Test and Diagnosis of Microprocessor Memory Arrays using Functional Patterns

by

Ya-Chieh Lai

Submitted to the Department of Electrical Engineering and Computer Science in Partial Fulfillment of the Requirements for the Degrees of

Bachelor of Science in Electrical Science and Engineering and Master of Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

May 26, 1996

Author .. ..
Department of Electrical Engineering and Computer Science
January 30, 1996

Certified by .....
Srinivas Devadas
...isor

Accepted by ....
...ler
Chairman, Department Committee on Graduate Theses

Test and Diagnosis of Microprocessor Memory Arrays using Functional Patterns

by
Ya-Chieh Lai

Submitted to the
Department of Electrical Engineering and Computer Science

January 30, 1996

In Partial Fulfillment of the Requirements for the Degrees of
Bachelor of Science in Electrical Science and Engineering
and Master of Engineering in Electrical Engineering and Computer Science

Abstract

As the complexity of microprocessors continues to increase, so too does the importance of test and diagnosis. The main goal of this thesis is a diagnostic decision-making procedure. Before this can be done, a fault model must be defined. For every fault, its manifestation or fault signature needs to be determined.

In this thesis we take the approach that if every fault can be shown to have a unique fault signature, then it can be uniquely diagnosed. This fault signature is presented graphically as a failure bitmap. By applying a set of criteria to the failure bitmap, a diagnosis can be made.

Thesis Supervisor: Srinivas Devadas
Title: Associate Professor, Research Lab for Electronics

Thesis Supervisor: Leendert Huisman
Title: Senior Engineer, IBM Microelectronics

Thesis Supervisor: Matthew Graf
Title: Microprocessor Development Manager, IBM Microelectronics

Acknowledgments

I would like to thank all the members of the IBM community from whom I have received assistance. I am eternally grateful to Dr. Leendert Huisman for being my mentor during the course of this thesis. He taught me all about memory testing and diagnosis and helped me to understand and clarify much of the work herein. I would like to thank Matthew Graf for helping me find this thesis topic just as I was beginning to lose hope of ever finding a topic and for being willing to be my manager summer after summer. I would like to thank Dean Adams for taking the time to explain to me many of the finer points of memory design and testing.

I am grateful to Professor Srinivas Devadas for advising this thesis.

Foremost, I am grateful to my Mother and Father. I thank you for your love and support and for helping me to reach where I am today.

I want to give a special thanks to Erika Chuang for putting up with me and loving me always.

Table of Contents

1 Introduction **13**
2 The Fault Model **15**
2.1 Introduction 15
2.2 Faults in the Memory Cell Array 17
2.2.1 Stuck-at Fault 18
2.2.2 Transition Fault 19
2.2.3 Coupling Fault 19
2.2.4 Non-Deterministic Faults 21
2.2.5 Destructive Read Fault 23
2.2.6 Pattern Sensitive Faults 24
2.2.7 Linked versus Unlinked faults 25
2.3 Faults in the Read/Write Logic 26
2.3.1 Bitline Precharge Faults 28
2.3.2 Bitline Faults 28
2.3.3 Column Multiplexor Faults 30
2.3.4 Data-in Faults 31
2.3.5 Sense Amplifier Faults 31
2.3.6 Data-out Faults 31
2.4 Faults in the Address Decoder 32
2.4.1 Row Decoder Faults 34
2.4.2 Column Decoder Faults 35
2.4.3 Wordline Faults 36
2.4.4 Bitline Select Faults 37
2.5 Faults that are not caught 38
3 Test **39**
3.1 Introduction 39
3.2 The Tests 40
3.2.1 Unique Address Ripple Word Test 42
3.2.2 Word Lines Stripe Test 42
3.2.3 Checkerboard Test 43
3.3 Implementation 44
3.3.1 Observability 45
3.3.2 Tester Idiosyncrasies 46
3.3.3 Failure Bitmaps 47
4 Diagnosis **49**
4.1 Introduction 49
4.2 Fault Detection and Manifestation: Memory Cell Array 50
4.2.1 Stuck-at Faults 51
4.2.2 Transition Faults 51
4.2.3 Coupling Faults 53
4.2.4 Non-Deterministic Faults 58
4.2.5 Destructive Read Faults 58
4.3 Fault Detection and Manifestation: Read/Write Logic 59

4.3.1 Bitline Precharge Faults 59
4.3.2 Bitline Faults 61
4.3.3 Column Multiplexor Faults 65
4.3.4 Data-in Faults 70
4.3.5 Sense Amplifier Faults 71
4.3.6 Data-out Faults 72
4.4 Fault Detection and Manifestation: Address Decoder 74
4.4.1 Row Decoder Faults 76
4.4.2 Column Decoder Faults 79
4.4.3 Wordline/Bitline Selector Faults 82
5 Summary and Conclusions 87
5.1 Diagnostic Tree 87
5.2 Test Optimization 93
A Shortening Test Simulation Time and Reducing Tester Buffer Memory Requirements 95
A.1 Introduction 95
A.2 Reducing the Simulation Time 95
A.2.1 Breaking up the Unique Address Ripple Word Test 96
A.2.2 Breaking up the Memory 96
A.3 Reducing the Tester Buffer Memory Requirements 98
A.3.1 Enabling the Cache 99
A.3.2 Memory Compression 99

List of Figures

Figure 2.1: Reduced functional model 15
Figure 2.2: Memory array architecture 16
Figure 2.3: A typical CMOS SRAM cell 18
Figure 2.4: Example of stuck-at fault and transition fault 19
Figure 2.5: Example of coupling fault 20
Figure 2.6: Examples of non-deterministic faults 22
Figure 2.7: Destructive read fault 24
Figure 2.8: Example of complex pattern sensitive fault 25
Figure 2.9: Example of linked faults 25
Figure 2.10: Unilateral state-coupling fault. 26
Figure 2.11: Faults in the read/write logic 27
Figure 2.12: Bitline precharge fault on complement wire 28
Figure 2.13: Bitline faults 29
Figure 2.14: Column multiplexor faults 30
Figure 2.15: Logical address decoder faults 32
Figure 2.16: Physical address decoder faults 33
Figure 2.17: Range of possible scopes of different faults 34
Figure 2.18: Wordline stuck-at 1 37
Figure 2.19: Bitline select stuck-at 38
Figure 3.1: Example of memory 41
Figure 3.2: Result of word line stripe test 42
Figure 3.3: Result of checkerboard test 43
Figure 3.4: Result of checkerboard under alternative memory implementation 44
Figure 3.5: Information encoding 46
Figure 4.1: List of Faults 50
Figure 4.2: Stuck-at fault detection and manifestation 51
Figure 4.3: Transition fault detection and manifestation 53
Figure 4.4: Fault signatures of idempotent faults 56
Figure 4.5: Fault signatures of inversion faults 57
Figure 4.6: Fault signatures of state-coupling faults 57
Figure 4.7: Fault Signatures of destructive read faults 59
Figure 4.8: Bitline precharge fault: manifestation in the test 60
Figure 4.9: Bitline precharge fault: manifestation in memory 61
Figure 4.10: Break in true wire of bitline 62
Figure 4.11: Fault signatures of break in bitline wire 63
Figure 4.12: Fault signature of true to complement short within one bitline 64
Figure 4.13: True to complement short in bitline 64
Figure 4.14: Fault signature of shorts across adjacent bitlines 65
Figure 4.15: Pass-gate stuck open in column multiplexor 66
Figure 4.16: Fault signature of stuck open pass-gate in column multiplexor 67
Figure 4.17: Pass-gate stuck-closed in column multiplexor 67
Figure 4.18: Column multiplexor stuck-closed: first case 69
Figure 4.19: Column multiplexor stuck-closed: second case 69

Figure 4.20: Fault signature of pass-gate stuck-closed in column multiplexor 70
Figure 4.21: Fault signature of data-in fault 71
Figure 4.22: Fault signature of sense amplifier fault 72
Figure 4.23: Data-out short 73
Figure 4.24: Fault signature of data-out short fault 74
Figure 4.25: Logical address decoder faults 75
Figure 4.26: Addressing clarification 75
Figure 4.27: Address decoder wordlines 76
Figure 4.28: Fault signature of row decoder faults A and B 76
Figure 4.29: Fault signature of row decoder fault C 78
Figure 4.30: Fault signature of row decoder fault D 79
Figure 4.31: Address decoder bitline selectors 80
Figure 4.32: Fault signature of column decoder faults A and B 80
Figure 4.33: Fault signature of column decoder fault C 81
Figure 4.34: Fault signature of column decoder fault D 82
Figure 4.35: Wordline/bitline selector stuck-at 1 fault 83
Figure 4.36: Fault signature of stuck-at 1 fault on wordline 84
Figure 4.37: Fault signature of wordline break 85
Figure 4.38: Fault signature of bitline selector break 85
Figure 5.1: Diagnostic flow 87
Figure 5.2: Scope of failure 89
Figure 5.3: Memory cell array faults 90
Figure 5.4: Read/write logic faults 91
Figure 5.5: Address decoder faults 92
Figure 5.6: Extra test for data-out faults 93

List of Tables

Table 4.1: Catching coupling faults assuming source cell precedes target cell..... 54
Table 4.2: Catching coupling faults assuming target cell precedes source cell..... 54
Table A.1: Required lines in tester buffer used for 16 entry dual-ported memory array with and without the cache..... 99
Table A.2: Required lines in tester buffer used for 16 entry dual-ported memory array with and without compression 100

Chapter 1

Introduction

Testing is the process of exercising a system and determining whether or not the system behaves correctly. If the system misbehaves, diagnosis is the further study of the system to determine the cause of the misbehavior. As the complexity of microprocessors continues to increase, so too does the importance of test and diagnosis. Though test is a fairly well-understood topic, there exists very little literature on diagnosis. During manufacturing, test is the most important component; bad chips need to be identified and discarded. Diagnosis becomes important when yield becomes low and it is necessary to pinpoint the defects or family of defects that are bringing down the yield. The purpose of this thesis is to create a methodology for the test and diagnosis of embedded memory arrays for one implementation of a microprocessor in the hope that conclusions reached here can be generalized to different implementations and technologies. At the final step of this thesis, a complete diagnostic decision-making process (a diagnostic tree) will be described.

This project involved the test and diagnosis of a number of small CMOS SRAM memory arrays (i.e., the translation look-aside buffer, segment look-aside buffer, etc.) embedded into a modern microprocessor. There were a number of approaches that could have been taken in this regard. Most of the larger memory arrays on the processor were tested using built-in self-test (BIST), in which a finite state machine exists on-chip with the ability to test the arrays. However, the number and size of these smaller arrays did not warrant the design time nor the chip space for BIST implementation. A second approach was to use scan techniques, where every latch is connected serially into one or more scan chains that can then be controlled and observed. However, the amount of time required for scan in and out of the patterns necessary to test every memory array cell made the scan tech-

nique impractical. The approach for this project was to use functional patterns, whereby the pins of the chip are driven with stimuli and then observed for responses while the chip is operating in a functional mode (i.e., as it would normally operate, as versus in a test mode). Essentially, the test algorithm is run on the chip in the form of a working program.

In general, functional patterns have their own shortcomings: "test pattern generation and fault simulation require too much time, and the manufacturing test suite would exceed the limits of parallel tester memory." [8]. However, these shortcomings are surmountable for the case of memory array testing and diagnosis. Pattern generation is not as much of a problem for memory array tests because memory is structurally very regular. The fault simulation time is acceptable because of the smaller size of the arrays under consideration. Finally, the limit on the tester memory is overcome by running these tests with the cache enabled. In this way, the tests can be fetched into the cache and run from within the processor without direction by the tester.

This thesis begins with a description of the specific faults that are to be targeted and how these faults will be modelled. The second section is a description of the proposed test algorithms designed to cover these faults. The third section is a description of the diagnostic information available from these tests. The way the fault manifests itself is it's fault signature. If every fault can be shown to have a unique fault signature, then a diagnosis can be made based upon the fault signature. From the findings of this thesis, a diagnostic decision making procedure will be presented in the conclusion section.

Chapter 2

The Fault Model

2.1 Introduction

"In order to develop any practically feasible test procedure, we should restrict ourselves to a subset of faults that are most likely to occur. This practice is known as 'selecting a fault model'" [1]. Even though all faults have some physical root cause, it is desired that these physical faults be modelled as logical faults. For instance, one physical fault in a memory cell is a short to ground.The logical equivalent of this fault might be that the cell is stuck at 0. There are two advantages to this layer of abstraction. First, methods to catch these logical faults can be valid across different physical implementations. Second, this abstraction simplifies the problem tremendously. For instance, there could be any number of physical reasons for a cell to be stuck at 0, but the idea of a cell stuck at 0 is cleaner and simplifies the reasoning.

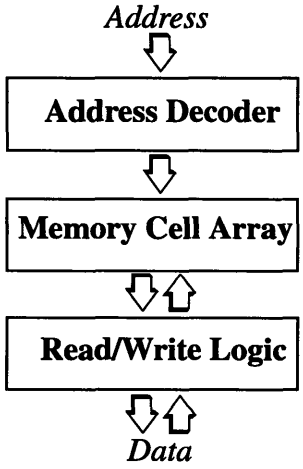


Figure 2.1: Reduced functional model

Borrowing from Van de Goor's "Reduced Functional Ram Chip Model" most memory arrays can be broken down into three parts: the address decoder, read/write logic, and memory cell array (Figure 2.1) [2]. A generic memory array architecture, similar to one

described by Weste and Eshraghian [4], is shown in Figure 2.2. The row decoders, column decoders, bitline selectors, and wordlines shown in Figure 2.2 form the address decoder. The bitlines, sense amplifiers, bitline precharge logic, column multiplexor (mux), write buffers, and data out lines form the read/write logic. The collection of individual memory cells form the memory cell array.

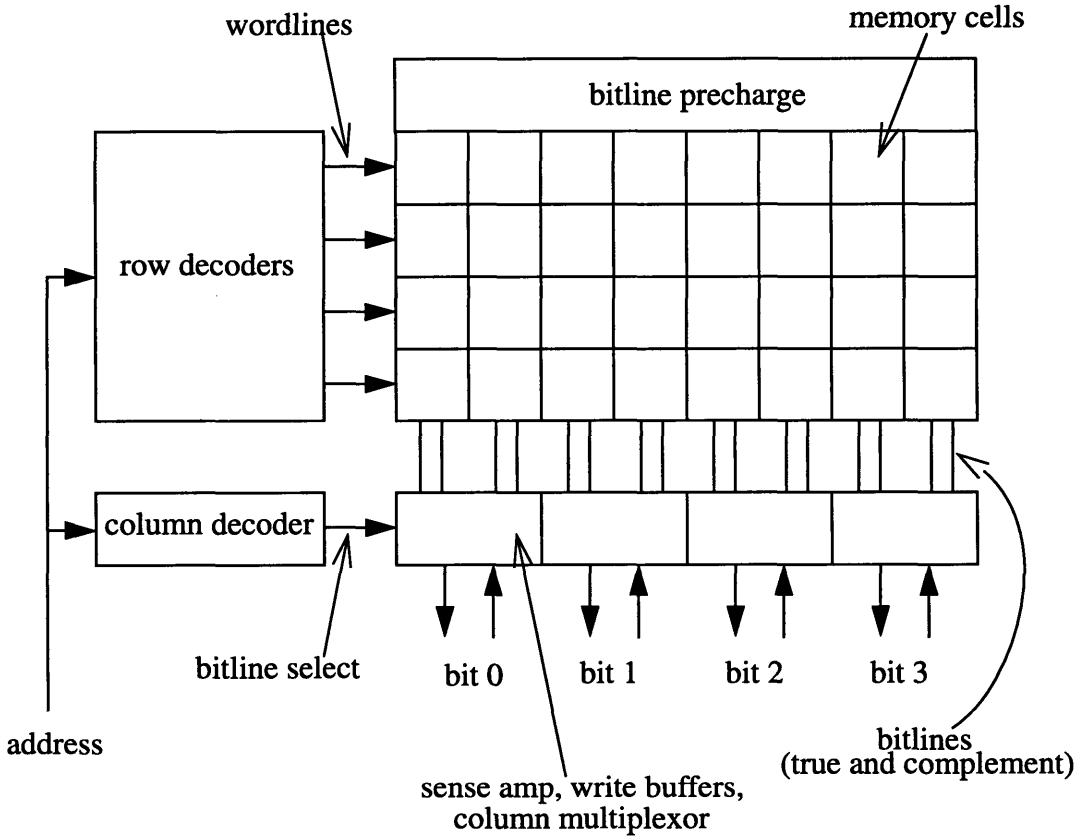


Figure 2.2: Memory array architecture

Van de Goor showed how faults in the read/write logic and address decoder can be mapped back onto the memory cell array for the purposes of test. Even though this "Reduced" model was originally developed for testing, it can also be used for diagnosis. The memory arrays on this chip were not designed with the capability to test the three parts of the memory array separately. Therefore, the memory array is essentially a black box whereby the only diagnostic information available is the requested address and the

result of a read/write on this requested address; this is equivalent to saying that the only information available is the observation of the memory cell array. The job of diagnosis is to take this information and work backwards to figure out whether the fault occurred within the address decoder, memory cell array, or read/write logic and then specifically what happened. Therefore, the "Reduced Functional Ram Chip Model" originally developed for testing, where all faults are modelled as faults in the memory cell array is also a valid model for diagnosis, where all faults are observed as faults in the memory cell array. If a fault, regardless of whether it is in the address decoder, memory cell array, or read/write logic, does not manifest itself as a fault or a set of faults in the memory cell array, it is not observable and hence not diagnosable.

This section begins by defining and describing the memory cell array faults. There will be a description of the faults being targeted in the read/write logic and the address decoder and how these faults are modelled as faults in the memory cell array. The logical faults will be discussed as well as the physical faults from which these logical faults are abstracted. The idea of scope will also be presented as well as its role in distinguishing between faults in the memory cell array, read/write logic, and address decoder. Furthermore, the idea that logical faults in the read/write logic are pattern sensitive is introduced. This means that a certain pattern of data in memory or pattern of accesses to memory sensitizes the read/write logic to certain faults, helping to expose them.

2.2 Faults in the Memory Cell Array

The memory cell array is the model for the collection of memory cells to which data is stored and from which data is retrieved. A typical memory cell from the memory cell array is shown in Figure 2.3. Initially, the precharge logic will raise both the bitline and $\overline{\text{bitline}}$ wires to VDD. On a read, if this memory cell is chosen (wordline is asserted), then either the bitline wire will begin to discharge (to read a 0) or the $\overline{\text{bitline}}$ wire will begin to dis-

charge (to read a 1). This discharge is detected by a sense amplifier which determine the value being read. On a write, if this memory cell is chosen (wordline is asserted), then either the bitline will be forced low while the $\overline{\text{bitline}}$ wire is forced high (to write a 0) or the bitline wire will be forced high while the $\overline{\text{bitline}}$ wire is forced low (to write a 1). Note that reads and writes are both assumed to occur through one port only.

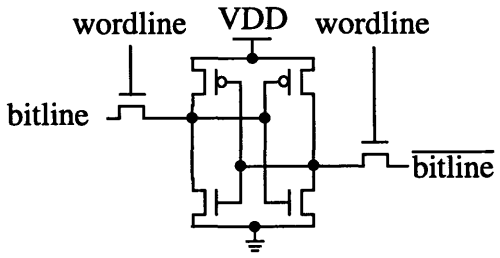


Figure 2.3: A typical CMOS SRAM cell

The five basic logical faults in the memory cell array are stuck-at faults, transition faults, coupling faults, non-deterministic faults, and destructive read faults. Each will be described in the following sections. Note that each of these logical faults may be caused by a variety of physical faults. The examples given here are not exhaustive nor guaranteed to be the most common. After the five basic logical faults have been introduced, another type of fault, a pattern sensitive fault, will be introduced and discussed. This fault is not considered one of the basic faults by this thesis. Finally, there will be a discussion of linked versus unlinked faults.

2.2.1 Stuck-at Fault

A cell that exhibits a stuck-at fault is a cell that is logically fixed at either a 0 or a 1. This cell cannot be written with another value and can only be read with one value. An example of how this can physically occur is shown in Figure 2.4. In this example, the short to VDD in "Cell A" causes the cell to be stuck in state 0. This is a stuck-at 0 fault.

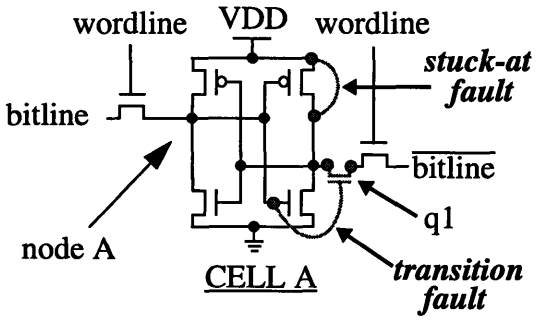


Figure 2.4: Example of stuck-at fault and transition fault

2.2.2 Transition Fault

A cell that exhibits a transition fault is unable to transition from a 0 to a 1 or from a 1 to a 0. An example of how this can physically occur is shown in "Cell A" in Figure 2.4. As demonstrated by Dekker et al, "A defect in the poly silicon layer covering a diffusion region may result in the creation of an extra pass transistor" [3]. This is the transistor, q1. The result of this defect is that the cell cannot transition from a 0 to a 1, but can transition from a 1 to a 0. Assume that node A begins with a value of VDD (the memory cell holds a value of 1). At this point, transistor q1 is on and data can pass freely into "Cell A" from the $\overline{\text{bitline}}$ wire. A value of 1 can continue to be written to and read out of "Cell A". Suppose the bitline wire then pulls low. This pulls node A to ground (the memory cell transitions from a value of 1 to a value of 0). This turns off the transistor q1. In order for "Cell A" to transition high again, a low value must be passed into the memory cell from the $\overline{\text{bitline}}$ wire; however, data can no longer enter through the $\overline{\text{bitline}}$ wire. Once "Cell A" has a value of 0, a value of 0 can continue to be written to and read out of "Cell A", but a value of 1 can be neither written to or read out of "Cell A".

2.2.3 Coupling Fault

The first two faults only involve one cell. Coupling faults are faults that involve two cells. The source cell of a coupling fault will, during a transition from 0 to 1 or from 1 to 0,

change the state of a second, target, cell. There are three different kinds of coupling faults: inversion coupling faults, idempotent coupling faults, and state-coupling faults. In an inversion coupling fault, a 0 to 1 or a 1 to 0 transition in the source cell inverts the contents of the target cell. In an idempotent coupling fault, a 0 to 1 or a 1 to 0 transition in the source cells forces the contents of the target cell to a certain value, 0 or 1. In a state-coupling fault, the state of one memory cell is directly linked to the state of another memory cell; writing a value into the first memory cell will force the other memory cell to either the same value or the inverse value.

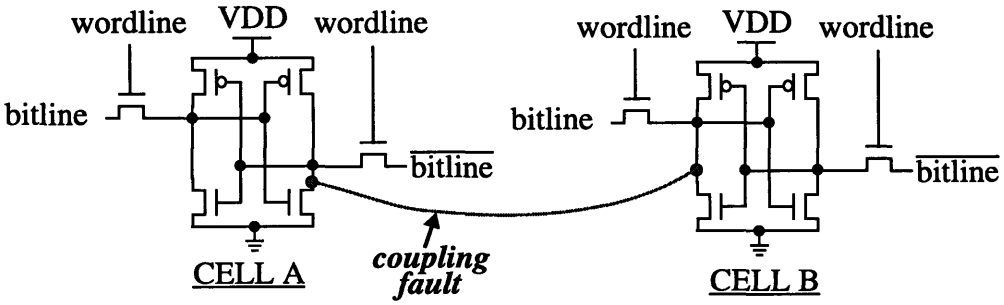


Figure 2.5: Example of coupling fault

An example of how a coupling fault can occur is shown in Figure 2.5. The result of the short between "Cell A" and "Cell B" is that when "Cell A" is written with a value of 0 (1), "Cell B" goes to the opposite value 1 (0), and when "Cell B" is written with a value of 0 (1), Cell A goes to the opposite value 1 (0). There are actually four different faults present in this example:

- a 1 to 0 transition in "Cell A" forces "Cell B" to a value of 1
- a 0 to 1 transition in "Cell A" forces "Cell B" to a value of 0
- a 1 to 0 transition in "Cell B" forces "Cell A" to a value of 1
- a 0 to 1 transition in "Cell B" forces "Cell A" to a value of 0

This is a special case of the state-coupling fault called a bilateral state-coupling fault. In this case, writing a value into either memory cell will force the other memory cell to a cer-

tain value. In a regular, unilateral, state-coupling fault, writing a value into "Cell A" forces a value into "Cell B", but writing a value into "Cell B" does not force a value into "Cell A". A bilateral state-coupling fault can be thought of as two unilateral state-coupling faults, one going from "Cell A" to "Cell B" and the other going from "Cell B" to "Cell A".

