National Library of Canada
Acquisitions and Bibliographic Services Branch
395 Wellington Street
Ottawa, Ontario
K1A 0N4

Bibliothèque nationale du Canada
Direction des acquisitions et des services bibliographiques
395, rue Wellington
Ottawa (Ontario)
K1A 0N4

Your file Votre référence

Our file Notre référence

NOTICE

The quality of this microform is heavily dependent upon the quality of the original thesis submitted for microfilming. Every effort has been made to ensure the highest quality of reproduction possible.

If pages are missing, contact the university which granted the degree.

Some pages may have indistinct print especially if the original pages were typed with a poor typewriter ribbon or if the university sent us an inferior photocopy.

Reproduction in full or in part of this microform is governed by the Canadian Copyright Act, R.S.C. 1970, c. C-30, and subsequent amendments.

AVIS

La qualité de cette microforme dépend grandement de la qualité de la thèse soumise au microfilmage. Nous avons tout fait pour assurer une qualité supérieure de reproduction.

S'il manque des pages, veuillez communiquer avec l'université qui a conféré le grade.

La qualité d'impression de certaines pages peut laisser à désirer, surtout si les pages originales ont été dactylographiées à l'aide d'un ruban usé ou si l'université nous a fait parvenir une photocopie de qualité inférieure.

La reproduction, même partielle, de cette microforme est soumise à la Loi canadienne sur le droit d'auteur, SRC 1970, c. C-30, et ses amendements subséquents.

Canada

Built-In Self-Diagnosis for Repairable Embedded RAMs

Robert Treuer

Department of Electrical Engineering

McGill University, Montreal

September 1992

A Thesis submitted to the Faculty of Graduate Studies and Research

in partial fulfillment of the requirements for the degree of

Doctor of Philosophy.

© Robert Treuer, 1992.

National Library of Canada
Acquisitions and Bibliographic Services Branch
395 Wellington Street
Ottawa, Ontario
K1A 0N4

Bibliothèque nationale du Canada
Direction des acquisitions et des services bibliographiques
395, rue Wellington
Ottawa (Ontario)
K1A 0N4

Your file Votre référence

Our file Notre référence

The author has granted an irrevocable non-exclusive licence allowing the National Library of Canada to reproduce, loan, distribute or sell copies of his/her thesis by any means and in any form or format, making this thesis available to interested persons.

L'auteur a accordé une licence irrévocable et non exclusive permettant à la Bibliothèque nationale du Canada de reproduire, prêter, distribuer ou vendre des copies de sa thèse de quelque manière et sous quelque forme que ce soit pour mettre des exemplaires de cette thèse à la disposition des personnes intéressées.

The author retains ownership of the copyright in his/her thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without his/her permission.

L'auteur conserve la propriété du droit d'auteur qui protège sa thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

ISBN 0-315-87677-8

Canada

Built-in self-diagnosis for repairable embedded RAMs

Abstract

Embedded random access memories (RAMs) are increasingly being tested using built-in self-test (BIST) circuits, because an embedded RAM's signals are not accessible through input/output pins. Given current trends, the size of embedded RAMs will eventually grow so large that yield considerations will require the use of redundant lines for repair. Then, BIST circuits will need to not only detect faults, but also locate faults for repair. The designs of two different BISD circuits (one with, and the other without, self-repair capability) appropriate for repairable, embedded, (single-metal and single-polysilicon layered) CMOS static RAMs, are presented. The implementation of a BISD circuit — with self-repair — requires 15% extra area in a 16K SRAM, and 5% extra area in a 64K SRAM. The BISD circuit contains a reduced-instruction-set processor, which executes diagnosis algorithms stored in a read-only memory, and which uses some extra lines to access the memory. The new algorithms employ "hybrid serial/parallel" operations to access the memory when external repair is available, or "modular" operations to access it when self-repair is required.

Résumé

De plus en plus souvent, les mémoires vives encadrées sont examinées pour la présence des défectuosités par moyen des circuits d'auto-vérification incorporés, parce que les signaux d'une mémoire vive encadrée ne sont pas accessible à travers les ports d'entrées/sorties. Suivant les tendances actuelles, la superficie des mémoires vives encadrées s'augmentera tellement que le rendement de fabrication diminué obligera l'usage des lignes redondantes pour la réparation. Alors, les circuits d'auto-vérification incorporés devront non seulement découvrir les défectuosités, mais devront aussi localiser les défectuosités pour la réparation. Les conceptions de deux circuits d'auto-vérification incorporés différents (l'un avec, et l'autre sans, le pouvoir d'auto-réparation), qui sont convenables pour les mémoires vives statiques, encadrées et réparables, et fabriquées en CMOS (avec une couche d'aluminium et une couche de polysilicone), sont présentées. Un circuit d'auto-diagnose incorporé — avec auto-réparation — exige 15% de superficie supplémentaire dans une mémoire vive à 16K, et 5% dans une mémoire à 64K. Le circuit d'auto-diagnose incorporé contient un processeur à peu d'instructions, qui exécute les algorithmes diagnostiques emmagasinés dans une mémoire morte, et qui se sert des lignes supplémentaires pour accéder à la mémoire vive. Les nouveaux algorithmes emploient des opérations "hybrides en série/parallèle" pour accéder à la mémoire quand la réparation externe est disponible, ou des opérations "modulaires" pour y accéder quand l'auto-réparation est requise.

Acknowledgements

I would like to express my sincere gratitude to my thesis supervisor, Prof. Vinod Agarwal, for his advice, his friendship, and his insight, throughout the entire duration of my studies. His enthusiasm and his boundless patience are greatly appreciated.

In addition, I would also like to thank all of the past and present members — both students and staff — of the Lab (i.e. the Microelectronics and Computer Systems Laboratory, formerly known as the VLSI Lab) with whom I have been friends over the years. In particular, special thanks to Jacek Slaboszewicz, Michael Howells, Salim Juma, Christine Marquis, and Charles Arsenault, for their assistance on numerous occasions; and further thanks to my long-time friends Yervant Zorian, André Ivanov, Abu Hassan, and Namhyung Kim, who made my stay in the Lab a more enjoyable experience.

On a personal level, I would like to thank my parents for their constant encouragement, forbearance and support.

This work was supported by scholarships from the Natural Sciences and Engineering Research Council of Canada, and from "le Centre de recherche informatique de Montréal." I also benefitted from an equipment and software grant made by Bell-Northern Research to our Lab.

Table of Contents

List of Figures

List of Tables

1 Introduction

1.1 Motivation for considering this subject

Random Access Memories (RAMs) are the densest circuits being fabricated today. Because their transistors and lines are packed so closely together, RAMs suffer from a very high average number of physical defects per unit area compared with other types of circuits, and this fact has motivated researchers to develop efficient RAM-test sequences that provide good fault coverage [Abadir and Reghbati 83], [van de Goor and Verruijt 90]. The published results have led some researchers to suggest that a better solution to the RAM testing problem can be obtained by redesigning and augmenting the peripheral circuits surrounding the cell array of the RAM, in order to improve the RAM's testability [Inoue *et al.* 87], [Sridhar 86]. Others have proposed the addition of even more extra circuitry to achieve a self-testing RAM, thereby dispensing with expensive external test equipment [Dekker *et al.* 89], [Franklin and Saluja 90], [Franklin *et al.* 90], [Mazumder and Patel 89], [Ohsawa *et al.* 87], [Ritter and Müller 87], [Saluja *et al.* 87], [Takeshima *et al.* 90], [You and Hayes 85]. The principal motivation behind the proposals for designed-for-testability RAMs and self-testing RAMs is the significant reduction in total testing time compared with conventional RAM testing procedures.

An *embedded* RAM usually cannot be tested by simply applying test patterns directly to the I/O pins, because the embedded RAM's data, address and control signals are not accessible through the I/O pins. As a result, there are some published design proposals for built-in self-testing embedded RAMs [Jain and Stroud 86], [Nadeau-Dostie *et al.* 90], [Nicolaidis 85], [Sun and Wang 84]. Given current trends, it is reasonable to expect that eventually the size of some embedded RAMs will grow so large that yield considerations will require these embedded RAMs to have spare rows and spare columns. When this happens, built-in test circuits will be needed that not only detect the presence of faults, but also specify the location of faults for repair purposes. A built-in self-diagnosis (BISD) method, which can test and locate faults in embedded RAMs, and which can be implemented using a small amount of extra area, is the subject of this thesis.

The heart of the proposed method is the Diagnosis and Repair Unit, which is composed of a small reduced-instruction-set-style processor, which executes instructions stored in a small read-only memory (ROM). When the embedded RAM is externally repairable (i.e., it has spare rows and columns that are programmed by blowing fuses with laser beam pulses), the Diagnosis and Repair Unit first locates all the faults, and then sends a repair procedure to the equipment which controls the laser beam. However, when the RAM is internally repairable (i.e., it has "soft fuses," which may be EEPROM cells or ordinary flip-flops), the Diagnosis and Repair Unit will program the "soft fuses" by itself, without help from any external equipment. The same Diagnosis and Repair Unit (DR-Unit) can be used to test, diagnose, and (optionally) repair more than one embedded RAM on the same chip.

1.2 Evaluation Criteria

The list below enumerates a possible set of parameters which can be used to evaluate the overall quality of a given built-in self-testing, self-diagnosing, and (possibly) self-repairing circuit design. Only methods which optimize most or all of these parameters are acceptable for implementation.

* The *area overhead* incurred by the extra circuitry (minimize).
* The effect of the extra circuitry on the *performance* of the original circuit (minimize).
* The time required to apply the tests and perform the diagnosis (minimize).
* The *fraction of faults* located by the built-in self-diagnosis (maximize).
* The repair algorithm's *allocation of spare resources* to repair faults (i.e., the *generated* repair-plan should deviate *minimally* from the *optimal* repair-plan).
* The *self-testability* of the extra circuitry which actually performs the self-diagnosis and self-repair (maximize).
* The *design time* needed by an engineer to implement the given method (by *minimizing* the amount of custom redesign required for each new memory block; ideally, any software that generates layouts of complete memory modules could also generate, as part of each layout, the additional built-in self-diagnosis circuitry).
* The *flexibility* of the design, in particular:
    - independence of the fabrication technology being used,
    - ability to accomodate to new and different fault types as the technology evolves, and
    - applicability, with minor changes, to different types of memories, such as dual-port Static RAMs, and Dynamic RAMs with various addressing capabilities.

1.3 Summary of thesis

This thesis presents the details of how to design both self-diagnosing and self-repairing embedded RAMs (applicable to both Static and Dynamic RAMs), that

can be implemented using a reasonable amount of extra area. Chapter 2 presents a summary of the variety of physical implementations used to obtain static and dynamic RAMs. Chapter 3 presents an organized list of the various fault models described in the literature. Chapter 4 contains a short survey of the most relevant previously published work on the topic of BIST for RAMs and embedded RAMs. Chapter 5 gives a general view of the software components of such a design. The details of the hardware designs are presented in Chapter 6. The specific details of the software components, as they relate to the hardware, are given in Chapter 7.

1.4 Claim of original results to appear in thesis

As will be seen from what follows, the self-diagnosis designs proposed in this thesis, largely satisfy the evaluation criteria enumerated above. In particular, the new proposed designs possess several qualities which make them more attractive than any of the *more obvious* self-diagnosis designs.

(1.) The new designs *make use* of the memory capacity *of the embedded RAM itself* during the execution of some of the diagnosis algorithms, and thereby save much silicon area which would otherwise be an additional part of the Diagnosis and Repair Unit, and which would only be used *to provide temporary storage for the fault maps* generated by some of the algorithms.

(2.) The new algorithms are based on a combination of serial and parallel BIST hardware structures, where previous algorithms dealt with only one kind of BIST hardware structure.

(3.) The new designs are *non-intrusive* because the only extra circuitry that is placed inside the memory array itself is a purely local bus; and because all the other extra circuitry can be placed *almost anywhere* on the same IC chip, provided that the local bus can be routed so that it links the embedded repairable RAM to the Diagnosis and Repair Unit.

(4.) The new designs are *programmable* because they use a small, easily-alterable ROM to store the diagnosis and repair algorithms.

(5.) With a very small allocation of additional hardware, the new designs can *execute relatively complex* diagnosis and repair algorithms, because the designs employ a data-path structure that functions almost like a small reduced-instruction-set processor.

(6.) Chapter 5 contains a general description of how to modify any single-bit march test so that the same algorithm can be applied to any RAM with B-bit words. Previously published books and papers dealing with march tests have not shown the precise details of how this transformation is carried out — in particular, the distinctions between "state-changing" and "state-retaining" march elements have not been made before, nor have the two types of Marching data backgrounds and two types of Walking data backgrounds been differentiated.

2 Physical Description of Embedded RAMs

2.1 General Structure of RAMs

Random access memories (also called read/write memories) allow binary data to be stored and fetched at approximately the same speed, currently ranging from a very fast 10 nanoseconds (ns) to a relatively slow 500 ns, depending on the fabrication technology, and the type of the memory. One measure of the operating speed of a memory is its *access time* t_a, which is the maximum time required to read a word from the memory. It is measured from the application of a new address on the address bus to the appearance of the valid data on the data bus. The process of reading or writing a word requires that various signals be applied to the address, control and data lines, and is called a read or write *memory cycle*. The *memory cycle time* t_c is the minimum time that must elapse between the initiation of two successive memory operations. Currently, the fastest DRAMs have access times of about 50ns, and cycle times of about 100ns.

Semiconductor random access memory (RAM) circuits are volatile; this means that if power to the RAM is shut off, then the stored information is lost. Semiconductor RAMs are mainly classified into two types: dynamic RAMs and static RAMs (the remaining type of RAM is the Content Addressable Memory, or CAM, which will not be considered in this thesis). Dynamic RAMs function by storing low or high voltages across capacitors. Because there is some leakage of

charge off these capacitors, the data must be periodically sensed and restored to their original voltages by *refresh* operations. These refresh cycles must periodically displace the normal data access cycles of a dynamic RAM circuit in order to prevent the irretrievable loss of stored data. Static RAMs do not require such refresh cycles nor any other kind of periodic data access (read or write), but can maintain stored information indefinitely, as long as the power supply is uninterrupted. In spite of the greater functional simplicity of static RAMs, it is dynamic RAMs which are more widely used in large quantities, because the price per bit and power per bit ratios are significantly less for dynamic RAMs than for static RAMs, with equivalent storage capacity. However in chips where the RAMs are embedded, mostly static RAMs are used, because static RAMs have *much faster access times* than dynamic RAMs with equivalent storage capacity.

The memory cell arrays of RAMs are organized into W words of B bits each, for a total information capacity of WB bits. The information capacity of a memory may also be quoted in terms of *bytes*, where 8 bits grouped together are known as a byte. Very often, the number of bits per memory word, designated by B, is a multiple of 8, such as 16 or 32, signifying that each memory word consists of a whole number of bytes, such as 2 or 4 bytes. Full-chip RAMs, that are used in the main memories of computers, are often organized to provide only a single data bit for read/write access at a time (i.e., $B = 1$). Embedded RAMs may have any number of bits per word.

A binary-encoded group of input bits provides an *address* that points to a particular word of B bits. The data will either be read from or written to the addressed word. The access time is independent of the particular address chosen, and is determined by the time it takes one of the address decoder circuits to activate exactly one row or column select line. In most memory designs there are separate row address decoders and column address decoders because of the layout characteristics of the memory cell array, although there are some RAMs which use only row address decoders. In DRAMs there is usually only one address latch to

store the incoming address value (it really is only *half of an address*), and the latch will transmit its contents to either the row decoder or column decoder, as required. This *multiplexing* of addresses in DRAMs was not adopted merely because it results in lower pin-counts, but because the slowness of the sense amplifiers in detecting and amplifying the tiny signals from the cells, allows plenty of time for the column address to be decoded after the row address has been decoded first. In SRAMs, which have much faster read/write circuitry than DRAMs do, the full address is usually stored in one address register, and the appropriate address bits will be simultaneously transmitted to both the row and column decoders.

2.2 Structure of Dynamic RAMs

In DRAMs, the address decoders are also subject to control by the *refresh logic*. When the refresh line is activated by the refresh clock (typically every 1 to 5 milliseconds), the column decoder selects *all* columns. The row decoder selects the row specified by the value in the address latch. All of the bits in the chosen row are read out and then written back ("refreshed") simultaneously. During such a refresh cycle, the refresh logic has the additional function of disabling all input to and output from the data register.

"Reading" or "writing" a bit of data from a DRAM, first requires the selection of a word line by raising it from its *precharged* level of 0 volts to the supply voltage value (usually designated by V_{DD}). All of the transfer gate transistors connected to the selected word line are turned on, and this causes charge to be transferred from each of the capacitors to their respective bit lines, and thus also causes the destructive reading of all the data in the chosen word. Before the charge from the cells was transferred to the bit lines, the bit lines were *precharged* to some pre-established value, usually to one half of the supply voltage. Each bit line has its own sense amplifier, whose function is to detect this charge transfer and to amplify the signal caused by it. The amplified signal ideally has a value of either

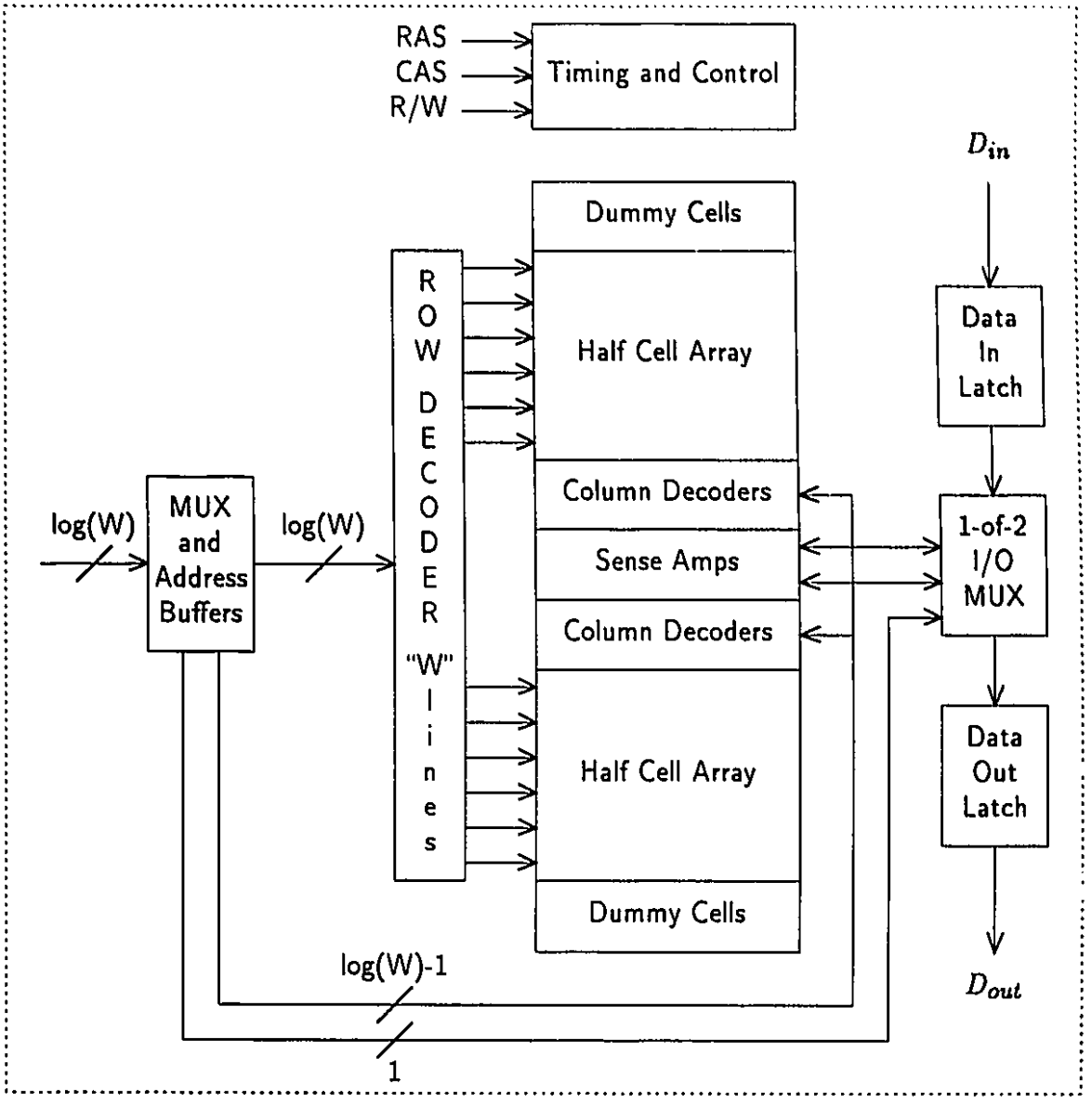

Fig. 2.1 A Typical Dynamic RAM

V_{DD} or ground. The transfer gate transistors continue to remain on throughout this period so that the amplified signals from the sense amplifiers can be fed back into their respective cells to restore the word of data.

