5-2018

Design and Verification of a Dual Port RAM Using UVM Methodology

Manikandan Sriram Mohan Dass
ms1289@rit.edu

Follow this and additional works at: http://scholarworks.rit.edu/theses

Recommended Citation

This Master's Project is brought to you for free and open access by the Thesis/Dissertation Collections at RIT Scholar Works. It has been accepted for inclusion in Theses by an authorized administrator of RIT Scholar Works. For more information, please contact ritscholarworks@rit.edu.
DESIGN AND VERIFICATION OF A DUAL PORT RAM USING UVM METHODOLOGY

by

Manikandan Sriram Mohan Dass

GRADUATE PAPER

Submitted in partial fulfillment
of the requirements for the degree of
MASTER OF SCIENCE
in Electrical Engineering

Approved by:

________________________________________
Mr. Mark A. Indovina, Lecturer
Graduate Research Advisor, Department of Electrical and Microelectronic Engineering

________________________________________
Dr. Sohail A. Dianat, Professor
Department Head, Department of Electrical and Microelectronic Engineering

DEPARTMENT OF ELECTRICAL AND MICROELECTRONIC ENGINEERING
KATE GLEASON COLLEGE OF ENGINEERING
ROCHESTER INSTITUTE OF TECHNOLOGY
ROCHESTER, NEW YORK

MAY, 2018
I would like to dedicate this work to my family, for their love and support during my masters.
Declaration

I hereby declare that except where specific reference is made to the work of others, that all content of this Graduate Paper are original and have not been submitted in whole or in part for consideration for any other degree or qualification in this, or any other University. This Graduate Project is the result of my own work and includes nothing which is the outcome of work done in collaboration, except where specifically indicated in the text.

Manikandan Sriram Mohan Dass
May, 2018
Acknowledgements

I would like to thank my professor, Mark A. Indovina, for his immense support, guidance and feedback on my project. I also would like to thank all my teachers who have helped me right from the start of my school. I would like to say special thanks to my friends cum motivators Nikhil Velguenkar and Deepak Siddharth for their valuable suggestions.
Abstract

Data-intensive applications such as Deep Learning, Big Data, and Computer Vision have resulted in more demand for on-chip memory storage. Hence, state of the art Systems on Chips (SOCs) have a memory that occupies somewhere between 50\% to 90\% of the die space. Extensive Research is being done in the field of memory technology to improve the efficiency of memory packaging. This effort has not always been successful because densely packed memory structures can experience defects during the fabrication process. Thus, it is critical to test the embedded memory modules once they are taped out. Along with testing, functional verification of a module makes sure that the design works the way it has been intended to perform. This paper proposes a built-in self-test (BIST) to validate a Dual Port Static RAM module and a complete layered test bench to verify the module’s operation functionally. The BIST has been designed using a finite state machine and has been targeted against most of the general SRAM faults in a given linear time constraint of $O(23n)$. The layered test bench has been designed using Universal Verification Methodology (UVM), a standardized class library which has increased the re-usability and automation to the existing design verification language, SystemVerilog.
Contents

Contents                                      v

List of Figures                               xii

List of Tables                                xiii

1 Introduction                                1
   1.1 Research Goals                           3
   1.2 Contributions                            3
   1.3 Organization                             4

2 Bibliographical Research                    5

3 Background on Memory Technologies           8
   3.1 Types of Memory.                        9
       3.1.1 Volatile memories:                9
           3.1.1.1 SRAM:                      10
           3.1.1.2 DRAM:                      11
       3.1.2 Non-Volatile Memories              11
           3.1.2.1 ROM:                       11
3.1.2.2 EEPROM: .............................................. 12
3.1.2.3 FLASH memory: ..................................... 12
3.1.3 Miscellaneous memories ................................ 12
3.2 Concerns in Memory Designs .......................... 13
  3.2.1 Aspect Ratio: ......................................... 13
  3.2.2 Access Time: ......................................... 13
  3.2.3 Power Dissipation: ................................... 14
  3.2.4 Memory Integration issues: ......................... 14
3.3 DUT Model ............................................... 15

4 Memory and Fault Models ........................................ 16
  4.1 Testing Methods ......................................... 16
    4.1.1 Embedded Microprocessor Access .................... 16
    4.1.2 Direct Memory Access ................................ 17
    4.1.3 Memory BIST ........................................ 17
  4.2 Memory Testing Model .................................... 18
    4.2.1 Memory Failure Modes ............................... 18
  4.3 Common Fault Models .................................... 19
    4.3.1 Stuck at Faults (SAF) ............................... 20
    4.3.2 Transition Fault (TF) ............................... 20
    4.3.3 Address Decoder Fault (AF) ......................... 20
    4.3.4 Stuck at Open Fault (SOF) .......................... 20
    4.3.5 Coupling Faults (CF) ............................... 21
      4.3.5.1 Types of Coupling faults ....................... 21
    4.3.6 Data Retention faults (DRF) ....................... 21
### Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.3.6.1 DRAM</td>
<td>22</td>
</tr>
<tr>
<td>4.3.6.2 SRAM</td>
<td>22</td>
</tr>
<tr>
<td>4.3.7 Multi-Port Faults</td>
<td>22</td>
</tr>
<tr>
<td>4.4 RAM Test Algorithm</td>
<td>23</td>
</tr>
<tr>
<td>4.4.1 ZERO-ONE Algorithm</td>
<td>23</td>
</tr>
<tr>
<td>4.4.2 Checkerboard Algorithm</td>
<td>23</td>
</tr>
<tr>
<td>4.4.3 GALPAT</td>
<td>24</td>
</tr>
<tr>
<td>4.4.4 WALPAT</td>
<td>24</td>
</tr>
<tr>
<td>4.4.5 MATS</td>
<td>24</td>
</tr>
<tr>
<td>4.4.5.1 OR-type Faults</td>
<td>24</td>
</tr>
<tr>
<td>4.4.5.2 AND-TYPE Faults</td>
<td>25</td>
</tr>
<tr>
<td>4.4.6 MATS+</td>
<td>25</td>
</tr>
<tr>
<td>4.4.7 MATS ++</td>
<td>25</td>
</tr>
<tr>
<td>4.4.8 March-1/0</td>
<td>25</td>
</tr>
<tr>
<td>4.4.9 MARCH C</td>
<td>25</td>
</tr>
<tr>
<td>4.4.10 MARCH Extended March C-</td>
<td>26</td>
</tr>
</tbody>
</table>

### 5 DUT and BIST Environment

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.1 DUT Introduction</td>
<td>27</td>
</tr>
<tr>
<td>5.2 Memory BIST</td>
<td>29</td>
</tr>
<tr>
<td>5.2.1 Hardware Resource.</td>
<td>30</td>
</tr>
<tr>
<td>5.2.1.1 Multiplexers:</td>
<td>30</td>
</tr>
<tr>
<td>5.2.1.2 Counter:</td>
<td>31</td>
</tr>
<tr>
<td>5.2.1.3 Address Selector:</td>
<td>31</td>
</tr>
<tr>
<td>5.2.1.4 Checker Block:</td>
<td>32</td>
</tr>
</tbody>
</table>
6 Verification Concepts

6.1 SystemVerilog

6.1.1 SystemVerilog Components

6.1.1.1 SystemVerilog for Design:

6.1.1.2 System Verilog for Verification (Test Benches):

6.1.1.3 SystemVerilog Application Programming Interface:

6.1.1.4 SystemVerilog for DPI:

6.1.1.5 SystemVerilog Assertions:

6.1.2 Basic Test Environment

6.1.2.1 Driver and Monitor:

6.1.2.2 Agent

6.1.2.3 Scoreboard and Checker

6.1.2.4 Generator

6.1.2.5 Environment

6.1.2.6 Test

6.1.2.7 Assertion

6.1.2.8 Coverage

6.2 UVM

6.2.1 Basic Tenets of UVM:

6.2.1.1 Functionality encapsulation

6.2.1.2 TLM (Transaction Level Modeling)

6.2.1.3 Using sequences for stimulus generation
6.2.1.4 Configuration .................................................. 40
6.2.1.5 Layering .......................................................... 40
6.2.1.6 Re-usability ...................................................... 40
6.2.2 UVM classes ....................................................... 40
  6.2.2.1 uvm_transaction ............................................. 41
  6.2.2.2 uvm_component .............................................. 41
  6.2.2.3 uvm_object .................................................. 41
6.2.3 UVM Phases ....................................................... 41
  6.2.3.1 function void build_phase ................................. 41
  6.2.3.2 function void connect_phase ............................. 42
  6.2.3.3 function void end_of_elaboration ........................ 42
  6.2.3.4 task run_phase ............................................ 42
  6.2.3.5 report_phase ............................................... 42

7 Test Methodology .................................................... 43
  7.1 Test Environment ................................................ 44
    7.1.1 Sequence .................................................. 45
      7.1.1.1 Variables .............................................. 45
    7.1.2 Constraints ............................................... 46
      7.1.2.1 Operation ............................................. 46
      7.1.2.2 Write Collision ...................................... 46
    7.1.3 Sequencer ................................................ 46
    7.1.4 Driver .................................................... 47
      7.1.4.1 Reset_BIST ............................................ 47
      7.1.4.2 Run_PortA ............................................. 47
Contents

7.1.4.3 Run_PortB ............................................. 47
7.1.4.4 Write task .......................................... 47
7.1.5 Monitor .................................................. 48
7.1.6 Agent ................................................... 48
7.1.7 Scoreboard ............................................. 48
7.1.8 Environment ........................................... 48
7.2 Test Planning ............................................ 48
7.2.1 Functional Coverage Plan ......................... 49

8 Result and Discussion ................................. 50
8.1 BIST Results ............................................ 50
8.1.1 Synthesis Report of BIST ......................... 50
8.1.2 Simulation Result of BIST ......................... 51
8.2 BIST UVM_Testbench ................................. 53

9 Conclusion ................................................ 55
9.1 Future Work ............................................ 56

References ............................................... 57

I Source Code ............................................ 61
I.1 BIST RTL ................................................. 61
I.2 Interface ................................................ 117
I.3 Driver ................................................... 119
I.4 Environment .......................................... 126
I.5 Agent ................................................... 128
I.6 Sequencer .............................................. 130
<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>I.7</td>
<td>Monitor</td>
<td>133</td>
</tr>
<tr>
<td>I.8</td>
<td>Scoreboard</td>
<td>136</td>
</tr>
<tr>
<td>I.9</td>
<td>SDP Test</td>
<td>141</td>
</tr>
<tr>
<td>I.10</td>
<td>Top Test</td>
<td>143</td>
</tr>
</tbody>
</table>
List of Figures

5.1 DUT Pin Diagram .................................................. 28
5.2 BIST Architecture ................................................. 30
5.3 BIST Multiplexer ................................................... 31
5.4 BIST State Machine ............................................... 33
6.1 Basic Test Environment[1] ....................................... 36
7.1 Test bench components ........................................... 44
7.2 Verification Environment .......................................... 45
8.1 BIST Simulation .................................................... 52
8.2 UVM Test Bench Simulation ....................................... 53
8.3 Coverage Results .................................................. 54
List of Tables

5.1 RAM Input/Output ........................................... 28

8.1 Logic Synthesis report of BIST ............................... 51
8.2 Comparative Results on Synthesis of BIST using different Libraries ............ 51
8.3 Timing Analysis coverage of BIST ............................. 52
Chapter 1

Introduction

With the scaling of technology to smaller transistors, the scope of manufacturing defects has increased. Most of the semiconductor manufacturers and process technologists are rated based on their memory array technology. The demand of memory arrays is growing to a stage where embedded memory structures are on the verge of occupying 90% of the die area [2]. This is at the cost of high vulnerability to fabrication faults. The percentage defects for memory structures in a die has a higher probability than the logic structures. This is due to the tight packing of transistors for memory design. Design for Tests or testability has always been a least concentrated field in Very-large-scale integration (VLSI). Memory testing takes a great deal of effort as it is difficult compared to testing the logic design. Insufficient test access in testing memory results in more significant problems as the memory array designs are mostly layered deeper on a chip. Structure memory units cannot be represented using their gate level equivalents. This is due to the large sizes of memory and the number of flip-flops required to model them. For example, an 8K memory array with 32-bit word size almost requires 262 K flip-flops and also large-scale combinational decoder blocks. Unlike logic testing, memory testing is more complicated due to its defect-oriented testing nature. Due to the high-density nature of memory, it has a defect-
sensitive pattern. Since memory is designed structurally and they are observable and controllable at the array level, test engineers develop an algorithmic test plan to identify the faults.

There are various technologies for designing memory. The selection of the technology depends on the design constraints such as area, power, and speed. For every such technology, the potential fault model changes according to its physical structure. A test engineer needs to consider those fault models and create the test support either co-existing inside the chip like built-in self-test (BIST) or accessing them through external structures. It must be noted that in the case of a BIST methodology, extra hardware resources are needed to be inbuilt inside the chip, and this should not violate the power, area, cost, and performance of the chip. The time required for the testing is also highly critical as the delay in bringing a chip to the market might even hit the return on the capital investment. Thus, there is a trade-off between the extensive fault testing and time constraints. Tests algorithms range from time constraint $O(\sqrt{n})$ to $O(n^3)$ [3]. The test engineers try to cover most of the faults in linear time complexity. This paper proposes a test with a time constraint of $O(23n)$ to test the faults.

Verification of the functional correctness of a design before the tape-out is more important than the validation after the tape-out. Verification takes more than 70% of the design cycle time. The languages preferred by the RTL design engineers and the verification engineers have a major difference in the aspect of synthesis. Verification engineers need a more simulation-friendly language while the design engineers need a synthesis friendly language. This made the design engineers prefer RTL languages such Verilog and VHDL, and verification engineers preferred languages such as Sugar OVA, VERA, SPECMAN, etc.

To unify design and verification languages, SystemVerilog (SV) was introduced as an extensive enhancement to IEEE 1364 Verilog standard. SystemVerilog has introduced many verification aides such as OOPS concepts, assertion, and coverage to the verification environment. To improve the reusability and automation, a uniform verification environment was required, and
thus many standard methodologies have been widely used in the industry [4]. Currently, the most widely accepted methodology is Universal Verification Methodology (UVM). Such methodologies ensure that the test bench is layered and separated into individual independent files. This layering initially takes up the verification cycle but will speed up the overall verification process and helps in reusability. In the proposed project, an industrial standard dual port RAM module is taken as the DUT and initially a memory BIST has been designed for the module. Then this module is verified using UVM methodology by creating independent transactors for each port of the RAM. The layered testbench has a scoreboard with an inbuilt checker. An associative array is implemented to work as an efficient model in the scoreboard. A neat coverage-driven constraint random sequence is generated by the sequencer for each port. The test is designed to run until it covers all the set cover-points.

1.1 Research Goals

The main goal of this project is to get hands-on experience on DFT concepts, and to design and implement an UVM based verification environment.

- To understand the importance of design for testability in a commercial chip.

- To understand the working of memory BIST and create an efficient BIST implementation.

- To be introduced to constraint random verification using UVM component classes and set up a proper testbench.

1.2 Contributions

The major contributions of this project are as below:
1.3 Organization

1. A parameterized BIST environment was designed to test the Dual Port RAM.

2. A UVM based layered test bench was designed to verify the functionality of the Dual Port Ram.

1.3 Organization

The structure of the paper is as follows:

- Chapter 2: This chapter briefs about previous work done on various memory test architectures and verification techniques.

- Chapter 3: This chapter describes about different memory technologies.