In order to make the terminology consistent, for the purpose of this paper, if cell X is said to be coupled or unilaterally state-coupled to cell Y, then cell X is the cell that exhibits the fault and cell Y is the trigger or source of the fault. Notice that in bilateral state-coupling, the coupling fault goes in both directions, so there is no difference between saying that cell X is bilaterally state-coupled to cell Y and saying that cell Y is bilaterally state-coupled to cell X.

2.2.4 Non-Deterministic Faults

There are two types of non-deterministic faults. In the first case, the state of a memory cell is completely non-deterministic. Stuck-open faults, referred to in some of the literature, fall under this category [2] [3]. This means that the contents of the memory cell can not be predicted ahead of time. Any read of a cell exhibiting this fault may be entirely random, may be always 0, or may always be 1. In the second case, the state of the memory is partially non-deterministic. In this case, under certain circumstances the memory cell behaves correctly and under other circumstances the memory cell becomes non-deterministic. For instance, writing a 1 into the memory cell will result in a 1 being read back out, but writing a 0 into the memory cell will result in an unpredictable result being read back out.

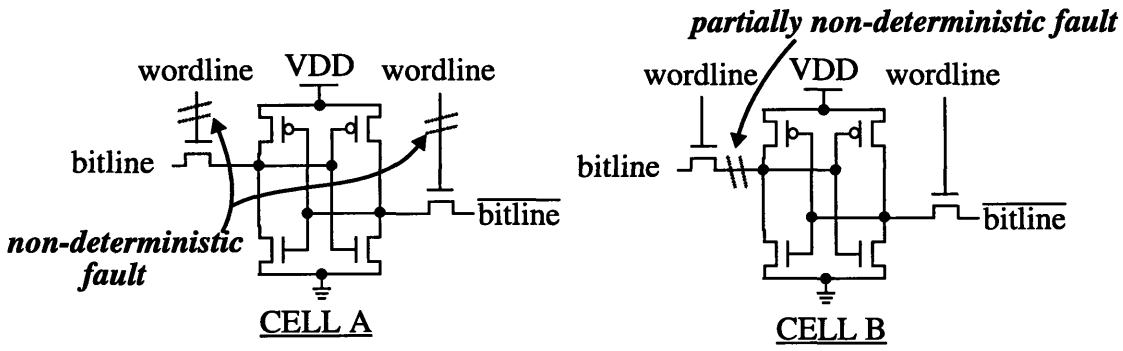


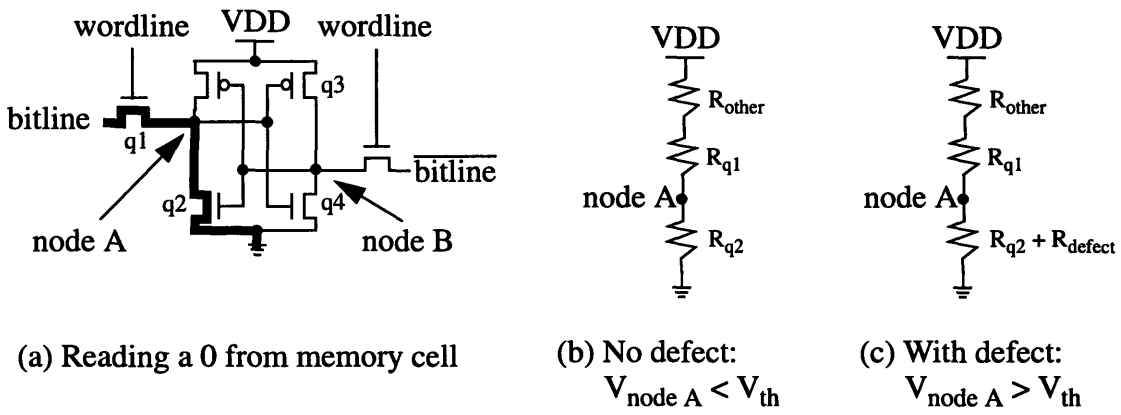
Figure 2.6: Examples of non-deterministic faults

An example of how a regular non-deterministic fault can manifest itself is shown in "Cell A" of Figure 2.6. The two breaks in the wordline will isolate the memory cell, rendering it inaccessible. The cell can be neither read from nor written to successfully. During a normal read of a memory cell, the true and complement bitline wires are initially precharged high. Then, the memory cell will pull down either the true or the complement side. The sense amplifier is then able to determine whether the cell has a value of 0 or 1. With an isolated cell, as in this example, neither the true nor the complement side will be pulled low. The sense amplifier will be unable to decide whether the cell is at a 0 or a 1. Sense amplifiers are generally designed without a preference for either a 0 or a 1 and therefore the cell will be read with a non-deterministic value. Though the exact value cannot be predicted, the actual value does depends on the final manufactured sense amplifier. Once a sense amplifier is manufactured, it may, due to process inconsistencies, show a preference for one value or another when presented with the same value from both the true and complement wires. This may result in a non-deterministic fault appearing as a stuck-at fault. This should be further qualified by noting that, depending on the noise in the system, the sense amplifier may not always be stuck-at the same value. Generally, non-deterministic faults rarely happen to one cell on its own. For instance, a broken wordline will affect all the memory cells that lie along that broken wordline.

An example of a partially non-deterministic fault is shown in "Cell B" of Figure 2.6. In this example, the true bitline has a break. However, the complement bitline is still operational, and the correct value can still be written into and read from that wire. The complement bitline can still be pulled low by the memory cell, so a read of 1 will work normally. However, the true bitline cannot be pulled low, so a read of 0 will not work normally. Again, this value cannot be predicted in advance. A read of 0 may result in a random value, a 0 (in which case the fault is completely masked), or a 1 (in which case, the memory cell appears to be stuck-at 1). Partially non-deterministic faults also rarely happen to only one cell. A break in a bitline will affect all memory cells that lie along that broken bitline.

2.2.5 Destructive Read Fault

A cell that exhibits a destructive read fault will, during a read operation, change the state of the cell being read. An example of how this happens is shown in Figure 2.7 [5]. Assuming the memory cell begins in state 0, a read of the memory cell should yield a value of 0. On a read of a normal cell (Figure 2.7a), the two transistors, q1 and q2, are on, and node A is at some value below the threshold voltage, V_{th} (Figure 2.7b). This keeps transistor q3 on and transistor q4 off, leaving the memory cell in state 0. There exists the possibility that a defect can cause the resistance of transistor q2 to be greater than normal. On a read of a defective memory cell, the two transistors, q1 and q2, are again on, but node A is now at some value greater than V_{th} (Figure 2.7c). This causes transistor q4 to turn on. This pulls down node B enough to forces the memory cell to flip state. Another read of this memory cell will yield an incorrect value of 1.



(a) Reading a 0 from memory cell

(b) No defect: $V_{node\ A} < V_{th}$

(c) With defect: $V_{node\ A} > V_{th}$

Figure 2.7: Destructive read fault

2.2.6 Pattern Sensitive Faults

In the literature [1] [2], there exists another fault called a pattern sensitive fault. The definition of a pattern sensitive fault is as follows: "The contents of a cell, or the ability to change the contents, is influenced by the contents of all other cells in the memory" [2]. The idea is that some pattern of data in the memory (i.e., a checkerboard pattern) will stress the array such that a certain cell is forced to an incorrect value. This paper argues that pattern sensitive faults are not a basic fault; the pattern is a means of exposing the fault, but the underlying fault lies elsewhere (in the read/write logic, for instance). For this project, many of the patterns being used were used to expose faults in the read/write logic. In reality, it is possible that a particular pattern of data around a cell (for example, Figure 2.8) will cause only a single cell (the middle cell in Figure 2.8) to an incorrect state, but this kind of linked fault is too complex to thoroughly test for and will not be considered in this thesis (see Section 2.2.7).

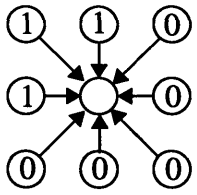


Figure 2.8: Example of complex pattern sensitive fault

2.2.7 Linked versus Unlinked faults

An important simplification in this thesis is that not all linked faults are addressed. A linked fault is a fault which influences the behavior of other faults. There are two types of linked faults (Figure 2.9). The first is where more than one fault affects the same cell (Figure 2.9a). The second is where one fault can propagate to cause another fault (Figure 2.9b). Including linked faults greatly increases the complexity of detecting and diagnosing for faults and will not be discussed. Including linked faults allows for the possibility that one fault can mask another and even detecting certain types of linked faults is impossible using march tests [2]. March tests are tests composed of a finite series of operations performed on every memory cell in order. They will be discussed in more detail in Chapter 3.

(a) Two faults affecting the same cell

(b) Propagating faults

Figure 2.9: Example of linked faults

One class of linked faults will be allowed. These are state-coupling faults. A description of state-coupled cells was given in section 2.2.3. The examples in Figure 2.9 all involve three cells. State-coupling faults are linked faults involving only two cells. This is

a case where two faults exist who share the same source and target (Figure 2.10). Writing both a 0 and a 1 to cell A forces a value into cell B.

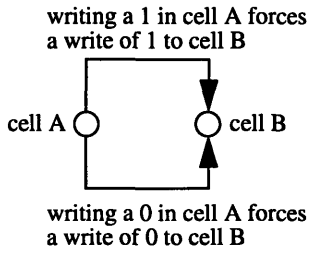


Figure 2.10: Unilateral state-coupling fault.

2.3 Faults in the Read/Write Logic

The read/write logic is the model for the bitlines, sense amplifiers, bitline precharge logic, column mux, write buffers, and data out lines in the memory array (Figure 2.2). The job of this logic is to write data into memory and to read data back out from memory. This section describes the different faults that are being targeted in the read/write logic and how they can be mapped onto faults in the memory cell array. The main idea is that while faults in the memory cell array will be manifested as individual memory cell faults, faults in the read/write logic will be manifested as a group of memory cell faults in a bit (i.e., a column of faults, a partial column of faults, or a group of columns of faults). A fault in a memory cell affects only that cell; a fault in the read/write logic will affect the group of memory cells that are connected to the faulty logic.

A general diagram of the read/write logic with the faults under consideration is shown in Figure 2.11 [5]. The numbered circles represent the faults. An important concept is that of the scope of a fault. The scope of a fault is the set of memory cells that are influenced by that fault. For instance, a fault on bitline 1 (as in #2 and #3 in Figure 2.11) will affect all the memory cells that lie along that bitline; therefore, the scope of this bitline 1 fault is the column of memory cells that lie along bitline 1. The scope of a sense amp fault are the two columns of memory cells that lie along bitline 1 and bitline 2. In general, the positioning

of the sense amps, bitline precharge, column mux, and write buffer, as well as the level of multiplexing that occurs (and therefore the scope of faults in the mentioned components) varies upon implementation, but they are conceptually similar to the model presented here. Though the individual memory cell faults that occur within the set of cells in the scope of the fault is dependent on the implementation, the idea that these faults will occur within a certain set of memory cells (the scope) is an important first step in diagnosis because it begins to narrow down the set of possible faults. The section on testing will then show how, by choosing the proper test patterns, the set of possible faults can be further narrowed down.

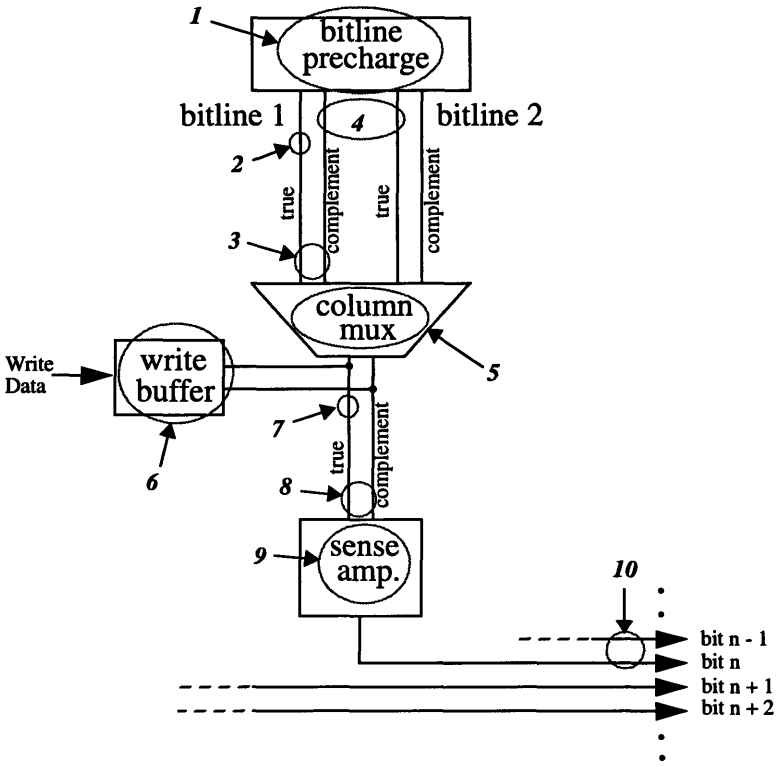


Figure 2.11: Faults in the read/write logic

As mentioned in an earlier section, pattern sensitive faults are better characterized as a means of exposing faults in the read/write logic rather than as actual faults themselves. Depending on the implementation of memory, faults in the read/write logic are complex

and their behavior is not entirely understood. However, certain patterns of data in memory or certain patterns of accesses to memory can expose the faults.

2.3.1 Bitline Precharge Faults

The bitline precharge logic is responsible for pulling up the bitlines before each memory read. The scope of the bitline precharge faults (#1) is one column of memory cells. A fault in the bitline precharge logic (#1 in Figure 2.11) can manifest itself as a temporary stuck-at fault in the memory cells in the scope of the bitline precharge [5]. These cells will be stuck-at the value of the previous write. Figure 2.12 shows the values of the true and complement wires during two consecutive memory access to two memory cells that share the same precharge logic. With a working precharge, the complement wire is precharged high after the write operation so that a successful read of a 0 can occur (Figure 2.12a). Assuming that there is a faulty precharge on the complement wire, the complement wire is not precharged correctly after the write operation, and an incorrect read of a 1 results (Figure 2.12b). Notice how this pattern of memory accesses serve to expose the fault.

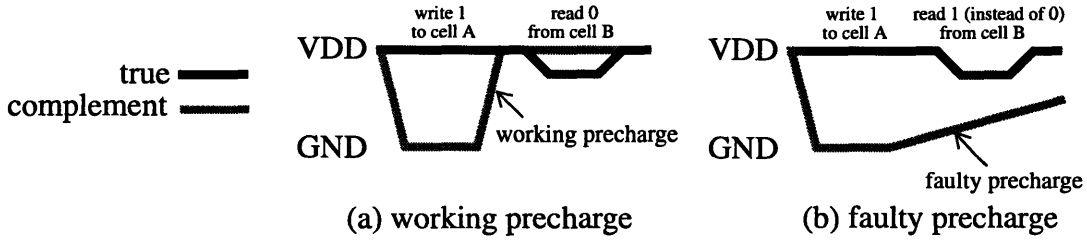


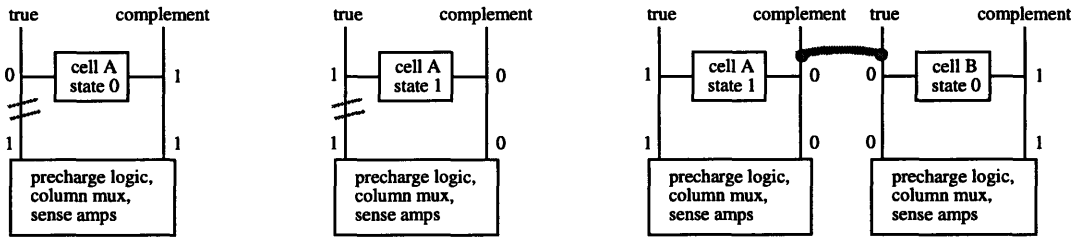
Figure 2.12: Bitline precharge fault on complement wire

2.3.2 Bitline Faults

The bitline is a wire that carries true and complement data to and from memory cells. There are essentially three types of bitline faults. The first two are bitline faults that only affects one bitline (#2 and #3 in Figure 2.11). This could be caused by a break in either the true or complement wire of the bitline (#2 in Figure 2.11) or a short between the true and complement wires of the bitline (#3 in Figure 2.11). The scope of this fault is the set of memory cells that lie along this faulty bitline. The specific details of how these faults will

manifest themselves depend on the implementation to some extent, but the idea that faults will manifest themselves within the scope of the faulty bitline is still useful.

For instance, with a bitline break in an implementation with the bitline precharge on the same side as the column muxes, it is possible that the memory cells before the break (i.e., closer to the sense amps) will still function properly. However, the cells after the break are inaccessible and will all exhibit a partially non-deterministic fault (Figure 2.13a). This figure demonstrates how a cell with a partially non-deterministic fault can be successfully read from and written to from one side (complement), but not from the other (true).



(a) fault #2, partially non-deterministic fault

(b) fault #4 coupling fault

Figure 2.13: Bitline faults

With the second type of bitline fault, a short across the true and complement wire of the same bitline, both wires will always be at the same value. The memory cells will be unable to decide what value is being written and the sense amplifier will be unable to determine the value being read. Therefore, this is a non-deterministic fault.

The third type of bitline fault is a short across different bitlines (i.e., #4 in Figure 2.11, a short between a complement wire of bitline 1 and the true wire of bitline 2). The scope of this fault is set of cells along the two bitlines involved. This fault will manifest itself as two columns that are coupled to each other. As demonstrated in Figure 2.13b, a write of a 1 to cell A will force cell B to a 0 and a write of a 0 to cell B will force cell A to a 1. This pattern of data in the memory will cause the fault to be exposed. Note that the layout of the

memory may be different from that shown in Figure 2.13b. There may also be the case where the short lies between a complement and a complement or between a true and a true. The coupling faults need to be changed accordingly, but the scope of this fault is still the same

2.3.3 Column Multiplexor Faults

The column multiplexor is responsible for selecting between bitlines (i.e., between bitline 1 and bitline 2 in Figure 2.11). An example of how the column multiplexor might be designed and some of the possible faults is shown in Figure 2.14. In general, the scope of a column multiplexor fault is the set of memory cells that lie along the bitlines which are inputs to the faulty column multiplexor.

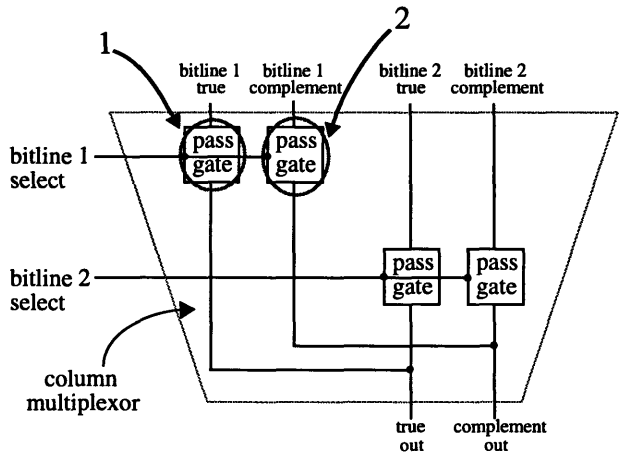


Figure 2.14: Column multiplexor faults

One type of fault that might occur is if one of the pass-gates is stuck open (#1 in Figure 2.14). In this case, data going down the true side of bitline 1 will never go through the pass-gate. The complement side will still work, so this will cause a partially non-deterministic fault all along bitline 1.

Another type of fault is if a pass-gate is always stuck-closed (#2 in Figure 2.14). In this case, regardless of which bitline has been selected, data will always pass through from the bitline 1 complement wire. There are two possible faults. The first fault is that bitline 1 is state-coupled to bitline 2. All writes to bitline 2 will force a write to bitline 1. The second

fault is a partially non-deterministic fault in bitline 2. During a read of bitline 2, the faulty pass-gate in bitline 1 (#2) will pass the value of the bitline 1 complement wire to the complement out wire. When the bitline 1 complement wire is trying to pass conflicting data from the bitline 2 complement wire, a fault can occur. The result is non-deterministic.

2.3.4 Data-in Faults

The write buffer is the component responsible for passing write data to the memory cells. If the data-in logic has a fault (#6 in Figure 2.11), the scope of this fault is every cell whose data is written by this logic (i.e., all the memory cells that lie along bitlines 1 and 2 in Figure 2.11). Therefore, a fault that exists in the data-in logic will cause all the memory cells along bitlines 1 and 2 to experience the same faults. For instance, if the write buffer is stuck-at 0, all memory cells in this bit will be stuck-at 0.

2.3.5 Sense Amplifier Faults

Faults #7, #8, and #9 in Figure 2.11 are all considered sense amplifier faults. Either the inputs to the sense amplifier (#7, #8) or the sense amplifier itself (#9) has a fault. The sense amplifier takes a true and complement wire and decides whether the memory cell being read is a 0 or a 1. The scope of sense amplifier faults is the set of all memory cells which are read with that sense amplifier. This is a fault that will cause all the memory cells in this bit to experience the same fault; if the sense amplifier has a stuck-at 0 fault, all memory cells will exhibit a stuck-at 0 fault. The sense amplifier is sensitive to the noise of neighboring sense amps. A fault may occur due to this noise. This fault may be manifested if all sense amps are trying to read the same value, all 1s for example.

2.3.6 Data-out Faults

There are two types of data-out faults. In the first case, the data-out logic exhibits a stuck-at fault. This fault manifests itself as a bit whose cells are all stuck-at 1 or stuck at 0. The scope of this fault is the set of memory cells in the bit.

The second type of data-out fault is a short between two output wires (#10 in Figure 2.11). The scope of this fault is the set of all memory cells whose value are read through those two wires. In Figure 2.11, the scope is set of memory cells along bitline 1 and bitline 2 for both bit n and bit n-1. This fault is only manifested if bit n and bit n-1 are different values. In this case, one of the values will be coupled to the other. If they are both 1s or both 0s, the fault will be masked.

2.4 Faults in the Address Decoder

The address decoder is the model for the row decoder, wordlines, bitlines, and column decoder (Figure 2.2). The address decoder is responsible for translating the lower order bits of the address into a particular choice of a wordline (row decoder) and for translating the higher order bits of the address into a particular choice of a bitline (column decoder). In this fashion, for any bit of data, a unique memory cell is accessed for each address. Faults in the address decoder all involve incorrect mapping of addresses to memory cells. As demonstrated by van de Goor, there are four basic types of logical address decoder faults [2] (Figure 2.15).

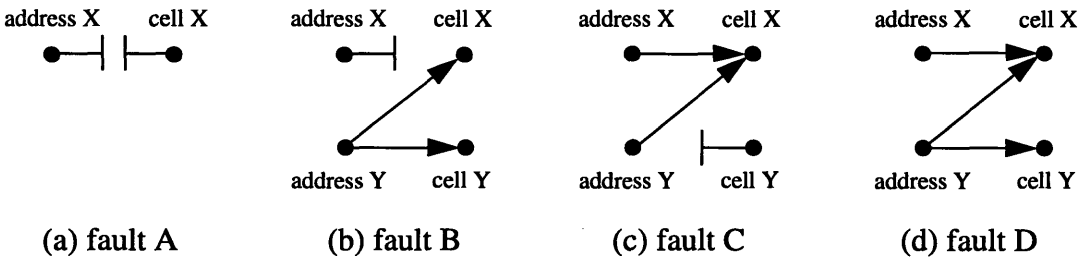


Figure 2.15: Logical address decoder faults

However, real address decoder faults generally affect all memory cells along a particular wordline or bitline. Therefore, the "cell X (Y)" labels in Figure 2.15 can be replaced with "wordline X (Y)" for row decoder faults and "bitline X (Y)" for column decoder faults. In reality, this is how the faults happen; an incorrect wordline or bitline is selected, or the correct wordline or bitline is selected, but for the wrong address. This will cause all the

cells along these wordlines or bitlines to appear faulty. An example of how each of these faults might physically occur in a row decoder is shown in Figure 2.16. The design of this row decoder is one of the simplest decoders presented by Weste and Eshraghian[4]. Note that there are a variety of possible decoder designs. This further illustrates why it is easier to use a logical fault model rather than a physical fault model.