"Writing" one bit to a DRAM is performed almost exactly like "reading" one bit. The only difference is that exactly one sense amplifier does not feed back the sensed bit line value to restore the charge to one cell, but it takes a new value from

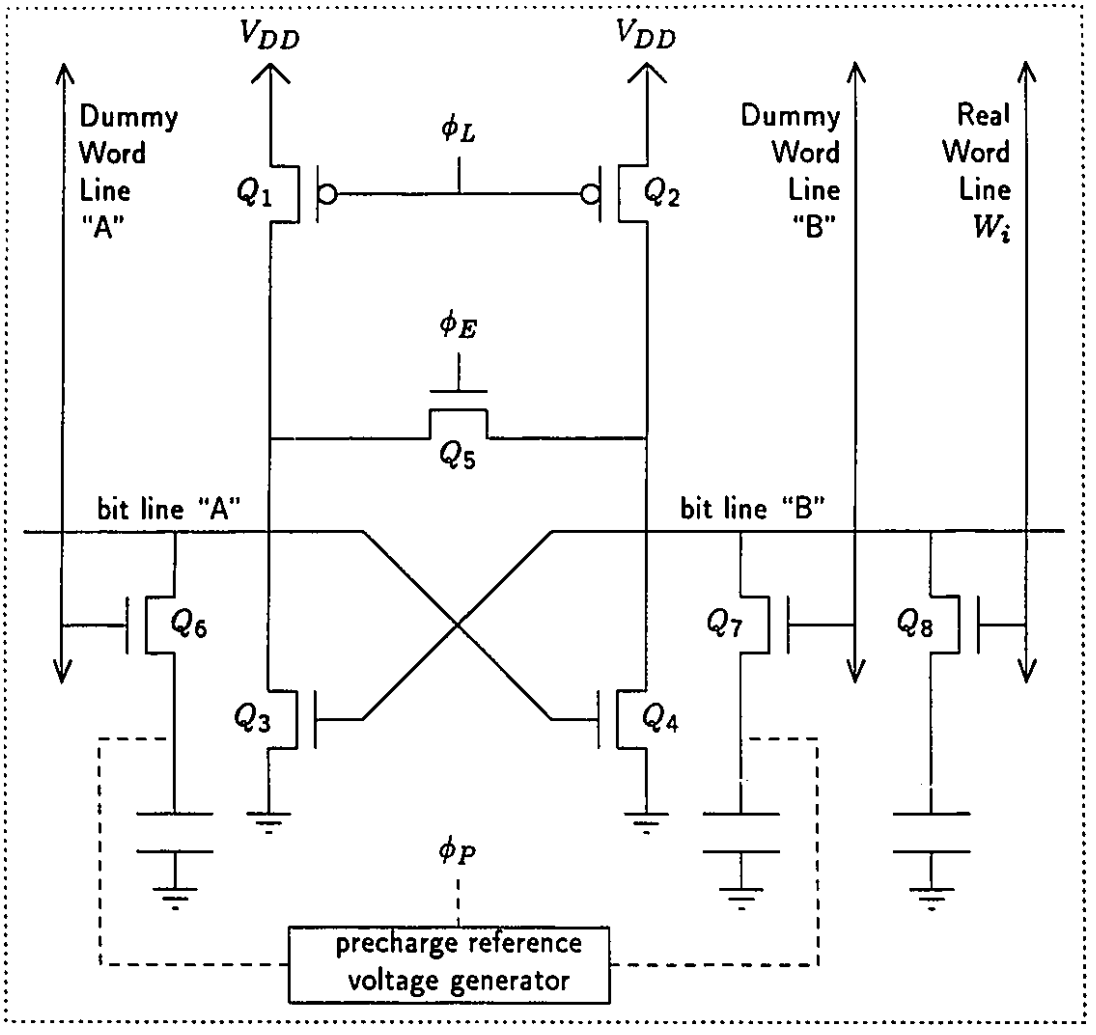

Fig. 2.2 Sense Amplifier with Dummy Word Lines

the data latch which stores a data bit that has been transmitted to the DRAM, and asserts this new value on the bit line to be stored into the one cell.

To maximize the signal into the sense amplifier, a large cell capacitance and a small bit line capacitance are desired. The bit line capacitance can be reduced by simply cutting the line in half and placing the sense amplifier in the middle between the two halves. As figure 2.1 shows, a DRAM makes use of half cell-arrays, dual column decoders, a central row of sense amplifiers, and dummy cells located on both sides, which are used to establish a voltage reference for the sense

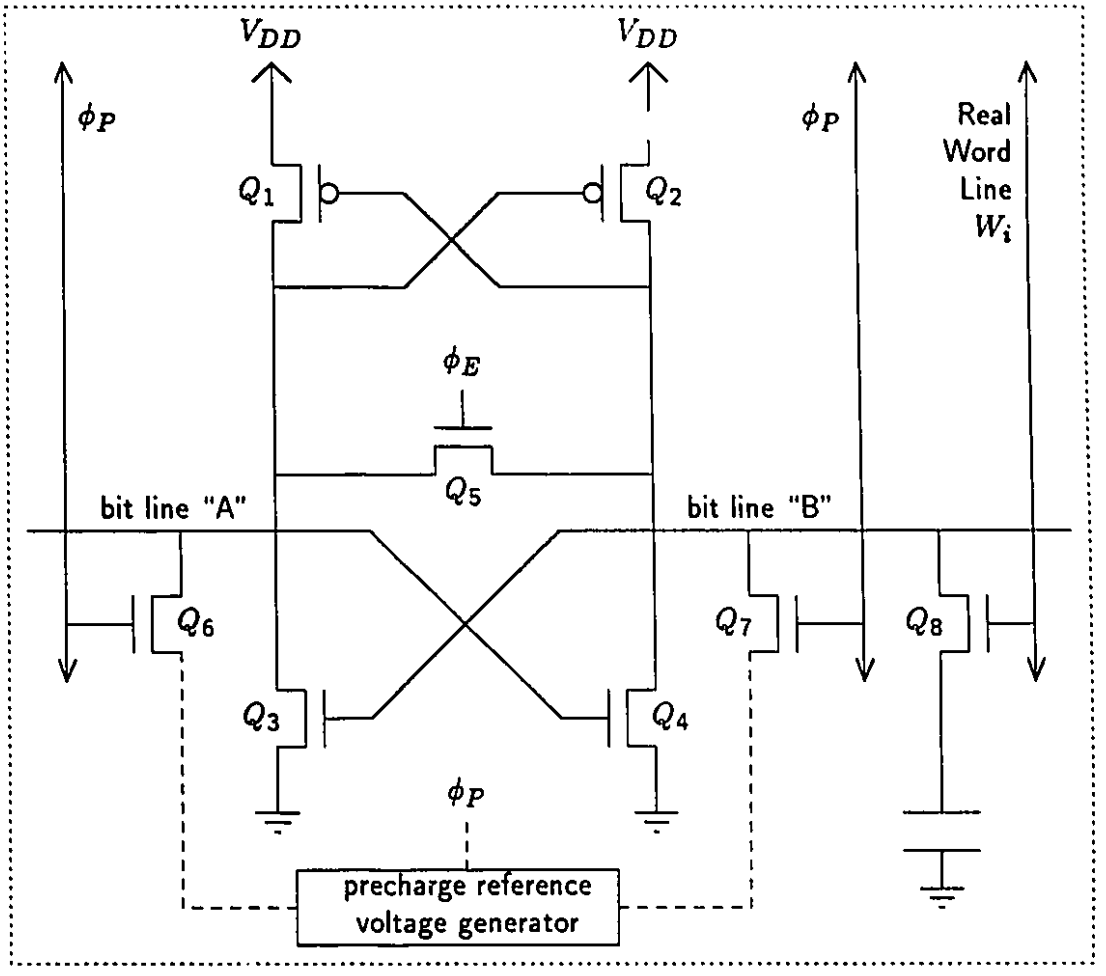

Fig. 2.3 Sense Amplifier with Direct Precharge

amplifiers. Because of how the sense amplifiers operate, one of the two half cell-arrays complements *all* incoming data bit, storing a "1" as a low voltage and a "0" as a high voltage. The output circuitry for that particular half cell-array performs another complementation to restore the data bit's original value.

As can be seen from figures 2.2 and 2.3, a sense amplifier is basically a differential amplifier that has been augmented with clock signals and a precharge reference voltage (typically $V_{DD}/2$). The precharge voltage can be applied through the access transistor of each dummy word line, as shown in figure 2.2, or it can be applied directly to the bit lines, as shown in figure 2.3. In the first case, the

capacitances of the storage cells connected to the dummy word line are identical to the capacitances of all the other storage cells; this results in (1.) the dummy cells storing a charge halfway between V_{DD} and ground, and (2.) the real storage cell's capacitance being exactly counterbalanced by the dummy cell's capacitance on the other bit line half. In the second case, the dummy storage cells (not shown in figure 2.3) usually have half the capacitance (and hence almost half the size) of real storage cells, but this means that the dummy cells must be pre-charged to V_{DD} instead of $V_{DD}/2$ in order to store the appropriate amount of charge; therefore, such dummy cells are precharged separately from the bit lines. All sense amplifiers contain an equalization transistor, gated by the clock ϕ_E, in order to even out any unintended charge differences between the bit line halves after the precharging phase, and before the actual sensing operation takes place. The sense amplifiers discussed above are merely meant to be representative; in fact, there are many distinctly different sense amplifier designs that have been published — each DRAM manufacturer uses a different design.

2.3 Structure of Static RAMs

In Static RAMs, as shown in figure 2.4, no periodic clock signals are needed to retain the stored data. There are two main types of memory cells used in CMOS SRAMs. The 4-transistor (4-T) cell has four *n*-channel transistors and two polysilicon pull-up resistors. The resistors are created in a high-resistivity second polysilicon layer, which allows the resistors to be *stacked* on top of the rest of the cell. The resistors must be properly insulated from the transistors below by an appropriately thick oxide layer. In the 6-transistor (6-T) cell, the two resistors are replaced by two *p*-channel transistors. Because transistors are more resistant to α-particle radiation and large temperature swings than resistors, the 6-T cell suffers from fewer soft errors than the 4-T cell. In addition, the 6-T cell consumes less power because its loads need very low standby current. As a result of these advantages, the 6-T cell CMOS SRAMs are preferred for military and other critical

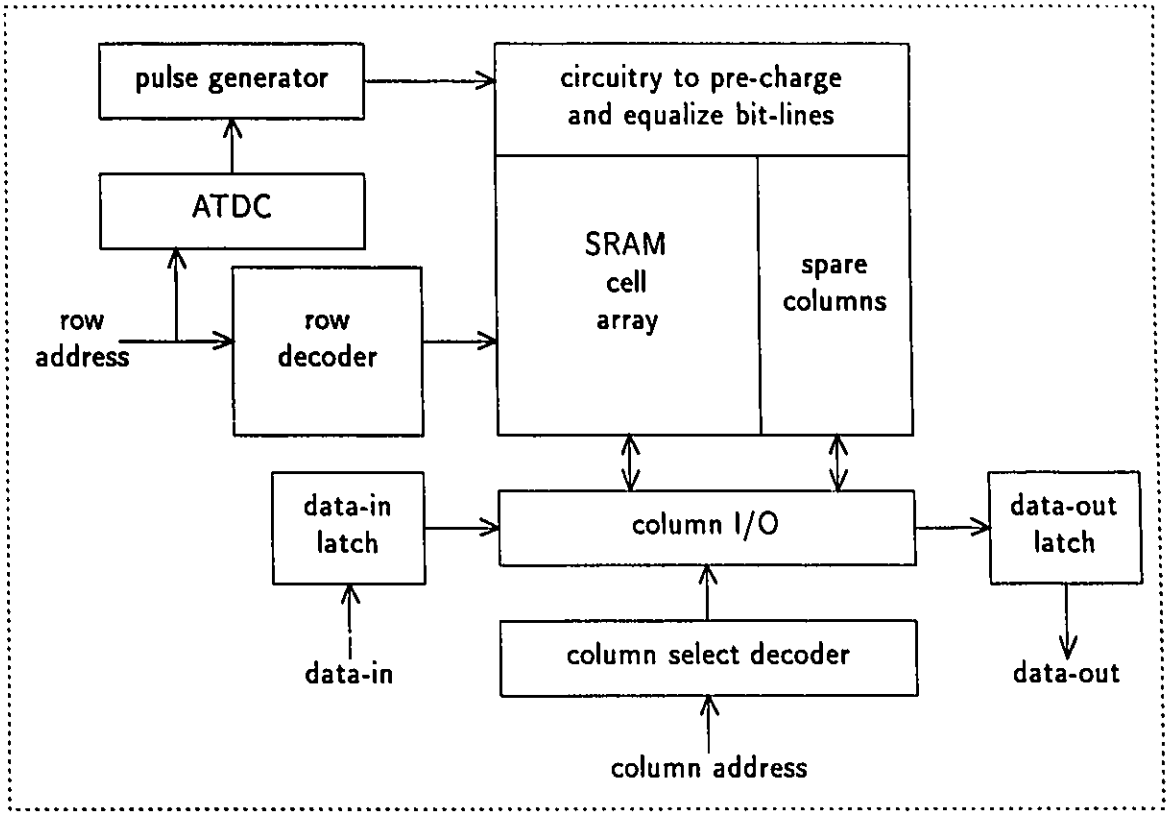

Fig. 2.4 A Typical Static RAM

applications. SRAMs using the 4-T cells have smaller die sizes, and hence cost less to manufacture, which explains their popularity in commercial applications where a low price is a major concern.

Figure 2.5 shows a typical 6-T cell in CMOS. The cell uses a pair of cross-coupled inverters, Q_1 through Q_4, as the storage latch, and two access transistors, Q_5 and Q_6. The 4-T cell of figure 2.6 uses two polysilicon resistors to replace the transistors Q_3 and Q_4 of the 6-T cell, which results in a smaller cell area. The selection of these resistances is critical. If the resistance is too low then the static (or standby) power consumption will be too high. If the resistance is too high then the charge representing the value "1" that is stored on the gate of either Q_1 or Q_2 may be indistinguishable (given the effects of noise) from the charge representing the value "0". Complementary bit lines D and $\overline{D}$ are used because it

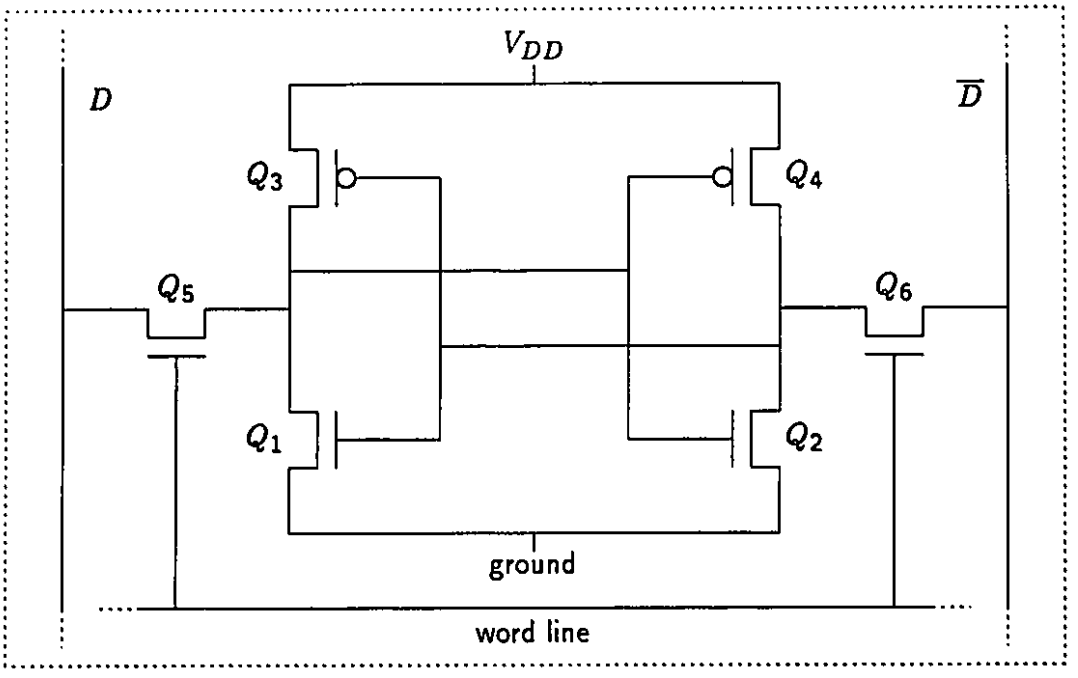

Fig. 2.5 Six Transistor (6-T) SRAM Cell

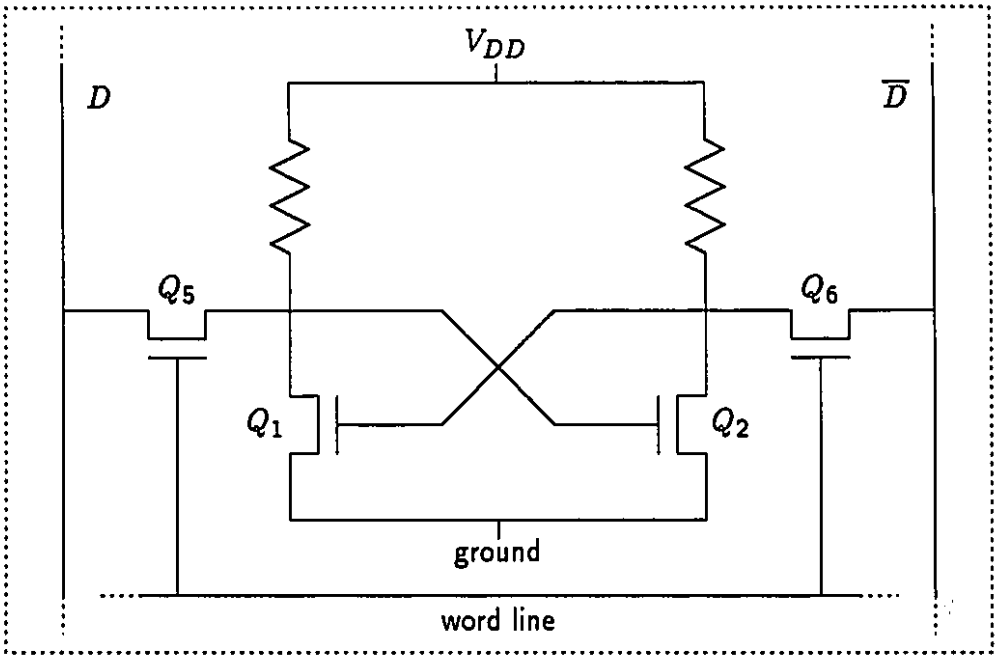

Fig. 2.6 Four Transistor (4-T) SRAM Cell

is difficult to achieve reliable operation with a single bit line at high speeds, when the wide variations in operating temperature and device parameters are taken into account. The word line is kept tied to ground until the cells connected to it are to be accessed for reading or writing, which is when the word line's voltage is raised to V_{DD}.

Writing is performed by assigning the value to be written and its complement to the D and $\overline{D}$ lines, respectively. To illustrate the write operation more clearly, assume that we are given a cell which currently stores a logical "0", and that we wish to now store a logical "1" in the same cell. The first step is to raise the D line to V_{DD} and to lower the $\overline{D}$ line to ground. Since the cell is storing a "0", this means that Q_1 and Q_4 are on, and that Q_2 and Q_3 are off. When the access transistors Q_5 and Q_6 are turned on, the two node voltages start to change. The transistor parameters are chosen in such a way that the resistances of Q_5 and Q_6 are much lower than the resistances of Q_1 through Q_4. Note that there is now a path from V_{DD} on line D, through Q_5, through Q_1, to ground. Because Q_1 has most of the resistance in this voltage divider, the voltage of the node between Q_5 and Q_1 has gone from 0 volts to greater than half of V_{DD}. This node is connected to the gates of Q_2 and Q_4, and its new voltage will tend to turn off Q_4 and turn on Q_2. Once Q_2 has started to conduct, the voltage of the node between Q_4 and Q_2 will start to drop from greater than half of V_{DD} down to ground, and this will tend to turn off Q_1 and turn on Q_3. Once Q_2 and Q_3 are completely turned on, and Q_1 and Q_4 are completely turned off, the write operation is successful, and the word line may then be returned to 0 volts.

Reading is performed by first *precharging* and *equalizing* the dual bit lines to half of V_{DD}, and then, just before the access transistors are turned on, allowing the bit lines to float. To illustrate the read operation more clearly, let us assume once more that the given cell currently stores a logical "0". When the word line is activated, the conducting path from bit line D through Q_5, through Q_1, to ground, pulls down the voltage of line D to 0 volts. Similarly, the conducting path

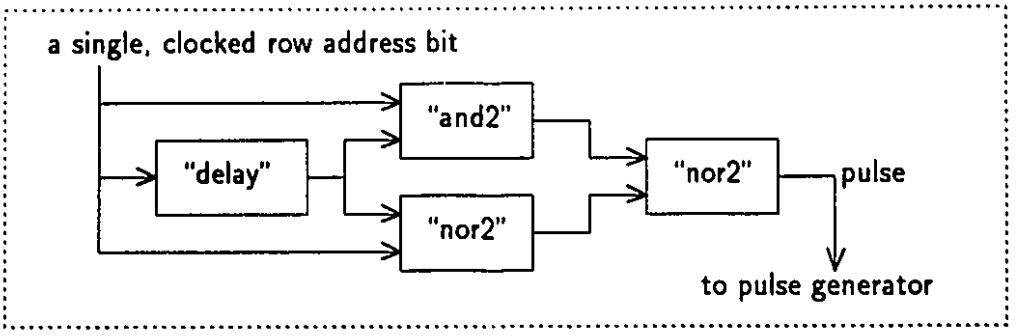

Fig. 2.7 One kind of Address Transition Detection Circuit

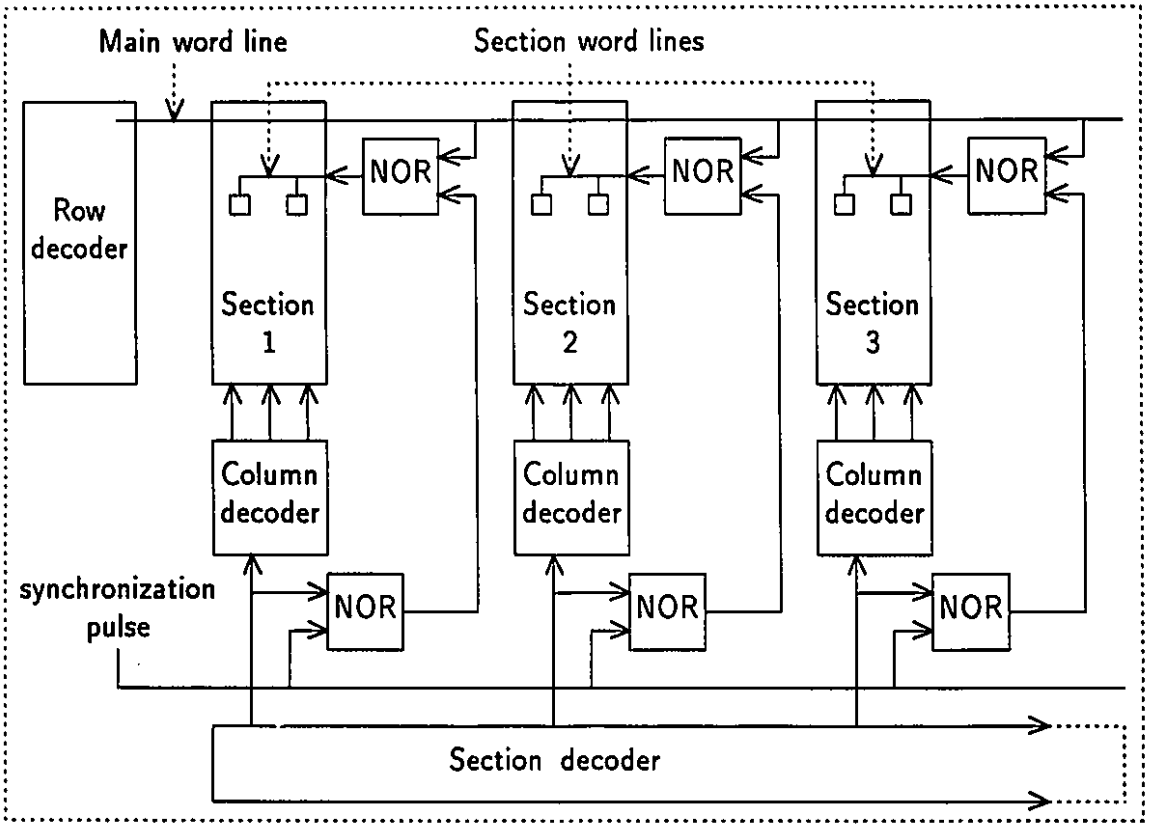

Fig. 2.8 Dynamic Multiple Word Line (DMWL) Scheme

from $\overline{D}$ through Q_6, through Q_4, to the supply voltage, pulls up the voltage of line $\overline{D}$ toward V_{DD}. In the fastest SRAMs, sense amplifiers are used to detect the initial voltage difference in order to complete the read operation more quickly.

