});
    desired_val = 1'h0;
```

Figure 6.0.1: Bit Mapping

When the svh file is available, error injecting needs to be done in sv file by manually adding forces. Below is the snippet:

```
 42     force soc_tb.soc.par_tcss.tcss_wrap.tcss.par_tbt_dma_0.tbt_dma_ip_wrap.tbt_dma_ip.tbt_dma_mem_wrap.tbt_dma_wra
ccm_ram_shell[0].ram_row_0_col_0.ip74sramfp2rsr2048x39m8r4c1.ip74sramfp2rsr2048x39m8r4c1_array.array[0][0][0] = 1'h1;
 43     force soc_tb.soc.par_tcss.tcss_wrap.tcss.par_tbt_dma_0.tbt_dma_ip_wrap.tbt_dma_ip.tbt_dma_mem_wrap.tbt_dma_wra
ccm_ram_shell[0].ram_row_0_col_0.ip74sramfp2rsr2048x39m8r4c1.ip74sramfp2rsr2048x39m8r4c1_array.array[1][1][0] = 1'h1;
 44     force soc_tb.soc.par_tcss.tcss_wrap.tcss.par_tbt_dma_0.tbt_dma_ip_wrap.tbt_dma_ip.tbt_dma_mem_wrap.tbt_dma_wra
ccm_ram_shell[0].ram_row_0_col_0.ip74sramfp2rsr2048x39m8r4c1.ip74sramfp2rsr2048x39m8r4c1_array.array[2][2][0] = 1'h1;
 45     force soc_tb.soc.par_tcss.tcss_wrap.tcss.par_tbt_dma_0.tbt_dma_ip_wrap.tbt_dma_ip.tbt_dma_mem_wrap.tbt_dma_wra
ccm_ram_shell[0].ram_row_0_col_0.ip74sramfp2rsr2048x39m8r4c1.ip74sramfp2rsr2048x39m8r4c1_array.array[3][3][0] = 1'h1;
```

Figure 6.0.2: Forces Applied

After the forces are added, simv_args needs to be added where enable signals needs to be 1 so that it will reflect the row addresses and repair will be done. Below is the snippet

```
"+tcss_tbt1_fuse_tbt_fuse_RedRowEn0=1",
"+tcss_tbt1_fuse_tbt_fuse_RedRowAddr0=00000000",
"+tcss_tbt1_fuse_tbt_fuse_RedRowEn1=1",
"+tcss_tbt1_fuse_tbt_fuse_RedRowAddr1=00000001",
"+tcss_tbt1_fuse_tbt_fuse_RedRowEn2=1",
"+tcss_tbt1_fuse_tbt_fuse_RedRowAddr2=00000010",
"+tcss_tbt1_fuse_tbt_fuse_RedRowEn3=1",
"+tcss_tbt1_fuse_tbt_fuse_RedRowAddr3=00000011",
```

Figure 6.0.3: Fuse Added

Below are the snippets of the waveforms where memory repair will be done and simulation will pass:

Figure 6.0.4: Simulation Waveform

Figure 6.0.5: Simulation Waveform

Figure 6.0.6: Simulation Waveform

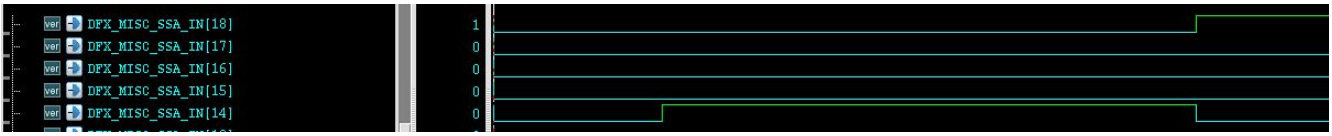

Figure 6.0.7: Simulation Waveform

Chapter 7

Post MBIST Simulation Debug

By default, VCS command scripts are written out during simulation to the path/ws/ETPlan/block_LVWS/ETA directory. These scripts make it easier to pull up the Verdi debugger for viewing the simulation waveforms in the event there are simulation FAILS. One script is written per algorithm simulated. When the script is executed, all of the required verilog files, testbenches, and collateral are loaded automatically and the viewer is launched.