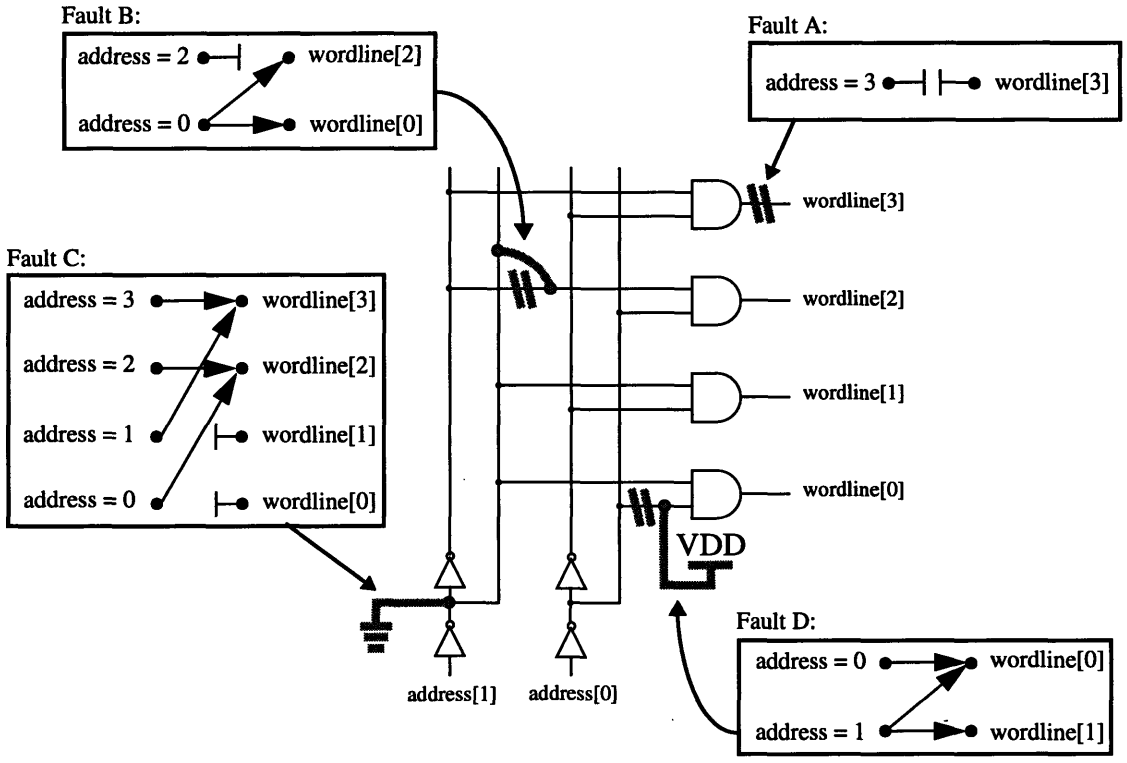


Figure 2.16: Physical address decoder faults

In general, the address decoder logic is shared across many and sometimes all bits. To recap, the scope of memory cell array faults can be either one cell or a pair of cells. The scope of read/write logic faults can be a set of memory cells from the group of cells that lie within one bit. The scope of address decoder faults, because the address decoder is shared among different bits, can be from the set of all memory cells (Figure 2.17). This is not to say that the faults will exist everywhere: an address decoder fault might result in a row of

faults across all bits or it might result in a column of faults that exist in the first column of every bit.

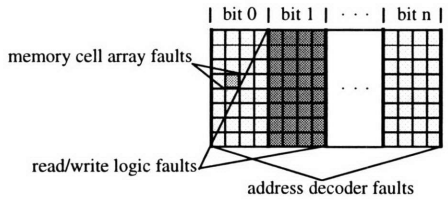


Figure 2.17: Range of possible scopes of different faults

2.4.1 Row Decoder Faults

The row decoder is responsible for selecting a mapping of addresses to a particular wordline. A row decoder fault results in an incorrect wordline being activated or the correct wordline is not activated. The scope of faults will be the set of memory cells along the faulty wordline(s).

Fault A (Figure 2.15a) results in a row of memory cells (by which I mean the set of memory cells that lie along one wordline) which all exhibit non-deterministic faults. The cells along wordline X are completely inaccessible.

Fault B (Figure 2.15b), though a different fault, is indistinguishable from fault A. In Fault B, address X is inaccessible and thus non-deterministic. Therefore, even though the cells along wordline X are coupled to the cells along wordline Y, this coupling fault is masked because of the non-deterministic fault. Similarly, reads from address Y will not show a problem because the cells along wordline X and along wordline Y will always be at the same value.

Fault C (Figure 2.15c) results in bilateral state-coupling between the cells along wordline X and along wordline Y. All accesses to address X and to address Y activate the same wordline. A write of a value in address X will appear in a read of address Y and vice versa.

Fault D (Figure 2.15d) results in two faults. In one row of faults, the cells accessed by address X all exhibit a unilateral coupling fault with the cells accessed by address Y. All

writes to cells accessed by address Y will force a value to cells accessed by address X. The row of cells accessed by address Y are more difficult to test. In the event that cell Y and cell X have identical values, no errors will be detected. However, in the case that cell X and cell Y have different values, the sense amplifier will be unable to decide the result of a read of address Y. This is equivalent to a non-deterministic fault, though only when the data in cell X and cell Y are different.

2.4.2 Column Decoder Faults

The column decoder is responsible for selecting a mapping of addresses to a particular choice of a bitline. Therefore the scope of column decoder faults is the columns of memory cells which are influenced by that choice of bitline. Note that this is different from the column mux faults in the read/write logic. While the column mux faults only influences one particular bitline (e.g. bitline 1 of bit 5), column decoder faults influence a bitline over all bits (e.g. bitline 1 across bits 1 through n). Note that these faults are just like the row decoder faults, except that instead of a row of faults, there are columns of faults.

Fault A results in a certain column of memory cells (by which I mean the set of memory cells that lie along one bitline) across many bits which are inaccessible and thus exhibit non-deterministic faults. As an example, the scope of fault A might be that bitline 1 is non-deterministic for all bits.

For the same reason as that mentioned in section Figure 2.4.1, fault B is indistinguishable from fault A. The scope is one column of non-deterministic faults across all bits.

Fault C results in bilateral state-coupling between the cells along bitline X and along bitline Y. The scope is thus two columns of faults across all bits.

Fault D results in two faults. The cells along bitline X, across all bits, will exhibit a unilateral-lateral state-coupling fault with the cells along bitline Y. Testing the column of cells along bitline Y is further complicated because in the event that cell Y and cell X have

identical values, no errors will be detected. However, in the case that cell X and cell Y have different data, the sense amplifier will be unable to decide the result of a read of address Y. This is equivalent to a non-deterministic fault, but only when the data in cell X and cell Y are different. The scope is two columns of faults across all bits.

2.4.3 Wordline Faults

The wordline is the wire that connects the row decoder to the pass-gates of the memory cells. The faults being targeted are wordline breaks and wordline stuck-at faults.

In the case of wordline breaks, all the rows of memory cells beyond the break away from the row decoder will be inaccessible and thus non-deterministic. The memory cells before the break and closer to the row decoder will still function normally. Unlike a regular row decoder fault, the wordline fault only occurs over some subset of all the memory cells that lie along that wordline.

Assuming that the logic is active high, when a wordline is stuck-at 0, that particular wordline is never active. This is equivalent to fault A in the decoder logic (see Figure 2.16). A wordline that is stuck-at 1 is always active. This is a special case of fault D. If a wordline X is stuck-at 1, then regardless of what other wordline should be accessed, wordline X will also be active. This is demonstrated in Figure 2.18. In the case the logic is active low, the same reasoning leads to a correspondence between wordline stuck-at 1 faults and fault A. Similarly, wordline stuck-at 0 faults correspond to the special case of fault D shown in Figure 2.18.

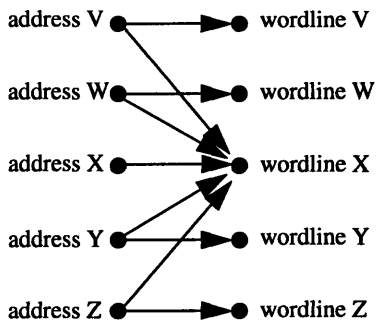


Figure 2.18: Wordline stuck-at 1

2.4.4 Bitline Select Faults

The bitline select wire is the wire that connects the column decoders to the column multiplexors. The targeted faults in the bitline select wire are bitline select breaks and bitline select stuck-at faults.

In the case of a bitline select break, the column multiplexors before the break will behave correctly, but the column multiplexors after the break will be stuck open and thus non-deterministic. Again, unlike a regular column decoder fault, the bitline select fault only occurs over some subset of the memory cells that lie in the bits away from the column decoders and on the other side of the break.

The description of the bitline select breaks is exactly identical to that of the wordline breaks. Assuming that the logic is active high, when a bitline select is stuck-at 0, that particular bitline select is never active. This is equivalent to fault A in the decoder logic (see Figure 2.16). On the other hand, a bitline select that is stuck-at 1 is always active. This is a special case of fault D. If a bitline select X is stuck-at 1, then regardless of what other bitline select should be accessed, bitline select X will also be active. This is demonstrated in Figure 2.18. In the case the logic is active low, the same reasoning leads to a correspon-

dence between bitline select stuck-at 1 faults and fault A. Similarly, bitline select stuck-at 0 faults correspond to the special case of fault D shown in Figure 2.18.

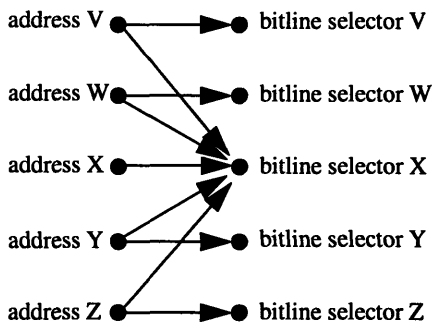


Figure 2.19: Bitline select stuck-at

2.5 Faults that are not caught

There are some faults that are not considered in this fault model. They may not be caught and their diagnosis will not be described. A major group of faults that are not caught are dynamic faults. These faults are time dependent and, though they will show up as functional faults, they cannot be described properly under the fault model outlined in this chapter because these faults do not take timing into account [7]. Examples of these types of faults are recovery faults, retention faults, and imbalance faults [7].

A second type of fault that is not caught is a coupling fault between the same address of different bits. In the memories considered in this study, on a read of an address X, an entire word of many bits is read out simultaneously. On a write of an address X, an entire word of many bits is written in simultaneously. The bits accessed by address X are accessed simultaneously; therefore, if coupling faults exist between these bits, then they may not be detectable. The bits may not be accessed at the same exact time and there is no way of guaranteeing that the source cell for the coupling fault will be accessed before the target cell. One cannot selectively decide which bit to access first.

Chapter 3

Test

3.1 Introduction

Testing is the process of exercising and evaluating a system to determine whether its behavior is correct. The goal is to determine whether the final manufactured chip behaves according to its design. There are two main forces driving the design of test algorithms. The first is a desire to keep the tests as short as possible. As memories continue to increase in size, so too does the time it takes to conduct the test. Furthermore, there is a fixed amount of memory that the test machinery has; the more memory each test takes up, the less memory is available for other tests. The second major force is a desire to maintain or increase test coverage. Traditionally, there have been a number of industry test algorithms which were not based on a fault model [6]. These traditional tests are generally longer than the newer algorithms being developed. However, there is always the possibility that changing the test algorithm may decrease the test coverage unless the test algorithm has been empirically tested and it's fault coverage compared with that of the traditional tests. There may be a trade-off between the test's length and the test's ability to catch all faults.

For the purpose of this project, three test algorithms were chosen: unique address ripple word test, word line stripe test, and checkerboard test. As a project that was performed in an industrial setting, this project uses algorithms that have evolved from the traditional tests. In academia, papers listing a number of more efficient tests have been proposed (for instance, in [2] [3] [4]), but they do not catch some of the faults targeted in this project. For instance, none test for a destructive read fault. Part of the goal of this thesis is to use these existing tests and decide how they can be used or modified for diagnosis of memory arrays.

The chapter begins by describing the three test algorithms. This is followed by a discussion on the implementation of these algorithms and the issues that arise when they are used for diagnosis.

3.2 The Tests

The test algorithms that have been chosen for this project are the following three march tests: unique address ripple word test, word line stripe test, and checkerboard test. The diagnostic capabilities of these tests will be discussed in the following chapter. This section begins with an overview of march tests and an example of the type of memory that these tests might be performed on.

Every march test is composed of march elements. Each march element is composed of a series of operations that are done to every memory cell before proceeding to the next operation [6]. These operations can be a combination of reading a 0 (r0), reading a 1 (r1), writing a 0 (w0), or writing a 1 (w1). Each march element can proceed in a forward direction through all the cells ($\Uparrow$) or in the opposite reverse direction ($\Downarrow$). Note that the exact order of the addresses is not important, so long as the order of the cells going in the forward direction is exactly opposite of the cells going in the reverse direction.

Four bits from a sample memory is given in Figure 3.1. This memory is the example that will be used for discussions in this thesis. The memory shown is for a cache which is two way associative. Furthermore, each way is separated into low and high addresses. Low addresses run from 0 (bottom) to 7 (top) while high addresses run from 8 (bottom) to 15 (top). The individual memory cells are labelled with their respective addresses. The highest order bit of the address is used to select between the low and high addresses. The lower order three bits are used to select one of the eight rows.

Ideally, a march test would proceed through each of the individual addresses, one bit at a time. Practically, however, the same address in every bit is accessed simultaneously. Memory is generally designed to retrieve a complete word from an address y (for example, a word at way B address 13 corresponds to the shaded boxes in the Figure 3.1), and therefore retrieves all bits simultaneously.

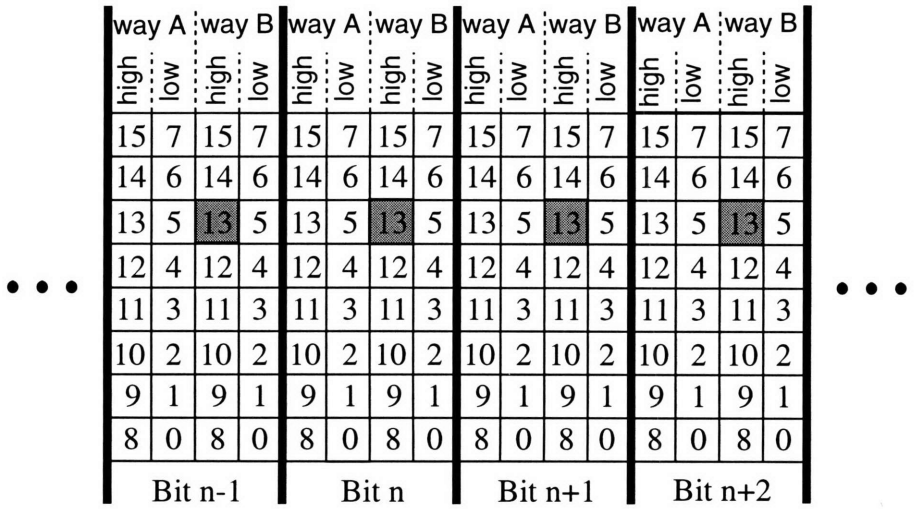


Figure 3.1: Example of memory

As mentioned earlier, it is not important that march tests follow the numerical order of the addresses, just that march elements that go forward (⇑) proceed in the opposite order from march elements that go backwards (⇓). For this project, a march element that goes forward (⇑) starts at the lower left corner of each bit, address 8 of way A high, works up the way A high column to address 15, then jumps to address 0 of way A low and works up the way A low column to address 7. The test then proceeds in a similar manner up the way B high column and finally up the way B low column. The last memory cell addressed going forward (⇑) is the top cell in way B low, address 7. A march element going in the backwards (⇓) direction goes in the reverse order, starting from the top cell of way B low (address 7), working down the way B low column, down the way B high column, down the way A low column, and finally down the way A high column. The last memory cell addressed going backwards (⇓) is the bottom cell of way A high (address 8).

3.2.1 Unique Address Ripple Word Test

The unique address ripple word test's algorithm is presented below. Each of the numbered steps correspond to a march element. For example, in step 3, this tests proceeds in a forward ($\Uparrow$) direction through each address. All the cells at each address (remembering that the same address of every bit is accessed simultaneously) is read (with a value of 1 expected), written with a 0, read again (with a value of 0 expected), and finally written with another 0. Once these four operations have finished, the test proceeds to the next address (in $\Uparrow$ order).

1. $\Uparrow$w0
2. $\Uparrow$r0w1r1w1
3. $\Uparrow$r1w0r0w0
4. $\Uparrow$r0
5. $\Downarrow$w1
6. $\Downarrow$r1w0r0w0
7. $\Downarrow$r0w1r1w1
8. $\Downarrow$r1

3.2.2 Word Lines Stripe Test

The word line stripe test's algorithm writes rows of 0s and 1s to consecutive addresses. Data will be written into the memory array such that we end up with the pattern shown in Figure 3.2.

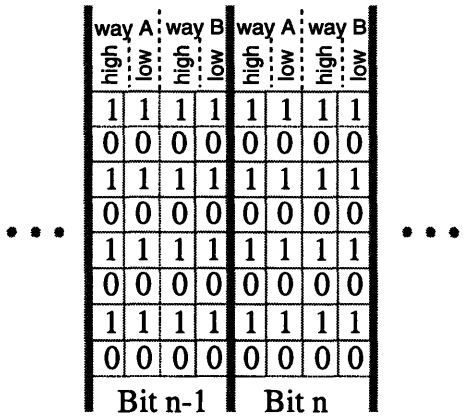


Figure 3.2: Result of word line stripe test

This data is then read and the opposite pattern (where in Figure 3.2, the 0s are replaced with 1s and the 1s replaced with 0s) is written into memory. In the last step, this new data

is read out. For the given memory implementation, this march test is shown below:

1. ⇑w0 (even addresses) and w1 (odd addresses)
2. ⇑r0 (even addresses) and r1 (odd addresses)
3. ⇑w1 (even addresses) and w0 (odd addresses)
4. ⇑r1 (even addresses) and r0 (odd addresses)

3.2.3 Checkerboard Test

The checkerboard test writes a physical checkerboard pattern of 0s and 1s into memory (see Figure 3.3).

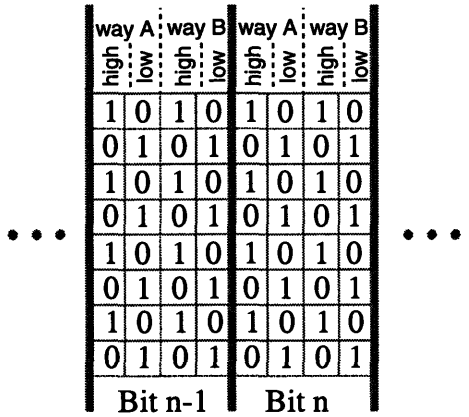


Figure 3.3: Result of checkerboard test

This pattern is read twice and then the opposite pattern (where in Figure 3.3, the 0s are replaced with 1s and the 1s replaced with 0s) is written into memory. In the last step, this new pattern is also read out twice. For the given memory implementation, this march test is shown below:

1. ⇑w0 (write 0s to even high and odd low addresses) and w1 (write 1s to even low and odd high addresses)
2. ⇑r0r0 (read 0s twice from even high and odd low addresses) and r1r1 (read 1s twice from even low/odd high addresses)
3. ⇑w1 (write 1s to even high and odd low addresses) and w0 (write 0s to even low and odd high addresses)
4. ⇑r1r1 (read 1s twice from even high and odd low addresses) and r0r0 (read 0s twice from even low and odd high addresses)

Note that this algorithm will vary depending on the implementation. The goal is to write a "physical" checkerboard pattern into memory. The algorithm shown above is not identical for all memory implementations. For this memory implementation, reading

address 0 in Figure 3.3 yields a word that is all 0s. If there is only one way and only eight addresses in memory (all in the same column), reading address 0 could yield a word that is 010101.... An example of this type of memory is shown in Figure 3.4.

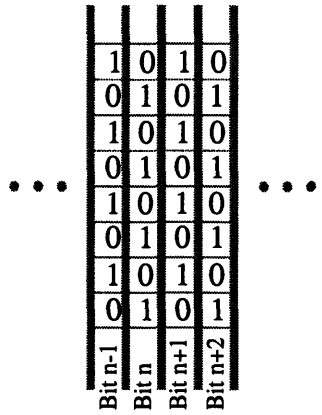


Figure 3.4: Result of checkerboard under alternative memory implementation

3.3 Implementation

The test algorithms were implemented as assembly programs. As mentioned earlier, there is a desire to keep the tests short so that the tests take up less tester buffer memory. This project involved the test and diagnosis of memory arrays embedded in a microprocessor. One way of shortening the tests was to take advantage of the microprocessor's functionality: these tests were run with the cache enabled. This way, the test program can be fetched into the instruction cache and the test run from within the processor rather than with continuous explicit direction by the tester (see Appendix A for data on efforts to shorten the tests on this project).

March tests can be used, not just to catch faults, but to also diagnose faults. This is accomplished by examining the fault signature of a fault due to the march tests: How does the march test fail due to a given fault? Fault signatures will be described in more detail in the next chapter. In order for the tests to support diagnosis, they have to be written to satisfy some requirements. First, there are requirements that must be met for the results of the tests to be observable. Secondly, there are requirements that must be met in order to satisfy

the tester equipment. The results of this test can then be used to generate a failure bitmap. A failure bitmap is a graphical representation of the memory with all the faults marked.

3.3.1 Observability

Diagnosis cannot occur unless the results of the tests are observable; the test must generate a complete picture of what is happening internally. As the test marches through the addresses writing and reading data, the results of each read must be put on the pins of the chip where it can be monitored off-chip. Furthermore, tests written for diagnosis cannot merely halt once an incorrect value is read from memory. These tests need to continue to run in order to present a complete picture of what is happening in the memory. Lastly, the tests have to provide a unique label for each read in order to identify what address has been read and what step of the test is being performed. Note that one way of doing this is to know which clock cycle corresponds to which address of which step of the test. However, this is complex. An easier solution is to present this information to the output pins of the chip as a tag along with the results of the read.

In this project, every read of a memory cell was presented to the output pins as a write to main memory. No comparison is ever done; regardless of whether the program is reading a 0 or a 1 from a memory cell, the output is presented to the bus without any decision-making on the part of the program as to whether the value read is correct or incorrect. The memory cell being read and the step number were encoded in the address of this write to main memory. This information will therefore appear off-chip as a memory request that the tester can monitor. For instance, for the two way, 16 address memory in Figure 3.1, this information can be encoded as shown in Figure 3.5. Bits 8:6 provide information about which step of the test is being performed. Bit 5 is the read number (the first or sec-

ond read of the step given by bits 8:6). Bit 4 is the way. Bits 3:0 correspond to the address (from 0 to 15).

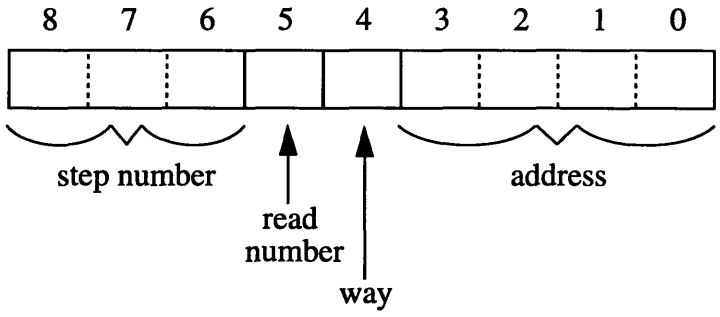


Figure 3.5: Information encoding

3.3.2 Tester Idiosyncrasies

Most general purpose testers have certain idiosyncrasies. Testers only know what data to apply on the pins of the chip and what data they should expect off the pins of the chip. The exact timing and voltages of this data can be very tightly controlled. However, the tester is not intelligent in the sense that it cannot adjust the data it applies to the chip based on the data it receives from the chip. To generate tester patterns, the test (a program) is simulated on a "good" model of the chip. The bus traces from this simulation are used by the tester as the signals it applies to and as the signals it expects from the chip. Therefore, if the program branches to X on the "good" model, but, due to a fault in the chip, the program branches to Y on the chip being tested, the tester will continue to supply signals to the chip as if a branch to X occurred because that is what happened on the "good" model. A problem is detected when the tester notices that the chip's outputs mismatches with the "good" model outputs. This is adequate for go/no go testing where on the first unsuccessfully branch on the chip, the tester will report a mismatch and stop. However, for diagnosis, further information is needed from the rest of the memory array in order to build a complete failure bitmap of the memory.