SRAMs contain an *address transition detection circuit (ATDC)*, which starts the precharging and equalization of the bit lines during the time it takes a new word line to become activated. This means that the row address lines are fed to both the row decoder and the ATDC, so that any change in row address is immediately followed by precharging and equalization. Since column access time is much faster than row access time, an ATDC is not connected to the column address lines. Figure 2.7 shows one bit slice of one type of ATDC.

An additional way to reduce the access time is to use a *dynamic multiple word line (DMWL)* scheme. As figure 2.8 illustrates, there are two levels of word lines. The *main* word line is not directly connected to any of the memory cells, which causes the line's capacitance to be quite small, resulting in a reduction of word-line delay. Only one *section* word line is active at a time, and thus only a few cells are accessed at a time, which also shortens delay and reduces the power dissipation. The ATDC generates a synchronization pulse that enables NOR-gates which allow the selection of exactly one section word line.

2.4 Redundancy for Static and Dynamic RAMs.

Because densely packed memory chips are so highly vulnerable to defects, the yield of RAM chips is crucial to the commercial success of their manufacture. Due to their regular structure, RAMs can be efficiently made fault tolerant through the addition of redundant (spare) bit lines and word lines. Once a fault in a RAM has been located, the RAM can usually be repaired by disconnecting some faulty bit line or word line (when they constitute replaceable units) and by connecting some spare bit line or word line.

The most widely used technique of memory repair is *permanent switching*: Here the repair is accomplished by using a laser beam or a large electric current to blow some permanent fuses which disconnect the faulty lines and which connect the spare lines. This kind of repair can be performed only by the manufacturer,

and the extra hardware needed for fault repair (i.e., fuses and special decoders) is separate from any extra hardware that may be added for self-testing purposes.

Another means of dealing with defects is to use the fault tolerance technique of *coding*: Here neither fault location nor repair are done explicitly, instead the faults are simply masked by redundancies in the stored-information.

A third technique, which has been largely ignored by commercial chip manufacturers, is called *soft switching*: Here the repair is accomplished by programming EEPROM cells (or some other non-volatile memory cells) that control multiplexers which can disconnect faulty lines and connect spare lines. The EEPROM cells are paired with flip-flops, as in commercially available *Shadow SRAMs*. When the chip is powered-up, the contents of the EEPROM cells are written into the flip-flops, thereby providing default configuration parameters. Whenever a new fault is discovered, the new configuration parameters are initially stored in the flip-flops, before the slow reprogramming (i.e. 5 to 10 milliseconds) of one or more EEPROM cells, in order to change the default setting.

2.5 *The meaning of "embedded".*

In order to enhance the speed performance of VLSI chips by reducing communication flow through the I/O pins, it is becoming increasingly common that logic circuits and memories are being fabricated on the same chip. When the data, address and control signals of RAMs cannot be directly controlled or observed through the I/O pins, such on-chip RAMs are called "embedded RAMs." These embedded RAMs cannot be properly tested by applying test patterns directly to the I/O pins, therefore self-testing methods must be used. Researchers have proposed self-testing embedded RAM designs using random test patterns and deterministic test patterns.

3 Fault Modelling

3.1 General Overview of Fault Models

The types of fault models that previous researchers (surveyed in [Abadir and Reghbati 83], [van de Goor and Verruijt 90], and chapters 2, 3, 10, and 11 of [van de Goor 91]) have considered when proposing fault detection and/or location algorithms, and that might be repairable given appropriate forms of redundancy, are consolidated and described in this chapter. Three broad classes of faults can be distinguished:

* *Functional faults* originate in permanent physical failures in the internal device structure of memories which will cause incorrect functional behavior. The detection and location of functional faults is accomplished by applying carefully chosen data patterns to memory cells in specific addressing sequences.

* *Dynamic faults* refer to failures that occur when the memory is operated at normal operational speed, but no failures occur when operated at significantly slower speeds. Such frequency dependent malfunctions originate mostly from *timing* inconsistencies internal to the memory circuits.

* *Parametric faults* are usually associated with voltage and current values that fall outside the expected margins. These faults refer to situations like:

output voltage being too high or too low, excessive power consumption, insufficient fanout capabilities, etc. The detection of parametric faults generally involves techniques and instruments that are highly dependent on the fabrication technology used to make the IC chip, therefore no self-test scheme for such faults is practical, and for this reason no further attention will be given to parametric faults.

Some *dynamic* tests can be combined with *functional* tests in the same self-test algorithm because the functional test patterns are applied at normal operational speed, and hence these test patterns will detect frequency dependant faults as well. The sections below enumerate a wide variety of *functional* faults, including as well those particular *dynamic* faults which can be detected without the use of external test equipment.

3.2 Functional Faults affecting Memory Cells

3.2.1 SAF: stuck-at fault

The logical value of a "stuck-at" memory cell is always 0 or 1, and can never be changed to the opposite value. A test to locate all SAFs must satisfy the condition: *from every cell, both a 0 and a 1 must be read.*

3.2.2 TF: transition fault

A memory cell fails to undergo either a $0 \rightarrow 1$ or a $1 \rightarrow 0$ "transition" when that same cell is being written-to; "transition faults" are not identical to "stuck-at faults", because the value stored in a TF cell may be changed occasionally when a *different* cell is being written-to, and a "pattern sensitive fault" exists between the *different* cell and the TF cell. Hence, a cell with an "up transition fault" is able to undergo a $1 \rightarrow 0$ transition when 0 is being written to it, but cannot undergo a $0 \rightarrow 1$ transition when 1 is being written to it, however there may exist a "pattern

A:	Functional Faults affecting Memory Cells
	1: SAF: stuck-at fault
	2: TF: transition fault
	3: CF: coupling fault
	a: CF/Inv: inversion coupling fault, $\langle\uparrow;\updownarrow\rangle$ or $\langle\downarrow;\updownarrow\rangle$
	b: CF/Id: idempotent coupling fault, $\langle\uparrow;0\rangle$ or $\langle\uparrow;1\rangle$ or $\langle\downarrow;0\rangle$ or $\langle\downarrow;1\rangle$
	c: CF/Dyn: dynamic coupling fault in SRAMs, $\langle r0,w0;0\rangle$ or $\langle r0,w0;1\rangle$ or $\langle r1,w1;0\rangle$ or $\langle r1,w1;1\rangle$
	d: CF/St: state coupling fault, $\langle 0;0\rangle$ or $\langle 0;1\rangle$ or $\langle 1;0\rangle$ or $\langle 1;1\rangle$
	e: BF: bridging fault
	4: NPSF: neighborhood pattern sensitive fault
	a: ANPSF: active neighborhood pattern sensitive fault
	b: PNPSF: passive neighborhood pattern sensitive fault
	c: SNPSF: static neighborhood pattern sensitive fault
B:	Functional Faults affecting Addressing
	1: AF/UuA: addressing fault — unused address
	2: AF/UrC: addressing fault — unreachable cell
	3: AF/MuA: addressing fault — multi-used address
	4: AF/MrC: addressing fault — multi-reachable cell
	5: SOAF: stuck-open addressing fault
C:	Faults affecting the Read/Write Circuitry:
	1: Stuck-open Access transistor
	2: Stuck-open Precharge transistor
	3: Stuck-open Equalization transistor
	4: Stuck-open Sensing transistor
D:	Line Faults
	1: SAF: stuck-at bit line fault
	2: BF: bridged lines fault
	3: SOF: stuck-open line fault
	4: CF: coupled bit line fault

Table 3.1 Summary of Functional Fault Models

sensitive fault" which can cause the $0 \rightarrow 1$ transition. Similarly, a cell with a "down transition fault" is able to undergo a $0 \rightarrow 1$ transition when 1 is being

E: Dynamic faults
1: RecF: recovery fault
a: RecF/SA: sense amplifier recovery fault
b: RecF/W: write recovery fault
2: RetF: retention fault
a: RetF/SS: sleeping sickness
b: RetF/RL: refresh line stuck-at fault
c: RetF/SDL: static data loss fault
3: IF: imbalance fault
4: SYNCF: synchronization fault

Table 3.2 Summary of Dynamic Fault Models

written to it, but cannot undergo a $1 \rightarrow 0$ transition when 0 is being written to it, however some kind of "pattern sensitive fault" may be able to cause the $1 \rightarrow 0$ transition. A test to locate all TFs must satisfy the condition: *every cell must undergo a* $1 \rightarrow 0$ *transition and a* $0 \rightarrow 1$ *transition, and must be read immediately after each transition before undergoing any additional transitions.*

3.2.3 CF: coupling fault

A pair of memory cells (x, y) are involved in a "coupling fault" when a write operation that causes a transition in one cell x (known as the *coupling cell*), also causes an unintended transition in another cell y (known as the *coupled cell*). However, the reverse action: a write operation that causes a transition in cell y, does not necessarily cause an unintended transition in cell x (if it does cause such a transition in cell x, then this action constitutes a second coupling fault which is distinct from the first). In the case of Dynamic RAMs, some of the coupling fault models are based on certain assumptions:

a) A read operation will not cause an error, which is reasonable since read signals are usually very weak and are therefore unlikely to influence the contents of other cells.

b) A non-transition write will not cause a fault, which is necessary to allow for read operations to be fault free; in DRAMs, a read operation is destructive, and must therefore be followed by a non-transition write operation to restore the original contents of the disturbed cells.

Coupling faults are classified as:

1) inversion coupling faults (CF/Inv)
2) idempotent coupling faults (CF/Id)
3) dynamic coupling faults in SRAMs (CF/Dyn)
4) state coupling faults (CF/St)
5) bridging faults (BF)

In what follows, we use the notation $\uparrow$ to signify a transition write to 1 in a cell which originally contained a 0. Analogously, the notation $\downarrow$ signifies a transition write to 0 in a cell which originally contained a 1. The notation $\updownarrow$ signifies an inversion of the original value stored in a cell. The notation $w0$ signifies a write to 0 in a cell whose original content is unspecified (i.e. the operation can be either a transition write or a non-transition write). Similarly, the notation $w1$ signifies a write to 1 in a cell whose original value is unspecified.

3.2.3.1 CF/Inv: inversion coupling fault, $\langle\uparrow;\updownarrow\rangle$ or $\langle\downarrow;\updownarrow\rangle$

An inversion coupling fault implies that an $\uparrow$ or $\downarrow$ transition write in one cell causes the contents of a second cell to always *invert*. A test to locate all CF/Inv *coupled* cells must satisfy the condition: *each coupled cell must be read after an odd number of inversions in the coupled cell that were caused by transition writes in the coupling cells.*

3.2.3.2 CF/Id: idempotent coupling fault, $\langle\uparrow;0\rangle$ or $\langle\uparrow;1\rangle$ or $\langle\downarrow;0\rangle$ or $\langle\downarrow;1\rangle$

An idempotent coupling fault implies that an $\uparrow$ or $\downarrow$ transition write in one cell causes the contents of a second cell to always take the same value, 0 or 1.

A test to locate all CF/Id *coupled* cells must satisfy the condition: *each coupled cell must be read after the coupled cell has been forced to the same value an unspecified number of times (but before the coupled cell could be forced to the opposite value), by transition writes in the coupling cells.*

3.2.3.3 *CF/Dyn: dynamic coupling fault in SRAMs,* $\langle r0|w0;0\rangle$ *or* $\langle r0|w0;1\rangle$ *or* $\langle r1|w1;0\rangle$ *or* $\langle r1|w1;1\rangle$

A dynamic coupling fault can be viewed as a generalization of the idempotent coupling fault. A dynamic coupling fault implies that a read operation of value x, or a non-transition write of x, or a transition write to x, may each cause the contents of a second cell to always take the same value, 0 or 1. A test to locate all CF/Dyn *coupled* cells must satisfy the condition: *each coupled cell must be read after the coupled cell has been forced to the same value an unspecified number of times (but before the coupled cell could be forced to the opposite value), by any types of read and write operations in the coupling cells.*

3.2.3.4 *CF/St: state coupling fault,* $\langle 0;0\rangle$ *or* $\langle 0;1\rangle$ *or* $\langle 1;0\rangle$ *or* $\langle 1;1\rangle$

A state coupling fault implies that if the *coupling* cell x contains a certain value, then the *coupled* cell y is forced to a particular value. At first glance, it would appear that state coupling faults are identical to dynamic coupling faults, but there is a clear distinction between them. State coupling faults can only occur between cells in the same word line, since all read and write operations take place concurrently among such cells. Dynamic coupling faults can only occur between cells in different word lines, since all read and write operations take place at different times. A test to locate all *state coupled* cells must satisfy the condition: *each pair of state coupled cells* (x, y) *must be written to, in such a way that all four logical states are supposed to occur, namely* $(0,0)$, $(0,1)$, $(1,0)$ *and* $(1,1)$.

3.2.3.5 *BF: bridging fault*

A set of memory cells $(x, y, z, \ldots)$ are involved in a "bridging fault" when the

cells have a single combined value which is either the *logical AND* or the *logical OR* of the values x, y, z, etc. A test to locate all *bridged* cells must satisfy the condition: *each possible pairing of bridged cells* (x, y) *must be written to, in such a way that all four logical states are supposed to occur, namely* $(0,0)$, $(0,1)$, $(1,0)$ *and* $(1,1)$. BFs are usually caused by short circuits between cells or lines.

3.2.4 NPSF: neighborhood pattern sensitive fault

The "neighborhood pattern sensitive fault" class is defined as follows: the value, or the ability to change the value, of a *base cell*, is influenced by the values of, or changes to the values of, a specific group of cells (known as the *neighborhood*). The set of values of the specific group of cells is called the *neighborhood pattern*. The two most common neighborhoods are known, somewhat arbitrarily, as Type-1 and Type-2:

Type-1: this neighborhood consists of the four "closest" cells to the base cell, namely the cells just above, just to the right, just below, and just to the left, of the base cell. Base cells along an edge of the cell array actually have only three physically adjacent neighbors, but in the interests of symmetry, such a base cell is *deemed* to have a fourth "adjacent" neighbor, which is the corresponding cell along the *opposite* edge of the array. Base cells at the corners of the cell array have only two physically adjacent neighbors, and they are *deemed* to have two more "adjacent" neighbors, which are in fact the horizontally opposite and vertically opposite corner cells (the remaining diagonally opposite corner cell is not in the neighborhood).

Type-2: this neighborhood consists of the eight "closest" cells to the base cell, namely the cells of the Type-1 neighborhood plus the four diagonally closest cells to the base cell. Hence, the eight cells of the neighborhood plus the base cell form a 3×3 square with the base cell as the center. Analogous to Type-1 neighborhoods, the base cells along an edge of the cell array have *deemed* "adjacent" neighbors along the opposite edges of the array.

The neighborhood pattern sensitive fault class is only used with reference to *dynamic* RAMs, and never with reference to *static* RAMs, because NPSFs model the effects of undesirable charge leakages between the capacitances in single-transistor dynamic RAM cells. NPSFs do not model any faulty behavior known to exist in static RAMs.

3.2.4.1 *ANPSF: active neighborhood pattern sensitive fault*

An active neighborhood pattern sensitive fault exists when: the base cell *changes* its value, in response to *changes* in the neighborhood pattern. A test to locate ANPSFs must satisfy this condition: *every base cell must be read in state 0 and state 1, for all possible changes in the neighborhood pattern.*

3.2.4.2 *PNPSF: passive neighborhood pattern sensitive fault*

A passive neighborhood pattern sensitive fault exists when: the base cell is forced to *retain* a specific value, in response to a *certain* neighborhood pattern. A test to locate PNPSFs must satisfy this condition: *every base cell must be written and read in state 0, and written and read in state 1, for all permutations of the neighborhood pattern.*

3.2.4.3 *SNPSF: static neighborhood pattern sensitive fault*

A static neighborhood pattern sensitive fault exists when: the base cell *changes* to a specific value, in response to a *certain* neighborhood pattern. A test to locate SNPSFs must satisfy this condition: *every base cell must be read in state 0 and state 1, for all permutations of the neighborhood pattern.*

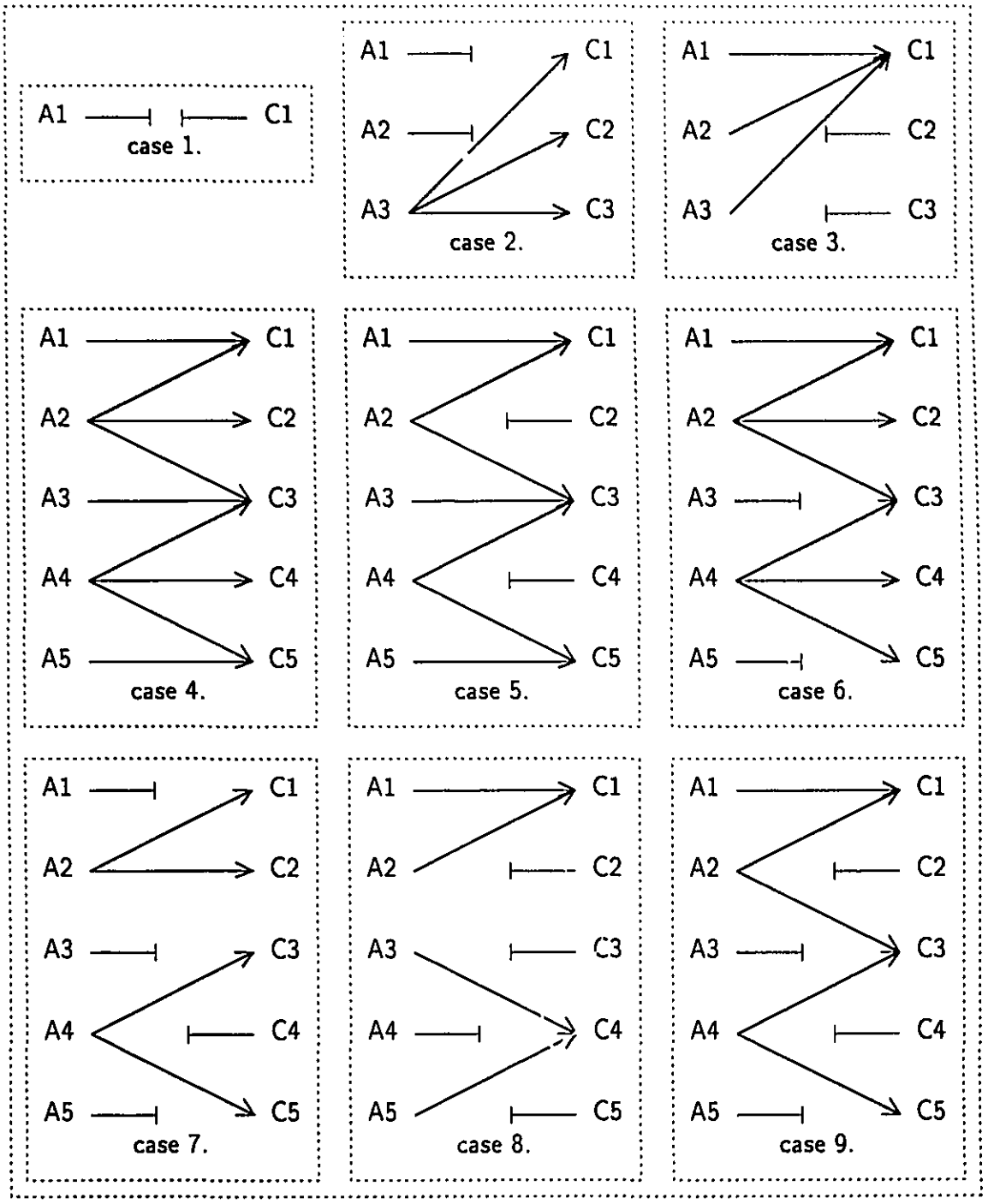

Fig. 3.1 Examples showing the Nine Combinations of Addressing Faults

3.3 Functional Faults affecting Addressing

These faults refer to physical failures in the row and column address decoders and in the address latch. Without loss of generality, only memories with one-bit words ($B = 1$) will be discussed in this section. There are four types of non-sequential "addressing faults":

3.3.1 AF/UuA: addressing fault — unused address

The "unused address" fault occurs when a given address is unable to access any cell at all. This looks like a shortage of one cell with respect to the number of addresses. When a read operation is performed at an "unused address", the response is technology dependant; in some cases, the response will consistently appear to be a logical 1 or a logical 0, but in other cases, the response may be completely unpredictable and may depend on neighboring leakage currents and other noise in the memory.