- Chapter 4: This chapter covers the various fault models in memory and testing procedure.

- Chapter 5: This chapter explains the implementation of the proposed BIST architecture.

- Chapter 6: This chapter briefly explains about general verification concepts.

- Chapter 7: This chapter discusses about the test plan and verification environment used to verify the given DUT.

- Chapter 8: This chapter comprises of the resultant simulations and findings from the tests.

- Chapter 9: The conclusion and possible future work are briefly discussed in this chapter.
Chapter 2

Bibliographical Research

Due to advanced development in the field of fabrication, the feature size of transistors has scaled down to few nanometers. This has dramatically increased the density of memory on a SoC die. Unfortunately, the yield on the fabrication of these dense designs has significantly reduced. Thus, it is essential to test these memories for failures after manufacturing. A lot of research in the field of Design for Testability (DFT) has been dedicated to memory failure models. Unlike logic testing, the memory testing is strongly influenced by the technology and the application. BIST is an effective testing method for embedded memories. Many algorithms for such BIST environments have been proposed by test engineers. Each algorithm tries to target a specific requirement such as extensive fault coverage, low power consumption, small area, minimum time complexity, and cost of the design. Even though the test algorithm remains same, the BIST configuration varies depending on the targeted applications.

[5] proposes a BIST configuration for block memories in reconfigurable Field Programmable Gate Array (FPGA) chips. The authors also discusses about a parametrized BIST which could be reused on multiple embedded models with least change to the configuration. [5] highlights the uses of FPGA’s or CPLD’s blocks to implement the Test pattern generators (TPGs), logic
structures and comparator circuits. This proposed design is highly effective for embedded signal or image processors. The tradeoff between flexibility and area of hardware design was addressed in [6]. It targeted multiple March algorithms and also an efficient repair mechanism for the faulty location. By implementing hardware-sharing, the authors also achieved area effectiveness. A dynamic march algorithm for testing was created by a software automation tool that was implemented using Tool Command language.

The test efficiency can be improved by identifying the actual geometric position of memory addresses[6]. An efficient way to identify coupling faults by exploiting the geometric nature of memory was proposed in [7]. Through thorough research on the physical design of the memory, an exact statistical model of possible coupling faults was planned [7]. Testing for faults only on the address predicted by the statistical model was efficient in the aspect of testing time. The limitation of the statistical model is the lack of support to target all memory technology. Micro-fabrication engineers have been coming up with various new physical concepts to understand and model random faults. Two such new fault models were proposed and an efficient algorithm was devised to cover them [8]. The new fault models were based on inter-port decoder faults and dynamic neighborhood coupling faults. An efficient algorithm for dynamic neighborhood faults were devised with the use of graph theory and statistics. An additional extension to the existing March algorithms were discussed in [9] for targeting cost-effective testing. The suggested algorithm in [9] also covers the recent fault models for simultaneous operation faults on dual-port DRAMs. Design constraints such as power, area, and cost are not directly affected by the test algorithm. Time is the only constraint directly dependent on test algorithm and is significantly reduced by implementing a fast memory simulator as explained in [10]. The authors have modeled plausible faults in a software model of memory to run possible test patterns and checked for the coverage. Thus, they came up with the shortest test pattern to cover the given fault models. The proposed system could achieve reducing complexity \(O(n^3)\) algorithms to \(O(n^2)\).
A more advanced test design is the Built-In Self-Repair (BISR) architecture. With growing usage of embedded memories in critical applications, it is not sufficient enough if a fault is diagnosed but important that a repairing strategy for the faulty memory location is devised. An efficient BISR architecture was proposed in [2] by using the least area and power. The BISR had a 100% fault repair with an area overhead of just 2%. With SV based layered test benches overtaking the traditional Verilog based test benches, testing has become simple and fast. Apart from the features for verification, SV was also useful for emulation testing through synthesizable assertion [11]. Verilog based test benches supported a directed testing approach while SV targeted a constraint-driven random testing approach. Using functional coverage, a proper test plan was designed to verify UART IP [12]. The authors clearly explain the differences in timing between directed and random testing. Directed testing takes a longer time to cover the functional cover points than a random testing [1]. A more efficient testing approach initially used a random test and later on targeted directed tests for uncovered function cover-points [12]. When UVM is used to verify SOC design, careful consideration is required since the engineers with different background experience on verification come together for a project. This issue is addressed in [13] and a practical structure of flow for SOC verification is explained.
Chapter 3

Background on Memory Technologies

Right from the beginning of semiconductor industry, storing information has been under constant research and improvement. There has been a steep improvement in memory technologies since the invention of transistors, but unfortunately, the memory requirement has always been ahead of the technology’s capability. For instance, during the boom of PC industry, Bill Gates stated that “640 kB memory ought to be enough for anybody”, but in the current scenario users need Terabytes of data. As we go through the timeline, around the year 1961 Texas instruments manufactured and shipped the first commercial memory spanning up to a few hundred bits. Then around the year, 1965 Moore came up with his famous Moore’s law stating memories would be built on integrated electronics. Later in less than a year’s time, 16-bit Transistor-Transistor-Logic (TTL) was commercialized by Honeywell. In the same year, DRAM cell was invented using a single transistor. This created a huge impact, wherein, a few megabytes of memory were stacked into a considerably smaller area. Few decades ahead the present DRAM technology is fabricated on 30nm technology as DDR4 with 8 GB of capacity [14]. Furthermore, NAND flash memories are fabricated on 20nm scale and come in capacities that span up to as much as 64 GB [15]. These memories are of variegated kinds and are specially designed to cater specific needs.
Ideal requirements of any memory technology are as follows:

1. Low Cost.

2. High speed.

3. Higher Capacity.

4. Lower power and Energy efficiency.

5. High reliability.

Since there is always a trade-off between these design constraints, based on specific constraint requirements different memory technologies are fabricated. For example, in case of cache memory higher speed is a critical constraint. In many cases power also plays an important role in deciding the type of memory to be used since few memory technologies need power to hold data. This is highly critical in embedded devices where hand-held batteries would be the only available source of power. Memories can be broadly classified into three kinds. Volatile memories, Electronic based Non-Volatile memories, and Mechanical based Non-Volatile memories.

3.1 Types of Memory.

3.1.1 Volatile memories:

Volatile memories require power to hold data. They hold data until power is provided and the moment the power is shut down they lose the data that they hold. Most of the embedded Complimentary Metal Oxide Semiconductor (CMOS) based memory arrays are volatile. The most commonly used volatile memories are Static Random Access memory (SRAM) and Dynamic Random access memory (DRAM).
3.1 Types of Memory.

3.1.1 SRAM:

SRAM is the most commonly used embedded RAM. The main advantage of SRAM is its ability to operate at high speed. For the same reason, it is preferred in the cache memory, tag memory, and Content Addressable Memory. Though SRAM, in a nutshell, can hold data permanently until power is available, on a long run it could discharge data to lose information. But on removing power it cannot retain data for more than a few microseconds. The volatility is generally preferred on SRAM to secure information when removed from a device. SRAM, unlike DRAM, doesn’t require a refresh circuit. It is also highly reliable compared to DRAM. Since it doesn’t necessitate the use of additional units to maintain information, it is highly power efficient. The power requirement is directly proportional to the frequency of accesses performed. When SRAM is used in a high-frequency unit it can almost consume the same power as a DRAM. At the same time in case of an embedded process running at a lower clock frequency, it consumes negligible power. SRAM is completely designed using transistors. The number of transistors depends upon the number of ports. The most common design of single port SRAM consists of six transistors to make a single bit cell. In case of dual port RAM, the SRAM requires at least 8 transistors. These transistors make SRAM bulky, thereby making it inefficient in space and cost. SRAM has three modes of operation. They are—Standby, Reading or Writing. In standby mode, the word line is not asserted and there is no data movement. In the reading mode, the word line is asserted, and the cell is read through the single access transistor. In the writing mode, the data is supposed to be written to the bit line. The DUT taken up in this project is a synchronous dual port static RAM.
3.1 Types of Memory

3.1.1.2 DRAM:

SRAM being bulky with 6 transistors was not the perfect solution for large memory. A more efficient solution was the DRAM. DRAM consists of only one transistor per bit memory, but it requires a complicated fabrication process to fabricate capacitor which actually holds the data. The capacitor is prone to leakage effects. This necessitates the DRAM circuitry to have a refreshing circuit to overcome leakage effects. The main task of the refresh circuit is, to make sure the capacitor holds the data. Therefore capacitances large enough to handle low leakage, high retention time and reliable sensing should be designed. The timing of the refresh circuit varies with capacitance and directly affects the power efficiency. Another disadvantage of capacitive storage is that, the reads are destructive i.e. reading a data, discharges the capacitor, that results in losing data and the capacitor should be recharged again to maintain the data. Sense amplifiers are used for reading purposes. DRAM’s are slower when compared to the accessing speed of corresponding SRAM circuits. Despite the disadvantages of DRAM, DRAM circuits are still preferred for their cost and space efficiency. Current DRAM technology is based on DDR4 technology and runs on 266 MHz with a bandwidth of 3200mbps.

3.1.2 Non-Volatile Memories

3.1.2.1 ROM:

Read only memory (ROM) is generally used in processors to hold the primary boot data. They are non-volatile in nature and can hold data even when they are not powered. The most common type of ROM is the mask programmable ROM and the contents are burnt on it at the time of fabrication itself. Most of ROM memories are not re-writable. In order to write into a ROM special re-processing is required.
3.1 Types of Memory.

3.1.2.2 EEPROM:

Electronically Erasable Programmable read-only memory (EEPROM) is also another non-volatile read only memory. They are floating-gate transistors organized as arrays. By applying control signals, they can be erased or reprogrammed with different data. They are written by applying higher than normal voltage. They are considerably slow as compared to DRAM and SRAM, and generally not used as an on-chip memory. The main advantage of EEPROM is that it is byte addressable in case of erasing or programming.

3.1.2.3 FLASH memory:

They belong to a non-volatile memory family. There are two most common FLASH memories, one is NAND and the other is NOR type FLASH memory. NOR memory has the ability to access random data and is byte addressable. It is comparatively slower to program/erase. They are direct alternatives to EEPROM. NAND type memories are faster to program/erase but are not directly byte addressable. They have relatively slower random memory access. They are the most common memory technology for file storage. The data storage demand recently has made NAND based FLASH the most preferred for their cheapest and smallest area per bit. It is said that the name FLASH came about due to how these memories are erased.

3.1.3 Miscellaneous memories

Other upcoming cutting edge, memory technologies such as Resistive RAM (RRAM), Ion Conducting RAM, Phase Change RAM (PCRAM), Spin Torque Transfer RAM (STT-RAM) and Magnetization based devices are under constant research and have the potential to replace the existing technologies. In Magnetization based RAM, data is stored in dielectric packed between two ferromagnetic plates. One of the plates is constantly charged while the other plate is charged
or discharged as per the bit to be stored. The PCRAM could substitute the existing NAND flash since they have a better endurance and are stable at higher frequencies. The PCRAM has the ability to change their impedance with a change in their material property, which can be induced through heat or by passing electric current.

3.2 Concerns in Memory Designs

3.2.1 Aspect Ratio:

A multitude of memory array technologies has a correlation between the height and the width. This ratio between the X-length and Y-width is called aspect ratio. The aspect ratio is more of a physical design constraint because in case of floor planning or while placing in a die it affects the ability to route or would affect the effective area of the cell. To meet the physical design constraints, designers try to change the orientation into square, horizontal or vertical rectangle, breaking down into smaller fragments. But this fragmentation can affect the performance of the overall memory. The number of rows or columns typically affects the decoder circuit which in turn affects the power dissipation and timing constraints.

3.2.2 Access Time:

Performance of memory is dependent on the access time of the memory. To reduce the read request time or write request time, access time must be kept in consideration. In few technologies, the write time is faster than the read time like NAND FLASH while in few memories like NOR flash the read access time is faster. The access time can be reduced by reducing the number of rows or columns and improving the efficiency of the decoder unit. Designers also increase or upsize the capacitance/drive strength to get faster circuits.
3.2 Concerns in Memory Designs

3.2.3 Power Dissipation:

An increased number of rows or columns can increase the power dissipation of the memory array. Stronger drive currents are required to make circuits faster which also results in higher power dissipation. The power dissipation is a direct function of the aspect ratio, timing and operation frequency and is represented in terms of $mW/MHz$ per operation.

3.2.4 Memory Integration issues:

The primary concern in integrating memories is the available area and required memory. This physical issue is directly related to the aspect ratio and the size of a single bit in the memory technology. Memory designers have started to prefer distributed memories over contiguous memory. Increase in the integration of modules per unit square area and reduction in the memory size has caused this trend. As the number of embedded memories are increasing, chip level decisions such as floor planning, centralized or distributed address selection and type of decoder unit must all be calculated initially. Memories have been historically placed in a specific side of a die and were accessed through a bus. Shrinking memory geometries have allowed memory a more of distributed locality. But this has posed potential problems regarding non-uniform routing delay. This routing delay on a sub-micron level is a major concern and directly affects the access time. Also, in case of a centralized memory, a simple address data bus would be sufficient, but distributed memory requires a wire intensive bus to be routed around the memories. The main point to consider while designing memory is the power dissipation and the ability of die to dissipate the power without affecting the performance. The worst-case power dissipation of the cell at clock’s maximum frequency should be considered to test the efficiency of the power structure of chip and the packages.
3.3 DUT Model

The given Device Under Test (DUT) is manufactured by Faraday Technology Corporation. It is a Synchronous Dual Port-Static Random-Access Memory. The model number is FSC0H_D SJ. It is possible to fabricate it in 0.13 micrometer UMC high-speed technology. It’s operating voltage ranges between 1.08 V and 1.32 V. The operating temperature is between 0 degree Celsius to 85 degrees Celsius. Minimum of 5 metals is required to fabricate it. Both read and write are synchronous to individual clock ports. The module supports both byte write, and word write. It utilizes a distributed memory model, to make the best use of the aspect ratio. More about the pins would be discussed in chapter 3.
Chapter 4

Memory and Fault Models

4.1 Testing Methods

Direct functional access to an individual memory in an array of embedded memories if possible, is not usually not practical. This poses a serious test problem. Test engineers have come up with three common methods for testing the circuit. Each of these methods has their own design trade-offs in the aspect of cost, area, pins required, power and timing.

4.1.1 Embedded Microprocessor Access

This was the first testing procedure initially followed by test engineers. An embedded microprocessor is connected and interfaced with the memory. The vectors are generated by compiled assembly code which is applied through the interface of the memory chip. These vectors need additional space inside the chip. These vectors are usually generated manually and not using an Automatic Test Pattern Generation (ATPG) tool. Only a functional or operational pattern can be generated using a processor-based tester. This cannot possibly guarantee complete testing for all faults. The result of the test is in processor level registers and the output from the processor must
be brought out to the external interface and further should be analyzed and processed to extract information. At times direct access to the memory might not exist after the integration and the vectors are developed after the tape-out.

### 4.1.2 Direct Memory Access

In this test architecture, direct access to embedded memory arrays is allowed through pins on the IC package. The main drawback of this technique is that it requires a large number of pins to provide address, data, control signals, and modified support to direct testing. To provide these, it requires the memory bus architecture to be route intensively i.e. it should cover all the embedded memory. Unlike processor-based testing, the vectors can be generated by automatic test equipment using a memory test function.