These tests were written so that regardless of the number of fails, the code does the same thing. This is guaranteed if the code is branch-free. As noted, no comparisons are

done in the tests. The values are merely read out of the memory array and placed on the bus.

3.3.3 Failure Bitmaps

Once the test algorithm has been performed on a memory array, it is useful to generate a failure bitmap. A failure bitmap is a graphical depiction of the memory array (for instance, as in Figure 3.1) with the faults shown right on the array. The next chapter will show how the different faults appear (their fault signatures). By matching the generated failure bitmap with the fault signatures of the known faults, a diagnosis can be made.

The failure bitmap can be generated if the result of the test is observable. In this project, a tag was associated with each memory read (Figure 3.5). In this way, every memory read can be associated with an address and bit in the memory array. Furthermore, the tag provides information about the step number of the test being performed. Knowing the locations and step numbers of the test for each fault gives enough information to generate a diagnosis. A diagnostic tree of the decision making steps will be given in Chapter 5.

Chapter 4

Diagnosis

4.1 Introduction

The goal of diagnosis is to discover the reason behind a system misbehavior: where did the fault occur and why? As a first step, this thesis outlined the possible faults that were being considered in the fault model (see Chapter 2). Next, it introduced the tests that were being used (see Chapter 3). This chapter provides a description of what tests catch the faults and how these faults manifest themselves in these tests. Fault manifestation is that fault's fault signature. The two components of a fault signature are the locations in the test where the fault manifests itself (e.g. reading the wrong value in the first read of step 2) and the locations in memory where the fault manifests itself (e.g. all cells of way A of bit 3). If different faults can be shown to have different fault signatures (i.e., manifest themselves differently), then knowing each fault's fault signature is all that is required to diagnose that fault. Once these fault signatures have been understood and cataloged, a decision making procedure can be created to make a diagnosis based upon the fault signature. This decision making procedure is called a diagnostic tree and will be described in the Conclusion. This thesis assumes that faults occur singly. The case where multiple faults occur in the same memory array adds extra complexity to both test and diagnosis and will not be considered here.

Note that the emphasis of this thesis is on diagnosis and not on test. Therefore, the description of what tests catch the faults is not directed towards the most efficient tests, but tests that allow for diagnosis. The fact that a test allows for the diagnosis of a certain fault implies that the tests can catch the fault.

As a brief review, Figure 4.1 presents a list of the faults considered in the fault model (from Chapter 2). Note that diagnosis assumes that these are the only faults that can exist. The goal is to attribute a different fault signature to each fault. However, there is a danger that an unlisted fault exists which manifests itself with the same fault signature as one of the listed faults. For test, an unknown fault is not a problem so long as this fault can be caught with the existing test. For diagnosis, an unknown fault is a problem because it may potentially have the same fault signature as a known fault, causing an incorrect diagnosis.

Memory Cell Array Faults:

- Stuck-at Faults
- Transition Faults
- Coupling Faults
- Non-Deterministic Faults
- Destructive Read Faults

Read/Write Logic Faults:

- Bitline Precharge Faults
- Bitline Faults
- Column Multiplexor Faults
- Data-in Faults
- Sense Amplifier Faults
- Data-out Faults

Address Decoder Faults:

- Row Decoder Faults
- Column Decoder Faults
- Wordline/Bitline Selector Faults

Figure 4.1: List of Faults

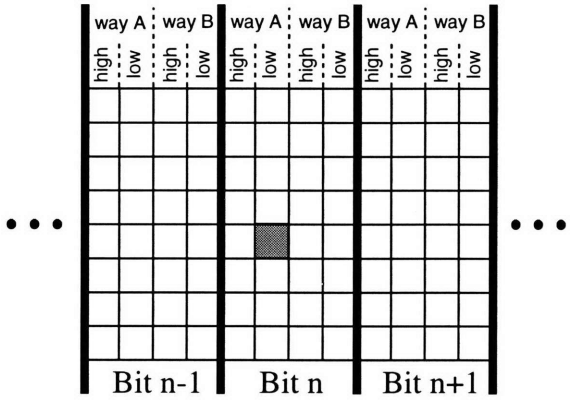
4.2 Fault Detection and Manifestation: Memory Cell Array

There are five types of memory cell faults: stuck-at faults, transition faults, coupling faults, non-deterministic faults, and destructive read faults. As mentioned in Chapter 2, the scope of these types of fault is just a single cell: all these faults manifest themselves as single cell fails.

4.2.1 Stuck-at Faults

Stuck-at faults are easy to catch. The tests only need to be written such that a value of both 1 and 0 are written to and read from each cell. A unique address ripple word test is adequate to test for all stuck-at faults. The circled reads in Figure 4.2a and Figure 4.2b are where mismatches occur for these tests in the event of a stuck-at 0 and a stuck-at 1 fault respectively. The scope of this fault is a single cell (see Figure 4.2c).

(a) S-a-0: All attempts to read 1 will fail (b) S-a-1: All attempts to read 0 will fail



(c) fault manifestation: single cell fail

Figure 4.2: Stuck-at fault detection and manifestation

4.2.2 Transition Faults

In order to catch transition faults, each memory cell must undergo a transition high and a transition low and be read after each transition. This fault is different from a stuck-at in that the cell may, for example, start with a value of 1, transition to 0, and then stay stuck-at 0. A stuck-at 0 fault would always be 0. A cell with a transition fault can transition in one direction but not the other, whereas a cell with a stuck-at fault cannot transition at all.

The problem with distinguishing transition faults from stuck-at faults is that the initial state of the memory cell is not known with any certainty. For instance, if the cell has a transition fault where it is unable to transition high but already starts in a 0 state, the fault will appear identical to a stuck-at 0 fault.

Moreover, every march test first initializes every memory cell to a value (i.e., step 1 of the unique address ripple word). A cell with a transition fault can transition in one direction but not in the other direction. The problem is that this first write may force the transition. Once a transition occurs, a transition fault cannot be distinguished from a stuck-at fault. The unique address ripple word test will never be able to distinguish a stuck-at 0 fault from a transition fault where the cell is unable to transition high because it initially forces all cells to a 0 state in step 1 (Figure 4.3a). The circled reads are reads where a mismatch will occur. The ability to transition low may be masked by the first write. On the other hand, a transition fault where the cell is unable to transition low may be caught, but only if the initial state of the memory cell is 0 (Figure 4.3b). As pointed out in the previous paragraph, if the memory cell starts in a state of 1, the original write would have been unable to force the transition low and the first read would read a 1 instead of a 0 (this is identical to the situation where the cell has a stuck-at 1 fault).

As with the stuck-at fault, this fault is manifested as a single cell fail (Figure 4.3c).

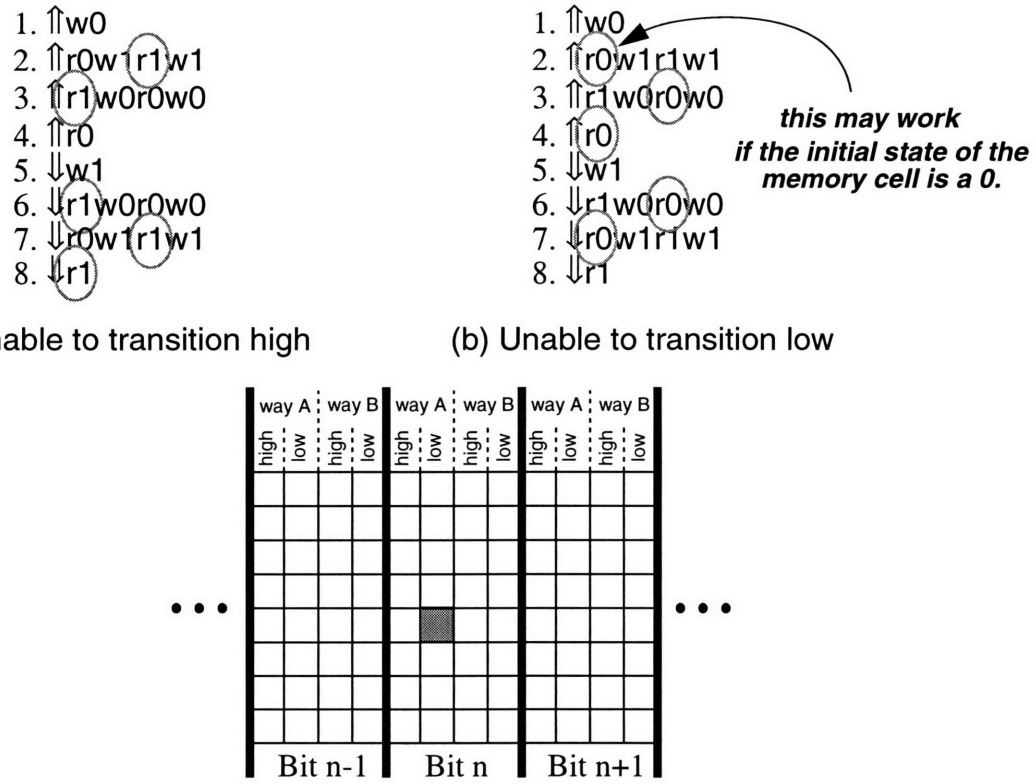


Figure 4.3: Transition fault detection and manifestation

4.2.3 Coupling Faults

As pointed out in Chapter 2, three types of coupling faults are being targeted: idempotent coupling faults, inversion coupling faults, and state-coupling faults. Certain patterns in the test can be used to expose these coupling faults. The source cell which causes the fault can be triggered by writing a 1 or a 0. Furthermore, we want to trigger the fault in both the case when we are expecting a 1 and when we are expecting a 0. In this way, all coupling faults can be caught.

There are thus four different combinations of triggers and expected values. The initial assumption is that the source cell precedes the target cell. Table 4.1 shows how a test can be written such that the combinations of triggers and expected values can arise. The two columns on the left show the combination of trigger and expected value. The columns on

the right show the two steps necessary to achieve this combination. As a reminder, ⇑r0w1 means that the memory cells are accessed in the forward direction and that at each memory cell a value of 0 is read and a value of 1 is written before proceeding to the next memory cell. ⇓r0w1 is identical to ⇑r0w1 except that the memory cells are being accessed in a reverse direction. If the target cell precedes the source cell, Table 4.2 shows how the tests can be written to expose the four combinations of triggers and expected values.

trigger at source	expected value at target	(step n-1)	trigger fault (step n)	(step n+1)
0	0	don't care	⇓w0	⇓r0
0	1	⇑w1	⇑r1w0	don't care
1	0	⇑w0	⇑r0w1	don't care
1	1	don't care	⇓w1	⇓r1

Table 4.1: Catching coupling faults assuming source cell precedes target cell

trigger at source	expected value at target	(step n-1)	trigger fault (step n)	(step n+1)
0	0	don't care	⇑w0	⇑r0
0	1	⇓w1	⇓r1w0	don't care
1	0	⇓w0	⇓r0w1	don't care
1	1	don't care	⇑w1	⇑r1

Table 4.2: Catching coupling faults assuming target cell precedes source cell

The reason for bringing up how these different combinations of triggers and expected values arise is that failures due to the three coupling faults can be thought of as failures in some set of these combinations. As the simplest case, consider an idempotent fault where writing a 1 in one cell forces a 0 in another cell. This fault can be caught with a test in which there is a trigger of 1 at the source and the expected value of 1 at the target. Each of the four types of idempotent faults have a corresponding test that will expose it:

1. writing a 1 in the source cell forces a 0 in the target cell:
• trigger of 1 in source, expect a 1 in target
2. writing a 1 in the source cell forces a 1 in the target cell:
• trigger of 1 in source, expect a 0 in target
3. writing a 0 in the source cell forces a 0 in the target cell:
• trigger of 0 in source, expect a 1 in target
4. writing a 0 in the source cell forces a 1 in the target cell:
• trigger of 0 in source, expect a 0 in target

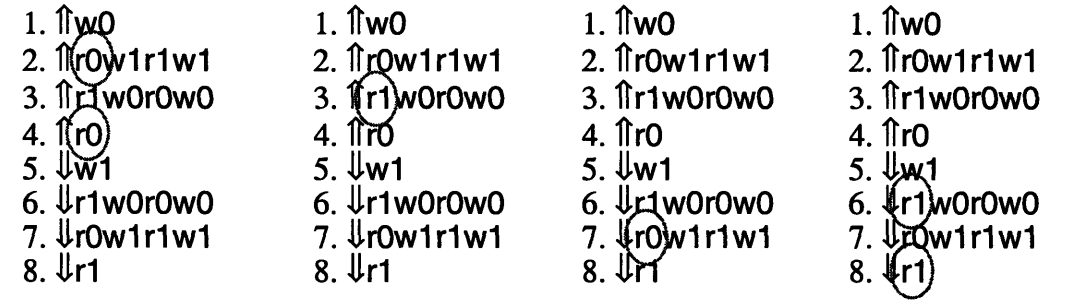
A similar correspondence can be made for inversion faults. Consider the case where writing a 1 in one cell inverts the state of another cell. It is necessary to check that writing a 1 in the first cells flips the target cell both when a 1 is expected and when a 0 is expected. If an incorrect value is read in both cases, then the cell exhibits an inversion fault. The two types of inversion faults and the tests needed to expose them are shown below:

1. writing a 1 in the source cell inverts the target cell:
• trigger of 1 in source, expect a 1 in target
• trigger of 1 in source, expect a 0 in target
2. writing a 0 in the source cell inverts the target cell:
• trigger of 0 in source, expect a 1 in target
• trigger of 0 in source, expect a 0 in target

There are two types of state-coupling faults. In the first type, writing a value, x, in the source cell will force a second, target, cell to this same value, x. In the second type, writing a value, x, in the source cell will force a second, target, cell to the opposite value, $\overline{x}$. To catch the first type of state-coupling fault, it is necessary to both write a 1 to the source cell while expecting a 0 in the target cell and write a 0 to the source cell while expecting a 1. An incorrect value should be read from the target cell in both cases for this type of state-coupling fault. The two types of state-coupling faults and the tests needed to expose them are shown on the following page:

1. writing x in the source cell forces x in the target cell:
• trigger of 1 in source, expect a 0 in target
• trigger of 0 in source, expect a 1 in target
2. writing x in the source cell forces $\overline{x}$ in the target cell:
• trigger of 1 in source, expect a 0 in target
• trigger of 0 in source, expect a 1 in target

The unique address ripple word test contains all the components in Table 4.1 and Table 4.2. Therefore this test will be able to catch these three types of coupling faults both when the target cell precedes and follows the source cell of the fault. Furthermore, because these three types of coupling faults are caught with different combinations of triggers and expected values, they can be uniquely diagnosed. Figure 4.4 shows the fault signature in the unique address ripple word test in the case of the four types of idempotent faults. The circled reads are the reads in the test where a mismatch will occur if an idempotent fault exists.



(a) writing a 1 in x forces a 1 to y or writing a 0 in y forces a 1 in x

(b) writing a 0 in x forces a 0 to y or writing a 1 in y forces a 0 in x

(c) writing a 0 in x forces a 1 to y or writing a 1 in y forces a 1 in x

(d) writing a 1 in x forces a 0 to y or writing a 0 in y forces a 0 in x

Figure 4.4: Fault signatures of idempotent faults

Figure 4.5 shows the unique address ripple word test with symptoms of the inversion faults. Again, the circled reads indicate where an incorrect value is read.

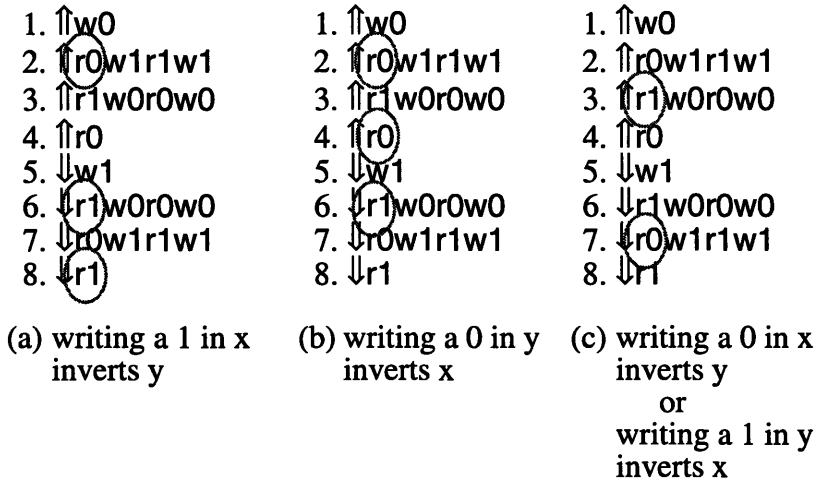


Figure 4.5: Fault signatures of inversion faults

Finally, Figure 4.6 shows the unique address ripple word test exhibiting the fault signatures of state-coupling faults. The circles indicate places in the test where an incorrect value is read.

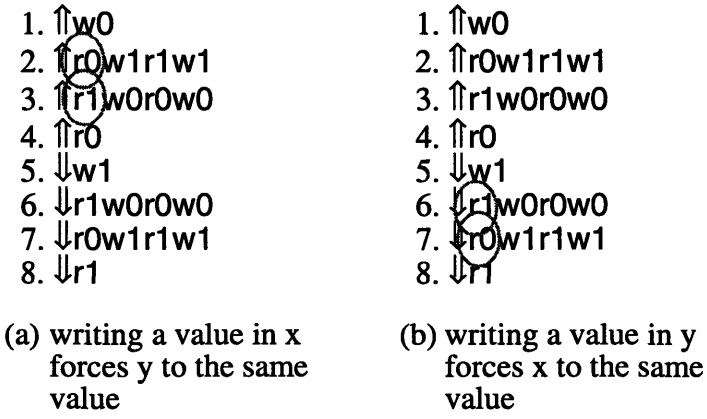


Figure 4.6: Fault signatures of state-coupling faults

As with the other memory cell array faults, coupling faults all manifest themselves as single cell fails (see Figure 4.3c). The main idea of Figure 4.4 through Figure 4.6 is that each of the three coupling faults (idempotent, inversion, state-coupling) have different fault signatures. This allows them to be distinguished from each other and thus diagnosed.

4.2.4 Non-Deterministic Faults

Non-deterministic faults are difficult to catch and even more difficult to diagnose. Depending on how the memory is designed and on the small variations that arise during manufacturing, the result of a memory cell that has a non-deterministic fault can vary. This is a result of the fact that an inaccessible memory cell (the usual cause of a non-deterministic fault) will not drive the sense amplifier with a value. Stated another way, the sense amplifier will be given the same value on both inputs. In this project, the sense amplifiers were designed without a preferred state. If the memory is manufactured perfectly, then the result of a non-deterministic memory cell will be random. On the other hand, if the sense amplifier has a preference, a non-deterministic fault may appear as a stuck-at 1 or a stuck-at 0. To qualify this even more, this preference may change depending on how much noise exists in the system.

Different memory implementations yield different manifestations of non-deterministic faults. Therefore, to properly understand the diagnosis of this fault (and even whether this fault can be uniquely diagnosed) for a particular chip requires an understanding of the design and implementation of that particular chip's memory. The fault signature of this fault is unknown and needs to be determined with experimental data from the actual implementation of the memory being tested.

The same is true of partially non-deterministic faults. As its name implies, only in certain circumstances (i.e., only when reading 0's) does a memory cell with a partially non-deterministic fault become non-deterministic. In other circumstances, the proper value is read from the memory cell.

4.2.5 Destructive Read Faults

The way to catch a destructive read fault is to read the same value twice from the same cell. The first read triggers the fault, and the second read catches the fault. A double read of both a 0 and a 1 needs to exist somewhere in the test. For instance, the unique address

ripple word could be modified such that it can catch this fault (Figure 4.7). Depending on the type of destructive read fault, either the second read of a 0 or the second read of a 1 will fail (the circled parts of Figure 4.7). This is also a single cell fail (see Figure 4.3c). Currently, the unique address ripple word test does not have the double read. However, this double read occurs in the checkerboard test.

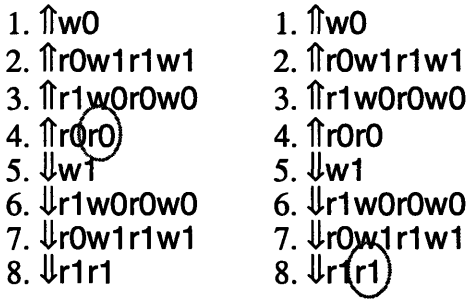


Figure 4.7: Fault Signatures of destructive read faults

4.3 Fault Detection and Manifestation: Read/Write Logic

There are six types of read/write logic faults: bitline precharge faults, bitline faults, column multiplexor faults, data-in faults, sense amplifier faults, and data-out faults. The scope of this type of fault is the set of cells or subset of cells from within a single bit or two adjacent bits. The model for the read/write logic was presented in Chapter 2 (see Figure 2.9). Note that this model showed the read/write logic for only one way of the two-way associative memory that is being used as an example. A separate read/write logic mechanism is being assumed for each way.

4.3.1 Bitline Precharge Faults

The bitline precharge fault was described in the fault model section in Figure 2.10. A very clear way of exposing this fault is to write some value x, then read the opposite value $\overline{x}$ from a cell that uses the same precharge logic. This is demonstrated in Figure 4.8. Note that cell A and cell B share the same precharge logic. The first command (the write to cell A) forces the true and complement wires in one direction. To write a 0, as shown in Figure 4.8a, the true wire is at GND while the complement wire is at VDD. The second

command (the read of cell B) expects the true and complement wires to be in the other direction. In Figure 4.8a, to read a 1 out of cell B, the true wire should be at VDD while the complement wire is at GND. However, the precharge logic is not working properly and a value of 0 is read instead.

This pattern of commands (writing x followed by reading $\bar{x}$) exists in the unique address ripple word test and is the reason for the second write in steps 2, 3, 6, and 7. Figure 4.8a and Figure 4.8b shows how this fault manifests itself in this test. The circled reads are the reads that will fail.

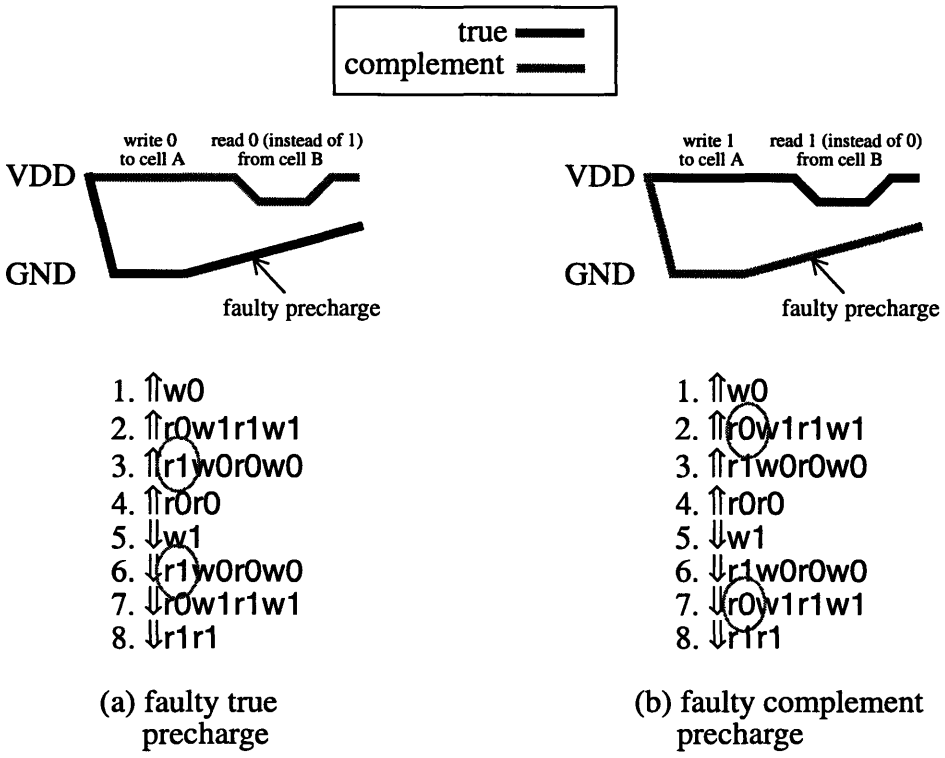


Figure 4.8: Bitline precharge fault: manifestation in the test

Note that the reads fail because a write in the preceding cell (cell A) forces the true and complement wires in the opposite direction than is required for the read of cell B. The first circled r1 in step 3 (Figure 4.8a) fails for a certain memory cell B because of the second w0 in step 3 of the previous memory cell A; the fault is triggered by a preceding memory cell. Figure 4.9 demonstrates how this faults shows up in memory. This fault is manifested

in memory as a column of memory cells which exhibits the faults shown in Figure 4.8. However, the fact that the fault is triggered by a write to the preceding memory cell means that the end points of the shaded column in Figure 4.9 behave a little differently. For example, if there was a faulty true precharge on this column (Figure 4.8a), most of the column will fail to read a 1 in step 3 and step 6. Assuming that tests going in the forward direction start from the bottom of the column and work up the column, the cell at the bottom of the column will only fail on step 6 (i.e., when the test is going backwards), while the cell at the top of the column will only fail on step 3 (i.e., when the test is going forwards). An analogous situation exists for a faulty complement precharge logic.