3.3.2 AF/UrC: addressing fault — unreachable cell

The "unreachable cell" fault occurs when there is no address which can reach a given cell. This given cell is never accessed. This looks like a surplus of one cell with respect to the number of addresses.

3.3.3 AF/MuA: addressing fault — multiused address

The "multiused address" fault occurs when a given address accesses two or more cells simultaneously. This looks like a surplus of cells with respect to the number of addresses. When a read operation is performed at a "multiused address", the response is technology dependant; in some cases, the response will consistently appear to be the logical AND or the logical OR of the cells actually accessed, but in other cases, the response may be completely unpredictable function of the accessed cells.

3.3.4 AF/MrC: addressing fault — multireachable cell

The "multireachable cell" fault occurs when two or more different addresses can access the same cell. This looks like a shortage of cells with respect to the number of addresses. Since write operations are performed on a "multireachable cell" using one or more illegal addresses, in addition to the single legal address, it is required that any test to locate such faults must read from the cell before any writing using the legal address can *mask out* the effect of writing from an illegal address.

3.3.5 Combinations of addressing faults

Because there are exactly as many addresses as there are cells, none of these faults can exist in isolation. The only possible combinations must balance the apparent shortage of cells caused by faults UuA and MrC, with the apparent surplus of cells caused by faults UrC and MuA. The following nine cases list all possible combinations, and examples illustrating these cases are presented in figure 3.1.

1: UuA, with UrC.

2: UuA, with MuA.

3: MrC, with UrC.

4: MrC, with MuA.

5: MrC, with a combination of UrC and MuA.

6: a combination of UuA and MrC, with MuA.

7: UuA, with a combination of UrC and MuA.

8: a combination of UuA and MrC, with UrC.

9: a combination of UuA and MrC, with a combination of UrC and MuA.

Those combinations which contain one or more MrCs (namely: 3, 4, 5, 6, 8 and 9) are susceptible to *fault masking* when a legally addressed write operation on a "multireachable cell" overwrites the faulty contents that resulted from an earlier, illegally addressed write operation on the same cell. Cases 6, 8 and 9 are further complicated by the possibility that a given "multireachable cell" may have only illegal addresses, when its sole legal address is subject to an "unused address" fault.

3.3.6 SOAF: stuck-open addressing fault

The above discussion of different addressing fault types is incomplete because it assumes that the address decoder circuits cannot be transformed into sequential logic by some physical defect. However, a "stuck-open fault" in one of the transistors can change the decoder into a sequential circuit (i.e., a circuit with some kind of memory capability). This type of fault is common in address decoders built out of CMOS technology. CMOS "fully complementary logic" gates are a combination of a pull-up network made from pMOS transistors, and a pull-down network made from nMOS transistors. In a CMOS complementary logic gate, exactly one of the two networks is always conducting, while the other is not conducting.

A single transistor is said to be *stuck-open* when it cannot conduct because of an open gate, open source or open drain connection. An entire CMOS logic gate is said to be *stuck-open* when either its pull-up or its pull-down network is prevented from conducting due to one or more stuck-open transistors in that network, while the remaining network is in a legally non-conductive state. As a consequence of neither network being able to conduct, the output line is not driven to 0 or 1, but takes on a *high impedance state*. The output level will therefore be determined by the charge stored by the capacitance of the output line, which implies that when the input signals change and the logic gate becomes "stuck-open", then the

output retains its *previous* logic value, regardless of which output value the new input signals are supposed to elicit.

Testing a CMOS complementary logic gate (also known as a "static" gate) for stuck-open faults requires that pairs of test patterns be applied in a particular order. The first of the pair of test patterns is the *initializing pattern* and the second one is the *sensitizing pattern*. The sensitizing pattern is chosen so as to activate a potential stuck-open fault. The initializing pattern is chosen so as to produce an output value which is the inverse of the (non-faulty) expected value of the sensitizing pattern. Hence, a stuck-open fault is detected when the wrong output value is produced in response to the sensitizing pattern. Let us designate the "initializing pattern" by IP and the "sensitizing pattern" by SP. To detect a stuck-open fault in the pull-up network, IP should set the output signal to 0, and SP should be selected to establish one or more conducting paths in the pull-up network, of which each chosen path contains a stuck-open fault. To detect a stuck-open fault in the pull-down network, IP should set the output signal to 1, and SP should be selected to establish one or more conducting paths in the pull-down network, of which each chosen path contains a stuck-open fault.

In a few special cases, the testing of "static" CMOS logic gates for stuck-open faults does not require a pair of ordered patterns, IP and SP. More specifically, if the pull-up network consists of exactly one path of series-connected transistors, then a stuck-open fault among those transistors is functionally indistinguishable from a stuck-at-0 fault on the output line. Analogously, if the pull-down network consists of exactly one path of series-connected transistors, then a stuck-open fault among those transistors is functionally indistinguishable from a stuck-at-1 fault on the output line. In more concrete terms, these results imply the following behavioral

(1) a stuck-open pMOS transistor in an *inverter* behaves like a stuck-at-0 output line fault,

(2) a stuck-open nMOS transistor in an *inverter* behaves like a stuck-at-1 output line fault,

(3) a stuck-open pMOS transistor in a *static NOR gate* behaves like a stuck-at-0 output line fault,

(4) a stuck-open nMOS transistor in a *static NAND gate* behaves like a stuck-at-1 output line fault.

The four most common implementations of address decoders are constructed with the components listed below:

(A) inverters and static NAND gates,

(B) inverters and static NOR gates,

(C) inverters, static NAND gates and static NOR gates,

(D) inverters and dynamic NOR gates.

The fastest address decoders (D in the list above) avoid the use of static logic gates, because of the inherent slowness of series-connected transistors (i.e. the pull-down network in static NAND gates, and the pull-up network in static NOR gates). The dynamic NOR gates used in decoders are based on the well-known "CMOS Domino logic" circuit design methodology, which consists of the pull-down network from the NOR function, sandwiched between a pMOS "precharge" transistor and an nMOS "evaluate" transistor, with the output being fed into two cascaded inverters that can drive a word line or control a bit line's transmission gate. Both the "precharge" and "evaluate" transistors are gated by the same clock signal. Since this "dynamic" gate is automatically precharged to 1, there is no need for IP when seeking faults in the pull-down network — only SP is required. If the "precharge" pMOS transistor is stuck-open, then the output will be stuck-at-0, since there is no other way to bring the output line upto V_{DD}. If the "evaluate" nMOS transistor is stuck-open, then the output will be stuck-at-1, since it controls access to the only path to ground.

From the observations above, it is evident that only stuck-open pMOS transistors in NAND gates, and stuck-open nMOS transistors in NOR gates, cannot behave like any kind of stuck-at-0/1 fault, and thus it is these kinds of stuck-open faults which will be considered below.

Suppose that there exists a single stuck-open transistor in the pull-up network of a NAND gate, or in the pull-down network of a NOR gate, which decodes the address $(x_1, x_2, x_3, \ldots, x_M)$, where each x-value represents either a 0 or a 1. Each (pull-up or pull-down) network contains M transistors, where each transistor corresponds to a different x-value. Suppose that the single stuck-open transistor corresponds to the value x_J, where J is an integer between 1 and M. On the basis of these suppositions, it can be shown that this single stuck-open transistor will be detected, in the context of an address decoder circuit, only by the pair of addresses: $IP = (x_1, x_2, \ldots, x_J, \ldots, x_M)$ followed by $SP = (x_1, x_2, \ldots, \overline{x_J}, \ldots, x_M)$.

The address IP causes only the correct word line (or bit line) corresponding to the desired address, to be selected (in a fault-free manner) by the gate containing the stuck-open fault. The address SP causes *both* of the lines corresponding to addresses $(x_1, x_2, \ldots, x_J, \ldots, x_M)$ and $(x_1, x_2, \ldots, \overline{x_J}, \ldots, x_M)$ to be selected at the same time. Hence, within an address decoder, a stuck-open fault can manifest itself only by specifying two addresses in sequence, such that the addresses are a Hamming distance of 1 apart. Given that we have an M-bit long address, then there are 2^M lines controlled by the decoder, and therefore the application of $M \times 2^M$ pairs of addresses are required to detect the presence of all possible stuck-open transistor faults.

3.4 Faults affecting the Read/Write Circuitry:

3.4.1 Stuck-open Access transistor:

In this case there is no charge transfer either to or from the memory cell with respect to the bit line (in figures 2.5 and 2.6, the access transistors are Q_5

and Q_6). The sense amplifier will not sense any voltage differential in such a case, unless there are second-order effects which become sufficiently pronounced to cause one of the sense amplifier's transistors to conduct more current than the remaining transistors. To test for this type of fault, we try to induce second-order effects by storing all ones in one half bit line, and all zeroes in the other half bit line, and vice versa. If the leakage currents are sufficient to cause a noticeable voltage imbalance, then a well-tuned sense amplifier will reinforce the values that have already leaked onto the bit lines. (Note that this fault is *not* the same as the DRAM "imbalance fault — IF", even though it has the same test pattern requirement as IF.)

Example: say that DRAM cell X, in bit line half D, has a stuck-open Access transistor, then it is impossible to store either a 0 or a 1 at cell X. Write all ones to bit line half D, and all zeroes to bit line half $\overline{D}$. Read cell X, and most likely, we get a value of 1, indicating that cell X is stuck-at-1. Now store all zeroes in bit line half D, and all ones in bit line half $\overline{D}$. Read cell X, and most likely, we get a value of 0, indicating that cell X is stuck-at-0. Note that cell X looks like it can be both stuck-at-1 and stuck-at-0. Conventional tests that look for stuck-at-0/1 memory cells will detect and locate this fault, but will report a paradox: the same cell appears to be stuck-at both 0 and 1.

3.4.2 Stuck-open Precharge transistor:

The reference voltage level on the bit line may be too low if only one of the two Precharge transistors is operational, since double the usual capacitance would now have to be charged by a single transistor. This may cause the pMOS Sensing transistors to conduct (because the voltage could be *low* enough to turn them on) even before the word lines have been activated. If both pMOS Sensing transistors conduct the same amount of current, then this fault may not even be noticed since the bit line voltage may have been pulled-up just enough to allow the correct operation of the nMOS Sensing transistors.

3.4.3 Stuck-open Equalization transistor:

The reference voltages may not be equal on both halves of the bit line (because one bit line half may have slightly more capacitance than the other bit line half, or one Precharge transistor may conduct more current than the other Precharge transistor), causing at least one Sensing transistor to conduct before any word line has been activated.

3.4.4 Stuck-open Sensing transistor:

If one of the four Sensing transistors (two pMOS and two nMOS) is open, and therefore cannot conduct, then all of the memory cells on the affected bit line may not have their stored values properly sensed, and these values may not be properly written in the first place. This fault will cause *sequential* behavior in most sense amplifier designs.

3.5 Line Faults

3.5.1 SAF: stuck-at bit line fault

A "stuck-at bit line" fault occurs when either a faulty write driver or a faulty sense amplifier causes a line to be stuck-at 0 or 1. This fault will behave the same way as an entire bit line of stuck-at memory cells.

3.5.2 BF: bridged lines fault

A "bridged bit lines" fault occurs when a pair of adjacent bit lines are shorted together. A "bridged word lines" fault occurs when a pair of adjacent word lines are shorted together, but this type of short is much less common than adjacent bit lines being shorted together. An even less common type of short — the "bridged bit line and word line" fault — is when a bit line and a word line, which run perpendicular to each other, are shorted at the point where one line crosses over the other.

3.5.3 SOF: stuck-open line fault

An "open line" fault occurs when the wire that makes up a bit line or a word line has a break in it. When a true-bit line or a complemented-bit line in a static RAM is open, then the corresponding sense amplifier will sometimes behave in a *sequential* manner, depending on whether the accessed cell is above or below the break in the bit line.

3.5.4 CF: coupled bit line fault

A "coupled bit lines" fault occurs when there is excessive capacitive coupling between a pair of bit lines. These faults may originate from defective write drivers or sense amplifiers, which control the bit lines.

In general, empirical evidence indicates that word lines are affected by fewer types of line faults than bit lines are.

3.6 Dynamic faults

3.6.1 RecF: recovery fault

Recovery faults occur when some circuit in the memory cannot recover fast enough from a previous state, with the result that the current operation is at least partly influenced by the previous operation. Such faults only manifest themselves when the memory is operated at high speed, close to its specified frequency limit. There are two types of recovery faults: sense amplifier recovery faults, and write recovery faults (which originate in the address decoder circuits).

3.6.1.1 RecF/SA: sense amplifier recovery fault

A sense amplifier recovery fault may occur when the sense amplifier has become "saturated" after reading a "long" sequence of bits of the same value x, just before the sense amplifier fails to correctly read a bit of the opposite value $\overline{x}$. Such a fault

may also occur when a "long" monovalued sequence of bits is written into memory, immediately followed by a read operation from a cell containing the opposite value. This latter fault occurs because the data to be written is placed on bit-lines which also serve as the inputs of the sense amplifiers. Therefore, the first requirement for a test to locate sense amplifier recovery faults involves *reading* a long string of 0s (or 1s), followed by reading a single 1 (or 0). The second requirement involves *writing* a long string of 0s (or 1s), followed by reading a single 1 (or 0). In both cases, the long string of monovalued operations should not be interrupted with even a single different operation, as this might prevent the sense amplifiers from failing.

3.6.1.2 RecF/W: write recovery fault

A write recovery fault occurs when a write operation is followed by a read or write operation at a different address, while the address decoder is too slow to react such that the second operation mistakenly uses the first address. Such faults are due to the fact that the signal path through the address decoder logic has different delays for different addresses. A *write-after-write recovery fault* occurs when a write operation to address A_2, which follows a write operation to address A_1, will actually write to address A_1. A *read-after-write recovery fault* occurs when a read operation at address A_2, following a write operation at address A_1, actually reads the contents at address A_1. A thorough test for such faults requires time of the order $O(n^2)$, because all $n \cdot (n-1)$ possible address combinations should be tested. A slightly modified Galpat algorithm could be used to locate these faults.

3.6.2 RetF: retention fault

Retention faults are not caused by read or write operations: they are the result of "spontaneous" data losses in the memory cells. Three main types of retention faults can be distinguished:

1) sleeping sickness in Dynamic RAMs,

2) refresh line stuck-at faults,

3) data losses in Static RAMs.

3.6.2.1 *RetF/SS: sleeping sickness*

Sleeping sickness occurs when the leakage current of a particular DRAM cell is higher than usual, such that when the read part of the refresh operation takes place, the wrong value will be read out and therefore the wrong value will be written back. Maximum leakage will be encountered by a cell when the data values stored by all its neighboring cells are opposite to the value which it is storing. This implies that a checkerboard pattern should be written into the memory array to test for this fault. The memory should not be accessed for at least the time interval between two refresh cycles, in order to allow the maximum leakage to take place.

3.6.2.2 *RetF/RL: refresh line stuck-at fault*

Refresh line stuck-at faults can only occur within DRAMs, not in SRAMs, because only DRAMs have refresh lines. A refresh line failure manifests itself by the inability of individual memory cells to retain their data values for a period much longer than the refresh period. Hence, the way to test for such a fault is to not access the memory, including no attempt at refreshing, for a "long" period of time. This test can be combined with the *sleeping sickness* test, if the time period that the memory is left alone is increased.

3.6.2.3 *RetF/SDL: static data loss fault*

Static data loss faults occur when SRAM cells lose their contents after not being accessed for some lengthy period of time. In principle, the contents of an SRAM cell should be retained indefinitely without the need for even reading the cell occasionally, but when a pull-up device in the cell is defective, there exists the

possibility that leakage currents from this pull-up device will eventually cause the cell's state to change. A test to locate static data loss faults must write a 0 to all cells, wait a "long" time (typically 100 milliseconds), and then read all cells to verify data retention; next, the test must write a 1 to all cells, wait a "long" time, and then read all cells.

3.6.3 IF: imbalance fault

Imbalance faults occur in DRAMs when there exists a bit-line precharge voltage imbalance. In DRAM cells, small leakage currents may exist between the cell capacitors and the bit-lines due to imperfect access transistors. These leakage currents can increase or decrease the bit-line voltage during the precharge phase and during the read operation. The worst case is when all the cells connected to one half bit-line contain the same data value x, and all the cells (except the to-be-read cell) connected to the other half bit-line contain the data value $\overline{x}$, and their combined leakage currents can sufficiently alter the precharge voltage levels on each half bit-line, so that the to-be-read cell's stored charge (of value x) cannot compensate sufficiently for the charge imbalance between the two bit-line halves which will cause the sense amplifier to produce the erroneous value $\overline{x}$. A test algorithm for this type of fault should create the worst case conditions on every bit-line.

3.6.4 SYNCF: synchronization fault

A "synchronization fault" occurs when there are defects in the clocking circuitry that generates various clock signals (such as Precharge and Equalize for the bit lines) and various enabling signals (such as Read and Write signals for the address decoders, sense amplifiers, and write drivers).

4 Brief Survey of Literature

4.1 Published Surveys

There are two, only partially overlapping, recently published surveys on the topic of "built-in self-testing of RAMs," namely: [Franklin and Saluja 90] and chapter 12 (pp. 341–423) of [van de Goor 91]. An exhaustive survey will not be attempted here because it would necessarily duplicate most of the material in the above-mentionned publications. Instead, some short comments are presented below, on the most recent papers, and how they apply to the subject of this thesis.

4.2 Experimental Results with SRAM March Tests

[Dekker *et al.* 88] and [Dekker *et al.* 89] deal with the issue of *electrical* fault modelling, in contrast with *functional* fault modelling. These papers describe some experimental results of a collection of various functional tests that were applied to 16K SRAMs fabricated by Philips. The "traditional" tests (like Zero-One, Sliding Diagonal, GALCOL) had fault coverages ranging from 50% to 70%. The SAF march tests found between 75% and 85% of all faults. The CF march tests had fault coverages from 80% to 100%. The tests for NPSFs found only 55% of all faults. From these results it can be concluded that coupling faults are closest to the actual *electrical* faults occurring within SRAMs.

[Dekker *et al.* 88] and [Dekker *et al.* 89] also deal with defect modeling at the layout level, using a technique called "Inductive Fault Analysis (IFA)". Layout defects manifest themselves in the following forms:

1. broken wires
2. shorts between wires
3. missing contacts
4. extra unwanted contacts
5. extra unwanted transistors

These electrical faults, in the context of an SRAM circuit, result in the following reduced functional faults:

a) SAF in a memory cell

b) TF in a memory cell

c) idempotent coupling fault, CF/Id, between two memory cells in different words

d) state coupling fault, CF/St, between two memory cells in the same word

e) inaccessible memory cell fault (causing one or more balanced pairs of AF/UrC and AF/UuA), improperly named "stuck-open fault", which is intended to mean that because of an *open word line*, the memory cell cannot be accessed,

f) an *open bit line*, which sometimes is a "stuck-open fault", because it might cause sequential behavior in the sense amplifier; to cope with this possibility of sequential sense amps, an extra (seemingly redundant) read operation is appended to every march element.

g) data retention fault, RetF/SDL, where a memory cell fails to retain its logic value after some time period, which is caused by a defective pull-up resistor.

Based on this functional fault model, three new march tests — namely IFA-9 (see Table 4.1), IFA-13 (see Table 4.2), and Static Data Retention Test (see Table 4.3) — are proposed, which have a somewhat better experimental fault coverage than a collection of well-known standard march tests for coupling faults. The most significant theoretical result from this work — which applies more frequently to SRAMs than to DRAMs — is that march tests can be made sensitive to *sequential* sense amplifier behavior by appending extra read operations to some march elements. For instance, the proposed algorithm IFA-9 is transformed into IFA-13 by appending a seemingly redundant read operation to every march element, except for the initialization march element $\Uparrow (w0)$. Many sense amplifier designs incorporate a data latch for storing the read-out data before sending it to the I/O port, such that the write driver for the same bit line pair has *no access* to this data latch (furthermore, a "virtual data latch" is sometimes created within a sense amplifier, that lacks a real data latch, by some types of CMOS stuck-open faults). Because the real or virtual data latch only stores the latest read-out data, and does not store the newest data to be written-in (since the write driver lacks access to the data latch), a stuck-open memory cell or faulty sense amplifier can be detected by employing a read operation *immediately after a transition write, within the same march element.*

$\Uparrow (w0)$
$\Uparrow (r0, w1)$
$\Uparrow (r1, w0)$
$\Downarrow (r0, w1)$
$\Downarrow (r1, w0)$

Table 4.1 IFA-9 Algorithm — combinational sense amps

The authors of [Dekker *et al.* 88] and [Dekker *et al.* 89] neglected to formalize their observation about sequential sense amplifiers into a generally applicable result, since they concentrated exclusively on their experimentally derived fault

$\Uparrow (w0)$
$\Uparrow (r0, w1, r1)$
$\Uparrow (r1, w0, r0)$
$\Downarrow (r0, w1, r1)$
$\Downarrow (r1, w0, r0)$

Table 4.2 IFA-13 Algorithm — sequential sense amps

$\Updownarrow (w0)$
Disable RAM and Wait
$\Updownarrow (r0, w1)$
Disable RAM and Wait
$\Updownarrow (r1)$

Table 4.3 Static Data Retention Algorithm — one-bit version

model which led directly to the IFA algorithms and the Static Data Retention test. Here is the general result implicit in their work:

Proposition: A march test will detect all faulty sequential behavior in the sense amplifiers of RAMs, if the test contains at least one instance of each of the following four march sequences, in any order:

1: $\Uparrow (\ldots, r0, w1, r1, \ldots)$

2: $\Uparrow (\ldots, r1, w0, r0, \ldots)$

3: $\Downarrow (\ldots, r0, w1, r1, \ldots)$

4: $\Downarrow (\ldots, r1, w0, r0, \ldots)$

Corollary: The two shortest march tests which satisfy the conditions of the Proposition are shown in Table 4.4. Note how conditions 1 and 2 have been merged into a single march element with five operations.