### 4.1.3 Memory BIST

To improve the speed of testing and to make testing more localized, engineers decided to include the tester function inside the silicon. This is done by synthesizing address generator, sequence data generator, and checker inside the silicon memory. A BIST environment is the most efficient testing method for memory in aspect of cost and time. The main advantage of BIST is that it requires relatively fewer pin level interfaces. The basic pins for a memory BIST environment are reset, initiate, clock (optional), done and result. Once implemented the BIST algorithm it cannot be changed as it is synthesized. They are few programmable BIST, where you could change the implementation, but they are very complex in designing. A simpler programmable finite state machine (FSM) based BIST capable of transiting between various march tests was proposed in [? ]. This was implemented using multiple FSMs where a primary FSM chooses between the algorithm and secondary FSM implements the corresponding algorithm[? ]. The FSM could also
be implemented on to a programmable CPLD or FPGA and fabricated along with the memory. A FPGA based FSM would be area efficient compared to synthesizing multiple FSM[16]. The clock to drive the BIST could be either taken from outside through a pin or could be driven through an internal clock generated from a PLL. Just like JTAG, the BIST environment could be serialized to connect multiple embedded memories. There are advanced BIST environments such as built-in self-repair (BISR) where it detects the instance where the memory fails and provides a redundant recovery option for those addresses. All the characteristic of the BIST test environment must be stored in ROM and executed by a processor or the entire BIST should be synthesized and placed next to the memory[17].

4.2 Memory Testing Model

Memory circuits are different to logic circuits both in the context of testing and designing. The regularity in memory allows a dense packaging. Thus, making it more susceptible to silicon defects. Basic memory fault is that a memory i.e. a bit does not have the right value when observed. As a tester, it does not matter how or why a cell is faulty. But detecting the fault is of more significance. But in order to diagnose the issue, it’s critical to know the mechanism or exact reason for the occurrence of the fault. Thus basic memory testing involves writing a value into a location and subsequently trying to read the same value from the location. Failure in memories are categorized into infant faults, faults during the initial stages of usage and failures due to wear-out[18].

4.2.1 Memory Failure Modes

Memory module being uniform in nature, the faults can be identified by writing a value in order and reading them in another order. This is because memory has certain failure modes which
could be identified by emulating different sequences. Generally, memory is susceptible to the following failure modes:

1. Data Storage: Bit cell can not accept a data.

2. Data Retention: Bit cannot hold on to a data, i.e. leakage effect.

3. Data Delivery: Bit delivers a wrong data through sense lines when read or written.

4. Data Decode: Bit is read by the wrong sense line.

5. Data Recovery: Delay in accessing or sensing time.

6. Address Recovery: Delay in decoding time.

7. Address Decode: Error in decoding the address.

8. Bridging: Errors that occur when writing into one bit affects another bit (Coupling faults).

9. Linking: Freezing of one part of memory when another part is accessed.

### 4.3 Common Fault Models

Logic testing fault models are not enough to model memory testing faults. This is because of regularity in memory design. When memory cells are close to each other new defects occur due to the coupling effects. Most of the faults are result of a phenomenon called spot defects[19]. Spot defects occur in regions with excess materials unintentionally used during the process of fabrication and could be modeled electrically. Few of the most common fault models are described below:
4.3.1 **Stuck at Faults (SAF)**

Similar to logic stuck at faults, memory also is inherent to stuck at faults. A stuck at fault occurs when a bit is always one or zero irrespective of the input. This can be detected by reading a zero and one.

4.3.2 **Transition Fault (TF)**

The memory cell fails to switch from one to zero or vice-versa. This can be detected by subsequent reads of different value from the same address.

4.3.3 **Address Decoder Fault (AF)**

A functional fault in the address decoder unit of a memory results in address decoder faults. There are 4 categories of address decoder faults:

1. For a given address no memory location is accessed.
2. A memory location is never accessed.
3. For a given address multiple memory locations are accessed.
4. For a given memory location multiple addresses can access.

4.3.4 **Stuck at Open Fault (SOF)**

This fault occurs when a memory line cannot be accessed due to breakage in the word line. This forces the circuit to read previous read value again. This is a port fault.
4.3.5 Coupling Faults (CF)

Coupling faults are the faults that occur when the value in a cell is influenced by a read or write operation in another cell. Coupling faults are caused by the dense fabrication of memory cells. Cells which get affected are called the victim cells and the cells that cause the fault are called the aggressors. Coupling faults are very difficult to model as they are totally unpredictable and the aggressor cell could be anywhere. Coupling effect need not necessarily be with a continuous address as the memory could be oriented in a totally different order compared to the physical memory address. In case of word-based memory, the coupling fault could be both inter-word or intra-word fault.

4.3.5.1 Types of Coupling faults

1. State Coupling Fault: The state of the aggressor cell forces the value in the victim cell.

2. Inversion Coupling Fault: When there is a transition in the aggressor cell victim cell gets inverted.

3. Idempotent Coupling Fault: A transition forces a value in the victim cell.

4. Disturb Fault: A continuous operation (read/write) on the aggressor cell forces victim cell to be either zero or one.

5. Read Disturb Fault: This is a special case of disturb fault that affects even ROM. A continuous read on the aggressor cell will flip the value in the victim cell.

4.3.6 Data Retention faults (DRF)

Since memories are actually charge stored in capacitance, they are always vulnerable to discharge. There is a specific time guaranteed by a manufacturer for cell to be valid without manual
4.3 Common Fault Models

recharging. When a cell discharges before the specific time it is called Data Retention fault. Depending upon the memory type different Data retention faults are possible.

4.3.6.1 DRAM

DRAM has both refresh fault and leakage fault. DRAM requires continuous refreshing to retain value. It also needs refresh every time there is a read. If a cell fails before the designated refresh rate it is a DRF fault. DRAM is also susceptible to Leakage fault.

4.3.6.2 SRAM

SRAM is only vulnerable to Leakage fault. A SRAM is supposed to hold on to a charge until there is power without needing any refresh unit. But if there is a internal leakage or shorting there are chances that a cell can lose its information before the expected time.

4.3.7 Multi-Port Faults

Mult-port memory cells are modules which have more than one port to access the data. Multi-Port cells are susceptible to all the above single port faults. These single port faults can be on individual ports or also as coupled between multiple ports. For example, address decoder faults could be coupled for two ports. Address faults in decoder circuit of multi-port RAM is highly complex to model since all individual address decoder blocks are coupled together [7]. Multi-port RAM are susceptible to coupling faults which are only visible when ports are accesses simultaneous [9]. Additionally, they have other faults like inter-word and intra-word line short fault. Inter-port faults cannot be covered when any of memory ports are idle and also it requires the exact port and exact word line to be triggered to identify a fault [20].
4.4 RAM Test Algorithm

A test algorithm is a set of finite sequence to test various faults in a memory. The test algorithm basically controls the memory by providing the address, data, operation and other control signals. Most of these algorithms are called March algorithms as address are incremented or decremented one after the other, marching throughout the entire memory multiple times. In general test engineers come up with a proper functional model of a memory’s faults to derive the testing algorithm [8]. The main characteristic of a test algorithm is the time constraint and the fault models covered by the algorithm. Few algorithms are capable of detecting faults while few algorithm also locate the fault and are useful in repair. Most cases the algorithm is implemented using FSMs. Once a test algorithm is decided a algorithmic state machine is designed which is later translated into FSM [21]. For Word accessible memory 0 and 1 represents any two complimentary words. The most common Test algorithms are

4.4.1 ZERO-ONE Algorithm

This is the simplest Test algorithm. It has a complexity of $O(4N)$. It covers SAFs and few address decoder faults. The basic test plan is to write 0 in all the location and read 0. Immediately write 1 and read 1.

$\uparrow(w0); \uparrow(r0); \uparrow(w1); \uparrow(r1);$ 

4.4.2 Checkerboard Algorithm

This is similar to Zero-One algorithm but the value written is not constant and a checker board is used to compare the value. This can be used to detect few CFs, SAFs, AFs.
4.4.3 GALPAT

Galloping Pattern is the most complex and the strongest test pattern. It is of the time constraint of $O(4 \times N^2)$. It is used to detect all AFs, TFs, CFs and SAFs. 1. 0 is written in all address. 2. Compliment one by starting from address 0 to N-1. 3. Once an address location is complimented the address is read and then all other cell other than that cell is read. 4. Write 1 in all cells. 5. Repeat Step 2.

4.4.4 WALPAT

It is similar to GALPAT but the in the step 3 of GALPAT it reads the address only after reading all other location thus taking the half the time of GALPAT. The time constraint of WALPAT is $O(2 \times N^2)$.

4.4.5 MATS

Modified Algorithmic Test Sequences are used to find OR-type or AND-type address faults. Address faults might occur when two address are coupled as AND or OR type.

4.4.5.1 OR-type Faults

In OR-type faults the two coupled addresses result in high logic if either of the address is written high. It is identified by reading a zero and writing a one in every address. If any of the address acts as an aggressor, then the subsequent read zero fails for the victim cell.

\[ \uparrow(W0); \downarrow(R0,W1); \uparrow(R1) \]
4.4.5.2 AND-TYPE Faults

In AND-type faults the two coupled addresses result in high logic if and only if both the address is written high. It is identified by reading a one and writing a zero in every address. If any of the address acts as an aggressor, then the subsequent read one fails for the victim cell.

\[
\uparrow(W1); \uparrow(R1,W0); \uparrow(R0)
\]

4.4.6 MATS+

Used for both OR and AND type faults.

\[
\downarrow(W0); \uparrow(R0,W1); \downarrow(R1,W0)
\]

4.4.7 MATS ++

This takes care of TFs along with AFs.

\[
\{\uparrow(W0); \uparrow(R0,W1); \downarrow(R1,W0,R0)\}
\]

4.4.8 March-1/0

It is a complete test and covers most of the faults. But it is not highly efficient as there are redundant stages.

\[
\uparrow(W0); \uparrow(R0,W1,R1); \downarrow(R1,W0,R0); \uparrow(W1); \uparrow(R1,W0,R0); \downarrow(R0,W1,R1)
\]

4.4.9 MARCH C

For most of AFs, TFs, SAFs and CFs.

\[
\downarrow(W0)\uparrow(R0,W1); \downarrow(R1,W0); \downarrow(R0)\downarrow(R0,W1); \downarrow(R1,W0); \uparrow(R0)
\]
4.4.10 MARCH Extended March C-

Reduced redundancy compared to MARCH C but also covers SOF.

\[ \uparrow(W0)\uparrow(R0,W1,W1);\uparrow(R1,W0)\downarrow(R0)\downarrow(R0,W1);\downarrow(R1,W0);\uparrow(R0) \]
Chapter 5

DUT and BIST Environment

5.1 DUT Introduction

The given memory model is FSC0H_D_SJ manufactured by Faraday Technology Corp. It is a Synchronous Dual Port SRAM. It is capable of synthesizing on UMC’s 0.13 um 1P8M library which could be powered on 1.2 V. It works on high-speed CMOS process. It allows various combinations of words, bits and aspect ratios. The complete module is re-configurable to adapt different size, word size, and timing constraint. The valiant features of the modules are

1. Read and write operation are synchronous
2. The layout is completely customizable.
3. Supports voltage between 1.08 ~ 1.32 V.
4. Power efficient: Turns Power downs to eliminate DC discharge
5. All the inputs are clocked with respect to the corresponding port’s clock input
6. Supports duty cycle less than 50 percent, provided setup and hold time are not violated.
7. Supports both Byte write or Word write.
5.1 DUT Introduction

Table 5.1: RAM Input/Output

<table>
<thead>
<tr>
<th>PIN Name</th>
<th>Direction</th>
<th>Size</th>
<th>Capacitance</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Input</td>
<td>[6:0]</td>
<td>0.018 pF</td>
<td>Port A address</td>
</tr>
<tr>
<td>CKA</td>
<td>Input</td>
<td>1</td>
<td>0.082 pF</td>
<td>Port A clock</td>
</tr>
<tr>
<td>CSA</td>
<td>Input</td>
<td>1</td>
<td>0.025 pF</td>
<td>Port A chip select</td>
</tr>
<tr>
<td>OEA</td>
<td>Input</td>
<td>1</td>
<td>0.017 pF</td>
<td>Port A output enable</td>
</tr>
<tr>
<td>WEAN</td>
<td>Input</td>
<td>[3:0]</td>
<td>0.075 pF</td>
<td>Port A write enable (active low)</td>
</tr>
<tr>
<td>DIA</td>
<td>Input</td>
<td>1</td>
<td>0.010 pF</td>
<td>Port A input data</td>
</tr>
<tr>
<td>DOA</td>
<td>Output</td>
<td>[31:0]</td>
<td>0.013 pF</td>
<td>Port A output data</td>
</tr>
<tr>
<td>B</td>
<td>Input</td>
<td>[6:0]</td>
<td>0.018 pF</td>
<td>Port B address</td>
</tr>
<tr>
<td>CKB</td>
<td>Input</td>
<td>1</td>
<td>0.082 pF</td>
<td>Port B clock</td>
</tr>
<tr>
<td>CSB</td>
<td>Input</td>
<td>1</td>
<td>0.025 pF</td>
<td>Port B chip select</td>
</tr>
<tr>
<td>OEB</td>
<td>Input</td>
<td>1</td>
<td>0.017 pF</td>
<td>Port B output enable</td>
</tr>
<tr>
<td>WEBN</td>
<td>Input</td>
<td>[3:0]</td>
<td>0.075 pF</td>
<td>Port B write enable (active low)</td>
</tr>
<tr>
<td>DIB</td>
<td>Input</td>
<td>1</td>
<td>0.010 pF</td>
<td>Port B input data</td>
</tr>
<tr>
<td>DOB</td>
<td>Output</td>
<td>[31:0]</td>
<td>0.013 pF</td>
<td>Port B output data</td>
</tr>
</tbody>
</table>

Figure 5.1: DUT Pin Diagram

With respect to the project, the DUT SDPRAM is of 128*8*4 bit memory i.e. 128 locations with word size of 32 bit.
5.2 Memory BIST

Dual Ported memory is difficult to test as there are additional faults to be covered compared to a single port. Most of the existing March algorithms for dual port RAM, apply march algorithm on independent ports. There are also attempts to test a dual port ram through writing in one port and reading in another port. But these are insufficient to check the complicated dual port designs, as there are compound coupling effect in the multi-port embedded memory designs. Thus special marching algorithms supporting concurrent operations are designed to test efficiently multi-port ram operations. A primary issue in concurrency test is to know the limitation of the design. A test engineer should know what level of concurrency is supported by the module. These limitations include concurrent writes, concurrent reads, and simultaneous read and write to the same address. The module tested in this paper supports only concurrent reads for the same address. Thus, the algorithm implemented does not try to read or write when other port writes to the same address. Another critical consideration is the effect of clock domain crossing. Usage of different clocks can result in meta-stability, and this could be falsely diagnosed as a fabrication fault. A basic BIST environment tries to run both the clocks at the same frequency and phase, so that clock domain crossing does not affect the testing. This is achieved by using multiplexers to choose the same clock for both the ports during testing. Complex coupling effects might be discovered when operating at different frequency but for simplicity purpose, the BIST implemented runs in only single frequency. The coupling effects are the most difficult faults to be identified and this BIST algorithm tests them by perturbing multitude of operations and arbitrating the address pair on the RAM. The testing environment includes conditions such as:

1. Port A and Port B writing to different address.

2. Port A and Port B reading from different address.
3. Port A and Port B reading from same address.

4. Port A writing and Port B reading from different address.

5. Port A reading and Port B writing from different address.

6. Also when a port is idle, it should not affect the operation of other port.

### 5.2.1 Hardware Resource.

The BIST environment consists of the following hardware elements.