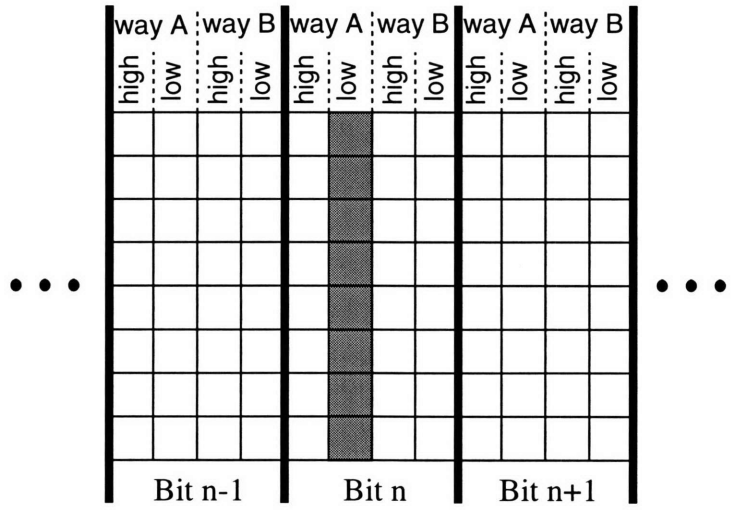


Figure 4.9: Bitline precharge fault: manifestation in memory

4.3.2 Bitline Faults

There are three types of bitline faults: breaks in one bitline, shorts across the true and complement wires of the same bitline, and shorts between wires on different bitlines. Each of these types of bitline faults has a slightly different fault signature.

The first type of fault is a break in the true or complement wire of one bitline. Figure 4.10 shows a break in the true wire. Assuming all the logic lies on the bottom part of the picture, the memory cells below the break and closer to the logic will still work properly. However, the memory cells beyond the break will be inaccessible and partially

non-deterministic. The complement wire still works correctly, so a value of 1 can still be written to these cells (the complement wire can pass a 0 into the memory cell on a write of 1) and read from these cells (the memory cell can still pull down the complement wire on a read of 1). However, attempts to write or read a 0 will fail. Reading such a cell yields a non-deterministic result.

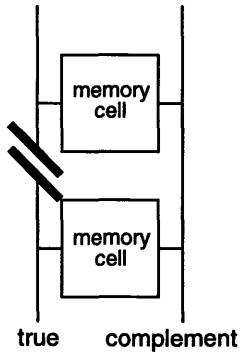


Figure 4.10: Break in true wire of bitline

Figure 4.11 shows the fault signature of this type of fault in memory. For the bitline with the break, the memory cells below the break work normally. The cells above the break will exhibit either the fault in Figure 4.11a or Figure 4.11b depending on whether there is a break in the true of the complement wire. Note that the reads that have been marked with X's are non-deterministic. Any test that can catch partially non-deterministic faults can catch this type of fault.

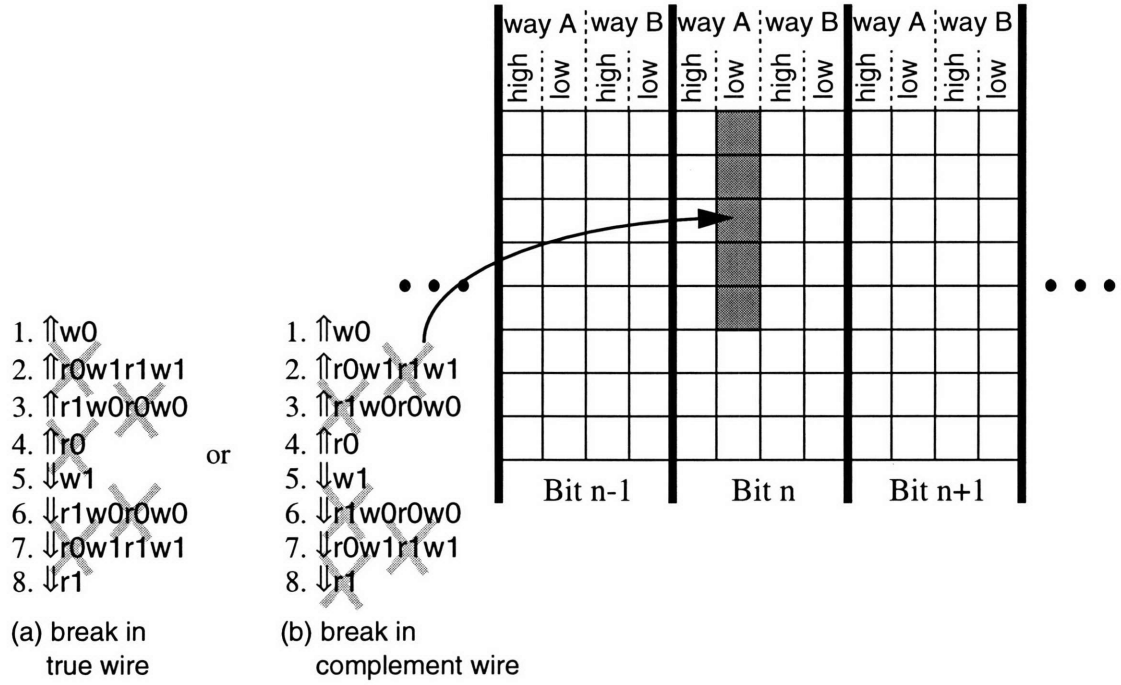


Figure 4.11: Fault signatures of break in bitline wire

The second type of bitline fault is a true to complement short within one bitline. The result of this fault is that both wires are at the same value; the sense amplifier does not know what value to return. This results in a column of memory cells that exhibit non-deterministic faults (Figure 4.12). These memory cells share the same sense amplifier so there is a high probability that all the memory cells will fail in a similar fashion. Any test that can catch non-deterministic faults can catch this type of fault.

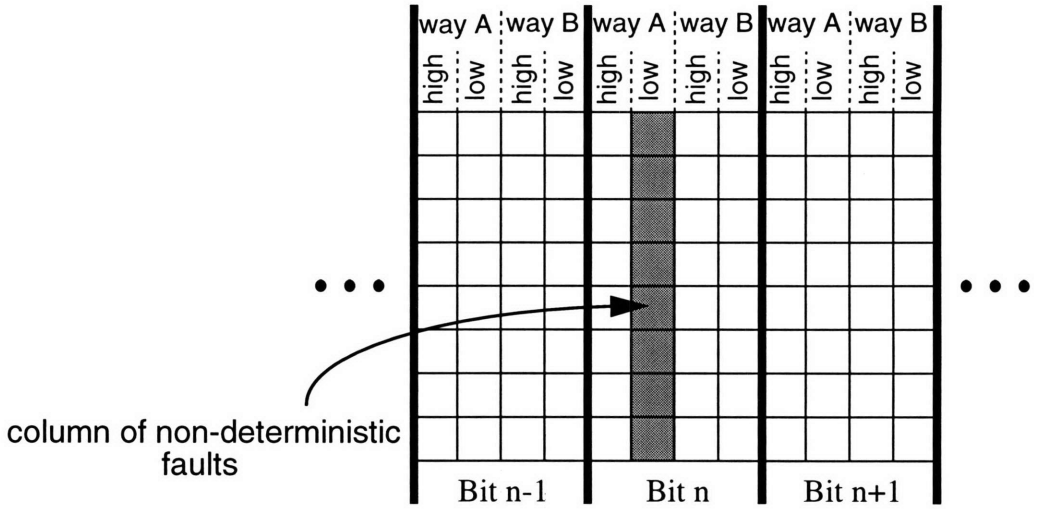


Figure 4.12: Fault signature of true to complement short within one bitline

The final type of bitline fault is a short across the wires of two adjacent bitlines. This can be a true to true, true to complement, complement to complement, or complement to true short. An example of this type of fault is shown in Figure 4.13. In this example, writing a 1 to a cell in bitline 0 forces the complement0 wire to 0. The short forces the true1 wire to 0. The result is that an adjacent cell in bitline 1 ends up forced to a state of 0. The arrows in Figure 4.13 indicate the idempotent faults that result from this fault. The arrows from left to right are idempotent faults where writing a 1 into A (or B) force a 0 into C (or D). The arrows from right to left are idempotent faults where writing a 0 in C (or D) force a 1 into A (or B).

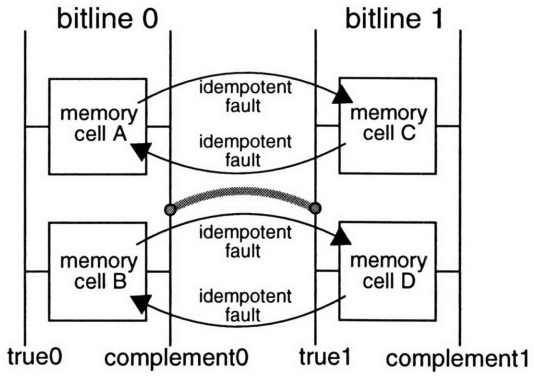


Figure 4.13: True to complement short in bitline

In this example, every cell in bitline 0 has an idempotent fault with an adjacent cell in bitline 1 and vice versa. The fault signature of a short across adjacent bitlines is shown in Figure 4.14. The manifestation of idempotent faults in the unique address ripple word test is shown in Figure 4.4. Any test that can catch all idempotent faults will be able to catch cross bitline shorts.

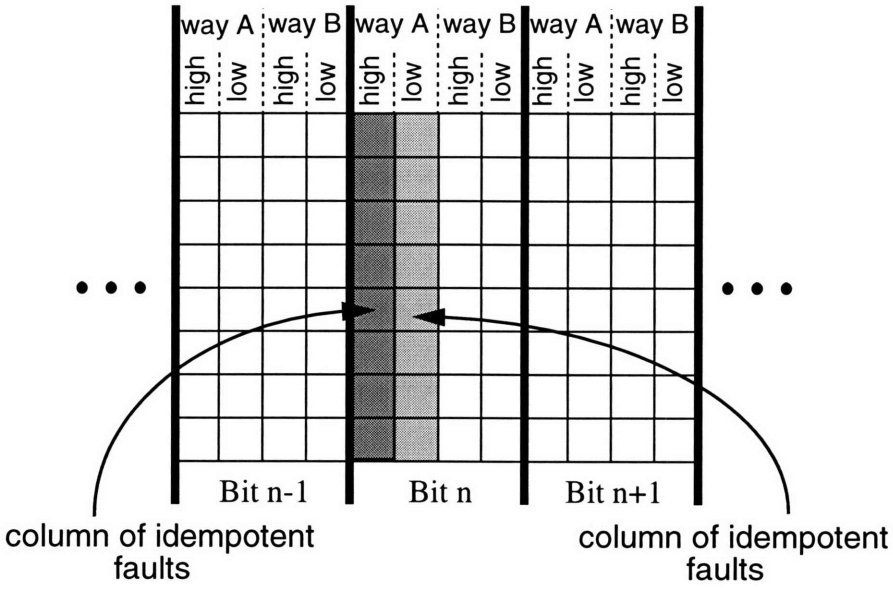


Figure 4.14: Fault signature of shorts across adjacent bitlines

4.3.3 Column Multiplexor Faults

The chapter on the fault model described two types of column multiplexor faults. In the first, a pass-gate for either the true or complement wire is stuck open. In the second, a pass-gate is stuck closed. Both types result in different fault signatures.

A stuck open pass-gate in the column multiplexor is equivalent to a bitline break right next to the pass-gate (Figure 4.15).

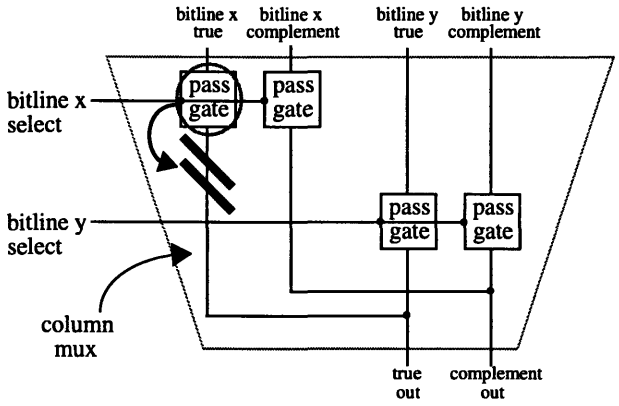


Figure 4.15: Pass-gate stuck open in column multiplexor

This fault manifests itself exactly as the bitline break did, except that now the entire column is affected (Figure 4.16). As with the bitline break, one of the wires, either the true or complement, becomes inaccessible. One of the wires will still be able to pass a value so, for example, if the pass-gate for the true wire is stuck open, the complement wire would still be able to pass a 0, which means that a value of 1 can still be read. A read of a 0 where the true wire is expected to pass a 0 is, however, non-deterministic (these are the X's in Figure 4.16a). Every cell in the column shares the same column multiplexor, so every cell should fail in the same way.

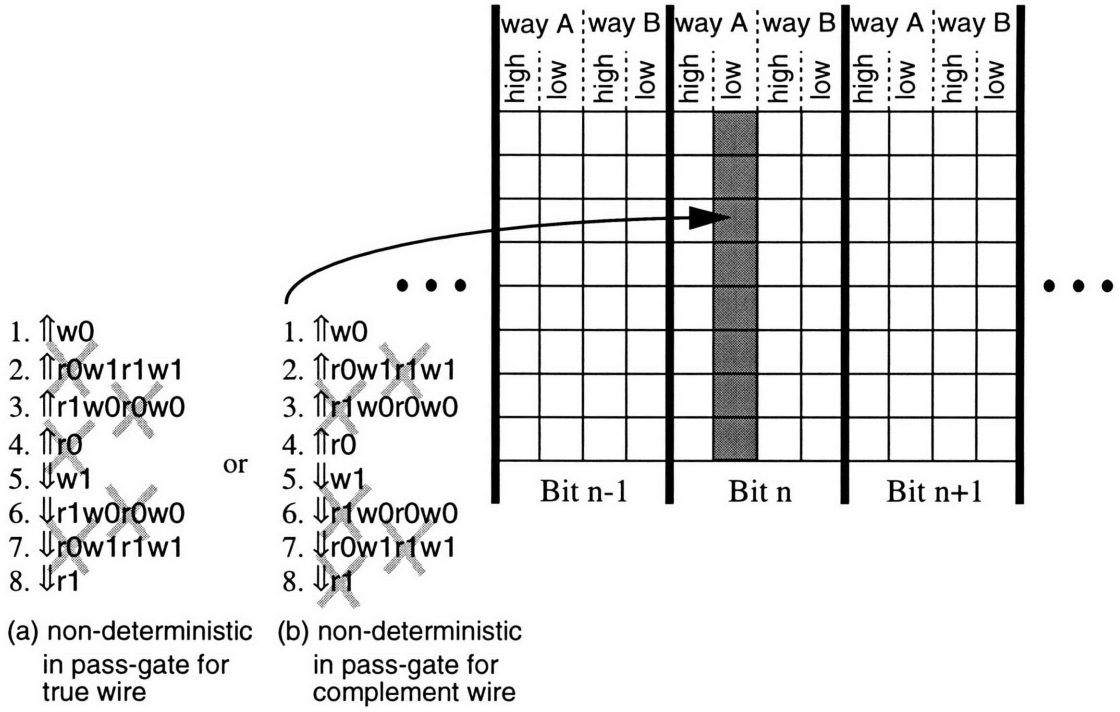


Figure 4.16: Fault signature of stuck open pass-gate in column multiplexor

The second type of column multiplexor fault arises when a pass-gate for either the true or complement wire of a bitline is stuck-closed (i.e, the pass-gate acts as a short-circuit). This is shown in Figure 4.17.

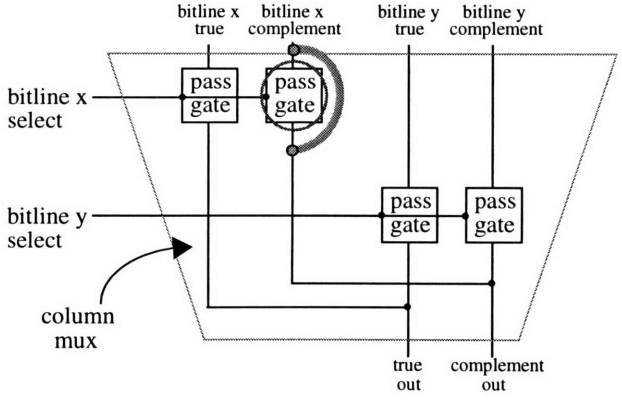


Figure 4.17: Pass-gate stuck-closed in column multiplexor

Bitline x, which contains the faulty pass-gate will exhibit an idempotent coupling fault.

Writing a 1 to a cell along bitline y (in which case the true out is high and the complement out is low) will force an adjacent cell along bitline x to 1 (the bitline x complement is driven low). If the fault existed in the pass-gate for the true wire, then writing a 0 to cell along bitline y would force a 0 to an adjacent cell along bitline x.

The behavior of bitline y is more uncertain and is implementation dependent. The short circuit in bitline x's complement wire means that a value from bitline x can corrupt the value being read from bitline y. However, this is only a problem when adjacent cells in bitline x and bitline y are at different values. When two adjacent cells in bitline x and y are at the same value, reading the value from the cell in bitline y will not be a problem. There are two possible cases where corruption of the read from bitline y can happen.

In the first case, a 1 is coming from the bitline y complement wire (when a value of 0 is read from bitline y) while the bitline x complement wire is trying to pull down to 0 (Figure 4.18). The transistor trying to pull the bitline x complement wire down to zero has a greater load than the transistor trying to pull the bitline y true wire down to zero. Therefore, the bitline y true wire will still fall a little faster than the bitline x wire. Depending on how sensitive the sense amplifier is, the correct value of 0 may still be read from bitline y. However, it is also possible that the sense amplifier will be unable to properly read the value (i.e., this may be non-deterministic).

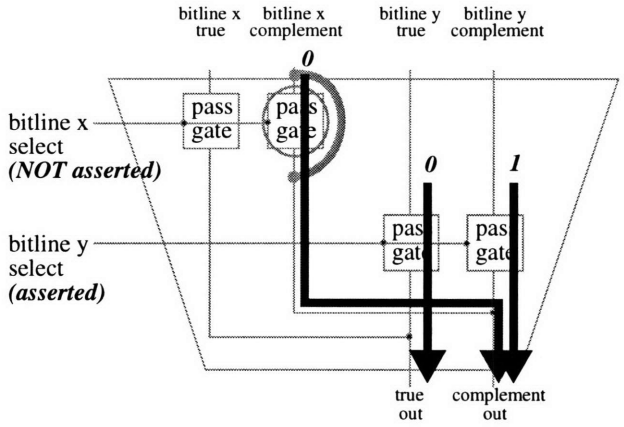


Figure 4.18: Column multiplexor stuck-closed: first case

In the second case, a 0 is coming from the bitline y complement wire (when a value of 1 is being read from bitline y) while the bitline x complement wire is trying to pass a 1 (Figure 4.18). The bitline y complement is still trying to pull down the wire, but it has a greater load than normal because it is shorted to the bitline x complement. Though it should be possible for the sense amplifier to properly distinguish a value of 1, this is not guaranteed, and depends on the sensitivity of the sense amplifier. Again, this may be non-deterministic.

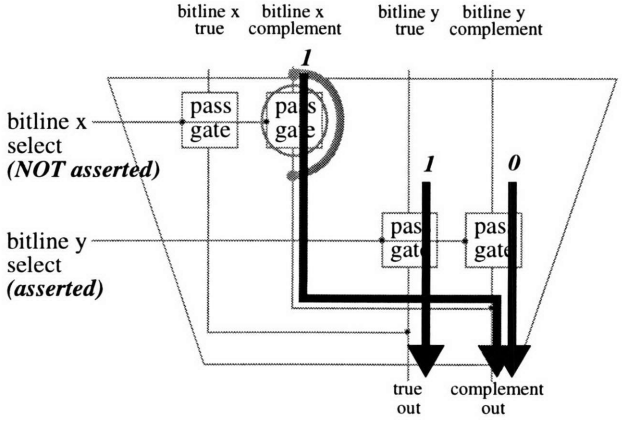


Figure 4.19: Column multiplexor stuck-closed: second case

In summary, the case of a column multiplexor pass-gate stuck-closed results in one column of cells which exhibit idempotent coupling faults. This is the column whose pass-

gate possesses the fault. Depending on the implementation, the other column may exhibit a partially non-deterministic fault or it may exhibit no fault at all. (Figure 4.20). Any test which can detect idempotent coupling faults should be able to catch this type of fault.

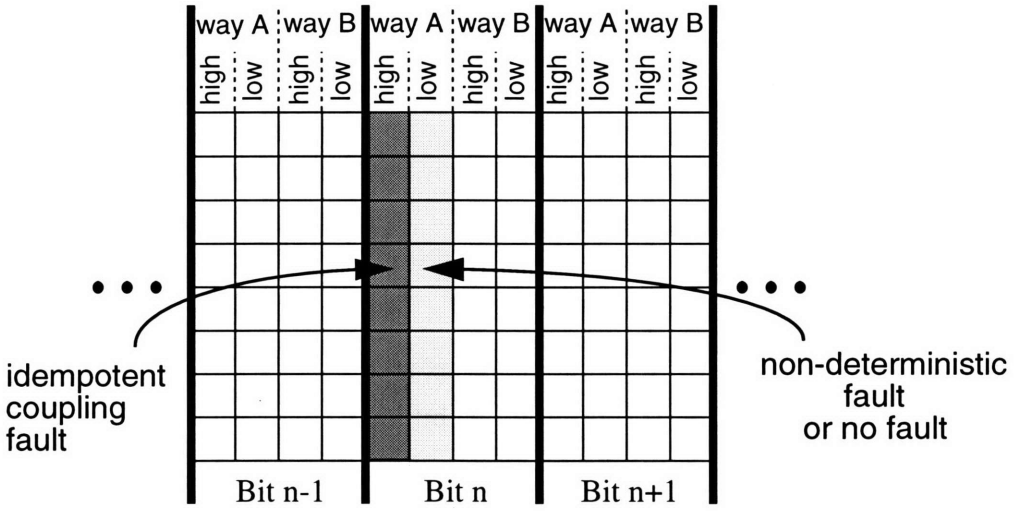


Figure 4.20: Fault signature of pass-gate stuck-closed in column multiplexor

4.3.4 Data-in Faults

Assuming that each way has its own read/write logic, a data-in fault in the read/write logic of way A of bit n will cause faults in all memory cells for way A of bit n. If there is a stuck-at 0 fault in the data-in logic, there will be a stuck-at 0 fault in all the memory cells. A similar reasoning applies to stuck-at 1 faults. In the case that the data-in logic is stuck open (and hence non-deterministic), all memory cells will be stuck-at their respective initial states. Any test that can catch stuck-at faults will catch data-in faults. The fault signature of this fault is shown in Figure 4.21. Note that the fault signature of this kind of fault will be slightly different depending on the memory implementation. This figure shows a two way associative memory where the fault only appears in way A. In the case that this memory only has one way, the fault would be apparent in all memory cells in that bit.

Also, in this example, each way had its own read/write logic. If the data-in logic was shared across ways, then the fault would appear in both ways of this bit.