$\Uparrow (r0, w1, r1, w0, r0)$ and $\Uparrow (r1, w0, r0, w1, r1)$, and conditions 3 and 4 have been merged, $\Downarrow (r0, w1, r1, w0, r0)$ and $\Downarrow (r1, w0, r0, w1, r1)$. These two tests obviously fail to detect many coupling faults (CFs), so they have lower fault coverage than IFA-13 and the other superior march tests shown in Chapter 5.

$\Uparrow (w0)$	$\Uparrow (w1)$
$\Uparrow (r0, w1, r1, w0, r0)$	$\Uparrow (r1, w0, r0, w1, r1)$
$\Downarrow (r0, w1, r1, w0, r0)$	$\Downarrow (r1, w0, r0, w1, r1)$

Table 4.4 Shortest "Sequential Sense-Amp" March Tests

4.3 More March tests for SRAMs

4.3.1 Direct BIST implementation of march tests

[Nicolaidis 85] implements march tests that are capable of detecting AFs, SAFs, TFs, and CFs, with hardware containing LFSRs and parity detectors. The overall design of the BIST circuit was intended to allow for fault detection only, however similar circuit components are used later in this thesis for fault location.

4.3.2 BIST based on march tests with shifting

[Nadeau-Dostie *et al.* 90] describes a serial interfacing technique for embedded SRAMs. Their scheme is closely related to the proposal by [You and Hayes 85] for self-testing DRAMs, because both schemes serially shift data in a fashion that makes an entire array of cells *appear* to behave like one very long shift register. A complete exposition of how this can be accomplished is presented in the next chapter (see section 5.4.1: "Serial Access to Memory").

4.4 Results for DRAMs only

4.4.1 Parallel testing with March tests

[Inoue *et al.* 87] propose line-mode testing, where an entire word line can be written in parallel, and where a string of EXOR gates compares the expected data with an entire word of read data. The parallel write operation is limited in the number of distinct patterns which can be applied, and the algorithms used are march tests, therefore NPSFs are not adequately covered by this scheme.

4.4.2 Parallel testing with Checkerboard tests

[Ohsawa *et al.* 87] is probably the first BIST implementation in a DRAM by industry (Toshiba). It uses only a checkerboard test. The fault coverage is even lower than that of the scheme in the previous paragraph. This test was probably selected for its speed and minimal area, because (to quote the authors) it is intended for "a *daily* start-up test of a memory in a system."

4.4.3 Microcoded march and checkerboard tests

[Takeshima *et al.* 90] is probably the second BIST implementation in a DRAM by industry (NEC). It uses a microprogrammed ROM to store both a march test and a checkerboard test. The fault coverage here is only slightly better than in the two previous schemes, but the most notable feature of this scheme is its ROM, which could be easily re-programmed to contain more complex march tests than the one reported. The authors even point out that with three additional circuits — a base register, a loop counter, and address switches — the ROM could be programmed to implement a "galloping" test (such as GALPAT, GALCOL or GALROW), although all this extra circuitry and the increased size of the ROM would nearly double the total BIST area overhead.

The use of a ROM to store algorithms as required by the BISD designs proposed in this thesis, is distinctly different from the use of the ROM as reported

in this paper. This is because the BISD ROM is only accessed by the BISD processor (which contains a finite-state machine), and it is the processor which directly controls all the remaining test circuitry, unlike the microprogram ROM which directly controls all test circuitry — in a sense, the microprogram ROM replaces the BISD processor with its internal finite-state machine. For a pure BIST scheme (i.e. no fault location), the microprogram ROM technique is more area efficient, but for BISD with self-repair, the much more elaborate algorithms (for fault location and computation of a repair plan) are more efficiently implemented in hardware using some kind of processor and stored-program-ROM combination, as was concluded in [Ritter and Müller 87].

4.4.4 *BIST for Restricted Active and Restricted Static NPSFs in a Type-2 neighborhood*

[Mazumder and Patel 87] employ a BIST SAMB architecture, which requires a modification of the DRAM's column decoder so that, during test mode, the same data value may be written in parallel to a selected group of cells in one row; in addition, the cells belonging to a particular group within one row can be read in parallel (a pair of deterministic comparators verify whether the read data consists of all 0's or all 1's). The choice of fault model and of neighborhood type result in the cells within a word being divided into two particular groups: the even numbered bits and the odd numbered bits.

4.4.5 *Fast BIST for RA and RSNPSFs in a Type-2 neighborhood*

[Mazumder and Patel 89] present essentially the same method as described in section 4.4.4, with the only difference being that *both* the even and odd groups of cells within a word can be read or written *simultaneously* rather than separately (i.e. all of the bits in a word are read or written at once, rather than only half of them).

4.4.6 BIST for ANPSFs and SNPSFs in a Type-1 neighborhood

[Mazumder and Patel 89] again use a BIST SAMB architecture, which requires the same kind of modified column decoder that is used by the methods described in sections 4.4.4 and 4.4.5, in order to effect multiple-bit read and write operations during test mode. The choice of fault model and neighborhood type result in the cells within a word being divided into five particular groups: (0 mod 5) numbered bits, (1 mod 5) numbered bits, (2 mod 5) numbered bits, (3 mod 5) numbered bits, and (4 mod 5) numbered bits. The algorithm to locate the faults is based on an Eulerian sequence.

4.5 Inapplicable Results

4.5.1 Parallel test with signature analysis

[Sridhar 86] uses an LFSR-based signature analyzer to read bit lines from one or more arrays simultaneously. Signature analysis is not practical for fault location, and even for fault detection there is a non-zero probability of aliasing errors. This scheme's principle advantages are: speed, very low area overhead, and the possibility of combining it with an external tester.

4.5.2 BIST for SNPSFs in a Type-1 neighborhood

[Kinoshita and Saluja 86] and [Saluja *et al.* 87] employ a BIST SASB architecture, which does not require changes to the DRAM layout, and therefore excludes the use of parallelism. As a result, this method can also be implemented using an external tester. The test response evaluation mechanism is based on compaction and thus does not allow for fault *location* but only fault *detection.*

4.5.3 DFT based on tree RAM

[Jarwala and Pradhan 87] and [Jarwala and Pradhan 88] describe an obvious application of the MASB architecture. This technique requires the use of an external tester — hence it is not directly applicable to this thesis.

4.5.4 BIST based on random testing

[McAnney *et al.* 84] and [McAnney *et al.* 85] are based on pseudo-random test techniques, and therefore their results cannot be applied here because faults cannot be *located* deterministically using pseudo-random methods.

4.5.5 Concurrent testing

Concurrent testing amounts to using error correcting codes [Chen and Hsiao 84]. This topic is not directly applicable to the subject of this thesis since it deals with "redundancy within the stored information" rather than "redundancy within the hardware" in the form of spare rows and spare columns intended for repair.

5 High-level Algorithms

5.1 Preview of Test-only algorithms: detection of faults.

All of the most important faults can be detected using the following sequence of five test algorithms:

(a) stuck-at faults, address decoder faults, transition faults, inversion coupling faults, idempotent coupling faults, and linked combinations of TFs and CF/Ids, linked combinations of some CF/Invs and some CF/Ids, and linked combinations of CF/Ids with other CF/Ids, can all be detected by a single march test, commonly called *March-B* (see Table 5.1, notation explained later in section 5.3). The *March-B* algorithm was originally proposed in [Suk and Reddy 81]. In order to augment the March-B algorithm so that it covers *sequential sense amplifier behavior* in static RAMs (i.e. the faults described in sections 3.4 and 3.5.3), we append extra read operations to satisfy the Proposition in section 4.2 to obtain a new algorithm called *March-B+* (see Table 5.2).

The original March-B algorithm requires $17n$ operations, assuming that the memory contains n cells. The new March-B+ algorithm requires $19n$ operations, and it constitutes a very thorough test of *static* RAMs. Note that the second march element, $\Uparrow (r0, w1, r1, w0, r0, w1)$, already satifies conditions 1 and 2 of the Proposition in section 4.2. The new fourth

$\Uparrow (w0)$
$\Uparrow (r0, w1, r1, w0, r0, w1)$
$\Uparrow (r1, w0, w1)$
$\Downarrow (r1, w0, w1, w0)$
$\Downarrow (r0, w1, w0)$

Table 5.1 March-B Algorithm — one-bit version

$\Uparrow (w0)$
$\Uparrow (r0, w1, r1, w0, r0, w1)$
$\Uparrow (r1, w0, w1)$
$\Downarrow (r1, w0, r0, w1, r1, w0)$
$\Downarrow (r0, w1, w0)$

Table 5.2 The *new* March-B+ Algorithm — one-bit version

march element, $\Downarrow (r1, w0, r0, w1, r1, w0)$, satifies conditions 3 and 4 of the Proposition.

(b) active NPSFs, passive NPSFs, static NPSFs, and linked combinations of these NPSFs with stuck-at faults and transition faults (and some address decoder faults), can be detected by a single test. Note that this test is only applicable to dynamic RAMs, since faulty behavior in static RAMs generally cannot be explained using NPSFs. This test is the "combined Active and Passive NPSF location algorithm", which uses an Eulerian sequence to order the write operations, and which has three options for address generation:

(1) "Type-1" neighborhood with the "Two-group" method, or

(2) "Type-1" neighborhood with the "Tiling" method, or

(3) "Type-2" neighborhood with the "Tiling" method.

(c) the remaining linked combinations of all NPSFs with address decoder faults, can be detected by the march test, MATS+ (see Table 5.3).

$\Uparrow (w0)$
$\Uparrow (r0, w1)$
$\Downarrow (r1, w0)$

Table 5.3 MATS+ Algorithm — one-bit version

$\Uparrow (w0)$
$\Uparrow (r0, w1, r1, w0, r0, w1)$
$\Downarrow (r1, w0, r0, w1, r1)$

Table 5.4 The *new* MATS++ Algorithm — sequential sense amps

If sequential sense amplifier behavior must be considered, then augment the MATS+ algorithm with the conditions of the Proposition in section 4.2, to obtain the *new* MATS++ algorithm shown in Table 5.4.

(d) linked combinations of CF/Invs with other CF/Invs, and linked combinations of some CF/Invs and some CF/Ids, can only be detected by a *non-march* test.

(e) stuck-open faults (SOAF) which transform address decoders into sequential circuits can be detected by a *non-march* test.

Further details of these algorithms are provided later in this thesis.

5.2 Preview of Diagnosis algorithms: location of faults

All of the most important faults can be located using a sequence of 8 algorithms, which locate the faults in the following order:

1. SAF in Read/Write circuitry,
2. SAF and dominant-0 BF in Address decoders,
3. dominant-1 BF in Address decoders,

4. dominant-0/1 BF in Read/Write circuitry,
5. SAF in Memory cells, dominant-0/1 BF in Memory cells,
6. CF in Memory cells,
7. SOAF, erroneous access of addresses that are a Hamming distance of 1 away,
8. combined Active and Passive NPSF location algorithm.

The details of the location algorithms will be presented later. For the time being, it suffices to be aware that the location algorithms are essentially extensions of the detection algorithms, and are significantly modified to use the built-in self-diagnosis hardware. Section 5.4 describes in considerable detail the heavy dependence between the location algorithms and the self-diagnosis hardware.

5.3 Notation for March Tests

The algorithms shown in Tables 5.1 and 5.3 are in the form of *march tests*, because they consist of sequences of "march elements". A single *march element* consists of a sequence of operations that are applied to each cell in the memory, before proceeding to the "next" cell, in either one of two addressing orders: an increasing order ($\Uparrow$), usually from address 0 to address $N-1$, or a decreasing order ($\Downarrow$), which must be exactly the reverse of the $\Uparrow$ addressing order.

The algorithm presented in Table 5.1 is appropriate when each address refers to exactly one bit of memory. When a word-oriented static RAM is tested, whole words of data are written to or read from the memory instead of only single bits. If we apply whole words of all 0's and all 1's, then we would fail to detect certain coupling faults between cells in the same word. In order to detect state coupling faults, we must apply the *Primary data backgrounds* to each word [Dekker *et al.* 88]. For example, if we wish to test a static RAM with eight bits per word (i.e. a byte-oriented SRAM), then the *Primary data backgrounds* are shown in Table

5.5. For a word with B bits, the number of Primary data backgrounds equals $2(\lceil \log_2 B \rceil + 1)$. The March-B+ algorithm for a byte-oriented static RAM, using the Primary data backgrounds, is shown in Table 5.6. Note how the original 1-bit algorithm has been replicated 4 times, with each replication using a different pair of complementary primary data backgrounds.

"0" replaced by:
01010101
00110011
00001111
00000000
"1" replaced by:
10101010
11001100
11110000
11111111

Table 5.5 8-bit "Primary" data backgrounds

The algorithm in Table 5.6 still would not detect some coupling faults within the same word (as mentionned in Appendix C of [van de Goor 91]). Idempotent coupling faults require different patterns to be written and read, namely: the *Marching data backgrounds*, which replace the write operations in "state-changing" march elements, as shown in Tables 5.7 and 5.8. A "state-changing" march element has an *odd* number of transition-writes in total; the first, third, fifth, and all subsequent odd-numbered writes in the original 1-bit algorithm are replaced by "Odd-Marching" data backgrounds, while the second, fourth, and all subsequent even-numbered 1-bit writes are replaced by "Even-Marching" data backgrounds. In an analogous manner, *Walking data backgrounds* replace the write operations in "state-retaining" march elements, as shown in Tables 5.9 and 5.10. A "state-retaining" march element has an *even* number of transition-writes in

$\Uparrow (w01010101)$
$\Uparrow (r01010101, w10101010, r10101010, w01010101, r01010101, w10101010)$
$\Uparrow (r10101010, w01010101, w10101010)$
$\Downarrow (r10101010, w01010101, r01010101, w10101010, r10101010, w01010101)$
$\Downarrow (r01010101, w10101010, w01010101)$
$\Uparrow (w00110011)$
$\Uparrow (r00110011, w11001100, r11001100, w00110011, r00110011, w11001100)$
$\Uparrow (r11001100, w00110011, w11001100)$
$\Downarrow (r11001100, w00110011, r00110011, w11001100, r11001100, w00110011)$
$\Downarrow (r00110011, w11001100, w00110011)$
$\Uparrow (w00001111)$
$\Uparrow (r00001111, w11110000, r11110000, w00001111, r00001111, w11110000)$
$\Uparrow (r11110000, w00001111, w11110000)$
$\Downarrow (r11110000, w00001111, r00001111, w11110000, r11110000, w00001111)$
$\Downarrow (r00001111, w11110000, w00001111)$
$\Uparrow (w00000000)$
$\Uparrow (r00000000, w11111111, r11111111, w00000000, r00000000, w11111111)$
$\Uparrow (r11111111, w00000000, w11111111)$
$\Downarrow (r11111111, w00000000, r00000000, w11111111, r11111111, w00000000)$
$\Downarrow (r00000000, w11111111, w00000000)$

Table 5.6 "Primary" 8-bit version of March-B+ Algorithm

total; the first, third, and all subsequent odd-numbered 1-bit writes are replaced by "Odd-Walking" data backgrounds, while the even-numbered 1-bit writes are replaced by "Even-Walking" data backgrounds. For a word with B bits, the number of Odd-Marching, Even-Marching and Odd-Walking data backgrounds equals $4B$ each; the number of Even-Walking data backgrounds equals 2, since they are all constant all-zeroes or all-ones patterns. The March-B+ algorithm for a byte-oriented static RAM that makes use of the Marching and Walking data backgrounds, is shown in Table 5.11. Note that, unlike Table 5.6, there is no repetitive replication of the original 1-bit algorithm, instead the various Marching and Walking patterns are *interleaved* within each march element, resulting in each

march element being lengthened by a factor of B. An algorithm with a suitable combination of Marching data backgrounds (such as in Table 5.11) will — as a byproduct — also detect all state coupling faults, because any two cells x and y will be forced into all four possible states: $(0,0)$, $(0,1)$, $(1,0)$, $(1,1)$.

$\Uparrow (w0)$ replaced by:	$\Downarrow (w0)$ replaced by:	$\Uparrow (w1)$ replaced by:	$\Downarrow (w1)$ replaced by:
01111111	11111110	10000000	00000001
00111111	11111100	11000000	00000011
00011111	11111000	11100000	00000111
00001111	11110000	11110000	00001111
00000111	11100000	11111000	00011111
00000011	11000000	11111100	00111111
00000001	10000000	11111110	01111111
00000000	00000000	11111111	11111111

Table 5.7 8-bit "Odd-Marching" data backgrounds

$\Uparrow (w0)$ replaced by:	$\Downarrow (w0)$ replaced by:	$\Uparrow (w1)$ replaced by:	$\Downarrow (w1)$ replaced by:
00000000	00000000	11111111	11111111
10000000	00000001	01111111	11111110
11000000	00000011	00111111	11111100
11100000	00000111	00011111	11111000
11110000	00001111	00001111	11110000
11111000	00011111	00000111	11100000
11111100	00111111	00000011	11000000
11111110	01111111	00000001	10000000

Table 5.8 8-bit "Even-Marching" data backgrounds

If the Marching and Walking data backgrounds are thought of as $B \times B$ square matrices, then there is a very simple way to represent them all using only 3 bits per background. A careful examination of the patterns reveals that each matrix has

$\Uparrow (w0)$ replaced by:	$\Downarrow (w0)$ replaced by:	$\Uparrow (w1)$ replaced by:	$\Downarrow (w1)$ replaced by:
01111111	11111110	10000000	00000001
10111111	11111101	01000000	00000010
11011111	11111011	00100000	00000100
11101111	11110111	00010000	00001000
11110111	11101111	00001000	00010000
11111011	11011111	00000100	00100000
11111101	10111111	00000010	01000000
11111110	01111111	00000001	10000000

Table 5.9 8-bit "Odd-Walking" data backgrounds

$\Uparrow (w0)$ replaced by:	$\Downarrow (w0)$ replaced by:	$\Uparrow (w1)$ replaced by:	$\Downarrow (w1)$ replaced by:
00000000	00000000	11111111	11111111
⋮	⋮	⋮	⋮
00000000	00000000	11111111	11111111

Table 5.10 8-bit "Even-Walking" data backgrounds

3 constant parts: an Upper triangle, a Diagonal, and a Lower triangle. The only difference between $\Uparrow (wX)$ and $\Downarrow (wX)$ is the direction of the Diagonal, namely $\searrow$ or $\swarrow$. Table 5.12 summarizes the data backgrounds using a compact notation.

	$\Uparrow (w0)$	$\Downarrow (w0)$	$\Uparrow (w1)$	$\Downarrow (w1)$
Odd-Marching	$(\triangledown 1, \searrow 0, \triangle 0)$	$(\triangledown 1, \swarrow 0, \triangle 0)$	$(\triangledown 0, \searrow 1, \triangle 1)$	$(\triangledown 0, \swarrow 1, \triangle 1)$
Even-Marching	$(\triangledown 0, \searrow 0, \triangle 1)$	$(\triangledown 0, \swarrow 0, \triangle 1)$	$(\triangledown 1, \searrow 1, \triangle 0)$	$(\triangledown 1, \swarrow 1, \triangle 0)$
Odd-Walking	$(\triangledown 1, \searrow 0, \triangle 1)$	$(\triangledown 1, \swarrow 0, \triangle 1)$	$(\triangledown 0, \searrow 1, \triangle 0)$	$(\triangledown 0, \swarrow 1, \triangle 0)$
Even-Walking	$(\triangledown 0, \searrow 0, \triangle 0)$	$(\triangledown 0, \swarrow 0, \triangle 0)$	$(\triangledown 1, \searrow 1, \triangle 1)$	$(\triangledown 1, \swarrow 1, \triangle 1)$

Table 5.12 (U, D, L) representation of data backgrounds

We call an address *orthogonal* if it is the zero address, or if its binary

$\Uparrow (w00000000)$
$\Uparrow (r00000000, w10000000, r10000000, w00000000, r00000000, w10000000,$ $r10000000, w11000000, r11000000, w10000000, r10000000, w11000000,$ $\vdots$ $r11111100, w11111110, r11111110, w11111100, r11111100, w11111110,$ $r11111110, w11111111, r11111111, w11111110, r11111110, w11111111)$
$\Uparrow (r11111111, w01111111, w11111111,$ $r11111111, w10111111, w11111111,$ $\vdots$ $r11111111, w11111101, w11111111,$ $r11111111, w11111110, w11111111)$
$\Downarrow (r11111111, w11111110, r11111110, w11111111, r11111111, w11111110,$ $r11111110, w11111100, r11111100, w11111110, r11111110, w11111100,$ $\vdots$ $r11000000, w10000000, r10000000, w11000000, r11000000, w10000000,$ $r10000000, w00000000, r00000000, w10000000, r10000000, w00000000)$
$\Downarrow (r00000000, w00000001, w00000000,$ $r00000000, w00000010, w00000000,$ $\vdots$ $r00000000, w01000000, w00000000,$ $r00000000, w10000000, w00000000)$

Table 5.11 "Marching" & "Walking" 8-bit version of March-B+ Algorithm

representation contains a single 1-bit, with all of the other bits being 0's. In decimal representation, the "orthogonal" addresses are 0, 1, 2, 4, 8, 16, 32, 64, 128, etc., including all of the powers of 2 that are within the address range.

5.4 *Detection & Diagnosis Algorithms and BISD Hardware*

Now that we have established some notation, we will consider four different approaches to built-in self-diagnosis hardware, and their impact on diagnosis time

and on the degree of diagnostic resolution. In all of what follows, it should be remembered that our first priority is to make the various algorithms as regular and symmetric as possible (hence, not necessarily as short as possible) so that the self-test circuitry which implements these algorithms may occupy as little silicon area as possible.

5.4.1 Serial Access to Memory

The types of march elements described so far assume that the circuitry performs read and write operations *directly* on all memory cells. However, in the context of *embedded* memories, it may be necessary to perform read and write operations *indirectly* during the testing process. For example, one could apply march elements *with shifting:* although the read/write circuitry may access an entire row of bits per operation, the addition of a bit-shifting operation to the data word would appear to implement a *single bit* march element. The shifting operation is obtained by adding multiplexers to the inputs of the write drivers, with each multiplexer selecting between the normal data input (originating outside the memory) or the stored value in the transparent latch (originating from the sense amplifier) from its *left-neighboring* bit [Nadeau-Dostie *et al.* 90]. During the "normal mode" of operation, the normal data input is selected. During the "test mode" of operation, the left-neighboring transparent latch is selected, which implies that shifting is indissolubly linked to the write operation during "test mode" (i.e. the shifting operation cannot be separated from the write operation).