7.1 Common Issues Encountered During Simulation

Unit level simulation using verilog testbench Flow option 1

- Miscompares on GO_ID_REG - LV_WSO is X
  - This usually indicates the memory or memories being tested were never written to in the first place. Ensure all signals required to be asserted to 0 or 1 to enable writing to the memory are either tied off in the RTL or asserted through the testbench by using the LV_ASSERT options in the mbist_setup.tcl file.
  - Another possibility is that the memory .lvlib file is incorrect and therefore the MBIST controller is trying to access addresses that dont exist. Verify the address mapping section in the .lvlib file is consistent with the memory RTL model.

- Miscompares on a GO_ID_REG LV_WSO is 0, expected to be 1
  - This indicates the test failed at some point during the running of the algorithm. This is usually more difficult to debug because it means the memory was written to but the data read back was not the data expected. At the point of failure, the MBISTPG_GO signal dropped low (see diagram below). This could be caused by any number of issues. Contact DA to get support for this type of failure.
- Miscompare on MBISTPG_DONE DONE is 0, expected to be 1
  - This indicates the pattern completed but the algorithm wasnt finished running. Try increasing the TEST_TIME_MULT setting in the mbist_setup.tcl file to 2.0 and re-generate your testbenches to see if that resolves the issue.

Unit level simulation using verilog testbench - Flow option 2

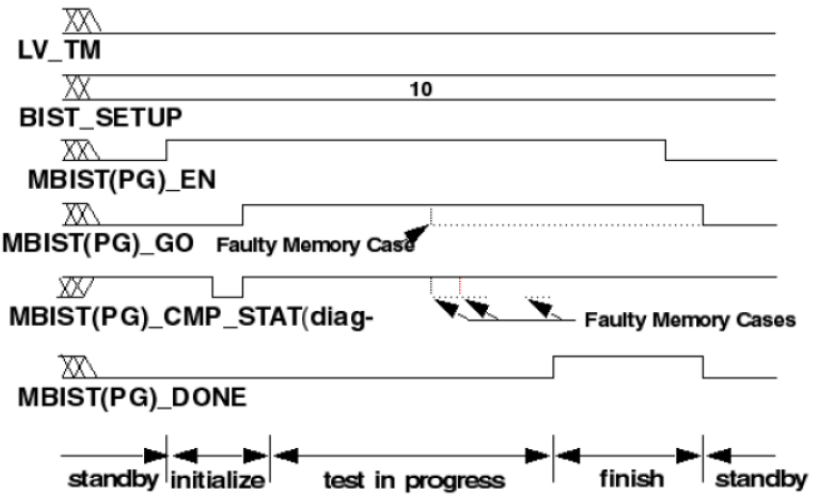

Figure 7.1.1: Unit Level Simulation

All of the above apply, but here are some other issues to watch out for:

- If GO_ID_REG is X, try to trace all signals from the bist wrapper to the memories and from the memories back to the bist wrapper to make sure everything is connected correctly.
- Make sure all required clocks are running at the memory and at the MBIST controllers and are not being blocked by clock gates upstream.
- Ensure the clock phase at the memories matches the clock phase at the controller (both the memory and the controller should be driven by the same clock).

Chapter 8

Conclusion

This thesis aims at understanding of why MBIST insertion is preferred over functional/atspeed testing. It will allow for robust testing of memories. It does simultaneous testing of multiple memories within the design. With this process all the memories can be tested in parallel. Running the Raster algorithm will help to find the faulty location by which memories can be repaired and at later stage they can be reused.

This thesis aims at inserting MBIST in the design so that all the defects present in the memories arrays caused during manufacturing process can be validated and the memories can be further repaired and reused also. This whole process of MBIST implementation is done by Tessent Mentor Graphics Tool which will reduce the test-time due to in-built test circuit.

References

[1] SoC DFT Handbook, Intel

[2] MBIST FLow and Debug Handbook, Intel

[3] Tessent Memory BIST User's and Reference Manual, Mentor Graphics