![BIST Architecture](image)

**Figure 5.2: BIST Architecture**

### 5.2.1.1 Multiplexers:

The multiplexers are used to choose between the external inputs of the RAM module and the BIST nets connecting to the RAM. The select signal to the Multiplexers are provided by the BIST control signals.
5.2 Memory BIST

5.2.1.2 Counter:

This counter is used to increment or decrement the address for marching purpose. The control signals to counter are sent through the BIST state machine. Usage of a linear feedback shift register would be a more efficient address generator but for simplicity purpose an add/sub counter is used in the paper. The counter resets only if the state machine requests it to do so or if it has reached the set target given by the state machine. An area efficient programmable BIST was implemented by using shared address generator and BIST controller in [6].

5.2.1.3 Address Selector:

This block is a simple mux that helps in choosing the exact address to be translated to the individual address pins of ports. This helps us to use just single counter for both the ports. The control signal to this address selector is provided by the state machine.
5.2 Memory BIST

5.2.1.4 Checker Block:

The only role of checker block is to compare the output of the data out port of the ram module with the compare word generated by the state machine and return the result to state machine. Enabling other control signals to this is also generated by the state machine. It is also the role of the comparator to report the address and the port information of where exactly the fault has occurred.

5.2.1.5 State Machine:

This is the main control part of the BIST environment. It has 23 states in it and instructs the other units of the BIST to perform their task. The state machine is also responsible to set the exact control signals to the RAM module for the required operation. The state transition occurs based on the counter value generated or error value generator by the counter and checker block respectively.

5.2.2 The Test algorithm.

This march testing algorithm tries to exploits the concurrency in operation by marching through each half of the addresses through two different ports thereby taking half execution time of complete marching. The address selector bus helps us to make sure there is no write collision. Address selector either inverts all the address bits between the two ports or inverts the most significant bit on the two ports, as per the control signal from the BIST state machine. Since the half address concept can make the testing faster, it should be noted that the individual ports must see Read 0 Write 1 followed by Read 1 Write 0 and vice versa for the entire address range for both directions. The 23 states ensure this and thus makes sure that the memory is tested for most of the faults in the given linear time. The state S1 writes the default constant word A in
all the ports through the two ports. The transition to next state occurs when the counter reaches the half-max value. In next state, it reads the value from all the addresses and compares with constant word A value and then writes the value B in all the ports. This process is repeated for multiple times in different directions. The states are designed to move to the Error_state and report the error in case a fault is detected. The main aspect of this BIST environment is that it tests the RAM by providing six different combinations of values and also at the end of the test, resets the memory to zero for easier operation of the memory. After the successful marching of all the 23 states, the BIST reports a success to the external interface.

Figure 5.4: BIST State Machine
Chapter 6

Verification Concepts

6.1 SystemVerilog

Since the introduction of SystemVerilog (SV), SV has been the preferred language for verification engineers. SV has provided a perfect verification environment by inheriting features from other existing verification languages. SystemVerilog has adapted the powerful OOPS concept from JAVA language and has eased the work of verification engineers. The first version of SV was introduced in the year 1983 as an addition to the existing Verilog language reference manual and then later an updated version was released in the year 2012 [IEEE 1800 Standard].

6.1.1 SystemVerilog Components

6.1.1.1 SystemVerilog for Design:

This is the synthesizable subset of the SystemVerilog language. Most constructs are similar to Verilog but a lot of additional features were added to SV. 4 state Logic and 2 state bit data type were introduced compared to the traditional reg data type in Verilog. Designers were now
able to control the synthesis of their code by specifying manually whether an always block is combinational or flip-flop, or latch. Though this added a lot visibility on the RTL designers side, design compilers are not completely successful in designing the intended specification.

6.1.1.2 System Verilog for Verification (Test Benches):

Introduction of Object Oriented Programming into verification has made it possible for engineers to layer their test bench and reuse components. SV only supports overriding and not overloading. Like JAVA, SV doesn’t support multiple inheritance. Inheritance, data abstraction and run time-overiding has made way for many methodologies such as OVM and UVM. These methodologies have abstracted the view of verification engineers and eased their work. Use of mailboxes, semaphores and event triggers have helped engineers synchronize despite having multiple threads running on the layered components. SV also has dynamic data structures such as queues, FIFOs, mailboxes and associative arrays to help in verification.

6.1.1.3 SystemVerilog Application Programming Interface:

These are additional features of SV that helps in integrating coverage and assertion with external application.

6.1.1.4 SystemVerilog for DPI:

This is similar to Verilog’s Programming Language Interface(PLI). But the Direct Programming Interface (DPI) is more sophisticated and has made easy integration of external languages such as C, C++, and more in SV. Most of the initial architectural design are programmed in C or SystemC. With DPI, engineers are able to use the C interface as a reference model for checking correctness of the design.
6.1.1.5 SystemVerilog Assertions:

Assertions are a huge subset of SV library and most of them are directly adapted from languages like Vera and SPEC. Assertions are basically categorized into immediate and concurrent assertion. Immediate assertion are used for verifying properties that happen in an instant, while concurrent assertions are used to assert properties that overlaps few cycles. SV provides a dynamic environment for assertion by turning them on or off selectively during the run time. This features has enabled design engineers to use assertion in the RTL code during the initial design phase that is later turned off. This is to ensure those assertions do not cost simulation cycles. A small subset of assertion are synthesizable and are used for emulation purpose [22].

6.1.2 Basic Test Environment

Figure 6.1: Basic Test Environment[1]
The main advantage of layered test bench is that it has allowed engineers to work independently and also simplified verification by allowing re-usability. The basic components of layered test bench are

6.1.2.1 Driver and Monitor:

Driver and monitor are the actual components that communicates with DUT. Driver is used to drive the inputs while the monitor monitors the outputs.

6.1.2.2 Agent

Agent internally consist of driver, monitor and sequencer in case of UVM. In simple SV test-benches, agents are considered as the sequence generator specific to the given DUT of bus functional model. The agents provide the sequence or number of transaction packets to both driver and scoreboard. The former drives the DUT while the later feeds it to model to predict the expected outcome.

6.1.2.3 Scoreboard and Checker

Scoreboard gets the transactions from agent and monitor. It has an internal reference model which predicts the expected output and the checker module verifies them.

6.1.2.4 Generator

The generator consists of the basic stimulus data and generates the actual packets that are later sequenced and packed by agent. The stimulus generator is governed by strict constraints.
6.1.2.5 Environment

The environment holds the entire test components. The environment instantiates other components and supports by providing the interface and other connection between the various test components.

6.1.2.6 Test

The test is the actual top module for verification. It instantiates the environment and other blocks.

6.1.2.7 Assertion

Assertion are written to ensure known properties of the DUT not violated. It is similar to software assertion found in languages like python. The assertion continuously observes every simulation cycle for correctness of asserted property.

6.1.2.8 Coverage

Function coverage is the fundamental goal of a constraint driven random testing. It is almost impossible to test a design completely for all the possible combination of inputs in the given allocated time for testing [23]. For simpler designs formal verification techniques were used to verify the functional aspect of the design to reduce the duration of verification [24]. But the mathematical equivalence of a complex design was difficult to model and also the formal verification was not able to verify the performance and the timing correctness of a design. This has forced verification engineers to come up with the function coverage [23]. The coverage blocks samples the random test variables at specified intervals and matches the bins. Finally, it reports if all coverpoints are covered. SV allows special features like cross coverage where bins are created for mutual intersection of two coverpoint bins.
6.2 UVM

Though SV was used as standard verification language, a new problem was created in industry with engineers implementing test environment arbitrarily. Although this enabled them to verify the individual module, it became difficult to reuse or integrate other modules to the existing test environment. For verifying large designs that have a lot of modules instantiated, an upgrade to the existing common techniques is needed. The basic test structure of SV shows all the modules are hardwired to each other and this reduces re-usability [25]. This is when UVM comes into picture. UVM is used by many IT industries as a standard verification methodology. It was developed by Accellera Systems Initiative for aiding the verification community [26]. It represents rapid advancements that enables the user to create reusable and robust verification IP and test bench components.

6.2.1 Basic Tenets of UVM:

6.2.1.1 Functionality encapsulation

UVM enables the user to compose an encapsulating functionality that extends a UVM defined block uvm_component. This block has a run task inbuilt that acts as an executable thread that helps to implement functionality of the DUT.

6.2.1.2 TLM (Transaction Level Modeling)

Various components of the testbench communicate to each other using TLM ports. This is possible because UVM defines various components in a standard way and can be modified as long as they are interfacing with the test bench architecture remaining the same. Ovm_sequence wrapper is used around a body().
6.2.1.3 Using sequences for stimulus generation

Transactions that run the DUT have to be generated by a class in the UVM environment. Letting a component generate these sequences would result in changing the component time and again to match the specification required.

6.2.1.4 Configuration

Having a configurable testbench empowers you to improve productivity of the testbench, and hence, it is a key element of a UVM test environment. The behavior of an already instantiated component can be changed by using a configurable API, factory overrides and callbacks.

6.2.1.5 Layering

Layering is a powerful way which takes care of the details that relate to specific layers. This layering can be applied to the UVM environment and is used to create hierarchy and compositions. Layering the stimulus is a very efficient way of reducing the complexity of generating the stimulus.

6.2.1.6 Re-usability

All the above points in this chapter reiterate reusability. Extending the contents of the classes to create more features, including configurability and layering is possible due to massive reuse of UVM classes.

6.2.2 UVM classes

UVM base class library supports three basic classes for building environment.
6.2.2.1 uvm_transaction.

It is the main base class for all transactions and are transient in nature. It extends to the base class uvm_object. Most of the simple transactions can be extended to uvm_transaction while advance sequences extend to uvm_sequence_item which in turn extends to uvm_transaction.

6.2.2.2 uvm_component

All the main component blocks of the UVM environment extends to uvm_component base class. Phases such as build, connect, report, config and others are called through function defined inside this base class.

6.2.2.3 uvm_object

uvm_object is the top most base class in UVM and all component and transaction extends to this class by default. All core transaction based task such deep copy, copy, create, etc. are defined inside the uvm_object.

6.2.3 UVM Phases

UVM execution is synchronized based on the phases. Unlike Verilog which has only 3 phases: build elaboration and run, UVM has number of phases to ease synchronization. The most common phases of UVM are

6.2.3.1 function void build_phase

The build phase is the first phase executed and it has the instantiation of all uvm_component objects declared inside the class. Instantiation in other phases are not possible.
6.2.3.2 function void connect_phase

In the connect phase, the instantiated object from the build phase are connected. For example, analysis export and import, subscriber port and FIFO ports are connected to one another classes through the connect phase.

6.2.3.3 function void end_of_elaboration

End of elaboration is used to change the configuration of test bench after the hierarchy is built. It is called after the Build phase and the connect phase.

6.2.3.4 task run_phase

This is the only task phase. All other phases are functions which implies they can’t take simulation cycles. All the actual executions are written in the run phase. It is important to know all the classes in the run phase are executed simultaneous as independent threads. The duration of run time of these run_phase tasks are decided by the function raise_object and drop_object.

6.2.3.5 report_phase

This phase is the final phase and is called when all the objection are dropped. UVM provides sophisticated inbuilt tasks for reporting information. They could be dynamically controlled through verbosity defined by UVM_ceiling.
Chapter 7

Test Methodology

A test-bench is an environment where a DUT can be verified for functional correctness. A basic test-bench should have an input stimulus generator, DUT output observer, and comparator to verify whether the intended output for corresponding input is received. For a very basic circuit, these three individual blocks are enough to verify a design. But as design gets complex, there are various condition to be tested and the testing also needs to be easily reconfigurable to a change in the implementation of the DUT. Basically testing could be done for the reset state of a DUT or could drive the DUT into a known state and then verify for a special scenario. Based on the test stimulus, testing is categorized into Directed Testing and Random testing. During Directed testing all the possible inputs are tested for a condition. This is a time intensive method. Random testing involves setting up a test plan and feeding random inputs until the test plan requirements are verified, In this verification plan, we use random testing with functional coverage on the inputs. Usually, the test is run until we get a 100% test coverage on the set coverpoints.
7.1 Test Environment

The environment is separated from the DUT using interface. The interface is the set of signals that are grouped together and are interfaced between the test environment and the DUT. Only the Driver and Monitor hold the pointer the interface. The components linked directly with the interface are DUT specific and are difficult to be reused with another DUT. A top-level outline of components are shown in Figure 7.1.

Figure 7.1: Test bench components
7.1 Test Environment

7.1.1 Sequence

The sequence or the transaction is a packet of stimulus data that is generated through randomization. All the interface signals need not be generated as few signals could be a function of the generated stimulus and can be emulated using a bus functional model.

7.1.1.1 Variables

1. opA: It is a 2-bit variable that decides the operation on Port A. 0 implies Idle, 1 implies reading, and 2 implies writing.

2. opB: It is a 2-bit variable that decides the operation on Port B. 0 implies Idle, 1 implies reading, and 2 implies writing.

3. datA: It is a 32-bit data that is written on write operation on Port A.

4. datB: It is a 32-bit data that is written on write operation on Port B.
5. wrA: It is a 4-bit write enable data for Port A.

6. weB: It is a 4-bit write enable data for Port B.

7. adrA: It is 7-bit variable that gives the write/read address for Port A.

8. adrB: It is a 7-bit variable that provides the write/read address for Port B

7.1.2 Constraints

There are two major constraints on the random generator

7.1.2.1 Operation

The operation variable for both port A and port B are 2-bit. But the possible operations are only 3: read, write, and idle respectively. So this constraint makes sure the variable opA and Op B doesn’t get a value other than the three.

7.1.2.2 Write Collision

As discussed earlier the SDPRAM doesn’t support concurrent operation on the same address other than reading simultaneously. Write_collision constraint makes sure when any one of the operation is write, then the address of the two port are not same.

7.1.3 Sequencer

The sequencer generates the sequences, stack them in order and then transmits them to the driver through analysis ports.
7.1.4 Driver

The main role of the driver is to drive the inputs on the DUT. The driver receives the transaction from the sequencer and based on the transaction information a bus function model performs the task of emulating all the inputs on the DUT interface.

The driver designed in this project has four tasks.

7.1.4.1 Reset_BIST

This task performs the operation of running the internal BIST environment of the DUT. By running the BIST the test could reset all the location in the memory to zero. In this task, the BIST_start pin is driven and the BIST_done signal is awaited.

7.1.4.2 Run_PortA

This task performs the PortA operation at the posedge of Port A Clock based on the opA, datA, wrA, and adrA variables from the transaction.

7.1.4.3 Run_PortB

This task performs the PortB operation at the posedge of Port B Clock based on the opB, datB, wrB, and adrB variables from the transaction.

7.1.4.4 Write task.

This task triggers the write block in the subscriber blocks. This is done using analysis ports. The scoreboard and the coverage blocks are connected to the driver through analysis port. Whenever write task is performed in the driver, individual write task in driver and scoreboard are triggered.
7.1.5 Monitor

The monitor samples the output of the RAM ports and sends them to the checker through the put_task provided by the TLM FIFO.

7.1.6 Agent

The agent has the driver monitor and the sequencer in it. The reason for categorizing them into a single block, is to separate the DUT dependent blocks from other blocks of the environment.

7.1.7 Scoreboard

The scoreboard receives the a transaction from the driver and performs the same task on the model. For the given memory DUT an associative array is used as model. The scoreboard also receives the output from the DUT and compares them with the expected output.