Table 5.13 displays these values: SDI (serial data input), four bit values representing an entire memory word, four bit values representing the output data latches, and SDO (serial data output). Values are displayed during both read and write operations, and lead to the following observations:

1. During a read operation, the contents of both the accessed memory word and the output data latches are identical.

operation	SDI	memory word	data latches	SDO
r	–	$abcd$	$abcd$	d
we	e	$eabc$	$abcd$	d
r	–	$eabc$	$eabc$	c
wf	f	$feab$	$eabc$	c
r	–	$feab$	$feab$	b
wg	g	$gfea$	$feab$	b
r	–	$gfea$	$gfea$	a
wh	h	$hgfe$	$gfea$	a
r	–	$hgfe$	$hgfe$	e

Table 5.13 Serialized march elements

2. During a write operation, only the contents of the accessed memory word are changed — the output data latches still contain their previous contents from the preceding read operation, and will only change their contents during the next read operation.

3. The SDO is identical to the contents of the rightmost bit in the output data latches, and therefore will change only during read operations, and can never change during write operations.

4. The SDI is only meaningful during write operations (obviously, SDI is identical to the contents of the leftmost bit in the memory word), and is thus inherently irrelevant during a read operation.

The coupling fault model selected must account for the fact that all operations inherently access an entire row of bits at once. It is impossible to read *only* one bit or to write *only* one bit. Thus, the coupling faults that exist between cells within the same row must be clearly distinguished from coupling faults between cells within different rows. The "same row" coupling faults are called *state coupling faults (CF/St)*, whereas the "different row" coupling faults are called *dynamic coupling faults (CF/Dyn)*.

The definition of a "serialized march element" must highlight the fact that our only controllable input is SDI, and our only observable output is SDO. In the "standard march element" we can directly control the input data bit to be written to each memory cell, and we can directly observe the output data bit which is read from each memory cell. The "serialized" notation $(W_1 \xrightarrow{\otimes B} R_0)$ means that the operation "read zero at SDO" (R_0) is performed concurrently with the operation "write one at SDI" (W_1), and that both are repeated together with the shifting operation B times (at the same word address). The direction of the arrow indicates that the shifting operation is from left to right, with SDI at the far left, and SDO at the far right. The actual sequence of "standard" operations performed on each bit of the chosen word is different for every bit:

1: for the first bit we apply $(r0, w1)(r1, w1)^{B-1}$

2: for the second bit $(r0, w0)(r0, w1)(r1, w1)^{B-2}$

3: for the third bit $(r0, w0)^2(r0, w1)(r1, w1)^{B-3}$

n: for the n^{th} bit $(r0, w0)^{n-1}(r0, w1)(r1, w1)^{B-n}$

B: for the final bit $(r0, w0)^{B-1}(r0, w1)$

In general, the "serialized" sequence of operations $(W_y \xrightarrow{\otimes B} R_x)$ performed on SDI and SDO, actually performs the "standard" sequence of operations

$$(rx, wx)^{n-1}(rx, wy)(ry, wy)^{B-n}$$

on the n^{th} bit of a B-bit word.

In the "serialized" mode of operation, faults within the output data latches can corrupt the data values which are intended to be written into the memory cells, and because we lack *direct* observability and controllability over each memory cell, this compels us to reinterpret our view of the "march element" in order to retain fault coverage. In order to detect (but not locate) faults in the B output data latches we need at least $B + 1$ serial read operations and B serial write operations [Nadeau-Dostie *et al.* 90]; however, $2B$ serial read operations and $2B$ serial write operations

are used for symmetry's sake because it minimizes the BIST circuit's area. The use of the one-bit-shifting write operation makes the application of "serialized" test algorithms a slow process, however the ability to detect and locate faults is the same as for "standard" test algorithms. The minimum non-redundant sequence is $(W_y \xrightarrow{\otimes B} R_x); (W_y \xrightarrow{\otimes 1} R_y)$, and the corresponding *symmetrical* sequence is $(W_y \xrightarrow{\otimes B} R_x); (W_y \xrightarrow{\otimes B} R_y)$, where the second half contains *entirely* redundant serial write operations and *mostly* redundant serial read operations.

The "serial read/write" operation that was used so far in the discussions above, can be described as implementing a "virtual" shift register, and is performed in two stages, as illustrated in Figure 5.1:

(1) We read the current contents of a given memory word and store it in the B bits of the Data Buffer; the rightmost bit of the Data Buffer is immediately available as the Serial Data Output.

(2) Now we force the multiplexers controlling the inputs to the write drivers to select the left-neighboring bits of the Data Buffer cells, along with the Serial Data Input at the leftmost bit position; we therefore write the SDI bit along with the $B-1$ leftmost bits of the Data Buffer back to the given memory word.

A different "serial read/write" operation exists, which is implemented using a "real" shift register, and is performed in three stages, as illustrated in Figure 5.2:

(1) We read the current contents of a given memory word and store the B bits in a Data Buffer, which is configured as a genuine shift register.

(2) We shift the contents of the Data Buffer by 1 bit to the right, we provide a new leftmost bit from the Serial Data Input, and we send the old rightmost bit to the Serial Data Output. When the Serial Data Output is compared with its expected value, this constitutes the "serial read" portion of the operation.

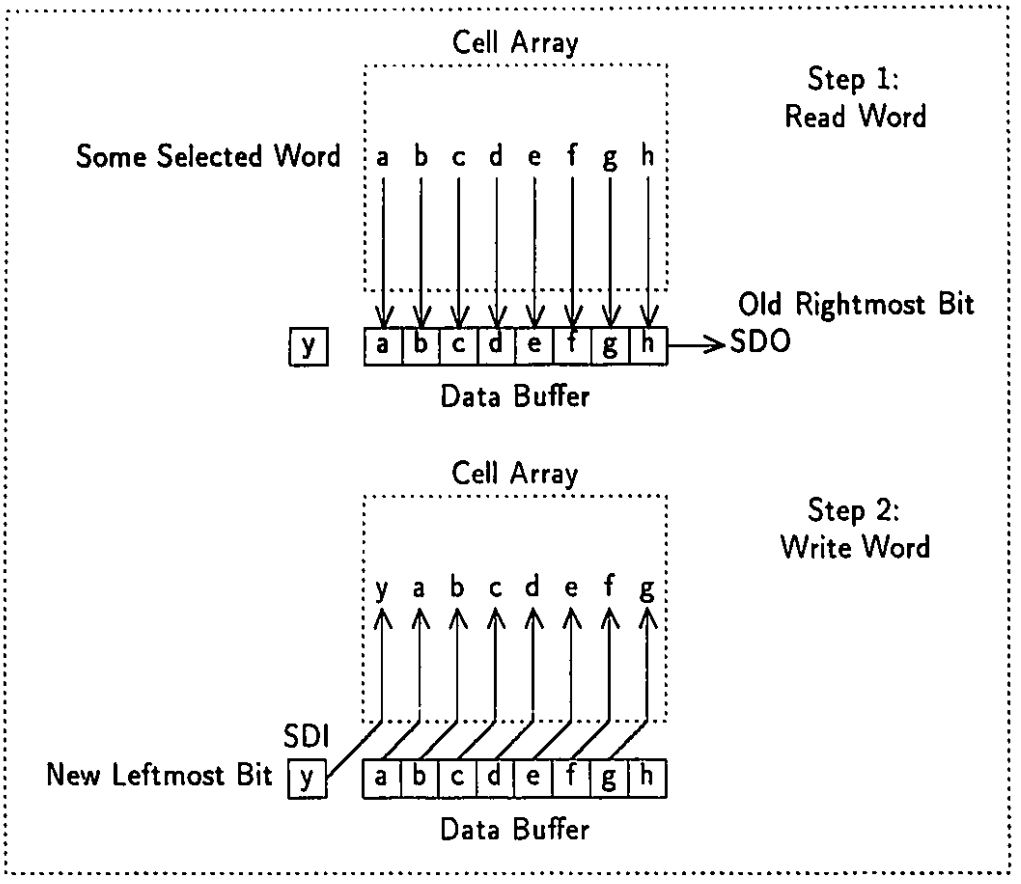

Fig. 5.1 Serial Read/Write in 2 steps, using "virtual" shift register

(3) Now we write the newly shifted contents of the Data Buffer back to the given memory word.

There exists an important *hardware* difference between the "real" and the "virtual" shifting circuits: the write drivers in the "real" shifting implementation always take their inputs from the Data Buffer — in both the BIST mode and the normal mode of operation; however, the write drivers in the "virtual" shifting implementation take their inputs from the Data Buffer only in the BIST mode — in the normal mode of operation, the Data Buffer is never connected to the write drivers.

Note that we cannot speak of a "serial write" or a "serial read" operation in

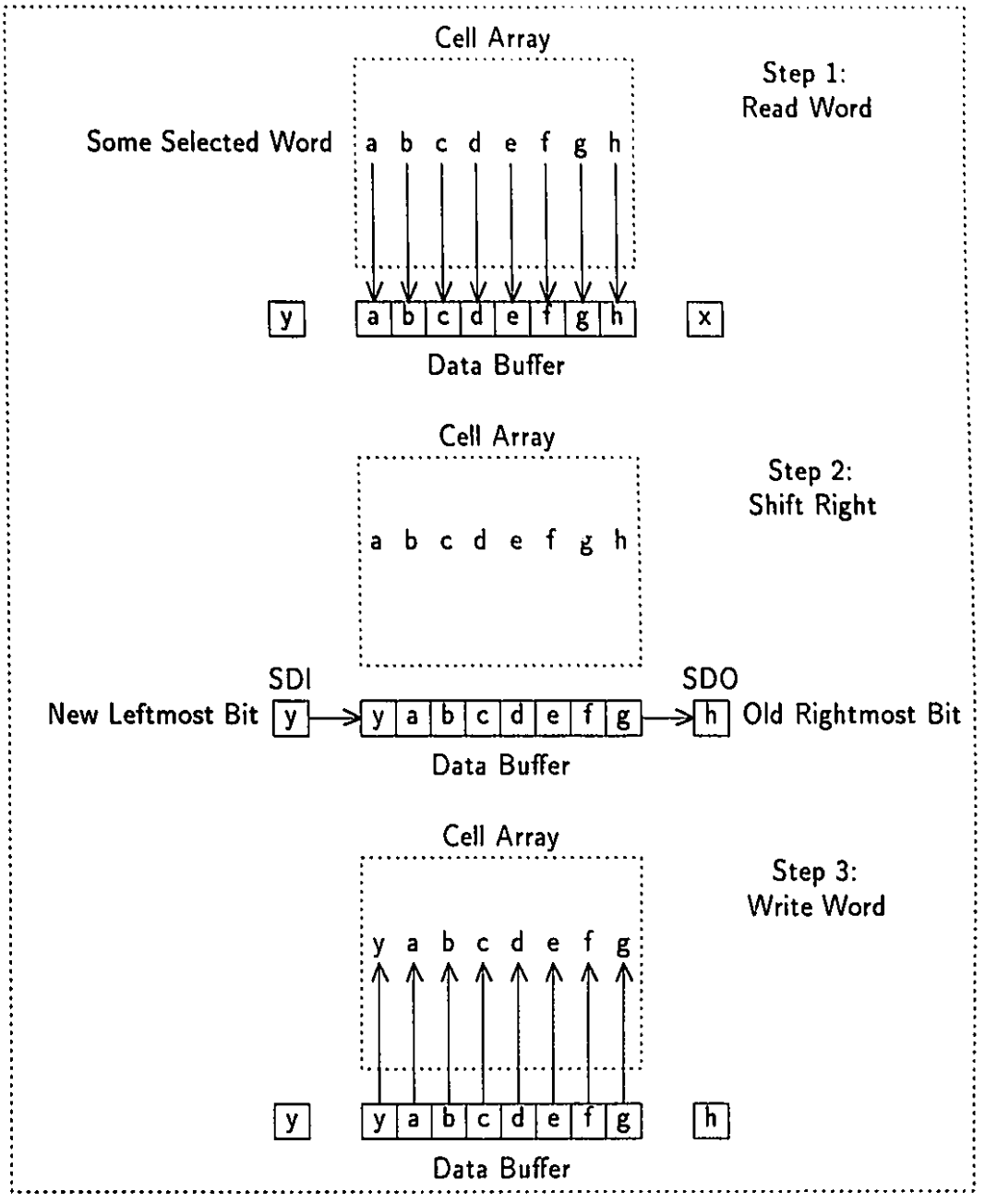

Fig. 5.2 Serial Read/Write in 3 steps, using "real" shift register

isolation, because the shifting of the Data Buffer's contents (in stage 2, above) produces an old rightmost bit (hence a serial read operation) and requires a new leftmost bit (hence a serial write operation). This is different from the "virtual"

shifting operation where the "serial read" occurs in stage 1, and the "serial write" occurs in stage 2.

5.4.2 Parallel Access to Memory

In the "parallel write" operation, B bits are written to a word of memory in one time-period. Since we are dealing with embedded static RAMs, we cannot supply this B-bit pattern from outside of the chip. Because of area constraints we are restricted in the number and the type of different B-bit patterns which we may generate on-chip. Usually, we generate patterns that are highly regular, such as this subset of the Primary data backgrounds: 11111111, 00000000, 01010101, 10101010. An easy way to obtain these patterns is to use a data buffer where each cell can be individually cleared to 0 or set to 1, and then to tie together the clear and set control lines of all the even cells and of all the odd cells. The "parallel read" operation tells us only whether such a regular pattern was correctly stored in a given memory word. If the B-bit pattern was incorrectly stored in the memory word, then the "parallel read" operation cannot identify which, or how many, of the bits are erroneous, only that at least one bit does not match the regular pattern [Mazumder and Patel 87]. The parallel read operation can be implemented using four high fan-in logic gates: two AND gates with $B/2$ fan-in and two OR gates also with $B/2$ fan-in. A different implementation uses B 2-input EXOR gates and two OR gates with $B/2$ fan-in, where half the inputs to the EXOR gates come from the memory cells, and the other half of the inputs are reference signals for comparison.

5.4.3 Hybrid Serial/Parallel Access to Memory

The use of B-bit all-0/1 comparators can speed up the application of test algorithms, and although the ability to *detect* faults using such "parallel comparators" is complete, the ability to *locate* faults is somewhat reduced for

specific kinds of faults. The purpose of using a hybrid BIST technique where both shifting operations and all-0/1 comparators are part of the extra hardware, is to combine the advantages and cancel-out the disadvantages of using either technique alone.

Serial Shifting Operations:

- advantage: can locate all kinds of faults
- disadvantage: a very slow technique for fault detection

Parallel All-0/1 Comparators:

- advantage: a very fast technique for fault detection
- disadvantage: cannot locate some kinds of faults

Therefore, the most logical use of a hybrid serial/parallel BIST method would be to (1) use the parallel all-0/1 comparators to quickly detect the presence of faults and partially locate them, and then (2) use the serial shifting operations to slowly complete the location of all faults.

5.4.4 *Modular Access to Memory*

The reading and writing of data backgrounds (for testing and diagnosis purposes) can also be performed in a different way that is neither parallel nor serial, but which is best described as "modular". For this type of write and read operations, we assume that the memory cell array has been segmented into M modules (i.e. each word consists of M groups of B/M bits). The "modular write" operation causes B/M bits to be written to exactly one of the M segments composing a word, in one time-period, leaving the remaining $B - B/M$ bits of the word untouched. Similar to the "parallel write" operation, we generate only highly regular patterns to be written to memory: 0101, 1010, 0011, 1100, 0000, 1111. The "modular read" operation tells us *more* than just whether such a reguiar pattern was correctly stored in the chosen one of M segments composing

a memory word; if the B/M-bit pattern was incorrectly stored in the given segment of the memory word, then the "modular read" operation (unlike the "parallel read" operation) does identify exactly which of the bits are erroneous. The "modular read" and "modular write" operations can be implemented by connecting each module's data buffer to a B/M-bitwide bus (called the Diagnosis and Repair Bus, described later). This bus is connected to a special circuit (called the Diagnosis and Repair Unit, also described later) with the ability, within the time for one memory cycle, to generate the B/M-bitwide regular patterns, or to identify (using EXOR gates) the individual erroneous bits of incorrect data.

All of the algorithms described below are available in two formats: first, using a combination of "parallel write", "parallel read", and "serial read/write" operations; second, using only the "modular write" and "modular read" operations. The hybrid serial/parallel format of algorithms is appropriate for the "Diagnosis Only" (i.e., using External repair) hardware design, and the modular format is appropriate for the "Diagnosis with Self Repair" hardware design.

5.5 Test-only algorithms: detection of faults.

Before we can write algorithms to detect the various faults enumerated in chapter 3, we must establish how the simultaneous presence of multiple faults can cause one fault to *mask* the effects of another fault in response to a given sequence of data patterns. When one fault *masks* the behavior of another fault, we say that the two faults are somehow "linked" to each other. A summary of functional test requirements is given below for various "linked" faults.

5.5.1 Homogeneous Linked Faults

The only linked faults where both faults are of the same type are *coupling faults*. There are two possibilities: (1) there is a single coupled cell, which is

influenced by two different coupling cells, or (2) there is a single coupled cell influenced in two distinct ways by a single coupling cell.

It can be shown that tests for *unlinked idempotent* CFs are more general than tests for *unlinked inversion* CFs. Unlinked inversion CFs can be viewed as behaving like linked idempotent CFs where the coupled cell is influenced by two idempotent CFs from a single coupling cell. More concretely, the $\langle\uparrow;\updownarrow\rangle$ CF/Inv can be viewed as a linked combination of a $\langle\uparrow;0\rangle$ CF/Id and a $\langle\uparrow;1\rangle$ CF/Id, and similarly the $\langle\downarrow;\updownarrow\rangle$ CF/Inv can be viewed as a linked combination of a $\langle\downarrow;0\rangle$ CF/Id and a $\langle\downarrow;1\rangle$ CF/Id.

It can also be shown that state coupling faults (CF/St) and bridging faults (BF) can be detected with a test that detects idempotent CFs. In tests for CF/Id faults, transition writes are the sensitizing operations, while in tests for BF and CF/St faults, states (and not transitions) constitute the sensitizing conditions. All BF and CF/St faults will be detected if the four state combinations of any two cells i and j can be reached; this condition is met by tests for CF/Id faults, as will be shown later.

It is important to note that, if we are restricted to using only *march tests*, then *"linked" inversion coupling faults (CFin)* cannot be detected in every possible case. Suppose that the three cells i, j and k, where $i < j < k$, are linked in such a way that k is the single coupled cell influenced by i with a $\langle\uparrow;\updownarrow\rangle$ CF/Inv fault, and is also influenced by j with a $\langle\uparrow;\updownarrow\rangle$ CF/Inv fault. Any $\Uparrow$ march element will generate two inversions in k, thus leaving the expected data value in k, and any $\Downarrow$ march element will not generate any inversions in k until after it has already read k. Therefore, an even number of linked $\langle\uparrow;\updownarrow\rangle$ CF/Inv faults or an even number of linked $\langle\downarrow;\updownarrow\rangle$ CF/Inv faults, on one side of the coupled cell, cannot be detected by march tests.

In the context of march tests, two of the four possible idempotent CFs may behave like inversion CFs, and therefore *some* linked combinations of an idempotent CF with an inversion CF are also undetectable by march tests. Let us

again consider three cells i, j and k, where $i < j < k$, and let k be coupled to i by a $\langle\uparrow;1\rangle$ CF/Id fault, and also let k be coupled to j by a $\langle\uparrow;\updownarrow\rangle$ CF/Inv fault. During a $\Uparrow$ march element, when cell i undergoes a $0 \rightarrow 1$ transition write, it also causes cell k to go from 0 to 1; then when cell j undergoes a $0 \rightarrow 1$ transition write, it causes cell k to invert from 1 back to its original value of 0; finally when cell k is read, no fault is detected. In an analogous manner, let k be coupled to i by a $\langle\downarrow;0\rangle$ CF/Id fault, and also let k be coupled to j by a $\langle\downarrow;\updownarrow\rangle$ CF/Inv fault. During a $\Uparrow$ march element, when cell i undergoes a $1 \rightarrow 0$ transition write, it also causes cell k to go from 1 to 0; then when cell j undergoes a $1 \rightarrow 0$ transition write, it causes cell k to invert from 0 back to its original value of 1; finally when cell k is read, no fault is detected. Therefore, an odd number of $\langle\uparrow;\updownarrow\rangle$ CF/Inv faults positioned between the coupling cell of a $\langle\uparrow;1\rangle$ CF/Id fault and the coupled cell common to all the linked faults, cannot be detected by march tests. Similarly, an odd number of $\langle\downarrow;\updownarrow\rangle$ CF/Inv faults positioned between the coupling cell of a $\langle\downarrow;0\rangle$ CF/Id fault and the coupled cell common to all the linked faults, cannot be detected by march tests.

Neighborhood pattern sensitive faults (NPSFs) cannot be classified into either "linked" or "unlinked" categories, due to the uniqueness of the neighborhood for every base cell. However, it is significant that a test for Static NPSFs is a subset of a combined test which detects both Active and Passive NPSFs. This can be seen by considering the following facts:

(1) a SNPSF implies that the base cell is "forced" to a certain value due to a certain neighborhood pattern,

(2) a test which detects ANPSFs, will detect all those SNPSFs where the base cell undergoes a transition from its "correct" value to its "forced" value,

(3) a test which detects PNPSFs, will detect all those SNPSFs where the base cell is already at its "forced" value and is being prevented from changing to its "correct" value.

Therefore, a test to detect the combination of Active and Passive NPSFs does in fact detect *all* NPSFs, including Static NPSFs.

5.5.2 Heterogeneous Linked Faults

When linked faults are of different fault types, the following results can be obtained:

(a) stuck-at faults, address decoder faults, transition faults, inversion coupling faults, idempotent coupling faults, and linked combinations of TFs and CF/Ids, linked combinations of some CF/Invs and some CF/Ids, and linked combinations of CF/Ids with other CF/Ids, can all be detected by a single march test.