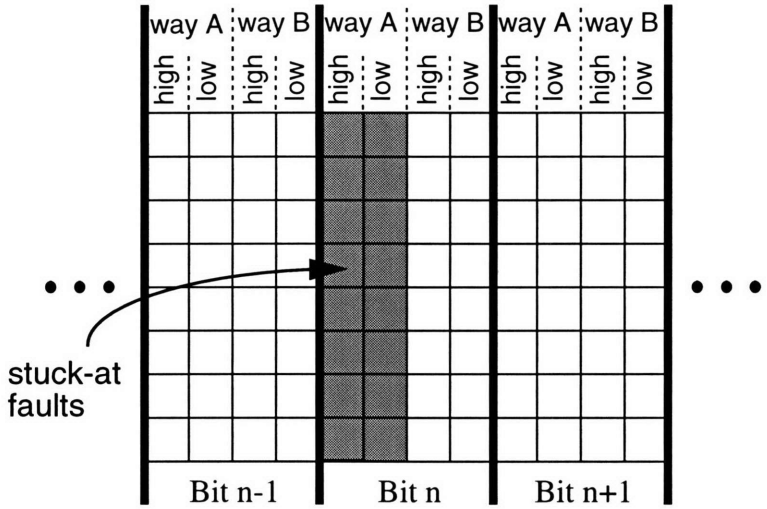


Figure 4.21: Fault signature of data-in fault

4.3.5 Sense Amplifier Faults

Unfortunately, sense amplifier faults may appear identical to data-in faults. If the sense amplifier of way A of bit n is stuck-at 0, then all the memory cells in way A of bit n will be stuck-at 0 (as was the case for a data-in fault). Similarly, if the sense amplifier is stuck-at 1, then all memory cells will be stuck-at 1 (Figure 4.22). A sense amplifier that is stuck open (non-deterministic) will cause an unknown fault, but one that is repeated in all memory cells. How this type of fault ultimately manifests itself depends on the circuitry connected to the sense amplifier. Tests that can detect stuck-at faults will be able to detect when the sense amplifier has a stuck-at fault.

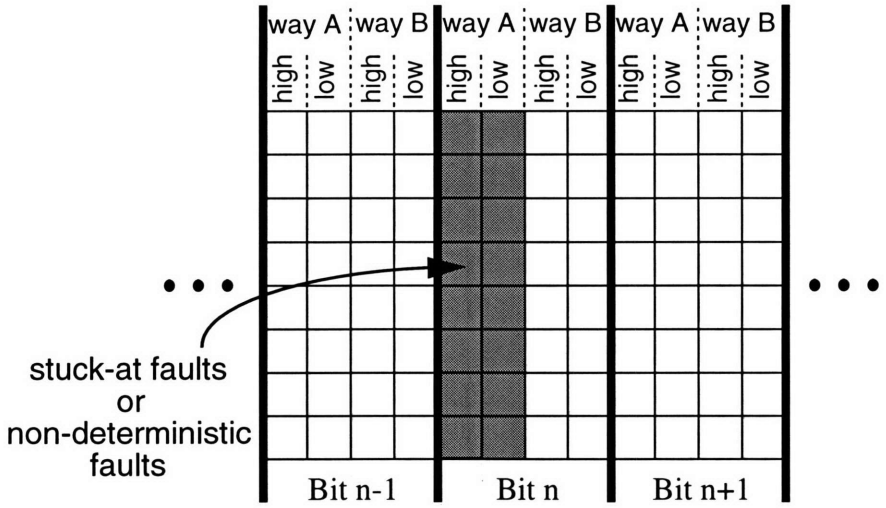


Figure 4.22: Fault signature of sense amplifier fault

The reasoning here must be qualified in the same fashion as for the data-in logic. The fault will affect only the memory cells that use this sense amplifier. If there is only one way, then all memory cells in this bit will be affected (rather than just in way A). If the sense amplifier is shared across ways, then memory cells in both ways of this bit will be affected.

Sense amplifiers are designed to catch small variations between its two inputs. These tight design parameters make them more susceptible to noise. Therefore, tests need to exist which will stress the sense amplifiers in different ways. One such test is the word line stripe test. In this test, a word, every bit of which is 1, is read out of memory so that every sense amplifier is reading a 1. Next, another word, every bit of which is 0, is read out of memory so that every sense amplifier is reading a 0. The next word is again an all 1s and so on. This pattern is repeated a number of times. This oscillation between reading all 1s and all 0s can cause noise in the system which can disrupt the sense amplifier [5].

4.3.6 Data-out Faults

The first type of data out fault, data-out stuck-at faults, are also indistinguishable from sense-amplifier stuck-at faults (Figure 4.22). If the data-out is stuck-at 1, then this fault

will manifest itself as a bit whose memory cells are all stuck-at 1. If the data-out is stuck-at 0, then this fault will manifest itself as a bit whose memory cells are all stuck-at 0.

The second type of data-out fault being considered here is the case where a data-out wire is short circuited to an adjoining data-out wire. This type of fault is shown in Figure 4.23.

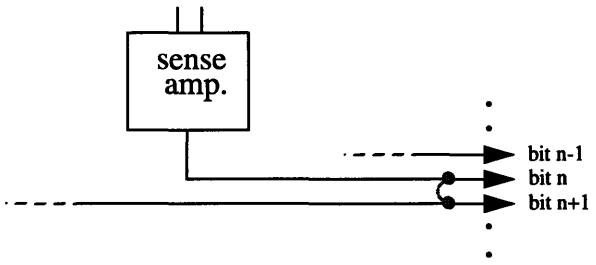


Figure 4.23: Data-out short

In this case, the best way to expose this fault is to make sure that different values are being read from consecutive bits. Depending on the memory implementation, the checkerboard test may satisfy this requirement. The main idea is that the word being read from the memory cell must be such that if bit n is a 1, bit n-1 and bit n+1 must be a 0. For the implementation shown in Figure 3.4, the checkerboard test does satisfy this requirement. However, for the implementation in Figure 3.3, the checkerboard test does not satisfy this requirement. If bit n is a 1, so too is bit n-1 and bit n+1.

For the current memory implementation, there needs to be an extra data-out fault test of four additional memory operations to test for data-out shorts. The first operation is a write into any address of a word, 010101... This number then needs to be read out of the selected address. The opposite word (101010...) then needs to be written to and read from memory. Note that unlike the march tests, these operations do not need to be repeated for all addresses because this is a test of the read/write logic only. Performing these operations on the rest of the memory cells adds no new information that the other tests do not already provide.

How this fault manifests itself depends on what logic comes after. In this figure, if bit n was trying to drive a value of 1 and bit n+1 was trying to drive a value of 0, then both wires would end up in a half-voltage state. Though the exact nature of how the individual memory cells fail is unknown, they will all fail in the same fashion. The memory cells who are affected by this fault are shown in Figure 4.24. If this memory only had one way, then all memory cells in bits n and n+1 would have been affected, rather than just in way A.

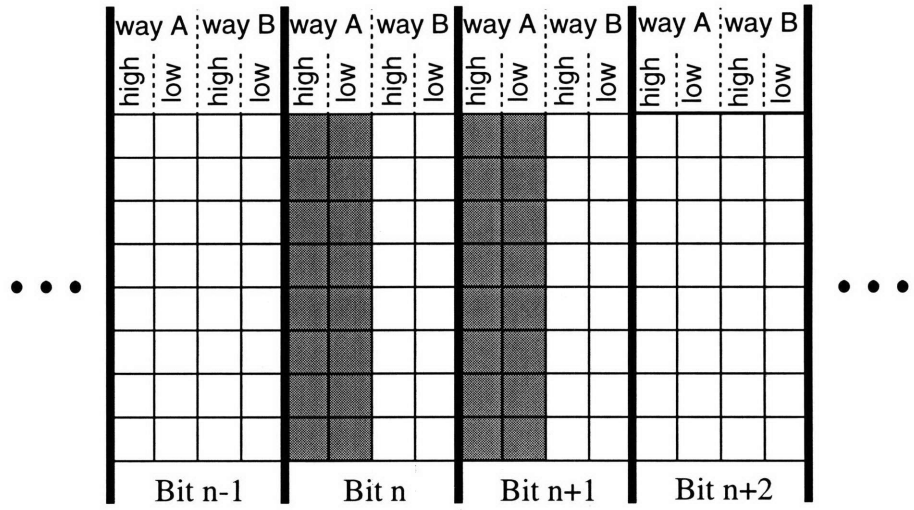


Figure 4.24: Fault signature of data-out short fault

4.4 Fault Detection and Manifestation: Address Decoder

Chapter 2 described three types of address decoder faults: row decoder faults, column decoder faults, and wordline/bitline selector faults. Each of these faults have their own fault signatures. As mentioned in Chapter 2, both the row decoder and column decoder faults can be further classified into four types of possible addressing faults (these are shown again in Figure 4.25).

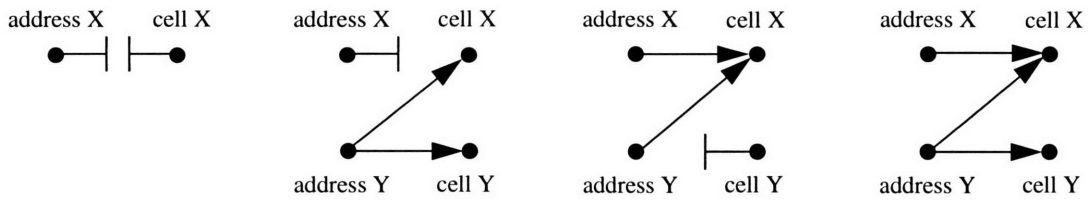


Figure 4.25: Logical address decoder faults

Note that although the diagram seems to indicate an address X that activates wordline X (or bitline selector X), in reality there are a set of address which all activate wordline X (or bitline selector X). "address X" is merely a convenient shorthand for all addresses that access wordline X (or bitline selector X). For instance, in Figure 4.26, if the cells that are shaded are accessed by wordline X, then the set of cells "address X" which activate wordline X are cells 2 and 10 for all bits and for both way A and way B.

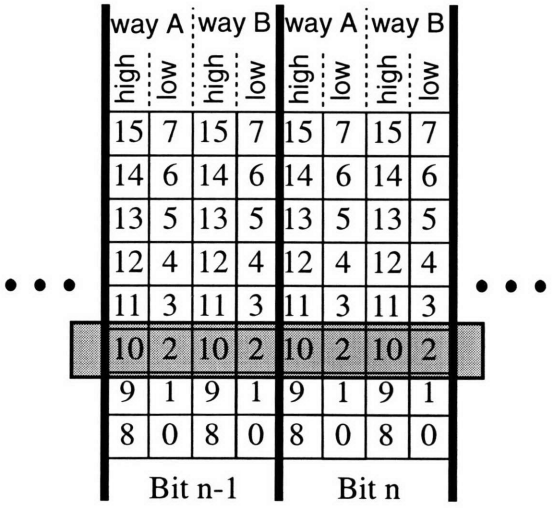


Figure 4.26: Addressing clarification

This section will show how each of these four addressing faults manifest themselves for the row decoder and column decoder and how they can be caught. This will be followed by a section describing how wordline/bitline selector faults manifest themselves and how they can be caught.

4.4.1 Row Decoder Faults

As described in 2.4.1, the result of faults A and B is that a row of memory cells will become inaccessible (therefore non-deterministic). Any attempt to write to or read from an any address X that is supposed to activates wordline X will fail; the wordline X that should have been active never fires (Figure 4.27). All addresses for all memory cells that should have been accessed with this wordline wire are never accessed. The result is that these two faults manifest themselves as shown in Figure 4.28. Any test that can catch non-deterministic faults can catch these faults.

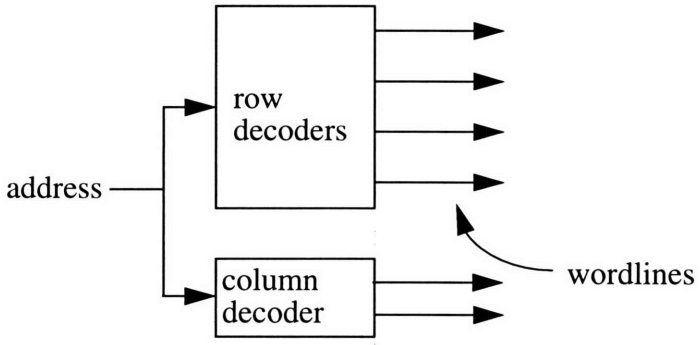


Figure 4.27: Address decoder wordlines

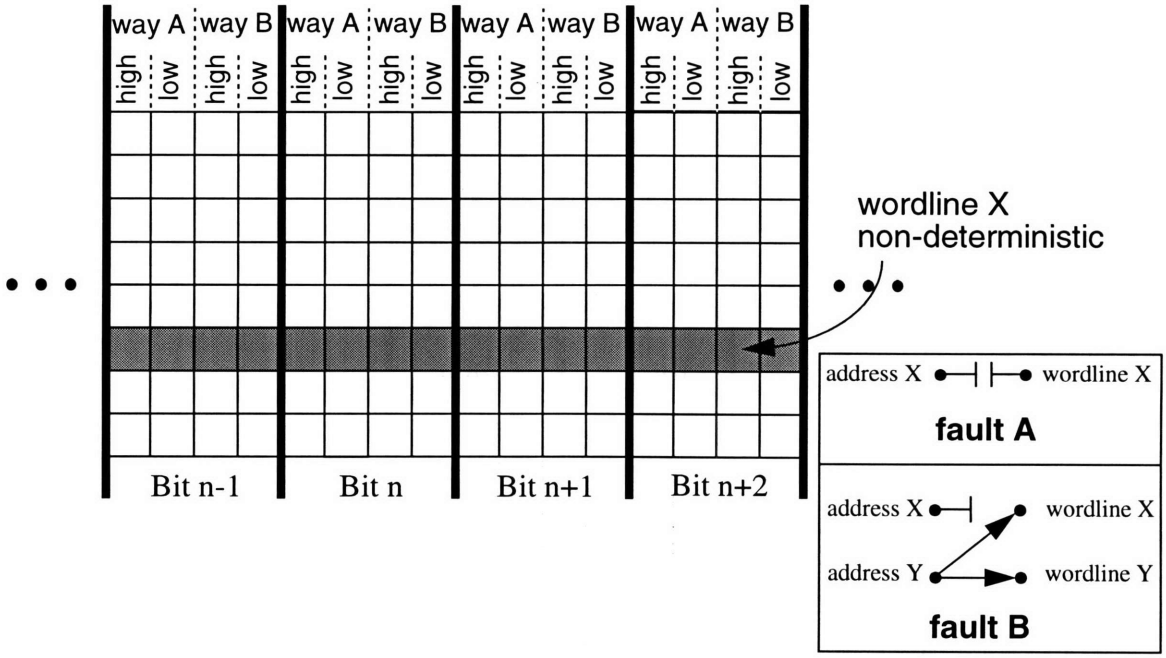


Figure 4.28: Fault signature of row decoder faults A and B

A row decoder with fault C is different. Referring back to Figure 4.27, while in fault A and B there exists a certain address X for which none of the wordline wires fire, in fault C there exists a certain address Y for which an incorrect wordline is fired. Two addresses end up activating the same wordline. Writes to addresses that are accessed by wordline X show up on reads from addresses that are accessed by wordline Y and vice versa; addresses accessed by wordline X appear to be state-coupled to addresses accessed by wordline Y and vice versa. The signature that results from performing a unique address ripple word test on a memory with this type of fault is shown in Figure 4.29. In bit n way B, address 2 is bilaterally state-coupled to address 5 while address 10 is bilaterally state-coupled to address 13. This is true in all ways of all bits all along wordline X and wordline Y. The parts of the test that are circled in Figure 4.29 are the reads from memory where an incorrect value is read. Note that these failures are the signatures for the state-coupling faults shown in Figure 4.6. Any test that can catch state-coupling faults can catch this fault.

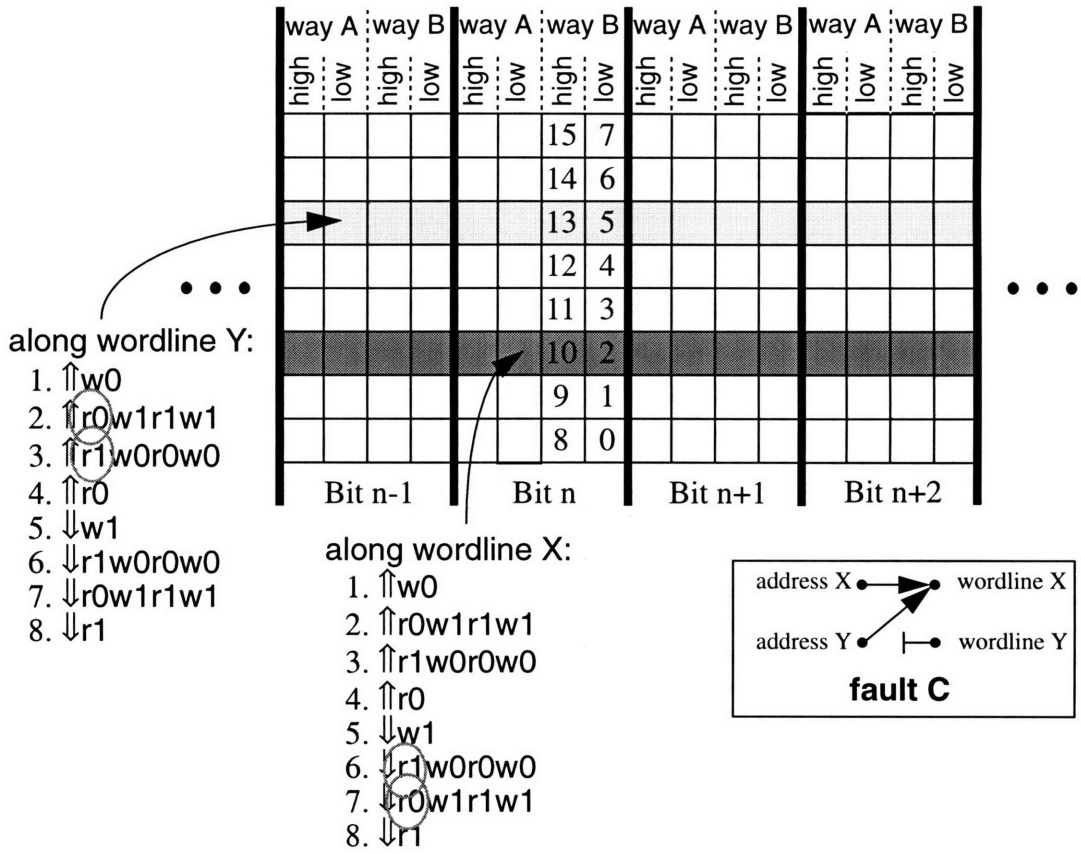


Figure 4.29: Fault signature of row decoder fault C

In fault D, wordline X is activated by both address X and address Y. Like fault C, this results in a state-coupling fault: the cells accessed by address X will have a state-coupling fault. However, accessing address Y activates wordline Y as well as wordline X. The result is that when cell X and cell Y are at different values, the cells that are accessed by address Y are partially non-deterministic. An attempt to read cell Y at this point will also read cell X. This is another case where the sense-amplifier will have identical values on both inputs. Figure 4.30 shows the fault signature for fault D in the row decoder under a unique address ripple word test. In way A of bit n+1, addresses 2 and 10 have state-coupling faults while addresses 5 and 13 have partially non-deterministic faults. This fault is replicated across both ways of all bits. The circled parts of the test indicate the locations where an incorrect value is read out. This is an example of a state-coupling fault. The

crossed out parts of the test indicate the locations where a non-deterministic value may be read. A test that can catch state-coupling faults and non-deterministic faults can catch this fault.

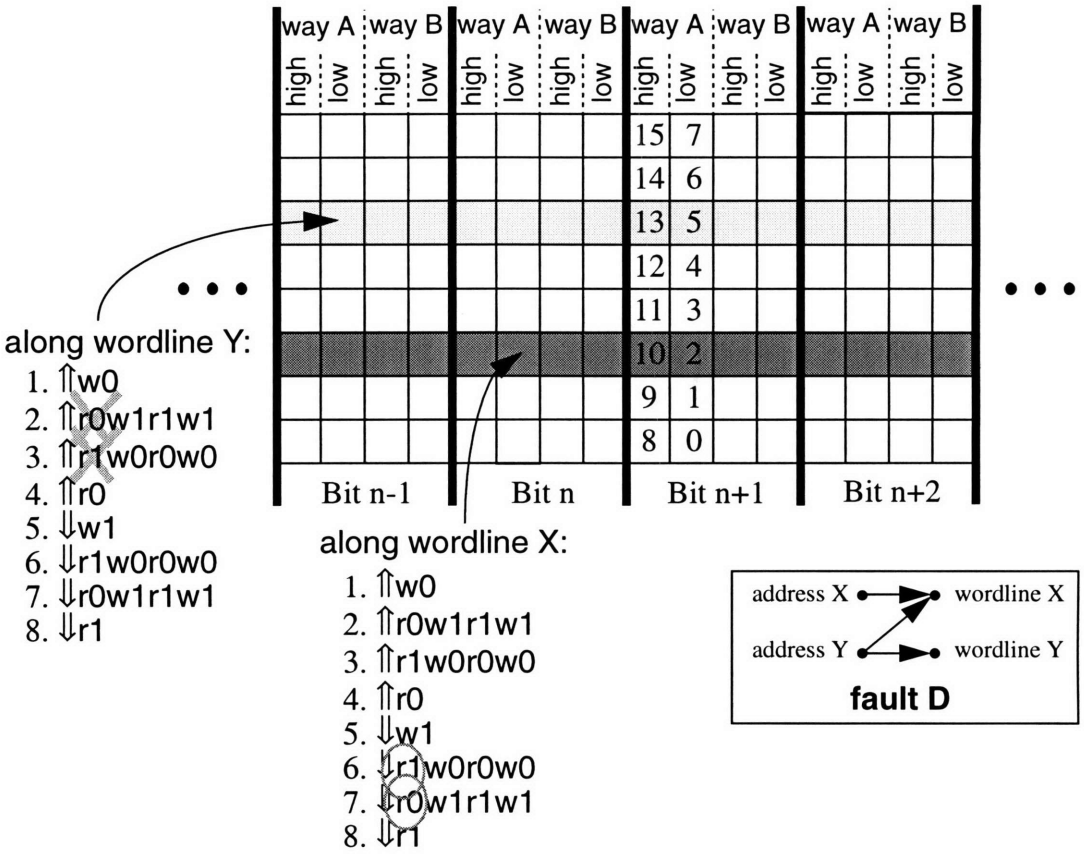


Figure 4.30: Fault signature of row decoder fault D

4.4.2 Column Decoder Faults

As with the row decoder, faults A and B in the column decoder appear to be identical. In both faults A and B, there will exist a bitline selector X (Figure 4.31) which will never become active. Thus, the set of addresses that are accessed by these bitline selectors will never be accessed for reads or writes. All reads to these cells are non-deterministic. The fault signature for these two types of faults is shown in Figure 4.32. Any test that can catch non-deterministic faults can catch this fault. This fault is repeated across all bits and for all

addresses that are accessed by this bitline, both in way A and way B (assuming they share the same address decoder).

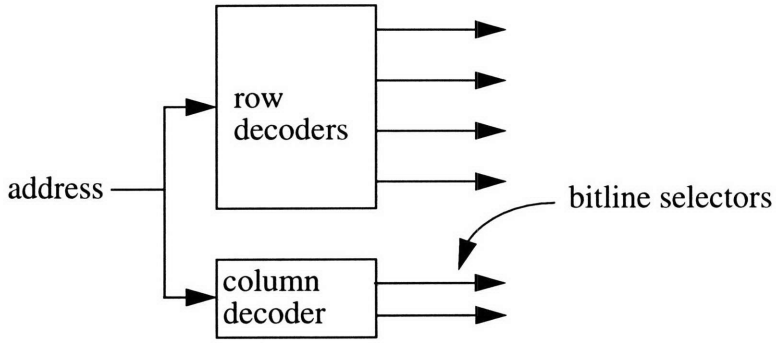


Figure 4.31: Address decoder bitline selectors

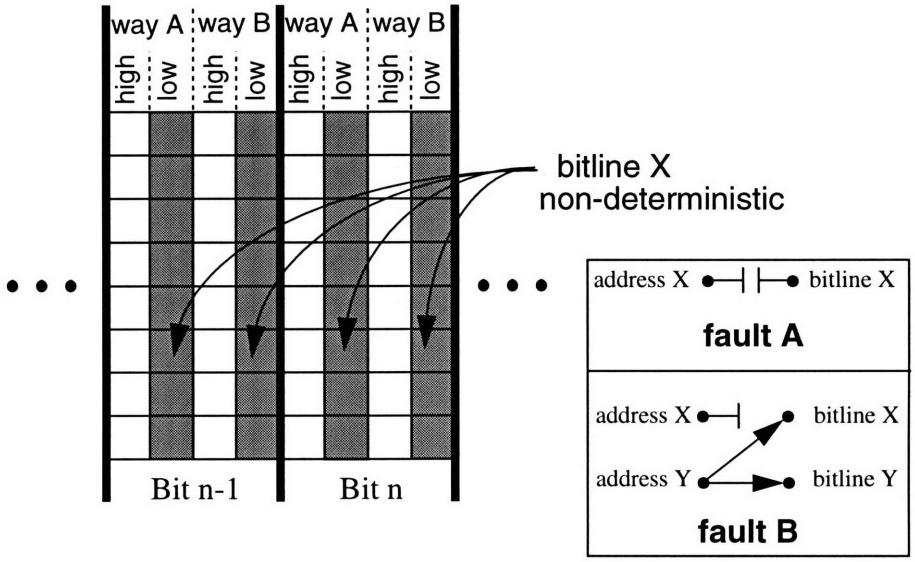


Figure 4.32: Fault signature of column decoder faults A and B

Referring to Figure 4.31, the result of fault C is that one of the bitline selectors will fire at the wrong time; two addresses will end up causing the same bitline selector to fire. Thus, writing to an address that is accessed by bitline X will appear to change the results of reading a memory cell from an address that is accessed by bitline Y, and vice versa. Using the unique address ripple word test, this fault manifests itself as the state-coupling fault shown in Figure 4.33. In this example, in way A of bit n-1, address 0 and 8 are bilaterally state-coupled (as are 1 and 9, 2 and 10, etc.). These faults are replicated across both ways of all bits. Note the circle indicates the locations where an incorrect value is read.