7.1.8 Environment

The environment instantiates all the other modules in build phase and also connect them in the connect phase.

7.2 Test Planning

Constraint random testing requires specific functional coverage goals set up. The test plan requires a good understanding of the DUT. Most of the simulators do not allow covering internal signals of the DUT. Apart from functional coverage, code coverage is automatically generated by the simulator.
7.2.1 Functional Coverage Plan

- Being a dual port RAM, all the three operations must be covered independently on both the ports.

- Also they should be crossed for seeing all the combination of operation on two ports.

- It's not critical to cover all the address ports in the memory. Thus 8 addresses are set for a single bin and also the extreme addresses are specified in separate bins.

- The data bus for each port is divided into 8 values for a bin and this coverpoint is crossed with the operation write and read for each individual port.
Chapter 8

Result and Discussion

The Synchronous Dual Port RAM was successfully tested using a BIST and verified using a UVM methodology in System Verilog.

8.1 BIST Results

The BIST starts executing when the BIST_TEST mode is selected and the start signal is asserted high. The completion of the BIST operation is implied by assertion of done signal. The result of BIST is implied by the result bit when the done signal is set. If the result bit is high the BIST test is successful and if the result bit is low the BIST has failed. The BIST test takes 1607 clock cycles for setting the done signal on the successful completion. The synthesis report of the BIST environment is tabulated below. The RTL code is synthesized using Synopsys Design Compiler.

8.1.1 Synthesis Report of BIST

The final design was synthesized on 180nm on TSMC library is tabulated in 8.1. The comparative results with TSMC 65nm library and SAED 32 nm are tabulated in 8.2. The timing report is
Table 8.1: Logic Synthesis report of BIST

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Pre-scan Netlist</th>
<th>Post-scan Netlist</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total cell area ($\mu m^2$)</td>
<td>86602.545931</td>
<td>89526.452209</td>
</tr>
<tr>
<td>Total no. of cells</td>
<td>1178</td>
<td>1159</td>
</tr>
<tr>
<td>Worst-case timing path</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Data required time (ns)</td>
<td>19.5153</td>
<td>19.4526</td>
</tr>
<tr>
<td>Data arrival time (ns)</td>
<td>-5.1775</td>
<td>-6.1963</td>
</tr>
<tr>
<td>slack (ns)</td>
<td>14.3378</td>
<td>13.2563</td>
</tr>
<tr>
<td>DFT coverage</td>
<td></td>
<td>87.41%</td>
</tr>
</tbody>
</table>

Table 8.2: Comparative Results on Synthesis of BIST using different Libraries

<table>
<thead>
<tr>
<th></th>
<th>32nm</th>
<th>65nm</th>
<th>180nm</th>
</tr>
</thead>
<tbody>
<tr>
<td>Area</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Combinational Area ($\mu m^2$)</td>
<td>2118.8</td>
<td>1447.20</td>
<td>12743</td>
</tr>
<tr>
<td>Non-Combinational Area ($\mu m^2$)</td>
<td>2143.9</td>
<td>2679.8</td>
<td>18434</td>
</tr>
<tr>
<td>Total Area ($\mu m^2$)</td>
<td>64416.76</td>
<td>62525.04</td>
<td>89526.45</td>
</tr>
<tr>
<td>Power</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Internal Power (W)</td>
<td>0.00668</td>
<td>0.000734</td>
<td>0.00813</td>
</tr>
<tr>
<td>Switching Power (W)</td>
<td>0.000450</td>
<td>0.0000597</td>
<td>0.0121</td>
</tr>
<tr>
<td>Leakage Power (W)</td>
<td>6.55E-04</td>
<td>2.32E-05</td>
<td>3.800E-06</td>
</tr>
<tr>
<td>Total Power (W)</td>
<td>1.37 e-3</td>
<td>8.17 e-4</td>
<td>0.0934</td>
</tr>
<tr>
<td>Timing</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Slack (ns)</td>
<td>15.90</td>
<td>18.70</td>
<td>13.2563</td>
</tr>
<tr>
<td>DFT Coverage (%)</td>
<td>88.73</td>
<td>85.41</td>
<td>87.41</td>
</tr>
<tr>
<td>Latency (BIST Test)</td>
<td>1607</td>
<td>1607</td>
<td>1607</td>
</tr>
</tbody>
</table>

tabulated in 8.3.

### 8.1.2 Simulation Result of BIST

The BIST is simulated using a normal Verilog test bench. The test bench sets up the basic legal values to the DUT input and provides the clock. The test bench also asserts the BIST_Mode signal and start signal to start the BIST mode operation. The test bench runs until the done signal is asserted. The simulation is done using Cadence NC Verilog simulator. The simulation of BIST is shown in 8.1.
### Table 8.3: Timing Analysis coverage of BIST

<table>
<thead>
<tr>
<th>Type of clock</th>
<th>Total</th>
<th>MET</th>
<th>V isolated</th>
</tr>
</thead>
<tbody>
<tr>
<td>Setup (41%)</td>
<td>754</td>
<td>444(59%)</td>
<td>0(0%)</td>
</tr>
<tr>
<td>Hold (41%)</td>
<td>754</td>
<td>444(59%)</td>
<td>0(0%)</td>
</tr>
<tr>
<td>Recovery(100%)</td>
<td>222</td>
<td>0(0%)</td>
<td>0(0%)</td>
</tr>
<tr>
<td>Min_period(100%)</td>
<td>2</td>
<td>0(0%)</td>
<td>0(0%)</td>
</tr>
<tr>
<td>Min_pulse_width(34%)</td>
<td>670</td>
<td>444(66%)</td>
<td>0(0%)</td>
</tr>
<tr>
<td>Out_setup(0%)</td>
<td>67</td>
<td>0(0%)</td>
<td>67(100%)</td>
</tr>
<tr>
<td>Out_hold(0%)</td>
<td>67</td>
<td>67(100%)</td>
<td>0(0%)</td>
</tr>
<tr>
<td>All check(42%)</td>
<td>2536</td>
<td>1399(55%)</td>
<td>67(3%)</td>
</tr>
</tbody>
</table>

![BIST Simulation](image)

Figure 8.1: BIST Simulation
8.2 BIST UVM_Testbench

The UVM Test bench is created using different components of UVM library and a layered test bench. The main goal was to randomize data and make sure the functional coverage bins were covered. The UVM test bench simulation is shown in 8.2. The Coverage on a run of 1000000 operation was 99.24 % and the detailed coverage report is shown in 8.3. The coverage clearly shows the individual statistics for each coverpoint and cross coverpoints.

![UVM Test Bench Simulation](image)

Figure 8.2: UVM Test Bench Simulation
Figure 8.3: Coverage Results
Chapter 9

Conclusion

An efficient BIST environment was successfully implemented for the given Synchronous Dual Port RAM. The BIST environment is parameterized for different block size and word size. The BIST has 23 states and it took 1607 cycles for a successful run. The initial state of designing BIST included understanding DFT concepts, MBIST algorithms and understanding the functioning of the module. An efficient approach for MBIST was selected and a finite state machine flow chart was created. The RTL coding started with non-synthesizable procedural coding and then complete code was separated into blocks and was written in synthesizable constructs. The BIST environment can be used for embedded memories by serializing the start and done signal.

The UVM test bench was capable of randomizing the operation and data. This ensured the memory module was verified for different addresses and different data. The test plan was set up based on constraint random testing. The testing achieved 99.4 % functional coverage and the testing was successful for all the runs. All the corner cases were set up as separate bin. Two assertions were proposed for asserting whether the done signal was received within 1607 cycles and asserting that the output matched with the expected output.
9.1 Future Work

- Complete capabilities of UVM were not explored in this project. The main advantage of UVM is re-usability. For most classes a base class library should be created and a working class module must extend the base rather than directly writing only single class.

- The complete test runs until an arbitrary number of data operation cycles are complete. This could be made dependent on the function coverage result by getting coverage result using get_coverage() function and stopping the simulation once functional coverage goals are achieved.

- The test performed in this project is not exhaustive. Neither BIST nor the Test Environment check for faults/errors when transitions occur on ports close enough to create clock crossing faults inside the RAM.

- The BIST environment is parameterized to adapt different word size and memory size. A Perl script could be implemented to generate BIST RTL code for memories based on the memory type, fault models to cover and the given time constraint. The RAMSES simulator to generate test function would be useful to implement this idea.

- The BIST covers exhaustive fault models, but not all the possible faults. There are ongoing researches in exploring faults and there is always a scope for development of a more efficient algorithm.

- The BIST could be extended to be implemented as BISR. BISR is more preferred for designs whose operations are critical and real time.

- The test engineers could use the current state of art technologies like Machine learning and deep learning to model the memory’s random faults.
References


Appendix I

Source Code

I.1 BIST RTL

```verilog
module SDPRAM (DOA, DOB, DIA, DIB, WEAN, WEBN, OEA, OEB, CSA, CSB, CKA, CKB, A, B, reset, clk, scan_in0, scan_en, start, test_mode, bist_mode, done, result, scan_out0);
```
I.1 BIST RTL

13
14 input
15    reset,                      // system reset
16    clk;                       // system clock
17
18 input
19    scan_in0,                  // test scan mode data input
20    scan_en,                   // test scan mode enable
21    test_mode,                 // test mode select
22    bist_mode;                 // bist mode test select
23
24 output
25    scan_out0;                 // test scan mode data output
26
27
28
29 parameter WSIZE = 32;
30 parameter MSIZE = 128;
31 parameter ASIZE = $clog2(MSIZE);
32 parameter WRSIZE = WSIZE/8;
33 parameter WRALL = {WRSIZE{1'b0}};
34 parameter WRNONE = {WRSIZE{1'b1}};
35
36
37 reg [4:0] state, nxt_state;
parameter S0=0, S1=1, S2=2, S3=3, S4=4, S5=5, S6=6, S7=7, S8=8, S9=9, S10=10, S11=11, S12=12, S13=13, S14=14, S15=15, S16=16, S17=17, S18=18, S19=19, S20=20, S21=21, S22=22, S23=23, S24=24, S25=25;

parameter DONE_STATE=26;

parameter ERROR_STATE=27;

parameter CONSA = {WSIZE / 2 { 2' b01 } } ;

parameter CONSB = {WSIZE / 2 { 2' b10 } } ;

parameter CONSC = {WSIZE / 8 { 8' b00001111 } } ;

parameter CONSD = {WSIZE / 8 { 8' b11110000 } } ;

parameter CONSE = {WSIZE / 16 { 16' hFFFF } } ;

parameter CONSF = {WSIZE / 16 { 16' h0000 } } ;

parameter ZERO = { ASIZE { 1' b0 } } ;

parameter UX= { WSIZE{ 1' bX } } ;

parameter UZ= { WSIZE{ 1' b0 } } ;

parameter MAX_SIZE={ ASIZE{ 1' b1 } } ;

parameter HALF_SIZE=MAX_SIZE>>1;

output [WSIZE – 1:0] DOA, DOB;

input [WSIZE – 1:0] DIA , DIB;

input [ASIZE – 1:0] A,B;
61 input OEA;
62 input OEB;
63 input [WRSIZE−1:0] WEAN;
64 input [WRSIZE−1:0] WEBN;
65 input CKA;
66 input CKB;
67 input CSA;
68 input CSB;
69 input start;
70 output reg done, result;
71
72
73 wire DOA0, DOA1, DOA2, DOA3, DOA4, DOA5, DOA6, DOA7, DOA8,
74 DOA9, DOA10, DOA11, DOA12, DOA13, DOA14, DOA15, DOA16, DOA17,
    DOA18,
75 DOA19, DOA20, DOA21, DOA22, DOA23, DOA24, DOA25, DOA26, DOA27,
    DOA28,
76 DOA29, DOA30, DOA31;
77 wire DOB0, DOB1, DOB2, DOB3, DOB4, DOB5, DOB6, DOB7, DOB8,
78 DOB9, DOB10, DOB11, DOB12, DOB13, DOB14, DOB15, DOB16, DOB17,
    DOB18,
79 DOB19, DOB20, DOB21, DOB22, DOB23, DOB24, DOB25, DOB26, DOB27,
    DOB28,
80 DOB29, DOB30, DOB31;
81 wire DIA0, DIA1, DIA2, DIA3, DIA4, DIA5, DIA6, DIA7, DIA8,
I.1 BIST RTL

82 DIA9, DIA10, DIA11, DIA12, DIA13, DIA14, DIA15, DIA16, DIA17, 
    DIA18, 
83 DIA19, DIA20, DIA21, DIA22, DIA23, DIA24, DIA25, DIA26, DIA27, 
    DIA28, 
84 DIA29, DIA30, DIA31; 
85 wire DIB0, DIB1, DIB2, DIB3, DIB4, DIB5, DIB6, DIB7, DIB8, 
86 DIB9, DIB10, DIB11, DIB12, DIB13, DIB14, DIB15, DIB16, DIB17, 
    DIB18, 
87 DIB19, DIB20, DIB21, DIB22, DIB23, DIB24, DIB25, DIB26, DIB27 
    , DIB28, 
88 DIB29, DIB30, DIB31; 
89  
90 wire A0, A1, A2, A3, A4, A5, A6; 
91 wire B0, B1, B2, B3, B4, B5, B6; 
92 wire WEAN0; 
93 wire WEAN1; 
94 wire WEAN2; 
95 wire WEAN3; 
96 wire WEBN0; 
97 wire WEBN1; 
98 wire WEBN2; 
99 wire WEBN3;
I.1 BIST RTL

103
104 wire [WSIZE-1:0] DOA_i, DOB_i;
105 wire [WSIZE-1:0] DIA_i, DIB_i;
106 wire [ASIZE-1:0] A_i, B_i;
107
108 wire [WRSIZE-1:0] WEAN_i;
109 wire [WRSIZE-1:0] WEBN_i;
110
111
112
113 reg [WSIZE-1:0] DIA_o, DIB_o, DIA_o1, DIB_o1;
114 wire [ASIZE-1:0] A_o, B_o;
115 reg OEA_o, OEA_o1;
116 reg OEB_o, OEB_o1;
117 reg [WRSIZE-1:0] WEAN_o, WEAN_o1;
118 reg [WRSIZE-1:0] WEBN_o, WEBN_o1;
119
120 reg CSA_o, CSA_o1;
121 reg CSB_o, CSB_o1;
122
123
124
125
126 SJHD130_128X8X4CM4 RAM(A0, A1, A2, A3, A4, A5, A6, B0, B1, B2, B3, B4, B5, B6, DOA0, DOA1,
DOA2, DOA3, DOA4, DOA5, DOA6, DOA7, DOA8, DOA9,
DOA10, DOA11, DOA12, DOA13, DOA14, DOA15,
DOA16,
DOA17, DOA18, DOA19, DOA20, DOA21, DOA22,
DOA23,
DOA24, DOA25, DOA26, DOA27, DOA28, DOA29,
DOA30,
DOA31, DOB0, DOB1, DOB2, DOB3, DOB4, DOB5, DOB6

DOB7, DOB8, DOB9, DOB10, DOB11, DOB12, DOB13,
DOB14, DOB15, DOB16, DOB17, DOB18, DOB19,
DOB20,
DOB21, DOB22, DOB23, DOB24, DOB25, DOB26,
DOB27,
DOB28, DOB29, DOB30, DOB31, DIA0, DIA1, DIA2,
DIA3, DIA4, DIA5, DIA6, DIA7, DIA8, DIA9, DIA10