(b) stuck-at faults, address decoder faults, transition faults, and active, passive and static NPSFs, can be detected by a single march test.

(c) linked combinations of CF/Invs with other CF/Invs, and linked combinations of some CF/Invs and some CF/Ids, can only be detected by *non-march* tests.

Linkage among the non-sequential address decoder faults can be analyzed with respect to all nine possible combinations (as shown in Fig. 3.1) of the four basic address decoder faults. When it is only important to *detect* and *not necessarily locate* faults, then it is useful to "map" faults occurring within the "read/write logic" and within the "address decoders" into faults occurring within the "memory cell array". The conditions which tests must satisfy in order to detect these "mapped" faults are presented below. These conditions apply only to march tests which are used to detect SAFs, TFs and CFs, and not for tests to detect NPSFs; after testing for NPSFs, a separate march test to detect AFs must be run.

A test to detect all possible combinations of AFs must contain march elements that match the following patterns:

0: $\Updownarrow(\ldots, wX)$

1: $\Uparrow (rX, \ldots, w\overline{X})$

2: $\Downarrow (r\overline{X}, \ldots, wX)$

It is possible to derive this result, by performing a detailed case by case analysis, of the nine possible combinations shown in Fig. 3.1.

When AFs are linked with TFs, the conditions change to:

0: $\Updownarrow (\ldots, wX)$

1: $\Uparrow (rX, \ldots, w\overline{X})$

2: $\Downarrow (r\overline{X}, \ldots, wX, rX)$

When AFs are linked with CFs, then the conditions become identical to the march elements required to detect CFs that are not linked to AFs.

It is impossible to devise a combined test which detects both NPSFs and AFs that are linked, because AFs may mask the NPSFs. A sequence of two tests, of which the first detects AFs and the second detects NPSFs, is also impossible, because NPSFs may mask the AFs. The only possible sequence of two tests is: to first test for NPSFs, followed by a test for AFs. The proofs of the above results are in Appendix B of [van de Goor 91].

Because tests to detect and/or locate neighborhood pattern sensitive faults require a large number of write operations, it is essential to select a sequence of applying the test patterns in order to minimize the total number of operations. The appropriate sequences for Static NPSFs are *Hamiltonian sequences.* The well-known *Gray code* is an example of a Hamiltonian sequence. The appropriate sequences for Active and Passive NPSFs are *Eulerian sequences.* An algorithm for constructing an Eulerian sequence from a given Gray code was first presented in [Hayes 80], and is reproduced on page 130 of [van de Goor 91]. Once the generation technique for test pattern data has been selected, the generation of memory cell addresses remains to be determined. For "Type-2" neighborhoods only the "Tiling" method is suitable, but for "Type-1" neighborhoods both the "Tiling"

method and the "Two-group" method are available for address generation. Each combination of (1.) fault model, namely APNPSF, ANPSF, PNPSF, and SNPSF; with (2.) test addressing, namely Type-1 tiling location, Type-1 two-group location, Type-1 two-group detection, and Type-2 tiling location; results in a collection of 15 different algorithms ($4 \times 4 = 16$, minus the PNPSF Type-1 two-group detection algorithm, which is identical to the PNPSF Type-1 two-group location algorithm). The only appropriate algorithms for self-diagnosis are the APNPSF Type-1 tiling location algorithm (with 194 steps per memory cell), the APNPSF Type-1 two-group location algorithm (with 196 steps per cell), and the APNPSF Type-2 tiling location algorithm (with 5122 steps per cell). The complete algorithms are reproduced in [van de Goor 91], pp. 136–153.

5.6 *Diagnosis algorithms: location of faults.*

Since the march-test notation is insufficient, we will now introduce some *new* notation, to be used in the remainder of this chapter, to specify the details of the diagnosis algorithms. In particular, the only type of read-action that is available in march-tests, namely $r0$ or $r1$ (also known as the "marching read-action"), must be augmented with two additional types of read-actions.

for moving-cell = 0 to $n-1$, excluding base-cell
if stored-value[moving-cell] $\neq a$, then error(moving-cell)
if stored-value[base-cell] $\neq b$, then error(base-cell)
endfor

Table 5.14 The "Galloping read-action:" $\binom{ra}{rb}$

The "galloping read-action" is represented by the symbol $\binom{ra}{rb}$, where the lower operation rb applies to the "base-cell" of this type of read-action, and the upper operation ra applies to its "moving-cell". As can be seen in Table 5.14, the

moving-cell and the base-cell are each read $n-1$ times, for a total of $2n-2$ read operations. This type of read-action is termed "galloping" because the read-action "gallops" back to the base-cell after every individual moving-cell is read.

```
for   moving-cell = 0 to n − 1, excluding base-cell
      if stored-value[moving-cell] ≠ a, then error(moving-cell)
endfor
if stored-value[base-cell] ≠ b, then error(base-cell)
```

Table 5.15 The "Walking read-action:" $(\overset{ra}{\cdot});rb$

The "walking read-action" is represented by the symbol $(\overset{ra}{\cdot});rb$, where the final operation rb applies to the "base-cell" of this type of read-action, and the upper operation ra applies to its "moving-cell". As can be seen in Table 5.15, the moving-cell is read $n-1$ times, and the base-cell is read only one time, for a total of n read operations. This type of read-action is termed "walking" because the read-action "walks" from each moving-cell to its neighboring moving-cell, and finally reads the base-cell only after all moving-cells have been read.

In practice, for both the "galloping read-action" and the "walking read-action", the values of a and b are always complements of each other, hence there are only two instances (out of a possible four) of each type of read-action that are actually used, namely $(\overset{r0}{\cdot},\overset{\cdot}{r1})$, $(\overset{r1}{\cdot},\overset{\cdot}{r0})$, $(\overset{r0}{\cdot});r1$, and $(\overset{r1}{\cdot});r0$. Furthermore, each of these read-actions also exists in numerous "restricted" forms, where the restriction amounts to a reduced number of moving-cells being selected (i.e. in Tables 5.14 and 5.15, the line *for moving-cell = 0 to $n-1$, excluding base-cell* is replaced by a shorter, more selective, FOR-loop). In such "restricted" read-actions, the set of moving-cells associated with any given base-cell is usually defined by the *electrical neighborhood* of the base-cell, which commonly includes the cells in the same row and/or column as the base-cell, and perhaps several adjacent rows and/or columns.

for base-cell = 0 to $n-1$ write-to[base-cell] $\leftarrow$ 0 endfor
for base-cell = 0 to $n-1$ write-to[base-cell] $\leftarrow$ 1 if stored-value[base-cell] $\neq$ 1, then error(base-cell) for moving-cell = 0 to $n-1$, excluding base-cell if stored-value[moving-cell] $\neq$ 0, then error(moving-cell) if stored-value[base-cell] $\neq$ 1, then error(base-cell) endfor write-to[base-cell] $\leftarrow$ 0 endfor
for base-cell = 0 to $n-1$ write-to[base-cell] $\leftarrow$ 1 endfor
for base-cell = 0 to $n-1$ write-to[base-cell] $\leftarrow$ 0 if stored-value[base-cell] $\neq$ 0, then error(base-cell) for moving-cell = 0 to $n-1$, excluding base-cell if stored-value[moving-cell] $\neq$ 1, then error(moving-cell) if stored-value[base-cell] $\neq$ 0, then error(base-cell) endfor write-to[base-cell] $\leftarrow$ 1 endfor
$\Uparrow(w0);\ \Uparrow(w1, r1, \binom{r0,}{\vdots\ r1}, w0);\ \Uparrow(w1);\ \Uparrow(w0, r0, \binom{r1,}{\vdots\ r0}, w1)$

Table 5.16 The GALPAT Algorithm

This means that the "restricted" read-actions only require $O(\sqrt{n})$ read operations in total, whereas the unrestricted read-actions require $O(n)$ read operations.

In order to allow the above notation to be used together with the "marching read-action" and the usual write operations, we introduce the *Base-cell convention*. This convention interprets the operations $r0$, $r1$, $w0$, and $w1$ as all applying to

the *base-cell*, and never to any of the moving-cells. Recall that the original "march elements" each consist of a single FOR-loop. The "augmented march elements", which now allow the use of galloping and walking read-actions, now contain two levels of FOR-loops:

(1.) a single outer FOR-loop, which acts on the variable "base-cell", and

(2.) an arbitrary number of inner (non-nestable!) FOR-loops, which act on local copies of the variable "moving-cell".

An example of this new, compact "augmented march" notation is shown in Table 5.16, which shows the widely-known GALPAT algorithm, first in pseudo-code format, followed by the "augmented march" format in the last line of Table 5.16. The galloping read-action can be further generalized by allowing more than one operation to apply to the moving-cell during the inner FOR-loop, and by allowing multiple reads of the base-cell, as shown in Table 5.17.

```
for   moving-cell = 0 to n − 1, excluding base-cell
      if stored-value[moving-cell] ≠ a, then error(moving-cell)
      if stored-value[base-cell] ≠ c, then error(base-cell)
      write-to[moving-cell] ← b
      if stored-value[moving-cell] ≠ b, then error(moving-cell)
      if stored-value[base-cell] ≠ c, then error(base-cell)
endfor
```

Table 5.17 An example of the "Generalized Galloping FOR-loop:" $\begin{pmatrix} ra, & \cdot, & wb, rb, & \cdot \\ \cdot, & rc, & \cdot, & \cdot, rc \end{pmatrix}$

In all of the algorithms discussed so far, all addressing has been exhaustive (i.e. *every* cell in the memory is accessed by march elements prefixed by the symbols ⇑ or ⇓), or nearly exhaustive (i.e. in the galloping FOR-loop, the moving-cell takes on the address of every cell in the memory, except for one — that of the base-cell). In the next few sections, algorithms are presented which only access the *orthogonal*

addresses, namely 1, 2, 4, 8, 16, 32, 64, ..., 2^n, and the address 0. We let the symbol $\perp$ refer to orthogonal addressing in the increasing order [1, 2, 4, 8, ...], and we let $\top$ refer to orthogonal addressing in the decreasing order [..., 8, 4, 2, 1]. We introduce the notation [*address list*](*operation list*) to represent the FOR-loop shown in Table 5.18. Using this notation, the symbol [0] refers to addressing only the cell with address 0.

```
for   base-cell in [address list] do
      apply (operation list) to base-cell
endfor
```

Table 5.18 Notation to represent selective addressing: [*address list*](*operation list*)

5.6.1 SAF in Read/Write circuitry

In order to locate stuck-at-0/1 faults in Data lines and Read/Write circuitry, the first step is to initialize a single data word at an arbitrary address to all zeroes. For simplicity, select the address 0. Since a stuck-at-1 data line would cause every single word, that should contain only zero data bits, to be read incorrectly, it does not matter which word is selected. See Table 5.19 for the algorithm $[0](w\vec{0});[0](r\vec{0})$.

parallel/serial version		
address	operation	word content
0	parallel-write	00000000
0	parallel-read	00000000 or X

Table 5.19 Detect stuck-at-1 data line

If we read a non-zero X from the word 0, then there may be some stuck-at-1 data lines, or there may be local faults affecting that word. Whatever the real cause of the incorrect data, we must repair the faulty word anyway, so it is quite unnecessary to probe further to discover the fault's source. Now we must locate the faulty bit position so that we can effect a repair.

Without loss of generality, assume the direction of shifting to be towards the right. We use a simple notation to describe the fault types and their locations, given an eight-bit word width: (--------) denotes no fault, (---1----) denotes that the fourth column is stuck-at-1, (-0--1---) denotes that the second column is stuck-at-0 and that the fifth column is stuck-at-1, (--0--10-) denotes that the third and seventh columns are both stuck-at-0 and that the sixth column is stuck-at-1. See Table 5.20 for the algorithm $[0](W_0 \xrightarrow{\otimes B} R_0)$, where B is the number of bit lines.

Fault-pattern:	---1----	-0-1----	---1-0--	-0-1-0--
step 1	00010000	00010000	00010000	00010000
step 2	00011000	00011000	00011000	00011000
step 3	00011100	00011100	00011000	00011000
step 4	00011110	00011110	00011000	00011000
step 5	00011111	00011111	00011000	00011000
step 6	00011111	00011111	00011000	00011000
	etc.	etc.	etc.	etc.

Table 5.20 Locate stuck-at-1 data line, in bit-serial fashion

Each step in Table 5.20 represents a Serial Read-0 / Write-0 operation. Recall that pattern detection with this type of hardware is limited to monitoring the rightmost bit after every shift operation (a simple way to monitor this bit is to send it off-chip through an I/O pin). It is easy to see that only the fault patterns ---1---- and -0-1---- have their stuck-at-1 fourth columns actually located at step 5 in the examples above (the faulty column number is obtained by subtracting

the step number where a 1-bit appears for the first time, from the number 9, which is simply the total number of columns plus one, i.e. $9-5=4$). The fault patterns ---1-0-- and -0-1-0-- fail to have their stuck-at-1 columns located because of the stuck-at-0 column on the right side which prevents the 1-bit generated by the stuck-at-1 column from propagating to column number 8 where it can be detected.

If we use Virtual shift registers, then there is a major limitation to our location procedures when there are multiple s-a-1 and/or multiple s-a-0 faults, because these complementary faults will mask out each other's effects and thereby prevent location. To perform complete location of all such faults without introducing a Real Shift Register (or having full-word access via the I/O pins), would require us to intermingle the repair process with the location procedures. In fact we would locate exactly one s-a-0/1 bit line fault at a time (namely, the rightmost such fault); then we would repair this fault before being able to locate another s-a-0/1 bit line fault.

Alternate Method: Here we use a Real Shift Register, where every register cell is further equipped with its own 2-input AND gate, 2-input OR gate, and a 1-out-of-3 multiplexer. The inputs to the AND and OR gates are (i) the value from the data line and (ii) the current value in the shift register cell. When we are looking for stuck-at-1(0) data lines, then we use the AND(OR) gate and send appropriate control signals to the multiplexer so that the output of the AND(OR) gate is the next value in the shift register cell.

Procedure to detect s-a-0 bit lines: We do exactly as for stuck-at-1 bit lines, except that the roles of 1 and 0 are reversed in the algorithm, namely $[0](w\vec{1});[0](r\vec{1});[0](W_1 \xrightarrow{\otimes B} R_1)$.

As indicated in Table 5.21, if we read X, then there are some stuck-at-0 data lines. Now we can locate these s-a-0 bit lines by shifting in both directions, using exactly the same techniques that we used for the location of s-a-1 bit lines.

The serial location procedure outlined in Table 5.20 is an original result. It locates SAFs in the Read/Write circuitry, using the "Even-Walking" data

address	operation	word
0	write	11111111
0	read	11111111 or X

Table 5.21 Detect stuck-at-0 data lines

backgrounds of Table 5.10.

5.6.2 SAF & dominant-0 BF in Address decoders

The procedures below allow us to locate stuck-at-0/1 faults in address decoders and address lines. We also locate "dominant-0 shorts" between address lines, because they behave like double s-a-0 address line faults.

address	operation	word
1	write	00000000
2	write	00000000
4	write	00000000
8	write	00000000
16	write	00000000
32	write	00000000
64	write	00000000
128	write	00000000
256	write	00000000
512	write	00000000
0	write	11111111

Table 5.22 Initialize orthogonal addresses

First we initialize the data words at the orthogonal addresses to all zeroes — in compact notation: $\perp(w\vec{0}); [0](w\vec{1})$ — as shown in Table 5.22. In what follows, we will exploit the fact that all orthogonal addresses are supposed to contain

00000000, except for address number 0 which contains 11111111. The example in Table 5.23 shows how to locate a stuck-at-1 address line, by simply reading each of the orthogonal addresses in sequence — in compact notation: $[0](r\vec{1}); \perp(r\vec{0})$. In this particular example, the fifth address-bit line is stuck-at-1, which means that when we refer to address number 0, we are really accessing address number 16 (in fact, every address whose fifth address-bit is supposed to be 0, will be aliased to another address whose fifth address-bit is 1). Looking back at the last line of Table 5.22, it is apparent that when we try to write all ones to address 0, we are really writing all ones to address 16.

desired address	real address	data word read
	-----1----	
0	0000010000	111...
1	0000010001	000...
2	0000010010	000...
4	0000010100	000...
8	0000011000	000...
16	0000010000	111... expected 000... fault located
32	0000110000	000...
64	0001010000	000...
128	0010010000	000...
256	0100010000	000...
512	1000010000	000...

Table 5.23 Locate a single stuck-at-1 address line

For stuck-at-0 address lines we get exactly the same behavior using the above procedure, as shown in the example in Table 5.24. In this example, the aliasing of addresses is reversed: now when we seek to read address 16, we really end-up accessing address 0, which contains all ones.

This procedure also has the advantage that the s-a-0/1 address line faults cannot mask each other, therefore any number of such faults can be precisely

desired address	real address	data word read
	-----0----	
0	0000000000	111...
1	0000000001	000...
2	0000000010	000...
4	0000000100	000...
8	0000001000	000...
16	0000000000	111...expected 000...fault located
32	0000100000	000...
64	0001000000	000...
128	0010000000	000...
256	0100000000	000...
512	1000000000	000...

Table 5.24 Locate a single stuck-at-0 address line

located, as shown in the example in Table 5.25. In this example, the addresses 0, 8, 64, and 256, are all aliased to the same address, namely 264 (= 8 + 256). This means that when the all ones data pattern was supposed to be written to address 0, it was really written to address 264. When addresses 8, 64, and 256, were supposed to be read, it was really address 264 that was repeatedly read.

The only information which this simple procedure fails to provide is whether a particular faulty line is stuck-at 0 or at 1. But such information is unnecessary because the repair procedure is the same for both types of faults. In the context of the procedure presented above, a dominant-0 short circuit between a pair of address lines is indistinguishable from a pair of stuck-at-0 address lines, because every address-bit —except for a single bit— has the value zero, and therefore for any pair of lines, they will never both have the value 1 at the same time. This means that the two addresses affected by a dominant-0 short circuit fault will both be aliased to the address 0.

The procedure presented in Table 5.22, namely $\perp(w\vec{0})$; [0]$(w\vec{1})$; [0]$(r\vec{1})$; $\perp(r\vec{0})$,

desired address	real address	data word read
	-1-0--1---	
0	0100001000	111...
1	0100001001	000...
2	0100001010	000...
4	0100001100	000...
8	0100001000	111...expected 000...fault located
16	0100011000	000...
32	0100101000	000...
64	0100001000	111...expected 000...fault located
128	0110001000	000...
256	0100001000	111...expected 000...fault located
512	1100001000	000...

Table 5.25 Locate several stuck-at address lines

is derived from the paper [Sarkany and Hart 87]. [Sarkany and Hart 87] only show how this procedure locates single SAFs, as shown in Tables 5.23 and 5.24, but they do not show that it locates *multiple* SAFs; therefore, Table 5.25 is an original result.

5.6.3 Dominant-1 BF in Address decoders

The next two procedures locate the dominant-1 short circuits between address lines. Recall that the procedure in the previous section covers the dominant-0 shorts between address lines, therefore only the remaining address line short circuits will now be located.

We locate one of the pair of dominant-1 shorted address lines using the procedure shown in Table 5.26 (note that this procedure does not require the use of address 0) — in compact notation: $\perp(w\vec{0}); \perp(r\vec{0}, w\vec{1}, r\vec{1})$. This procedure has located the faulty address line number 16. In order to locate the remaining faulty address line number 2, we use the procedure illustrated in Table 5.27, namely

desired address	real address	operation	real data	expected data
	-----*--*-			
1	0000000001	read	000...	ok
1	0000000001	write	111...	
1	0000000001	read	111...	ok
2	0000010010	read	000...	ok
2	0000010010	write	111...	
2	0000010010	read	111...	ok
4	0000000100	read	000...	ok
4	0000000100	write	111...	
4	0000000100	read	111...	ok
8	0000001000	read	000...	ok
8	0000001000	write	111...	
8	0000001000	read	111...	ok
16	0000010010	read	111...	000... half-fault located
16	0000010010	write	111...	
16	0000010010	read	111...	ok
	etc.			

Table 5.26 Locate one of a pair of dominant-1 shorted address lines

$\top(r\vec{1}, w\vec{0})$. The dominant-1 short causes addresses 2 and 16 to both be aliased to the address 18, and the reversal of addressing order is required to locate the fault at address 2. The procedure shown in Table 5.26 is from [Sarkany and Hart 87]. However, the procedure outlined in Table 5.27 is an original result. In summary, dominant-1 short circuits between address lines are located by applying the algorithm: $\bot(w\vec{0}); \bot(r\vec{0}, w\vec{1}, r\vec{1}); \top(r\vec{1}, w\vec{0})$.

5.6.4 Dominant-0/1 BF in Read/Write circuitry

The procedures in this section cover short circuits between data lines and in the read/write circuitry. Now that we have located all of the stuck-at faults

desired address	real address	operation	real data	expected data
	-----*--*-			
16	0000010010	write	000...	
8	0000001000	read	111...	ok
4	0000000100	read	111...	ok
2	0000010010	read	000...	111...other half located
1	0000000001	read	111...	ok

Table 5.27 Locate remaining dominant-1 shorted address line

affecting both the address lines and the data lines, and the short circuits affecting address lines, we are left with short circuits affecting data/bit lines, before we look for faults affecting memory cells.

In order to locate Dominant-0 shorted bit lines, we first want to initialize some of the words in the memory so as to contain either of the two data patterns shown in Table 5.28.

address	data word	alternate data word
1	00000001	10000000
2	00000010	01000000
3	00000100	00100000
4	00001000	00010000
5	00010000	00001000
6	00100000	00000100
7	01000000	00000010
8	10000000	00000001

Table 5.28 Data background for dominant-0 shorted bit lines

One possible way to generate these data patterns is to use Virtual shift registers. The data word stored at address 1 is obtained by first initializing it to all zeroes and then using the Serial input to inject a single 1-bit at the least significant

or most significant bit position. The data word stored at address 2 is obtained by reading address 1, then shifting the contents of the word by one bit position (in the appropriate direction), and finally writing the shifted word to address 2. Similarly, the data words stored at addresses 3, 4, 5, etc., are obtained by shifting the previous word's contents by one bit position and writing them into the next address. This procedure generates the desired data pattern correctly when there are no faults present. Tables 5.29 and 5.30 show what kinds of data patterns are generated, using the above outlined procedure (based on Virtual shifting), when there are short circuits between the bit lines. We *redefine* the notation ---0-0-- to signify that two bit lines are shorted together, with *zeroes* dominant (it no longer signifies two stuck-at-0 faults), and ---1-1-- to signify that two bit lines are shorted together, with *ones* dominant (it no longer signifies two stuck-at-1 faults). In the serialized notation shown in Table 5.29, the symbol $[n](W_a \xrightarrow{[m]} R_b)$ signifies that the new contents of $[n]$ are the right-shifted old contents of $[m]$, with the leftmost new bit given by W_a, and the rightmost old bit sent to R_b.