The faults shown are the same as the state-coupling faults shown in Figure 4.6. This fault repeats itself across all bits and for all addresses that are accessed by bitline X and bitline Y. Any test that can catch state-coupling faults can catch this fault.

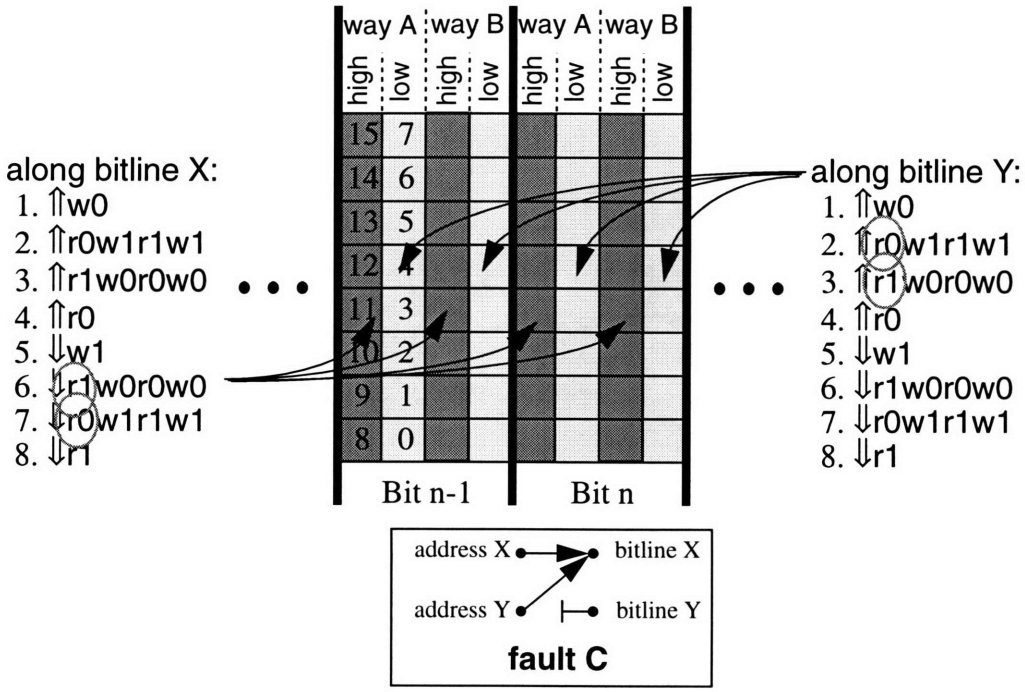


Figure 4.33: Fault signature of column decoder fault C

In fault D, bitline selector X is accessed, correctly, by address X. However, another address Y will also access bitline selector X. Referring to Figure 4.31, in the column decoder an incorrect bitline selector will fire for address Y. An attempt to read a value from address X will yield a value that is dependent not just on what was written to address X but what was written to address Y. Therefore, all addresses that access bitline selector X will exhibit a state-coupling fault with corresponding addresses that access bitline selector Y. This is similar to fault C. However, in fault D, an access of address Y does not just activate the faulty bitline selector X, but also activates bitline selector Y. If the two cells being read have the same value, the correct value will be read out of address Y. However, if the two cells read are at different values, the sense amplifier will be trying to read contradicting data from the two cells. A non-deterministic result is read. The fault signature for fault

D is shown in Figure 4.34. In this example, in way A of bit n, address 0 (and 1 and 2, etc.) has a non-deterministic fault. Address 8 (and 9 and 10, etc.) has a state-coupling fault. These faults are replicated across both ways of all bits. Any test that can catch state-coupling faults and non-deterministic faults can catch this fault.

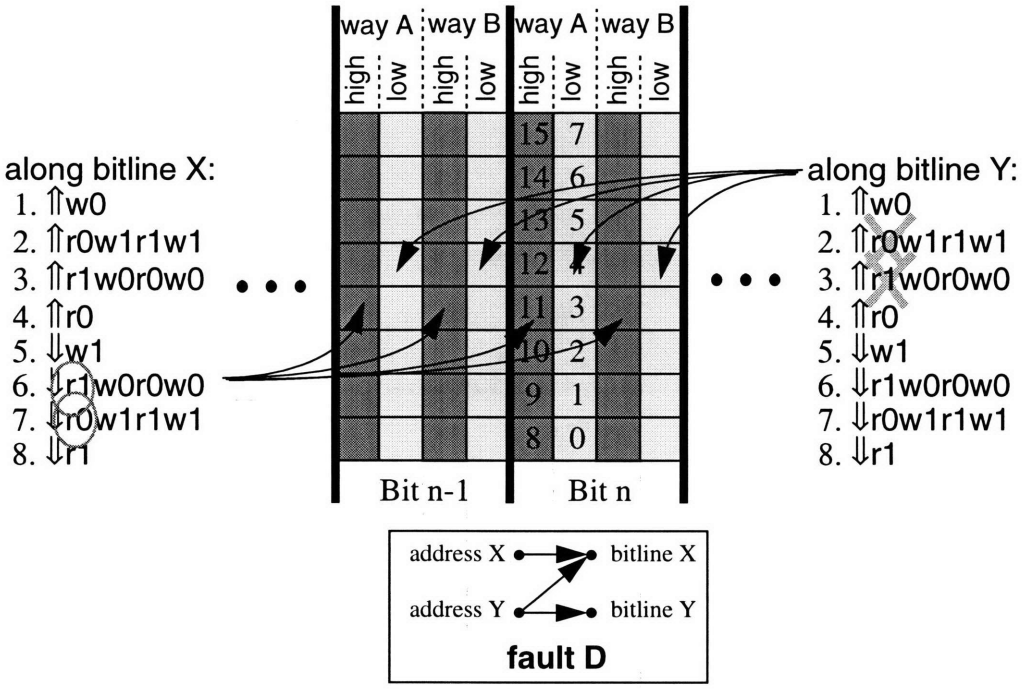


Figure 4.34: Fault signature of column decoder fault D

4.4.3 Wordline/Bitline Selector Faults

There are two types of faults for both the wordline and for the bitline selector. The first type of fault is a stuck-at fault (e.g., the wordline may be stuck-at 1 or stuck-at 0). The second type of fault is a break in the wordline or bitline selector.

In the case that either the wordline or bitline selector is stuck-at 0, that particular wire is never activated. This is identical to fault A in the address decoders. Therefore the fault signature for a wordline stuck-at 0 fault is the same as shown in Figure 4.28. Similarly, the fault signature for a bitline selector stuck-at 0 fault is the same as shown in Figure 4.32.

In the case that either the wordline or bitline selector is stuck-at 1, that particular wire is always active. This is equivalent to a special case of fault D (see Figure 4.35). Regard-

less of what other wordline (or bitline selector) is supposed to be active, the wordline (or bitline selector) with the stuck-at fault (in this case X) will always be active.

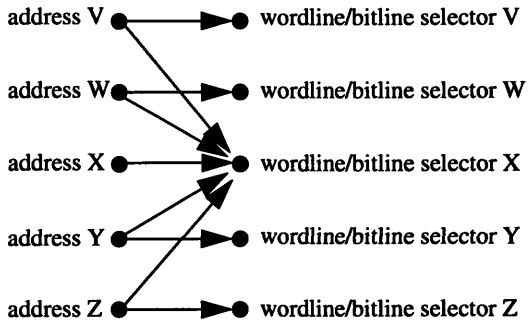


Figure 4.35: Wordline/bitline selector stuck-at 1 fault

Assuming that the fault is a wordline stuck-at fault (the case of the bitline selector is entirely analogous), the faulty wordline will be affected by all writes to the memory, both to addresses before and after it. Therefore, the cells along wordline X will exhibit the symptoms of both state-coupling faults from a previous cell (Figure 4.6a) and state-coupling faults from a following cell (Figure 4.6b). Accessing the cells along the other wordlines will also access cells along wordline X. The result will be non-deterministic should the cell's value be different from the value of a corresponding cell in wordline X. The fault signature for this fault is shown in Figure 4.36 for a unique address ripple word test. The circled reads are reads where an incorrect value is read. The crossed reads are reads where the result is non-deterministic. Note that if wordline X was either the top-most or bottom-most wordline, it would exhibit a state-coupling fault from one direction only. Any test that can catch non-deterministic faults and state-coupling faults will catch this fault.

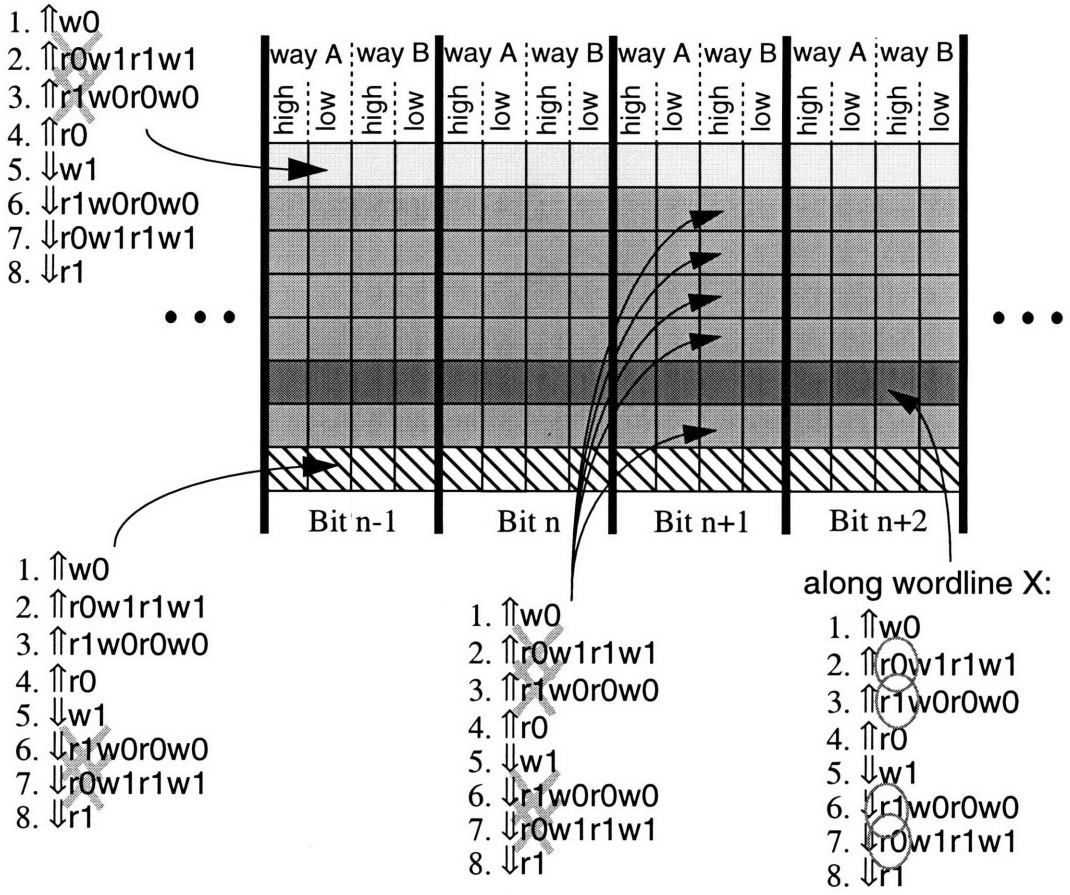


Figure 4.36: Fault signature of stuck-at 1 fault on wordline

The fault signature for a wordline and bitline break is fairly straightforward. All memory cells for all bits beyond the break and away from the decoder will be effectively stuck open (non-deterministic). They will never be accessed. Therefore the fault signature for the wordline break will appear as it does in Figure 4.37. Cells before the break can be accessed normally. Cells beyond the break are non-deterministic.

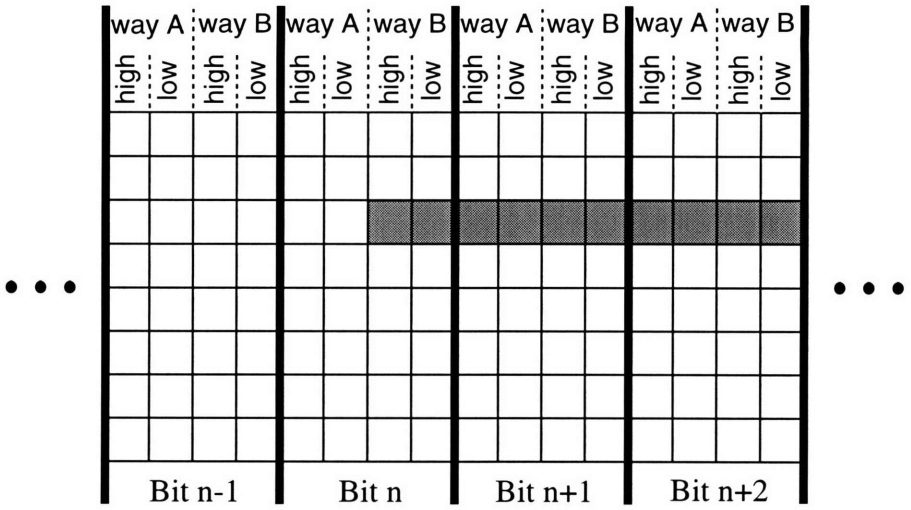


Figure 4.37: Fault signature of wordline break

An analogous situation holds for the bitline selector breaks. For addresses beyond the break that use the bitline selector with the fault, no cell will be accessed. Addresses before the break will function normally. This is illustrated in Figure 4.38.

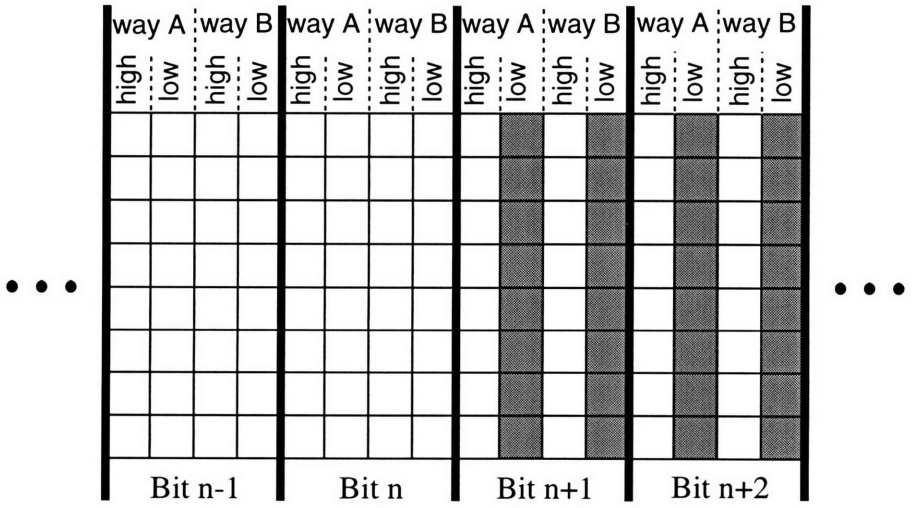


Figure 4.38: Fault signature of bitline selector break

Chapter 5

Summary and Conclusions

The objective of this thesis is a methodology for memory array diagnosis. Initially, a fault model was described (Chapter 2). The tests being used to catch these faults was then presented (Chapter 3). Finally, the way the faults manifested themselves in the tests (each faults' fault signature) was described (Chapter 4). Now that these fault signatures have been introduced, the complete diagnostic flow can be shown (see Figure 5.1).

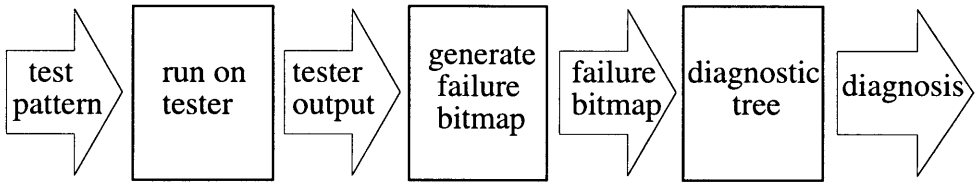


Figure 5.1: Diagnostic flow

First, the test patterns need to be generated. These patterns are applied by the tester to the chip being tested. The resulting tester output is then analyzed to generate a failure bitmap. By applying the criteria given by the diagnostic tree to the failure bitmap, a final diagnosis can be made. The most important result of this study was the creation of this diagnostic decision-making process or the diagnostic tree. Furthermore, examining tests and the faults they catch from the point of view of diagnosis forces a more detailed look at the specific faults encountered and therefore a better understanding of what tests are unnecessary or how the tests can be optimized.

5.1 Diagnostic Tree

The information about Diagnosis contained in Chapter 4 can be condensed into a diagnostic tree (see Figure 5.2 for base of tree). This is a decision making guide that allows a diagnosis to be made based upon a given failure bitmap. The following four figures show how a failure bitmap can be interpreted to make a diagnosis. Initially, the memory is tested with

the unique address ripple word test. This version of the unique address ripple word test contains the double read necessary to catch destructive read faults (see Section 4.2.5). Figure 5.2 shows how the failure bitmaps resulting from this test can be sorted depending on the scope of the fault. Faults in the memory cell array, read/write logic, and address decoder all have different scopes and Figure 5.2 gives the appropriate next figure to look at depending on the scope of the faults in the failure bitmap. The fault signatures for memory cell array faults are shown in Figure 5.3. The fault signatures for read/write logic faults are shown in Figure 5.4. Finally, the fault signatures for address decoder faults are shown in Figure 5.5.