DIA11, DIA12, DIA13, DIA14, DIA15, DIA16,
DIA17,
DIA18, DIA19, DIA20, DIA21, DIA22, DIA23,
DIA24,
DIA25, DIA26, DIA27, DIA28, DIA29, DIA30,
DIA31,
DIB0, DIB1, DIB2, DIB3, DIB4, DIB5, DIB6, DIB7,
DIB8, DIB9, DIB10, DIB11, DIB12, DIB13, DIB14,
DIB15, DIB16, DIB17, DIB18, DIB19, DIB20, DIB21, DIB22, DIB23, DIB24, DIB25, DIB26, DIB27, DIB28, DIB29, DIB30, DIB31, WEAN0, WEAN1, WEAN2, WEAN3, WEBN0, WEBN1, WEBN2, WEBN3, CKA_i, CKB_i, CSA_i, CSB_i, OEA_i, OEB_i) ;

assign DOA_i={DOA31, DOA30, DOA29, DOA28, DOA27, DOA26, DOA25, DOA24, DOA23, DOA22, DOA21, DOA20, DOA19, DOA18, DOA17, DOA16, DOA15, DOA14, DOA13, DOA12, DOA11, DOA10, DOA9, DOA8, DOA7, DOA6, DOA5, DOA4, DOA3, DOA2, DOA1, DOA0} ;

assign DOB_i={DOB31, DOB30, DOB29, DOB28, DOB27, DOB26, DOB25, DOB24, DOB23, DOB22, DOB21, DOB20, DOB19, DOB18, DOB17, DOB16, DOB15, DOB14, DOB13, DOB12, DOB11, DOB10, DOB9, DOB8, DOB7, DOB6, DOB5, DOB4, DOB3, DOB2, DOB1, DOB0} ;
I.1 BIST RTL

DOB22, DOB21, DOB20, DOB19, DOB18, DOB17, DOB16, DOB15, DOB14, DOB13,

DOB12, DOB11, DOB10, DOB9, DOB8, DOB7, DOB6, DOB5, DOB4, DOB3,

DOB2, DOB1, DOB0;

assign {DIA31, DIA30, DIA29, DIA28, DIA27, DIA26, DIA25, DIA24, DIA23, DIA22, DIA21, DIA20, DIA19, DIA18, DIA17, DIA16, DIA15, DIA14, DIA13, DIA12, DIA11, DIA10, DIA9, DIA8, DIA7, DIA6, DIA5, DIA4, DIA3, DIA2, DIA1, DIA0} = DIA_i;

assign {DIB31, DIB30, DIB29, DIB28, DIB27, DIB26, DIB25, DIB24, DIB23, DIB22, DIB21, DIB20, DIB19, DIB18, DIB17, DIB16, DIB15, DIB14, DIB13, DIB12, DIB11, DIB10, DIB9, DIB8, DIB7, DIB6, DIB5, DIB4, DIB3, DIB2, DIB1, DIB0} = DIB_i;

assign {A6, A5, A4, A3, A2, A1, A0} = A_i;

assign {B6, B5, B4, B3, B2, B1, B0} = B_i;

assign {WEAN3, WEAN2, WEAN1, WEAN0} = WEAN_i;

assign {WEBN3, WEBN2, WEBN1, WEBN0} = WEBN_i;

assign DOA = (test_mode)?DIA:(bist_mode)?UZ:DOA_i;
assign DOB=(test_mode)?DIB:(bist_mode)?UZ:DOB_i;

assign DIA_i=(bist_mode)?DIA_o:DIA;
assign DIB_i=(bist_mode)?DIB_o:DIB;

assign A_i=(bist_mode)?A_o:A;
assign B_i=(bist_mode)?B_o:B;
assign OEA_i=(bist_mode)?OEA_o:OEA;
assign OEB_i=(bist_mode)?OEB_o:OEB;
assign WEAN_i=(bist_mode)?WEAN_o:WEAN;
assign WEBN_i=(bist_mode)?WEBN_o:WEBN;
assign CKA_i=(bist_mode)?clk:CKA;
assign CKB_i=(bist_mode)?clk:CKB;
assign CSA_i=(bist_mode)?CSA_o:CSA;
assign CSB_i=(bist_mode)?CSB_o:CSB;

reg [ASIZE−1:0] Counter, Counter1;
reg [1:0] operation;
reg [ASIZE−1:0] O_Reset_Value;
reg [1:0] A_Address_Generator, B_Address_Generator;
assign A_o = (A_Address_Generator==2’b00)? Counter1:
       (A_Address_Generator==2’b01)? {Counter1[6],~Counter1[5:0]}:
       (A_Address_Generator==2’b10)? {!Counter1[6],Counter1[5:0]}:
       (A_Address_Generator==2’b11)?~Counter1:Counter1;

assign B_o = (B_Address_Generator==2’b00)? Counter1:
       (B_Address_Generator==2’b01)? {Counter1[ASIZE-1],~Counter1[(ASIZE-2):0]}:
       (B_Address_Generator==2’b10)? {!((Counter1[ASIZE-1])),Counter1[(ASIZE-2):0]}:
       (B_Address_Generator==2’b11)?~Counter1:Counter1;

/*
always@ (A_Address_Generator)
begin
  if (reset)
    A_o<=ZERO;
  else if (bist_mode)

I.1 BIST RTL

225 begin
226 case :
227 */
228
229 always@ (posedge clk or posedge reset) begin
230 begin
231 if (reset ) begin
232 begin
233 Counter <=ZERO;
234 Counter1 <=ZERO;
235 DIA_o<=0;
236 DIB_o<=0;
237 OEA_o<=0;
238 OEB_o<=0;
239 WEAN_o<=WRNONE;
240 WEBN_o<=WRNONE;
241 CSA_o<=0;
242 CSB_o<=0;
243 end
244
245 else if (bist_mode )
246 begin
247 begin
248 Counter1 <=Counter;
249
case (operation)
    0: Counter <= Counter;
    1: Counter <= Counter +1;
    2: Counter <= Counter -1;
    3: Counter <= O_Reset_Value;
    default: Counter <= Counter;
endcase

DIA_o<=DIA_o1;
DIB_o<=DIB_o1;
OEA_o<=OEA_o1;
OEB_o<=OEB_o1;
WEAN_o<=WEAN_o1;
WEBN_o<=WEBN_o1;
CSA_o<=CSA_o1;
CSB_o<=CSB_o1;
end
else
begin
    DIA_o<=DIA_o;
    DIB_o<=DIB_o;
    OEA_o<=OEA_o;
    OEB_o<=OEB_o;
    WEAN_o<=WEAN_o;
    WEBN_o<=WEBN_o;
    CSA_o<=CSA_o;
CSB_o <= CSB_o;
Counter <= Counter;
Counter1 <= Counter1; end
end

reg [ASIZE-1:0] AADR1, BADR1;
reg Error_Reg1, Error_Reg2;
reg [ASIZE:0] Error_ADR1, Error_ADR2;
reg chk_A, chk_B, chk_A1, chk_B1, chk_A2, chk_B2;
always@(posedge clk or posedge reset)
begin
if (reset)
begin
AADR1 <= ZERO;
BADR1 <= ZERO;
chk_A1 <= 0;
chk_B1 <= 0;
chk_A2 <= 0;
chk_B2 <= 0;
I.1 BIST RTL

299     Error_Reg1 <= 1'b0;
300     Error_ADR2 <= {1'b0, ZERO};
301     Error_Reg2 <= 1'b0;
302     Error_ADR1 <= {1'b0, ZERO};
303     Exp_OP_B1 <= 0;
304     Exp_OP_A1 <= 0;
305     Exp_OP_B2 <= 0;
306     Exp_OP_A2 <= 0;
307         end
308     else if (bist_mode)
309         begin
310             AADR1 <= A_o;
311             chk_A1 <= chk_A;
312             chk_B1 <= chk_B;
313             chk_A2 <= chk_A1;
314             chk_B2 <= chk_B1;
315             BADR1 <= B_o;
316             Exp_OP_B1 <= Exp_OP_B;
317             Exp_OP_A1 <= Exp_OP_A;
318
319             Exp_OP_B2 <= Exp_OP_B1;
320             Exp_OP_A2 <= Exp_OP_A1;
321
322             if (chk_A2 && !Error_Reg1)
323                 begin
if (DOA_i\neq\text{Exp\_OP\_A2})

\begin{verbatim}
begin
  // $display("A %d Error found %h Expected @ %h 
  \quad but we got %h ", state, Exp\_OP\_A2, AADR1, DOA_i 
  );
  Error\_Reg1 <=1'b1;
  Error\_ADR1 <=\{1'b0, AADR1\}; end
  else
    begin Error\_Reg1 <=Error\_Reg1;
      Error\_ADR1 <=Error\_ADR1; end
end
else
  begin
    Error\_Reg1 <=Error\_Reg1;
    Error\_ADR1 <=Error\_ADR1; end

if (chk\_B2 && !Error\_Reg2)

begin
  if (DOB_i\neq\text{Exp\_OP\_B2})

  // $display("B %d Error found %h Expected @ %h 
  \quad but we got %h ", state, Exp\_OP\_B2, BADR1, 
  DOB_i);
  begin Error\_Reg2 <=1'b1;
    Error\_ADR2 <=\{1'b0, BADR1\}; end
  else begin

\end{verbatim}
Error_Reg2 <= Error_Reg2;
Error_ADR2 <= Error_ADR2;
end
else begin
  Error_Reg2 <= Error_Reg2;
  Error_ADR2 <= Error_ADR2;
end end
else begin
  Error_Reg2 <= Error_Reg2;
  Error_ADR2 <= Error_ADR2;
  Error_Reg1 <= Error_Reg1;
  Error_ADR1 <= Error_ADR1;
end end
end

always@(posedge clk or posedge reset)
begin
  if (reset)
    state <= S0;
  else if (bist_mode)
begin
  if ( Error_Reg1 || Error_Reg2 )
    state <= ERROR_STATE;
  else
    state <= nxt_state;
  end
else
  state <= state;
end

always@(*)
begin
  if ( bist_mode )
    begin
      case ( state )
      S0: begin
if (start == 1) begin
  nxt_state=S1;
  operation=2'b11;
end
else
begin
  
nxt_state=S0;
  operation=2'b00;
end

end

begin
  
A_Address_Generator=2'b00;
  B_Address_Generator=2'b10;
  DIA_o1=CONSA;
  DIB_o1=CONSA;
  CSA_o1 =1;
  CSB_o1 =1;
  OEA_o1 =0;
  OEB_o1 =0;
  Exp_OP_A =0;
  Exp_OP_B =0;
  chk_A =0;
  chk_B =0;
  done =0;
  result =0;
  WEAN_o1=WRALL;
  WEBN_o1=WRALL;
  O_Reset_Value=0;
if (Counter==HALF_SIZE)
begin
   nxt_state=S2;
   operation=2'b11;
end
else
begin
   nxt_state=S1;
   operation=2'b01;
end

S2:
begin

   A_Address_Generator=2'b00;
   B_Address_Generator=2'b10;
   OEA_o1=1;
   OEB_o1=1;
   CSA_o1=1;
end
CSB_o1 = 1;
WEAN_o1 = WRNONE;
WEBN_o1 = WRNONE;
Exp_OP_A = CONSA;
Exp_OP_B = CONSA;
chk_A = 1;
chk_B = 1;
done = 0;
result = 0;
O_Reset_Value = ZERO;
DIA_o1 = CONSA;
DIB_o1 = CONSA;
if (Counter == HALF_SIZE)
begin
    nxt_state = S3;
    operation = 2'b11;
end
else
begin
    nxt_state = S2;
    operation = 2'b01; end
end
S3:

begin

A_Address_Generator=2'b00;
B_Address_Generator=2'b10;
DIA_o1=CONSB;
DIB_o1=CONSB;
OEA_o1=0;
OEB_o1=0;
WEAN_o1=WRALL;
WEBN_o1=WRALL;
CSA_o1=1;
CSB_o1=1;
Exp_OP_A=0;
Exp_OP_B=0;
chk_A=0;
chk_B=0;
result=0;
O_Reset_Value=0;
done=0;

if (Counter==HALF_SIZE)
begin

nxt_state=S4;
operation=2'b11;

end
end

else

begin

operation = 2'b01;
nxt_state = S3;

end

end

S4:

begin

DIA_o1 = 0;
DIB_o1 = 0;
A_Address_Generator = 2'b00;
B_Address_Generator = 2'b10;
OEA_o1 = 1;
OEB_o1 = 1;
WEAN_o1 = WRNONE;
WEBN_o1 = WRNONE;
CSA_o1 = 1;
CSB_o1 = 1;
Exp_OP_A = CONSB;
Exp_OP_B = CONSB;
O_Reset_Value = 0;
result = 0;
done = 0;
if (Counter == HALF_SIZE)
    begin
        chk_A = 0;
        chk_B = 0;
        nxt_state = S5;
        operation = 2'b11;
    end
else begin
    operation = 2'b01;
    chk_A = 1;
    chk_B = 1;
    nxt_state = S4;
end
end
S5:
begin
    operation = 2'b00;
    A_Address_Generator = 2'b00;
    B_Address_Generator = 2'b10;
OEA_o1 = 1;
OEB_o1 = 1;
WEAN_o1 = WRALL;
WEBN_o1 = WRALL;
CSA_o1 = 1;
CSB_o1 = 1;
Exp_OP_A = CONSB;
Exp_OP_B = CONSB;
DIA_o1 = CONSC;
DIB_o1 = CONSC;
chk_A = 0;
chk_B = 0;
result = 0;
done = 0;
nxt_state = S6;
O_Reset_Value = 0;

end

S6:

begin

operation = 2'b00;
A_Address_Generator = 2'b00;
B_Address_Generator = 2'b10;
OEA_o1 = 1;
OEB_o1 = 1;
WEAN_o1 = WRNONE;
WEBN_o1 = WRNONE;
CSA_o1 = 1;
CSB_o1 = 1;
Exp_OP_A = CONSC;
Exp_OP_B = CONSC;
chk_A = 1;
chk_B = 1;
DIA_o1 = CONSC;
DIB_o1 = CONSC;
nxt_state = S7;
result = 0;
done = 0;
O_Reset_Value = 0;

end
S7:
begin
A_Address_Generator = 2'b00;
B_Address_Generator = 2'b10;
result = 0;
done = 0;
OEA_o1 = 1;
OEB_o1 = 1;
WEAN_o1 = WRALL;
WEBN_o1=WRALL;
DIA_o1=CONSD;
DIB_o1=CONSD;
CSA_o1=1;
CSB_o1=1;
Exp_OP_A=CONSB;
Exp_OP_B=CONSB;
chk_A=0;
chk_B=0;
O_Reset_Value=0;
if (Counter==HALF_SIZE)
begin
  nxt_state=S8;
  operation=2'b11;
end
else
begin
  operation=2'b01;
  nxt_state=S5;
end
end
S8:

begin

operation = 2'b00;
A_Address_Generator = 2'b00;
B_Address_Generator = 2'b10;
OEA_o1 = 1;
OEB_o1 = 1;
WEAN_o1 = WRNONE;
WEBN_o1 = WRNONE;
CSA_o1 = 1;
CSB_o1 = 1;
Exp_OP_A = CONSD;
Exp_OP_B = CONSD;
chk_A = 1;
chk_B = 1;
nxt_state = S9;
result = 0;
done = 0;
DIA_o1 = 0;
DIB_o1 = 0;
O_Reset_Value = 0;
670       end
671
672
673       S9:
674       begin
675           operation = 2'b00;
676           A_Address_Generator = 2'b00;
677           B_Address_Generator = 2'b10;
678           OEA_o1 = 1;
679           OEB_o1 = 1;
680           WEAN_o1 = WRALL;
681           WEBN_o1 = WRALL;
682           CSA_o1 = 1;
683           CSB_o1 = 1;
684           Exp_OP_A = CONSB;
685           Exp_OP_B = CONSB;
686           chk_A = 0;
687           chk_B = 0;
688           DIA_o1 = CONSE;
689           DIB_o1 = CONSE;
690           nxt_state = S10;
691           result = 0;
692           done = 0;
693           O_Reset_Value = 0;
694       end
S10:

begin

A_Address_Generator=2'b00;
B_Address_Generator=2'b10;
OEA_o1=1;
OEB_o1=1;
WEAN_o1=WRNONE;
WEBN_o1=WRNONE;
CSA_o1=1;
CSB_o1=1;
Exp_OP_A=CONSE;
Exp_OP_B=CONSE;
O_Reset_Value=MAX_SIZE;
result=0;
done=0;
DIA_o1=0;
DIB_o1=0;
if (Counter==HALF_SIZE)

begin

nxt_state=S11;
operation=2'b11;
chk_A=0;
chk_B=0;
720
721        end
722    else
723    begin  operation = 2'b01;
724
725
726        chk_A = 1;
727        chk_B = 1;
728        nxt_state = S8;
729    end
730    end
731
732
733
734    S11:
735
736    begin
737
738        A_Address_Generator = 2'b00;
739        B_Address_Generator = 2'b10;
740        DIA_o1 = CONSA;
741        DIB_o1 = CONSB;
742        CSA_o1 = 1;
743        CSB_o1 = 1;
744        OEA_o1 = 0;
745     OEB_o1=0;
746     Exp_OP_A=0;
747     Exp_OP_B=0;
748     chk_A=0;
749     chk_B=0;
750     O_Reset_Value=MAX_SIZE;
751     result=0;
752     done=0;
753     if (Counter==HALF_SIZE)
754       begin
755         nxt_state=S12;
756         operation=2'b11;
757         WEAN_o1=WRNONE;
758         WEBN_o1=WRNONE;
759
760       end
761     else
762       begin
763         nxt_state=S11;
764         WEAN_o1=WRALL;
765         WEBN_o1=WRALL;
766         operation=2'b10;
767
768       end
769     end
S12:
begin

A_Address_Generator=2'b00;
B_Address_Generator=2'b10;
OEA_o1=1;
OEB_o1=1;
CSA_o1=1;
CSB_o1=1;
WEAN_o1=WRNONE;
WEBN_o1=WRNONE;
Exp_OP_A=CONSA;
Exp_OP_B=CONSB;
DIA_o1=0;
DIB_o1=0;
result=0;
done=0;
O_Reset_Value=HALF_SIZE;
if (Counter==HALF_SIZE)
begin  chk_A=0;
chk_B=0;
nxt_state=S13;
operation=2'b11;
chk_A=0;
chk_B=0;end
else
    begin  nxt_state=S12;
        chk_A=1;
        chk_B=1;
        operation=2'b10;
    end
end
e1

S13:

begin
    A_Address_Generator=2'b00;
    B_Address_Generator=2'b10;
    DIA_o1=CONSC;
    DIB_o1=CONSD;
    CSA_o1=1;
    CSB_o1=1;
OEA_o1 = 0;
OEB_o1 = 0;
result = 0;
done = 0;
Exp_OP_A = 0;
Exp_OP_B = 0;
chk_A = 0;
chk_B = 0;
O_Reset_Value = HALF_SIZE;

if (Counter == 7'h00)
begin
  nxt_state = S14;
  operation = 2'b11;
end
else
begin
  nxt_state = S13;
  WEAN_o1 = WRONE;
  WEBN_o1 = WRONE;
end
end

WEAN_o1 = WRALL;
WEBN_o1 = WRALL;
operation = 2'b10;

S14:
begin

A_Address_Generator = 2'b00;
B_Address_Generator = 2'b10;
OEA_o1 = 1;
OEB_o1 = 1;
CSA_o1 = 1;
CSB_o1 = 1;
WEAN_o1 = WRNONE;
WEBN_o1 = WRNONE;
Exp_OP_A = CONSC;
Exp_OP_B = CONSD;
result = 0;
done = 0;
O_Reset_Value = MAX_SIZE;
DIA_o1 = 0;
DIB_o1 = 0;
if (Counter == 7'h00)
begin
    chk_A = 0;
end
I.1 BIST RTL

870       chk_B = 0;
871       nxt_state = S15;
872       operation = 2'b11;
873
874       end
875       else
876       begin  nxt_state = S14;
877       chk_A = 1;
878       chk_B = 1;
879       operation = 2'b10; end
880       end
881
882       S15:
883
884       begin
885       operation = 2'b00;
886       A_Address_Generator = 2'b00;
887       B_Address_Generator = 2'b10;
888       OEA_o1 = 1;
889       OEB_o1 = 1;
890       CSA_o1 = 1;
891       CSB_o1 = 1;
892       WEAN_o1 = WRALL;
893       WEBN_o1 = WRALL;
894       DIA_o1 = CONSE;
DIB_o1 = CONSF;

Exp_OP_A = CONSE;

Exp_OP_B = CONSF;

chk_A = 0;

chk_B = 0;

result = 0;

done = 0;

O_Reset_Value = MAX_SIZE;

nxt_state = S16;

end

S16:

begin

A_Address_Generator = 2'b00;

B_Address_Generator = 2'b10;

OEA_o1 = 1;

OEB_o1 = 1;

CSA_o1 = 1;

CSB_o1 = 1;

WEAN_o1 = WRNONE;
920       WEBN_o1=WRNONE;
921       DIA_o1=CONSE;
922       DIB_o1=CONSF;
923       Exp_OP_A=CONSE;
924       Exp_OP_B=CONSF;
925       chk_A =1;
926       chk_B =1;
927       nxt_state=S17;
928       operation=2’b00;
929       result=0;
930       done=0;
931       O_Reset_Value=MAX_SIZE;
932
933
934
935
936       end
937
938
939       S17:
940       begin
941
942
943       A_Address_Generator=2’b00;
944       B_Address_Generator=2’b10;
I.1 BIST RTL

945 OEA_o1=1;
946 OEB_o1=1;
947 CSA_o1=1;
948 CSB_o1=1;
949 WEAN_o1=WRALL;
950 WEBN_o1=WRALL;
951 DIA_o1=CONSF;
952 DIB_o1=CONSE;
953 Exp_OP_A=CONSF;
954 Exp_OP_B=CONSE;
955 chk_A =0;
956 chk_B =0;
957 result =0;
958 done =0;
959 O_Reset_Value=MAX_SIZE;
960 nxt_state=S18;
961 operation=2’b00;

end

S18:
begin

A_Address_Generator=2’b00;
B_Address_Generator = 2'b10;
OEA_o1 = 1;
OEB_o1 = 1;
WEAN_o1 = WRNONE;
WEBN_o1 = WRNONE;
CSA_o1 = 1;
CSB_o1 = 1;
Exp_OP_A = CONSF;
Exp_OP_B = CONSE;
O_Reset_Value = HALF_SIZE;
result = 0;
done = 0;
DIA_o1 = 0;
DIB_o1 = 0;
if (Counter == HALF_SIZE)
begin
  nxt_state = S19;
  operation = 2'b11;
  chk_A = 0;
  chk_B = 0;
end
else
begin  operation=2'b10;
    chk_A=1;
    chk_B=1;
    nxt_state=S15;
end

S19:
begin
operation=2'b00;
A_Address_Generator=2'b00;
B_Address_Generator=2'b10;
OEA_o1=1;
OEB_o1=1;
CSA_o1=1;
end
CSB_o1 = 1;
WEAN_o1 = WRALL;
WEBN_o1 = WRALL;
DIA_o1 = CONSA;
DIB_o1 = CONSB;
Exp_OP_A = CONSA;
Exp_OP_B = CONSB;
chk_A = 0;
chk_B = 0;
result = 0;
done = 0;
O_Reset_Value = MAX_SIZE;
nxt_state = S20;

end

S20:
begin

operation = 2'b00;
A_Address_Generator = 2'b00;
B_Address_Generator = 2'b10;
OEA_o1 = 1;
1045    OEB_o1=1;
1046    CSA_o1=1;
1047    CSB_o1=1;
1048    WEAN_o1=WRNONE;
1049    WEBN_o1=WRNONE;
1050    DIA_o1=CONSA;
1051    DIB_o1=CONSB;
1052    Exp_OP_A=CONSA;
1053    Exp_OP_B=CONSB;
1054    chk_A=1;
1055    chk_B=1;
1056    nxt_state=S21;
1057    result=0;
1058    done=0;
1059    O_Reset_Value=MAX_SIZE;
1060
1061
1062
1063
1064    end
1065
1066
1067    S21:
1068    begin
1069
A_Address_Generator = 2'b00;
B_Address_Generator = 2'b10;
OEA_o1 = 1;
OEB_o1 = 1;
CSA_o1 = 1;
CSB_o1 = 1;
WEAN_o1 = WRALL;
WEBN_o1 = WRALL;
DIA_o1 = CONSB;
DIB_o1 = CONSA;
Exp_OP_A = CONSF;
Exp_OP_B = CONSE;
chk_A = 0;
chk_B = 0;
result = 0;
done = 0;
O_Reset_Value = MAX_SIZE;
nxt_state = S22;
operation = 2'b00;
end
S22:
begin

A_Address_Generator='b00;
B_Address_Generator='b10;
OEA_o1=1;
OEB_o1=1;
WEAN_o1=WRNONE;
WEBN_o1=WRNONE;
CSA_o1=1;
CSB_o1=1;
Exp_OP_A=CONSB;
DIA_o1=0;
DIB_o1=0;
Exp_OP_B=CONSA;
O_Reset_Value='h00;
result=0;
done=0;
if (Counter=='h00)
begin
    nxt_state=S23;
    operation='b11;
    chk_A=0;
    chk_B=0;
end
I.1 BIST RTL

```verilog
1120
1121       end
1122
1123        else
1124        begin
1125           operation = 2'b10;
1126           chk_A = 1;
1127           chk_B = 1;
1128           nxt_state = S19;
1129        end
1130
1131
1132
1133       end S23:
1134        begin
1135           A_Address_Generator = 2'b00;
1136           B_Address_Generator = 2'b10;
1137           OEA_o1 = 1;
1138           OEB_o1 = 1;
1139           CSA_o1 = 1;
1140           CSB_o1 = 1;
1141           WEAN_o1 = WRALL;
1142           WEBN_o1 = WRNONE;
1143           DIA_o1 = CONSC;
1144           DIB_o1 = CONSF;
```
Exp_OP_A=CONSB;

Exp_OP_B=CONSA;

chk_A=0;

chk_B=1;

O_Reset_Value=MAX_SIZE;

result=0;

done=0;

if (Counter==HALF_SIZE)

begin

nxt_state=24;

operation=2'b11;

end

else

begin

nxt_state=S23;

operation=2'b01;

end

end

S24:

begin

A_Address_Generator=2'b00;

B_Address_Generator=2'b10;

OEA_o1=1;
OEB\_o1=1;
CSA\_o1=1;
CSB\_o1=1;
WEAN\_o1=WRNONE;
WEBN\_o1=WRALL;
DIA\_o1=CONSC;
DIB\_o1=CONSF;
Exp\_OP\_A=CONSA;
Exp\_OP\_B=CONSC;
result=0;
done=0;
chk\_B=0;
O\_Reset\_Value=7'h00;
if (Counter==HALF\_SIZE)
  begin
    nxt\_state=S25;
    operation=2'b11;
    chk\_A=0;
  end
else
  begin
    nxt\_state=S24;
    operation=2'b10; chk\_A=1;
  end
end
S25: begin
  A_Address_Generator = 2'b00;
  B_Address_Generator = 2'b10;
  DIA_o1 = CONSF;
  DIB_o1 = CONSF;
  CSA_o1 = 1;
  CSB_o1 = 1;
  OEA_o1 = 0;
  OEB_o1 = 0;
  Exp_OP_A = 0;
  Exp_OP_B = 0;
  chk_A = 0;
  chk_B = 0;
  done = 0;
  result = 0;
  WEAN_o1 = WRALL;
  WEBN_o1 = WRALL;
  O_Reset_Value = 0;
  if (Counter == HALF_SIZE)
    begin
      nxt_state = DONE_STATE;
operation = 2'b11;

end

else

begin

nxt_state = S25;

operation = 2'b01;

end

end

ERROR_STATE:

begin

// $display("Error State");

done = 1;

result = 0;

WEAN_o1 = WRNONE;

WEBN_o1 = WRNONE;

operation = 2'b00;

nxt_state = S0;
1245     OEA_o1=0;
1246     OEB_o1=0;
1247     CSA_o1=0;
1248     CSB_o1=0;
1249     DIA_o1=0;
1250     DIB_o1=0;
1251     Exp_OP_A=0;
1252     Exp_OP_B=0;
1253     chk_B=0;
1254     chk_A=0;
1255     O_Reset_Value=7’h00;
1256     A_Address_Generator=2’b00;
1257     B_Address_Generator=2’b00;
1258          end
1259
1260
1261
1262  DONE_STATE:
1263    begin
1264
1265          $display ("Done");
1266          done=1;
1267          result=1;
1268          WEAN_o1=WRNONE;
1269          WEBN_o1=WRNONE;
operation = 2'b00;
nxt_state = S0;
OEA_o1 = 0;
OEB_o1 = 0;
CSA_o1 = 0;
CSB_o1 = 0;
DIA_o1 = 0;
DIB_o1 = 0;
Exp_OP_A = 0;
Exp_OP_B = 0;
chk_B = 0;
chk_A = 0;
O_Reset_Value = 7'h00;
A_Address_Generator = 2'b00;
B_Address_Generator = 2'b10;
end

default: begin
nxt_state = S0;
DIA_o1 = 0;
DIB_o1 = 0;
OEA_o1 = 0;
OEB_o1 = 0;
WEAN_o1 = WRNONE;
WEBN_o1 = WRNONE;
CSA_o1 = 0;
CSB_o1 = 0;
chk_A = 0;
chk_B = 0;
operation = 0;
O_Reset_Value = ZERO;
A_Address_Generator = 0;
B_Address_Generator = 0;
Exp_OP_A = 0;
Exp_OP_B = 0;
done = 0;
result = 0; end
dcase
end
else
begin
next_state = S0;
DIA_o1 = 0;
DIB_o1 = 0;
OEA_o1 = 0;
OEB_o1 = 0;
WEAN_o1 = WRNONE;
WEBN_o1 = WRNONE;
CSA_o1 = 0;
CSB_o1 = 0;
1320       chk_A = 0;
1321       chk_B = 0;
1322       operation = 0;
1323       O_Reset_Value = ZERO;
1324       A_Address_Generator = 0;
1325       B_Address_Generator = 0;
1326       Exp_OP_A = 0;
1327       Exp_OP_B = 0;
1328       done = 0;
1329       result = 0;
1330       end
1331
1332
1333       end
1334
1335
1336       endmodule  // SDPRAM
I.2 Interface

interface intf;

parameter WSIZE = 32;
parameter MSIZE = 128;
parameter ASIZE = $clog2(MSIZE);
parameter WRSIZE = WSIZE/8;
parameter WRALL = {WRSIZE{1'b0}};
parameter WRNONE = {WRSIZE{1'b1}};

wire [WSIZE−1:0] DOA, DOB;
bit [WSIZE−1:0] DIA, DIB;
bit [WRSIZE−1:0] WEAN, WEBN;
bit OEA, OEB, CSA, CSB, CKA, CKB;
bit [ASIZE−1:0] A, B;
bit clk;
bit test_mode;
bit bist_mode;
wire result;
bit start;
wire done;
bit reset;
bit scan_in0, scan_en, scan_out0;
24 end interface: intf
class sdp_driver extends uvm_driver #(sdp_transaction);