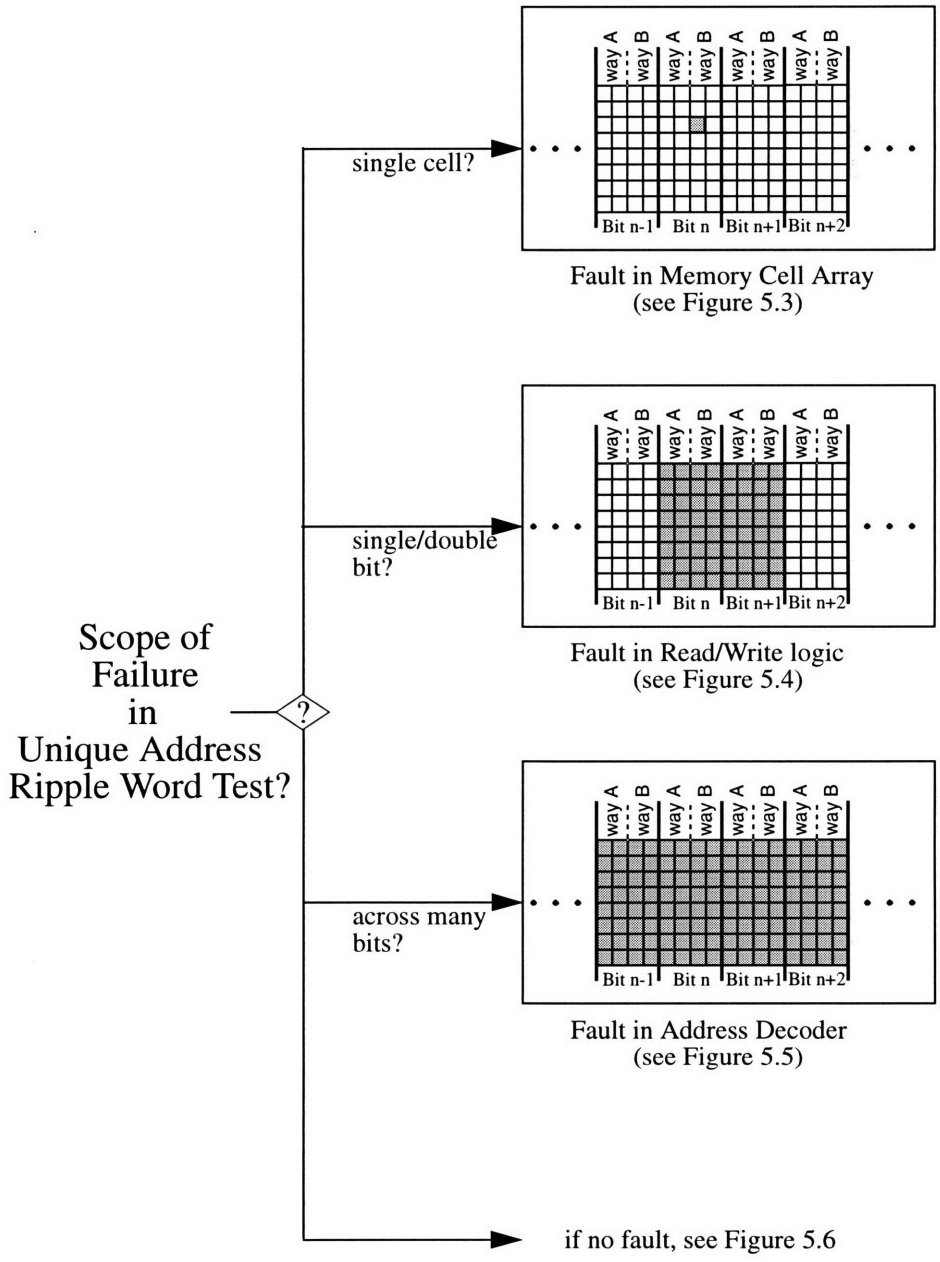


Figure 5.2: Scope of failure

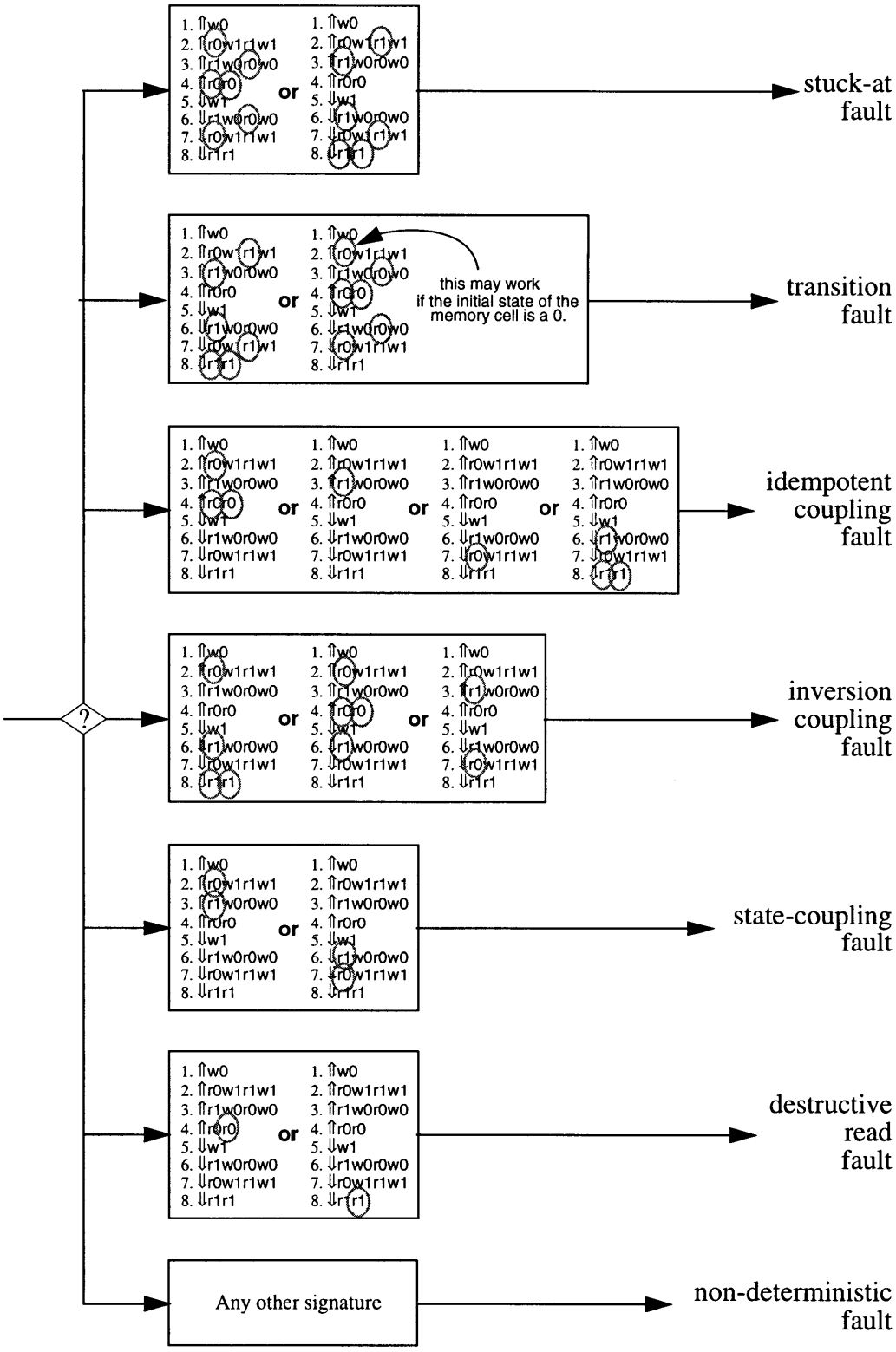


Figure 5.3: Memory cell array faults

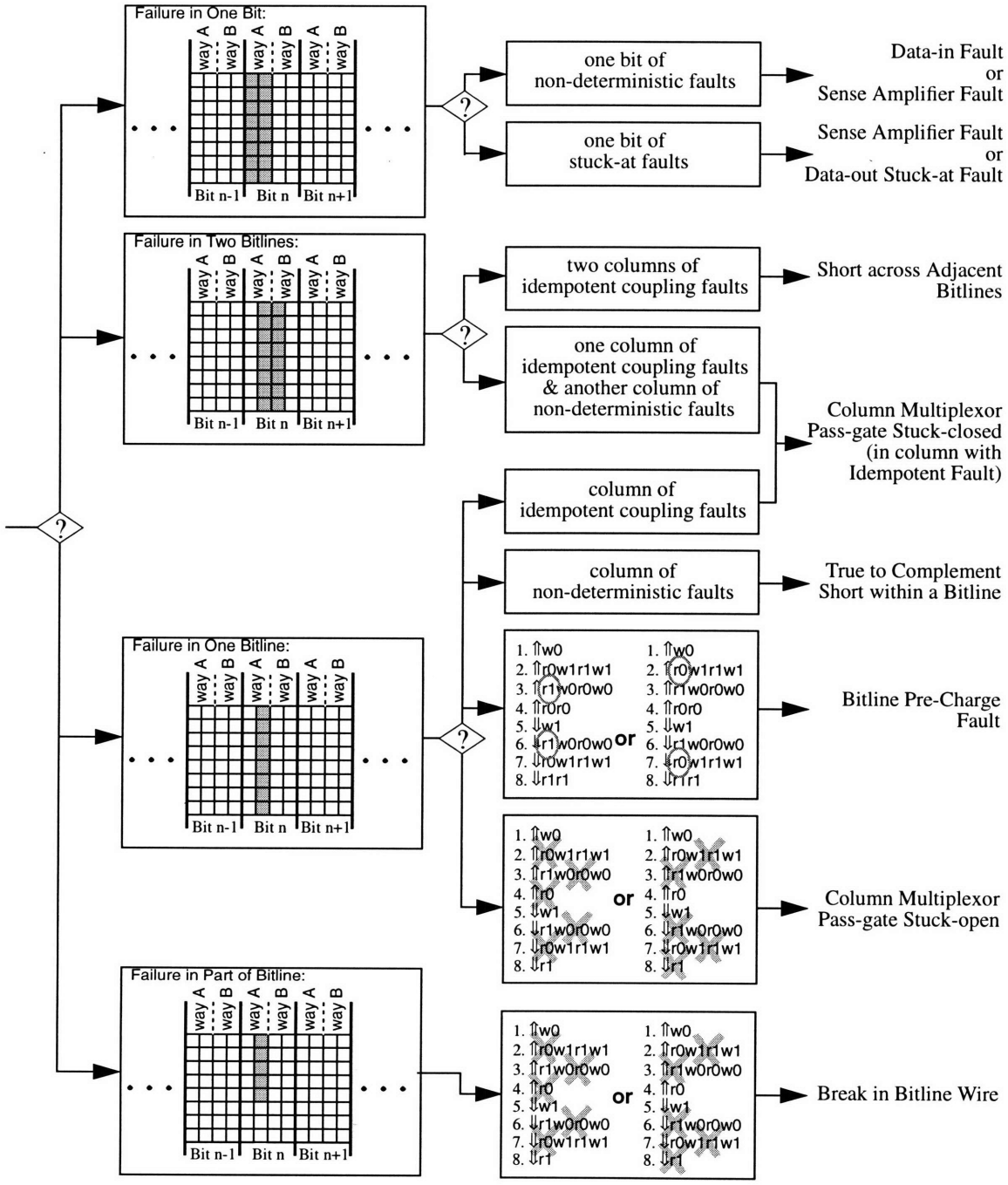


Figure 5.4: Read/write logic faults

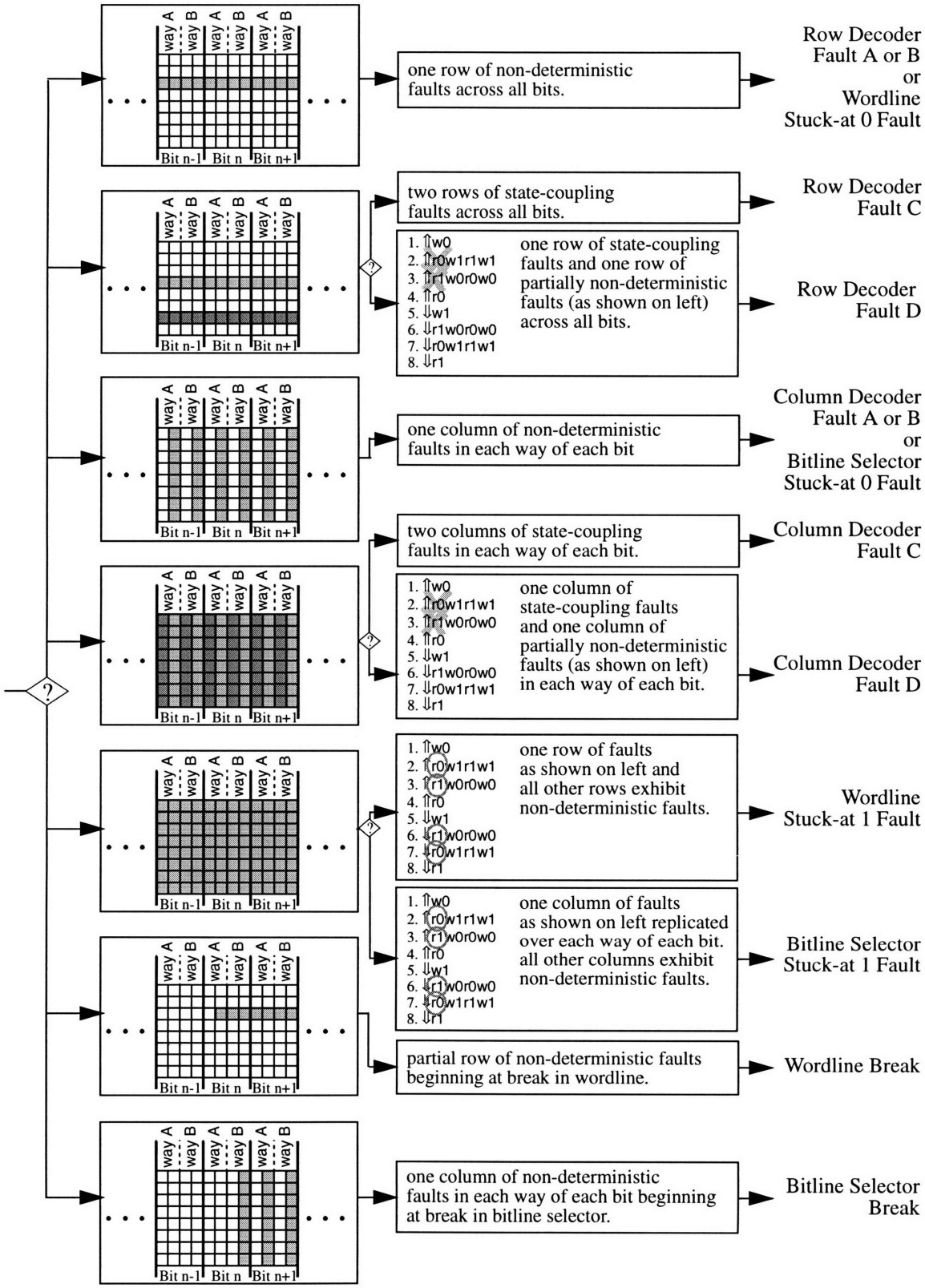


Figure 5.5: Address decoder faults

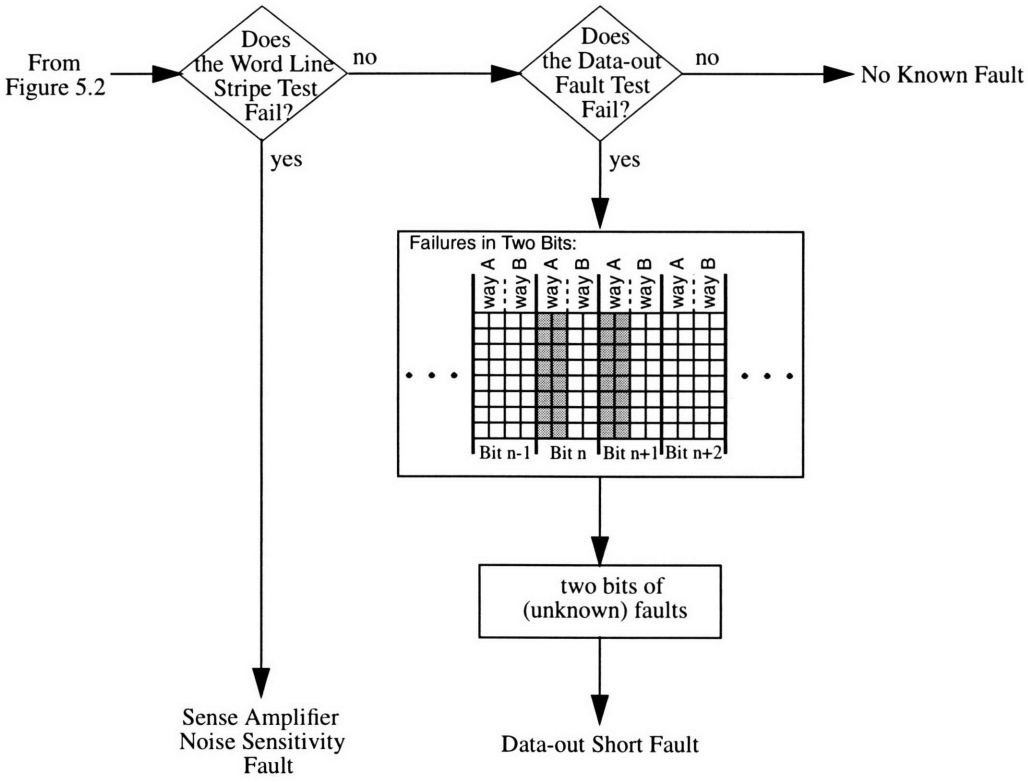


Figure 5.6: Extra test for data-out faults

If there are no faults in the unique address ripple word test, then the word line stripe test is applied to test for sense amplifier sensitivities to noise (see Section 4.3.5). If this test is also successful, the additional data-out fault test described in Section 4.3.6 is used to test for data-out short faults. This is the test whereby consecutive bits are written with different values and then read out of memory. If a fault manifests itself as shown in Figure 5.6, then a data-out fault exists. Otherwise, no known fault exists. Using this diagnostic tree, a diagnosis can be made from the failure bitmap.

5.2 Test Optimization

The study of memory array diagnosis leads to a better understanding of the faults encountered in memory arrays. This knowledge can be applied to discard or optimize memory

tests. Of the original three tests presented, the unique address ripple word test can be used to catch most faults. The remaining faults do not need to be caught with full march tests like the checkerboard or word line stripe tests.

For the given implementation of memory, the checkerboard test did not catch any additional faults once the double read was also included in the unique address ripple word test. This checkerboard test can be discarded.

The word line stripe test was only used to catch sense amplifier noise sensitivity faults (described in Section 4.3.5). This is significant because the word line stripe test did not need to march through every address. To test the sense amplifier, all that is required is that a value of all 1s and all 0s is read out one after the other, for some k number of times. Thus, instead of a march test of order n (where n is the number of addresses), this tests can be of order k (a constant).

The last fault, the data-out fault, can be caught with the data-out fault test described in Section 4.3.6. This was another test where the addresses used were irrelevant; a full march test would merely be a waste of tester time and memory. This test only required that a few words be written to and read from memory; each of these words must have alternating bits at different values. Again, this is also a test of order k.

Appendix A

Shortening Test Simulation Time and Reducing Tester Buffer Memory Requirements

A.1 Introduction

The tests described in this thesis are functional tests. They run on the microprocessor as assembly programs. Most of these tests are march tests and their length is dependent on the number of memory cells that the test must march through. These tests can become considerably large and the amount of time required to simulate and run these tests as well as the amount of tester buffer memory required to store these tests can be enormous. The amount of time required to simulate these tests is important because all the assembly programs need to be simulated in order to generate the patterns that the tester applies to the processor. The amount of time required to run these tests is important because during manufacturing the test patterns need to be applied to every processor and so any decrease in test run time increases the throughput for the manufacture of the processors. The amount of tester buffer required is important because there is only a limited amount of tester buffer memory available in the test equipment and buying extra memory is either costly or not possible. Shortening the run time of these tests involves optimizing the tests themselves, which is discussed in Section 5.2. This appendix deals with shortening the tester simulation time and reducing tester buffer memory requirements, given a particular set of tests.

A.2 Reducing the Simulation Time

Two strategies to reduce the simulation time were considered. The first was to split the unique address ripple word test (the longest of the three tests) into four parts. The second was to pretend that the memory was broken into parts and simulate the tests on the separate parts. In this way, though the total simulation time may be unchanged or longer, each

of these parts is independent; therefore, many parts can be simulated in parallel.

A.2.1 Breaking up the Unique Address Ripple Word Test

One method of shortening the amount of simulation time was to break up the longest test, the unique address ripple word test. The test can be easily broken into four parts without loss of diagnosis as shown in Figure A.1. Notice how in part 2 and part 4, steps 2b and 6b have been added, respectively. These two extra steps are necessary to continue to trigger all the appropriate coupling faults.

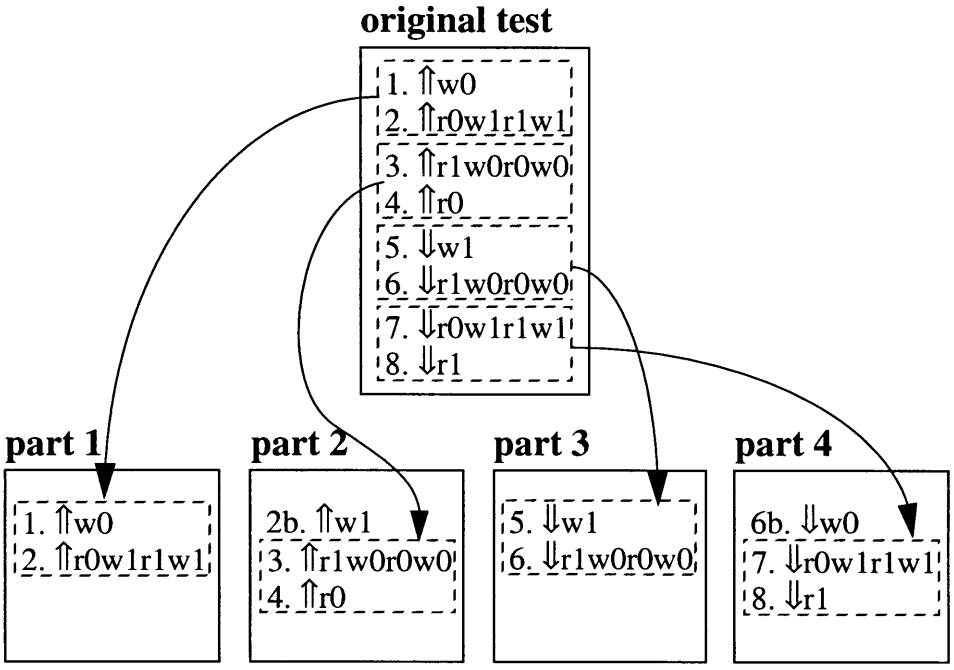


Figure A.1: Breaking up the unique address ripple word test

All four parts need to be run in order to do diagnosis. Their results can be put together to generate a complete failure bitmap and diagnosis can be done as before. Notice again that only the simulation time for each part has been shortened. The amount of tester buffer memory required and the total simulation and run time of all four parts has been increased due to the two extra steps, 2b and 6b.

A.2.2 Breaking up the Memory

A second method of shortening the amount of simulation time was to divide the memory and simulate the tests on each piece separately (each piece being a certain set of

addresses). However, the memory cannot be broken up haphazardly, because both test and diagnosis will be affected. There are three rules, enumerated below.

The memory cannot be broken into pieces as shown in Figure A.2a. In this case, the test is run separately on the top half and the bottom half. The problem is that not all coupling faults will be caught. For example, in Figure A.2a, a coupling fault between memory cell X and memory cell Y will never be caught. The source that triggers the fault and the target of the fault are in different parts. Assuming that coupling faults only occur between adjacent memory cells (otherwise, breaking memory into pieces will not work at all), the first rule is that adjoining pieces must overlap; the memory cells at the boundary between any two pieces must be a part of both pieces. This is illustrated in Figure A.2b, in which case the darker row is tested as part of both the top piece (part 1) and the bottom piece (part 2). Coupling faults between memory cell X in Figure A.2b and memory cells both above and below it will be caught.

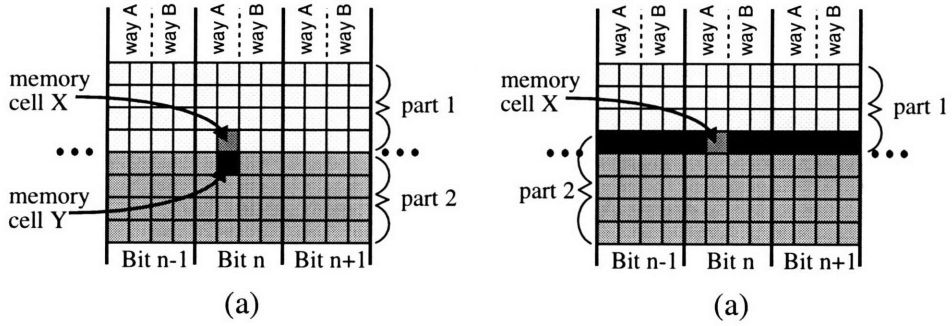


Figure A.2: Preserving coupling faults

The second rule is that regardless of how many smaller pieces the memory is broken into, there needs to be a piece where all the wordlines are accessed in order to test for row decoder faults. For instance, in addition to the two pieces shown in Figure A.2b, there

needs to be a third piece as shown in Figure A.3 where every one of the wordlines is accessed.

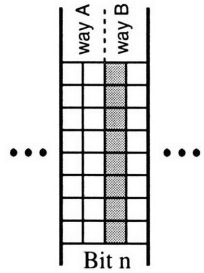


Figure A.3: Catching row decoder faults

An analogous rule is that a piece must exist where all the bitline selectors are accessed so that column decoder faults can be caught. Fortunately, this is already true in Figure A.2b. In general, there needs to be a piece as shown in Figure A.4 where every one of the bitline selectors is accessed.

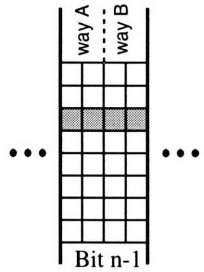


Figure A.4: Catching column decoder faults

As with breaking up the unique address ripple word test, breaking up memory only decreases the simulation time. It increases both the run time and the required tester buffer memory because of the redundant testing of certain memory cells (i.e., the overlapped memory cells, or testing the same memory cells to account for row decoder or column decoder faults).

A.3 Reducing the Tester Buffer Memory Requirements

Two strategies were pursued to reduce the tester buffer memory requirements. The first was to turn on the microprocessor's instruction cache during these tests. The second was to compress the data in the tester memory.

A.3.1 Enabling the Cache

The assembly code for the memory array tests is not very long. These tests are written as iterative loops that increment or decrement through memory. Part of the reason for the huge tester memory requirements is that the tester is constantly feeding the processor with the next instruction. With the cache disabled, the tester needs to continuously supply the next instruction to the processor. The looping nature of the code means that the same instruction will be repeatedly supplied by the tester for each iteration of the loop. With the cache enabled, only the first iteration of the loop needs to be supplied to the processor where it can be cached. An example of the savings that enabling the cache provides is shown in Table A.1.

test type	cache disabled	cache enabled
checker-board	34,656	*17,251*
unique address ripple word	107,345	*56,242*
word line stripe	22,696	*11,492*
TOTAL	164,697	*84,985*

Table A.1: Required lines in tester buffer used for 16 entry dual-ported memory array with and without the cache

Note that this method only provides savings in the amount of tester buffer memory required. It does not affect the amount of time it takes for the tests to run or to simulate.

A.3.2 Memory Compression

Once cached, the tester is often idle, supplying the same information to the processor every clock cycle; turning on the cache allows the processor to do a lot of work without needing information from the bus. Instead of having the tester supply the same exact information each clock cycle, this information can be compressed into one line along with a repeat value that tells the tester how many times to repeat it. This compression is the sec-

ond method of addressing the tester buffer memory requirements. The further savings from compressing the tester memory is shown in Table A.2.

test type	just cached	cached & compressed
checker-board	*17,251*	*2,140*
unique address ripple word	*56,242*	*4,025*
word line stripe	*11,492*	*1,150*
TOTAL	*84,985*	*7,315*

Table A.2: Required lines in tester buffer used for 16 entry dual-ported memory array with and without compression

As with enabling the cache, the savings here are only in terms of the amount of required tester buffer memory. The time required to run and simulate remain the same.

Bibliography

[1] M.S. Abadir and H.K. Reghbati, Functional Testing of Semiconductor Random Access Memories, *ACM Computing Surveys*, 15(3), September 1983, p. 177.

[2] A.J. van de Goor, *Testing Semiconductor Memories: Theory and Practice*, sections 2.4-2.5, p. 45-59, John Wiley & Sons, West Essex, England, 1991.

[3] R. Dekker, F. Beenker, and L. Thijssen, Fault Modeling and Test Algorithm Development for Static Random Access Memories, *Proceedings of the IEEE International Test Conference*, 1988, p. 345.

[4] N.H.E. Weste and K. Eshraghian, *Principles of CMOS VLSI design*, Second Edition, p. 564-565, Addison-Wesley Publishing Company, Reading, Massachusetts, 1993.

[5] Based on personal discussions with R. Dean Adams, IBM Microelectronics, Burlington, VT, Summer, 1995.

[6] A.J. van de Goor, *Testing Semiconductor Memories: Theory and Practice*, section 3.2, Chapter 4, p. 69,93-170, John Wiley & Sons, West Essex, England, 1991.

[7] A.J. van de Goor, *Testing Semiconductor Memories: Theory and Practice*, chapter 11, p. 323-337, John Wiley & Sons, West Essex, England, 1991.

[8] Crouch, Alfred L. et al, "Testability Features of the MC68060 Microprocessor", International Test Conference, p.60, Computer Society Press of the IEEE: Washington, D.C., 1994.