'uvm_component_utils(sdp_driver)

uvm_analysis_port #(sdp_transaction) drv2sb;
virtual intf vif;
sdp_transaction sd_tx;

function new(string name, uvm_component parent);
  super.new(name, parent);
endfunction: new

function void build_phase(uvm_phase phase);
  super.build_phase(phase);
  drv2sb = new("drv2sb", this);
  if (!uvm_config_db #(virtual intf)::get(null, "*", "vif", vif))
    'uvm_fatal("DRIVER", "Failed to get interface")
endfunction: build_phase

task run_phase(uvm_phase phase);
  phase.raise_objection(this);
  drive();
  phase.drop_objection(this);
endtask: run_phase
task run_reset_bist();

vif.OEA=0;
vif.CSA=0;
vif.DIA=0;
vif.WEAN=4'hF;
vif.A=0;
vif.OEB=0;
vif.CSB=0;
vif.DIB=0;
vif.B=0;
vif.WEBN=4'hF;
vif.test_mode=0;
vif.scan_in0 = 0;
vif.scan_en = 0;
vif.reset=0;
vif.bist_mode =0;
repeat(5) @(posedge vif.clk);
vif.reset=1;
repeat(5) @(posedge vif.clk);
vif.reset=0;
@(posedge vif.clk);
vif.bist_mode =1;
@(posedge vif.clk);
vif.start =1 ;
@(posedge vif.done);
@(posedge vif.clk);

vif.bist_mode =0;
repeat(5) @(posedge vif.clk);
endtask

task run_PortA;
 @(posedge vif.CKA)
begin

case (sd_tx.opA)
2'b00: begin
vif.OEA=1;
vif.CSA=0;
vif.DIA=0;
vif.WEAN=4'hF;
vif.A=0;
end
2'b01: begin
vif.OEA=1;
vif.CSA=1;
vif.DIA=0;
vif.WEAN=4'hF;
vif.A=sd_tx.adrA;
end
2'b10: begin
vif.OEA=1;


I.3 Driver

vif.CSA=1;
vif.DIA=sd_tx.data;
vif.WEAN=4'h0;
vif.A=sd_tx.adrA;
end

default: begin
vif.OEA=0;
vif.CSA=0;
vif.DIA=0;
vif.WEAN=4'hF;
vif.A=sd_tx.adrA;
end
endcase
end
task run_PortB;
@(posedge vif.CKB)
begin
case(sd_tx.opB)
2'b00: begin
vif.OEB=1;
vif.CSB=0;
vif.DIB=0;
end
endcase
end

task run_PortB;
I.3 Driver

99 \[\text{vif}.\text{WEBN}=4\text{'}\text{hF};;\]
100 \[\text{vif}.\text{B}=0;;\]
101 \[\text{end}\]
102 \[2\text{'}\text{b01}: \text{begin}\]
103 \[\text{vif}.\text{OEB}=1;;\]
104 \[\text{vif}.\text{CSB}=1;;\]
105 \[\text{vif}.\text{DIB}=0;;\]
106 \[\text{vif}.\text{WEBN}=4\text{'}\text{hF};;\]
107 \[\text{vif}.\text{B}=\text{sd}\_\text{tx}.\text{adrB};\]
108 \[\text{end}\]
109
110 \[2\text{'}\text{b10}: \text{begin}\]
111 \[\text{vif}.\text{OEB}=1;;\]
112 \[\text{vif}.\text{CSB}=1;;\]
113 \[\text{vif}.\text{DIB}=\text{sd}\_\text{tx}.\text{datB};\]
114 \[\text{vif}.\text{WEBN}=4\text{'}\text{h0};;\]
115 \[\text{vif}.\text{B}=\text{sd}\_\text{tx}.\text{adrB};\]
116 \[\text{end}\]
117
118 \[\text{default}: \text{begin}\]
119 \[\text{vif}.\text{OEB}=0;;\]
120 \[\text{vif}.\text{CSB}=0;;\]
121 \[\text{vif}.\text{DIB}=0;;\]
122 \[\text{vif}.\text{WEBN}=4\text{'}\text{hF};;\]
123 \[\text{vif}.\text{B}=\text{sd}\_\text{tx}.\text{adrB};\]
I.3 Driver

124         end
125     endcase
126     end
127 endtask
128
129
130 virtual task drive();
131     run_reset_bist();
132
133
134
135     repeat (10000) begin
136         seq_item_port.get_next_item (sd_tx);
137         fork
138         run_PortA;
139         run_PortB;
140         join
141         drv2sb.write (sd_tx);
142         seq_item_port.item_done();
143
144         end
145
146
147 endtask: drive
148
149 endclass: sdp_driver
I.4 Environment

```cpp
class sdp_env extends uvm_env;

'uvm_component_utils(sdp_env)

sdp_agent sd_agent;
sdp_scoreboard sd.sb;
sdp_coverage sd_cov;
uvm_tlm_fifo #(mon2sb) fifo_h;

function new(string name, uvm_component parent);
   super.new(name, parent);
endfunction: new

function void build_phase(uvm_phase phase);
   super.build_phase(phase);
   sd_agent = sdp_agent::type_id::create("sd_agent", this);
   sd_sb = sdp_scoreboard::type_id::create("sd_sb", this);
   sd_cov = sdp_coverage::type_id::create("sd_cov", this);
   fifo_h = new("fifo_h", this);
endfunction: build_phase

function void connect_phase(uvm_phase phase);
   super.connect_phase(phase);
   sd_agent.sd_drvr.dr2sb.connect(sd_sb.analysis_export);
   sd_agent.sd_drvr.dr2sb.connect(sd_cov.analysis_export);
```
I.4 Environment

24    sd_sb.get_port_h.connect(fifo_h.get_export);
25    sd_agent.sd_mntr.put_port_h.connect(fifo_h.put_export);
26    // sd_agent.sd_drvr.connect(sd_sb.sb_export_before);
27
28    endfunction: connect_phase
29    endclass: sdp_env
# I.5 Agent

```cpp
class sdp_agent extends uvm_agent;

'uvm_component_utils(sdp_agent)

sdp_sequencer sd_seqr;
sdp_driver sd_drvr;
sdp_monitor sd_mntr;

function new(string name, uvm_component parent);
    super.new(name, parent);
endfunction: new

function void build_phase(uvm_phase phase);
    super.build_phase(phase);
endfunction: build_phase

    // agent_ap_before = new("agent_ap_before", this);

    sd_seqr = sdp_sequencer::type_id::create("sd_seqr", this);
    sd_drvr = sdp_driver::type_id::create("sd_drvr", this);
    sd_mntr = sdp_monitor::type_id::create("sd_mntr", this);
```

endfunction: build_phase

function void connect_phase(uvm_phase phase);
    super.connect_phase(phase);
    sd_drvr.seq_item_port.connect(sd_seqr.seq_item_export);
endfunction: connect_phase
endclass: sdp_agent
I.6 Sequencer

```plaintext
import uvm_pkg::*;

#include "uvm_macros.svh"

class sdp_transaction extends uvm_sequence_item;

function new(string name = "");
    super.new(name);
endfunction: new

rand bit [1:0] opA;
rand bit [1:0] opB;
rand bit [31:0] datA;
rand bit [31:0] datB;
rand bit [3:0] wrtA;
rand bit [3:0] wrtB;
rand bit [6:0] adrA;
rand bit [6:0] adrB;
logic [31:0] outA;
logic [31:0] outB;

constraint write_collision {
```
(((opA==2'b10 || opB==2'b10)) -> (adrA != adrB)); }

constraint operation {
    opA<2'b11;
    opB<2'b11;
}

'uvm_object_utils_begin(sdp_transaction)
'uvm_field_int(opA, UVM_ALL_ON)
'uvm_field_int(opB, UVM_ALL_ON)
'uvm_field_int(datA, UVM_ALL_ON)
'uvm_field_int(datB, UVM_ALL_ON)
'uvm_field_int(wrtA, UVM_ALL_ON)
'uvm_field_int(wrtB, UVM_ALL_ON)
'uvm_field_int(adrA, UVM_ALL_ON)
'uvm_field_int(adrB, UVM_ALL_ON)
'uvm_object_utils_end
endclass: sdp_transaction

class sdp_sequence extends uvm_sequence #(sdp_transaction);
'uvm_object_utils(sdp_sequence)

function new(string name = "");
```plaintext
49     super.new(name);
50     endfunction: new
51
52     task body();
53     sdp_transaction sd_tx;
54
55     forever begin
56     sd_tx = sdp_transaction::type_id::create("sd_tx");
57
58     start_item(sd_tx);
59     assert(sd_tx.randomize());
60     //                              `uvm_info("sd_sequence", sa_tx.sprint(), UVM_LOW);
61     finish_item(sd_tx);
62     end
63     endtask: body
64 endclass: sdp_sequence
65
66
67     typedef uvm_sequencer #(sdp_transaction) sdp_sequencer;
68
69     typedef logic [1:0] [31:0] mon2sb;
```
class sdp_monitor extends uvm_monitor;

'uvm_component_utils(sdp_monitor)

uvm_put_port #(mon2sb) put_port_h;
virtual intf vif;

mon2sb Aval;

function new(string name, uvm_component parent);
    super.new(name, parent);
    // Aval.outA=0;
    // Aval.outB=0;
endfunction: new

function void build_phase(uvm_phase phase);
    super.build_phase(phase);
endfunction: build_phase
if (!uvm_config_db #(virtual intf)::get(null, "*", "vif", vif))
    'uvm_fatal("Monitor", "Failed to get interface")

put_port_h = new("put_port_h", this);
endfunction: build_phase

task run_phase(uvm_phase phase);

forever begin
    @(posedge vif.clk)
    begin
        #3;
        Aval[0] = vif.DOA;
        Aval[1] = vif.DOB;
        // $display("MONITOR %d %d", vif.DOA, vif.DOB);
        // Send the transaction to the analysis port
        put_port_h.put(Aval);
    end
end
endtask : run_phase

class : sdp_monitor
class sdp_scoreboard extends uvm_subscriber #(sdp_transaction);

'uvm_component_utils(sdp_scoreboard)

uvm_get_port #(mon2sb) get_port_h;

// logic [32:0] aa, bb;

mon2sb aaa;

logic [31:0] predicted_data1, predicted_data2, t1, t2;

bit check1, check2;

function new(string name, uvm_component parent);
    super.new(name, parent);
    check1 = 0;
    check2 = 0;
    t1 = 0;
    t2 = 0;
    predicted_data1 = 0;
    predicted_data2 = 0;
endfunction: new
function void build_phase(uvm_phase phase);
    get_port_h = new("get_port_h", this);
endfunction : build_phase

bit [31:0] mem [int];

function void write(sdp_transaction t);

if (t.opA==2'b10)
begin
  // $display(" A port %d %d", t.adrA, t.datA);
  mem[t.adrA] = t.datA;
  check1 = 0;
  predicted_data1 = t.datA;
end
else if (t.opA==2'b01)
begin
  check1 = 1;
  if (mem.exists(t.adrA))
    predicted_data1 = mem[t.adrA];
  else
    predicted_data1 = 0;
end
I.8 Scoreboard

49          end
50    else begin
51          check1 = 0;
52          predicted_data1 = predicted_data1;
53          end
54
55
56
57    if (t.opB==2'b10)
58        begin
59          // $display("B Port %d %d", t.adrB, t.datB);
60          mem[t.adrB] = t.datB;
61          check2 = 0;
62          predicted_data2 = t.datB;
63        end
64    else if (t.opB==2'b01)
65        begin    check2 = 1;
66          if (mem.exists(t.adrB))
67            predicted_data2 = mem[t.adrB];
68          else
69            predicted_data2 = 0;
70
71        end
72    else begin
73        check2 = 0;
predicted_data2 = predicted_data2;
end

eendifunction : write

task run_phase(uvm_phase phase);
check1 = 0;
check2 = 0;
forever begin
get_port_h.get(aaa);
if (check1 == 1)
begin
if (aaa[0] != t1)
begin // $display("Port 1 value %d Predicted data %d ", aaa[0], t1);
'uvm_fatal("SCOREBOARD", "Failed at Port A ")
end
end
if (check2 == 1)
begin
if (aaa[1] != t2)
begin

I.8 Scoreboard

97    // $display("Port 2 %d Predicted data %d", aaa[1], t2

98    `uvm_fatal("SCOREBOARD", "Failed at Port B")

99    end

100 end

101

102 // $display("%d %d", predicted_data1, predicted_data2);

103    t1=predicted_data1;

104    t2=predicted_data2;

105    end

106

107 endtask: run_phase

108

109

110

111

112

113

114 endclass: sdp_scoreboard
I.9 SDP Test

```h臻
class SDP_test extends uvm_test;

    `uvm_component_utils(SDP_test)

sdp_env sd_env;

function new(string name, uvm_component parent);
    super.new(name, parent);
endfunction: new

function void build_phase(uvm_phase phase);
    sd_env = sdp_env::type_id::create("sd_env", this);
endfunction: build_phase

task run_phase(uvm_phase phase);
    sdp_sequence sdp_seq;
    // phase.raise_objection(.obj(this));
endtask: run_phase
```
sdp_seq = sdp_sequence::type_id::create(.name("sdp_seq"), .ctxt(get_full_name()));
assert(sdp_seq.randomize());
sdp_seq.start(sd_env.sd_agent.sd_seqr);
// phase.drop_object(.obj(this));
endtask: run_phase

class: SDP_test
I.10 Top Test

`// `include "sdp_monitor.sv"

`// `include "./sdp_scoreboard.sv"

`// `include "./sdp_config.sv"

module test;

import uvm_pkg::*;

intf vif();

SDPRAM top(.result(vif.result),
            .done(vif.done),
            .start(vif.start),
.clk (vif.clk),
.reset (vif.reset),
.scan_in0 (vif.scan_in0),
.scan_en (vif.scan_en),
.bist_mode (vif.bist_mode),
.test_mode (vif.test_mode),
.scan_out0 (vif.scan_out0),
.DOA (vif.DOA),
.DOB (vif.DOB),
.DIA (vif.DIA),
.DIB (vif.DIB),
.WEAN (vif.WEAN),
.WEBN (vif.WEBN),
.OEA (vif.OEA),
.OEB (vif.OEB),
.CSA (vif.CSA),
.CSB (vif.CSB),
.CKA (vif.CKA),
.CKB (vif.CKB),
.A (vif.A),
.B (vif.B)
);

initial
begin

uvm_config_db #(virtual intf)::set(null, "*", "vif", vif);
run_test();
end

initial begin
vif.clk = 'b1;
vif.CKA = 'b1;
vif.CKB = 'b1;
end

initial begin
$set_coverage_db_name("SDPRAM");
end

always
#5 vif.CKA = ~vif.CKA;
always
#5 vif.CKB = ~vif.CKB;
always
#5 vif.clk = ~vif.clk;
endmodule : test