The faults can be located by reading each address and checking for the all zeroes condition using the all-zeroes Detector (i.e. by inserting the parallel read operation $[n](\bar{r}\vec{0})$ after each shifting operation). The notation $\bar{r}\vec{0}$ means *do NOT* expect to read all-zeroes in the *fault-free* case, because in the fault-free case, none of the data words [1,2,3,4,5,...,B] is all-zeroes. In Table 5.29, the first word to contain all-zeroes corresponds to the position of one of the shorted lines, and by this technique location is accomplished.

We can locate Dominant-1 shorted bit lines in a similar manner, except that the roles of 0 and 1 in the data words are reversed (see Table 5.30).

When we get multiple Dominant-0 or Dominant-1 shorts then the two outermost bit lines are located easily. When we have a mixture of dominant-0 and dominant-1 shorts, then we will observe one of the five cases shown in Tables 5.31 to 5.35.

address	data word		shift right's
	shift left	shift right	serialized-
	---0-0--	---0-0--	notation
0	00000000	00000000	$[0](w\vec{0})$
1	00000001	10000000	$[1](W_1 \xrightarrow{[0]} R_0)$
1	parallel read		$[1](\bar{r}\vec{0})$
2	00000010	01000000	$[2](W_0 \xrightarrow{[1]} R_0)$
2	parallel read		$[2](\bar{r}\vec{0})$
3	00000000	00100000	$[3](W_0 \xrightarrow{[2]} R_0)$
3	parallel read		$[3](\bar{r}\vec{0})$
4	00000000	00000000	$[4](W_0 \xrightarrow{[3]} R_0)$
4	parallel read		$[4](\bar{r}\vec{0})$
5	00000000	00000000	$[5](W_0 \xrightarrow{[4]} R_0)$
5	parallel read		$[5](\bar{r}\vec{0})$
	etc.	etc.	etc.
$8 = B$	00000000	00000000	$[B](W_0 \xrightarrow{[B-1]} R_0)$
B	parallel read		$[B](\bar{r}\vec{0})$
$9 = B + 1$	00000000	00000000	$[B+1](W_0 \xrightarrow{[B]} R_1)$

Table 5.29 Locate dominant-0 shorted bit lines

address	data word	
	shift left	shift right
	---1-1--	---1-1--
1	11111110	01111111
2	11111101	10111111
3	11111111	11011111
4	11111111	11111111
5	11111111	11111111
	etc.	etc.

Table 5.30 Locate dominant-1 shorted bit lines

	shift left	shift right	shift left	shift right
	--10-01-	--10-01-	--10-01-	--10-01-
1	00000001	10000000	11111110	01111111
2	00100010	01000000	11111111	10111111
3	01000000	00100010	11111111	11111111
4	10000000	00000001	11111111	11111111
5	00000000	00000000	11111111	11111111

Table 5.31 Case 1: Dominant-0 short sandwiched by Dominant-1 short

	shift left	shift right	shift left	shift right
	--01-10-	--01-10-	--01-10-	--01-10-
1	00000001	10000000	11111110	01111111
2	00000000	01000000	11011101	10111111
3	00000000	00000000	10111111	11011101
4	00000000	00000000	01111111	11111110
5	00000000	00000000	11111111	11111111

Table 5.32 Case 2: Dominant-1 short sandwiched by Dominant-0 short

	shift left	shift right	shift left	shift right
	-0--101-	-0--101-	-0--101-	-0--101-
1	00000001	10000000	11111110	01111111
2	00001010	00000000	11111111	10111011
3	00010000	00000000	11111111	11011111
4	00100000	00000000	11111111	11101111
5	00000000	00000000	11111111	11111111

Table 5.33 Case 3: overlapping Dominant-0 and Dominant-1 shorts, different widths

In cases 1, 2 and 3 both the Shift Left and Shift Right operations, with

	shift left	shift right	shift left	shift right
	--01-01-	--01-01-	--01-01-	--01-01-
1	00000001	10000000	11111110	01111111
2	00010010	01000000	11111111	10111111
3	00100100	00000000	11111111	11011011
4	01001000	00000000	11111111	11101101
5	10010010	00000000	11111111	11110110
6	00100100	00000000	11111111	11011011
7	01001000	00000000	11111111	11101101
8	10010010	00000000	11111111	11110110

Table 5.34 Case 4: overlapping Dominant-0 and Dominant-1 shorts, equal widths

	shift left	shift right	shift left	shift right
	--00-11-	--00-11-	--00-11-	--00-11-
1	00000001	10000000	11111110	01111111
2	00000110	01000000	11111111	10111111
3	00001110	00000000	11111111	11001111
4	00001110	00000000	11111111	11000111
5	00001110	00000000	11111111	11000111

Table 5.35 Case 5: completely separate Dominant-0 and Dominant-1 shorts

both Zeroes and Ones backgrounds, provide us with information that allows the location of the two outermost faulty lines. However, in cases 4 and 5, the Zeroes background Shift Left operations and the Ones background Shift Right operations provide us with no useable information since the data words never become all zeroes or all ones (therefore the all zeroes/all ones comparator will not report the presence of any faults). Only the Zeroes background Shift Right operations and the Ones background Shift Left operations show that there is something wrong, allowing us again to locate only the two outermost faulty lines.

This procedure, based upon Virtual shifting, requires partial repair before there can be further location of faulty lines. However, if we apply the same procedure with Real shift registers being used to generate the required data patterns, then all faulty lines can be located concurrently.

The procedures presented above are a completely original application of the "Odd-Walking" data backgrounds (see Table 5.9). [Sarkany and Hart 87] use *only one* of the four "Odd-Walking" data backgrounds — namely the central column in Table 5.28 (also the last column in Table 5.9) — and their algorithm applies this data background only to address 0, whereas the procedures illustrated in Tables 5.28 through 5.35 apply *all four* data backgrounds to a *sequence* of addresses, excluding address 0, equal in number to the total number of data bit lines. [Sarkany and Hart 87] also do not mention the cases of multiple shorted lines, as shown in Tables 5.31 through 5.35.

5.6.5 SAF & BF in Memory cells

The procedures shown below locate stuck-at memory cells, short circuits between memory cells, and passively coupled transition faults. In the following text, we use a small 16-bit (i.e. 4 words of 4 bits each) memory cell array to more clearly illustrate the functioning of the procedures.

First, we initialize the entire memory to all zeroes (see Table 5.36), $\Uparrow (w\vec{0})$.

0000
0000
0000
0000

Table 5.36 Initialize memory

Then, we perform a Read of each word to certify that the initialization is complete and correct. In the following, we use the abbreviation $\Uparrow (r\vec{x})$ to mean:

"Read each word in the memory, and verify that it is all zeroes or all ones, as appropriate" (in other words, employ either $r\vec{0}$ or $r\vec{1}$, as appropriate). We use the abbreviation $\Uparrow (r\vec{1})$ to mean: "Read each word in the memory, and verify that it is all ones or is NOT all ones, as appropriate" (in other words, employ either $r\vec{1}$ or $\bar{r}\vec{1}$, as appropriate).

1111		1111		1111		1111	
0000	$\Uparrow (r\vec{x})$,	1111	$\Uparrow (r\vec{x})$,	1111	$\Uparrow (r\vec{x})$,	1111	$\Uparrow (r\vec{x})$.
0000		0000		1111		1111	
0000		0000		0000		1111	

Table 5.37 Detect stuck-at-0 cell faults

The short procedure in Table 5.37 checks quickly for the existence (but not the location) of stuck-at-0 cell faults. Now we use the longer procedure in Table 5.38 to locate (1) stuck-at-1 cell faults and (2) transition faults, where a transition fails because it is passively coupled to one or more neighboring cells.

If undesired transitions occur (i.e. there is more than a single zero-bit in the memory array), this happens because the Cell Under Test is Actively coupling the remaining cells which are undergoing transitions. The location of these particular coupling faults is by another procedure.

Now we repeat the procedure in Table 5.38 with a background of all zeroes and a single walking-1 that appears and disappears at every cell, in order to locate the stuck-at-0 faults and the remaining passively coupled cells.

The procedures above use hybrid serial/parallel operations only, and they are original results. If modular operations are to be used instead, then an abbreviated version of March-B+ can be used to locate exactly the same faults, namely the algorithm shown in Table 5.39, which is also a previously unpublished march test.

0111		1111		1011		1111	
1111	$\Uparrow(\dot{r}\vec{1})$,	1111	$\Uparrow(\dot{r}\vec{1})$,	1111	$\Uparrow(\dot{r}\vec{1})$,	1111	$\Uparrow(\dot{r}\vec{1})$,
1111		1111		1111		1111	
1111		1111		1111		1111	
1101		1111		1110		1111	
1111	$\Uparrow(\dot{r}\vec{1})$,	1111	$\Uparrow(\dot{r}\vec{1})$,	1111	$\Uparrow(\dot{r}\vec{1})$,	1111	$\Uparrow(\dot{r}\vec{1})$,
1111		1111		1111		1111	
1111		1111		1111		1111	
1111		1111		1111		1111	
0111	$\Uparrow(\dot{r}\vec{1})$,	1111	$\Uparrow(\dot{r}\vec{1})$,	1011	$\Uparrow(\dot{r}\vec{1})$,	1111	$\Uparrow(\dot{r}\vec{1})$,
1111		1111		1111		1111	
1111		1111		1111		1111	
et cetera							
1111		1111		1111		1111	
1111	$\Uparrow(\dot{r}\vec{1})$,	1111	$\Uparrow(\dot{r}\vec{1})$,	1111	$\Uparrow(\dot{r}\vec{1})$,	1111	$\Uparrow(\dot{r}\vec{1})$,
1111		1111		1111		1111	
0111		1111		1011		1111	
1111		1111		1111		1111	
1111	$\Uparrow(\dot{r}\vec{1})$,	1111	$\Uparrow(\dot{r}\vec{1})$,	1111	$\Uparrow(\dot{r}\vec{1})$,	1111	$\Uparrow(\dot{r}\vec{1})$.
1111		1111		1111		1111	
1101		1111		1110		1111	

Table 5.38 Locate stuck-at-1 cell faults

$\Updownarrow(w0)$
$\Updownarrow(r0, w1, r1, w0, r0)$
$\Updownarrow(w1)$
$\Updownarrow(r1, w0, r0, w1, r1)$

Table 5.39 The *new* Diagnostic March-B± Algorithm

Cycle 1:							
1111		0111		0111		0111	
1111	$\Uparrow(r\vec{1})$	1111	$\Uparrow(W_\star \overset{\otimes B}{\rightarrow} R_\star)$	0111	$\Uparrow(W_\star \overset{\otimes B}{\rightarrow} R_\star)$	0111	$\Uparrow(W_\star \overset{\otimes B}{\rightarrow} R_\star)$
1111		1111		1111		0111	
1111		1111		1111		1111	
0111		0011		0011		0011	
0111	$\Uparrow(W_\star \overset{\otimes B}{\rightarrow} R_\star)$	0111	$\Uparrow(W_\star \overset{\otimes B}{\rightarrow} R_\star)$	0011	$\Uparrow(W_\star \overset{\otimes B}{\rightarrow} R_\star)$	0011	$\Uparrow(W_\star \overset{\otimes B}{\rightarrow} R_\star)$
0111		0111		0111		0011	
0111		0111		0111		0111	
et cetera							
0000		0000		0000		0000	
0001	$\Uparrow(W_\star \overset{\otimes B}{\rightarrow} R_\star)$	0000	$\Uparrow(W_\star \overset{\otimes B}{\rightarrow} R_\star)$	0000	$\Uparrow(W_\star \overset{\otimes B}{\rightarrow} R_\star)$	0000	$\Uparrow(r\vec{0})$
0001		0001		0000		0000	
0001		0001		0001		0000	

Table 5.40 Locate Actively coupled and coupling cells

5.6.6 CF in Memory cells

At this point, the combination of all the procedures above allows us to locate all types of stuck-at faults, all types of shorts, and all passively coupled cells. This leaves actively coupled and coupling cells to be located. The procedures in Tables 5.40 to 5.43 locate both the coupled and coupling cells, using hybrid serial/parallel operations.

The notation $\Uparrow(W_\star \overset{\otimes B}{\rightarrow} R_\star)$ signifies a succession of *wraparound-shifted* read and write operations performed on every word, in order to serially scan the contents of every bit. Between these wraparound-shifted operations, which require a total of BW shifting operations, we perform a pair of full-word read and write operations to implement a column-wise shifting operation. We reinitialize to all zeroes before continuing with Table 5.42.

This four-stage procedure for hybrid serial/parallel operations is an original

Cycle 2:							
0000		0000		0000		0000	
0000	$\Uparrow(r\vec{0})$	0000	$\Uparrow(W_\star \overset{\otimes B}{\rightarrow} R_\star)$	0000	$\Uparrow(W_\star \overset{\otimes B}{\rightarrow} R_\star)$	0001	$\Uparrow(W_\star \overset{\otimes B}{\rightarrow} R_\star)$
0000		0000		0001		0001	
0000		0001		0001		0001	
et cetera							
0111		0111		0111		1111	
0111	$\Uparrow(W_\star \overset{\otimes B}{\rightarrow} R_\star)$	0111	$\Uparrow(W_\star \overset{\otimes B}{\rightarrow} R_\star)$	1111	$\Uparrow(W_\star \overset{\otimes B}{\rightarrow} R_\star)$	1111	$\Uparrow(r\vec{1})$
0111		1111		1111		1111	
1111		1111		1111		1111	

Table 5.41 Locate Actively coupled and coupling cells

Cycle 3:							
0000		1000		1000		1000	
0000	$\Uparrow(r\vec{0})$	0000	$\Uparrow(W_\star \overset{\otimes B}{\rightarrow} R_\star)$	1000	$\Uparrow(W_\star \overset{\otimes B}{\rightarrow} R_\star)$	1000	$\Uparrow(W_\star \overset{\otimes B}{\rightarrow} R_\star)$
0000		0000		0000		1000	
0000		0000		0000		0000	
et cetera							
1111		1111		1111		1111	
1110	$\Uparrow(W_\star \overset{\otimes B}{\rightarrow} R_\star)$	1111	$\Uparrow(W_\star \overset{\otimes B}{\rightarrow} R_\star)$	1111	$\Uparrow(W_\star \overset{\otimes B}{\rightarrow} R_\star)$	1111	$\Uparrow(r\vec{1})$
1110		1110		1111		1111	
1110		1110		1110		1111	

Table 5.42 Locate Actively coupled and coupling cells

fault location algorithm. If modular operations are to be used instead, then a two-stage procedure can be used to locate the same faults. A suitable first stage is the march algorithm IFA-13 (see Table 4.2). From IFA-13, a list of faulty cells is compiled, and this list contains only the *coupled* cells and not the *coupling* cells.

To locate the *coupling* cells, the second stage employs an algorithm with Galloping FOR-loops, as shown in Table 5.44. The list of IFA-13–discovered

Cycle 4:							
1111		1111		1111		1111	
1111	$\Uparrow(r\vec{1})$	1111	$\Uparrow(W_\star \overset{\otimes B}{\rightarrow} R_\star)$	1111	$\Uparrow(W_\star \overset{\otimes B}{\rightarrow} R_\star)$	1110	$\Uparrow(W_\star \overset{\otimes B}{\rightarrow} R_\star)$
1111		1111		1110		1110	
1111		1110		1110		1110	
et cetera							
1000		1000		1000		0000	
1000	$\Uparrow(W_\star \overset{\otimes B}{\rightarrow} R_\star)$	1000	$\Uparrow(W_\star \overset{\otimes B}{\rightarrow} R_\star)$	0000	$\Uparrow(W_\star \overset{\otimes B}{\rightarrow} R_\star)$	0000	$\Uparrow(r\vec{0})$
1000		0000		0000		0000	
0000		0000		0000		0000	

Table 5.43 Locate Actively coupled and coupling cells

coupled cells is denoted by [failed cells]. This algorithm is clearly different from GALPAT (see Table 5.16), and it constitutes an original result.

$\Uparrow(w0)$
[failed cells]$(w1, r1, (\overset{r0,}{\cdot}\ \overset{w0,}{\cdot}\ \overset{r0,}{\cdot}\ \overset{\vdots}{r1}), w0)$
$\Uparrow(w1)$
[failed cells]$(w0, r0, (\overset{r1,}{\cdot}\ \overset{w1,}{\cdot}\ \overset{r1,}{\cdot}\ \overset{\vdots}{r0}), w1)$

Table 5.44 A Galloping algorithm to locate Coupling cells

5.6.7 Sequential SOAF

Using the procedure in Table 5.45 we can locate stuck-open faults in the address decoders. Recall from section 3.3.6 that this kind of fault manifests itself by selecting two address lines at the same time, such that the addresses are a Hamming distance of 1 apart. Given that we have an M-bit long address, then there are 2^M lines controlled by the decoder, and therefore $M \times 2^M$ pairs of addresses are required to detect the presence of all possible SOAFs.

In order to locate such faults, we must exploit the fact that two word lines are being activated at the same time. If the two words being accessed are supposed to store different values, then by trying to read each word individually we may obtain erroneous readings because of a wired-AND or wired-OR effect between the two lines. In the procedure shown in Table 5.45, we assume a wired-OR effect, and we let w_i stand for the binary representation of word address i. The time required to run this procedure, assuming W words, is: $W + W(1 + 3 \log W + 1) = 3W + 3W \log W$ steps.

for $w_i = 1 \rightarrow W$ do { write 000... to w_i}
for $w_i = 1 \rightarrow W$ do { write 111... $\rightarrow w_i$; read w_i ; expect value 111... ; read $(w_i \oplus 1000...)$; expect value 000... else stuck-open ; read $\overline{w_i}$; ignore value ; read w_i ; expect value 111... ; read $(w_i \oplus 0100...)$; expect value 000... else stuck-open ; read $\overline{w_i}$; ignore value ; et cetera; read w_i ; expect value 111... ; read $(w_i \oplus ...0010)$; expect value 000... else stuck-open ; read $\overline{w_i}$; ignore value ; read w_i ; expect value 111... ; read $(w_i \oplus ...0001)$; expect value 000... else stuck-open ; read $\overline{w_i}$; ignore value ; write 000... $\rightarrow w_i$; }

Table 5.45 Locate Stuck-open addressing faults

As was explained in section 3.3.6, such faults are located by first applying an "initializing pattern" IP, followed by a "sensitizing pattern" SP. In any algorithm that uses a sequence of addresses, such that any two adjacent addresses are a Hamming distance of 1 apart, the location of faults is ambiguous. The located

fault is within the decoder whose address is IP, and the faulty bit position within decoder IP is given by $IP \oplus SP$. For example, using 4-bit addresses, and letting the base-address be 0000, the sequence mentioned above is 0000, 0001, 0000, 0010, 0000, 0100, 0000, 1000. Note that 0001 can be an SP to the first instance of 0000, or 0001 can be an IP to the second instance of 0000. To prevent this multiplicity of IP/SP status, we should insert a "buffering pattern" BP right after the intended SP. Using a BP of 1111, the above sequence becomes 0000, 0001, 1111, 0000, 0010, 1111, 0000, 0100, 1111, 0000, 1000, 1111. This sequence will locate SOAFs only within the 0000–decoder.

The cycle of three read operations shown in Table 5.45 is analogous to the Galloping FOR-loop, but since the addressing follows a "unit Hamming distance" scheme, we call it the Hamming FOR-loop. We can use a similar notation to the Galloping loop for the Hamming loop by replacing the parentheses with angle brackets $\langle\rangle$. The algorithm of Table 5.45 can be rewritten in this notation as follows: $\Uparrow(w0)$; $\Uparrow(w1, \left\langle \dot{r1}, \overset{r0,}{\cdot}, \dot{\Re x} \right\rangle, w0)$. We use the notation $\dot{\Re x}$ to represent $[\overline{base\ cell}](rx)$, where we ignore the read value, because this operation is only inserted to buffer the address pairs. (We could also use $\overset{\Re x}{\cdot}$, which represents the operation $[\overline{moving\ cell}](rx)$, to play the role of buffering pattern BP.)

5.6.8 APNPSF location algorithms

The hardware design described in the next chapter is intended for use with Static RAMs, for which NPSFs are not applicable (hence, for the remainder of this thesis, the location of NPSFs is not required). The same BISD circuit can be used with Dynamic RAMs, with appropriate changes to the address generators and a reprogrammed ROM. Since the 3 types of APNPSF location algorithms are long and complicated, they will not be reproduced here — complete details of these 3 algorithms are available in [van de Goor 91], as was mentionned previously.

5.7 Repair algorithms: allocation of spares

The computational complexity of optimal reconfiguration is known to be $\mathcal{NP}$-complete [Kuo and Fuchs 87]. Many different heuristic algorithms for spare allocation have been published [Day 85], [Kuo and Fuchs 87], and each one is appropriate in different situations. The actual type and amount of redundancy available, in a given RAM design, will determine which heuristic algorithm should be used; the selection criteria for heuristic algorithms involves graph-theoretic analysis, which is beyond the scope of the topic of this thesis.

The DR-Unit diagnoses and repairs each (b+1)-bit line memory module completely before starting a diagnosis of the next (b+1)-bit line module. First, the Address Mode bit is set to 1, the b Data Bits are set to the current module number, and the module-number decoder's flip-flop is activated only in the appropriate Programmable Module. Then by setting the Test Mode bit to 1 (Address and Program Modes are 0), the b Data Bits are connected to the b bit lines (excluding the $(b+1)^{th}$ spare bit line) in the chosen module, and the DR-Unit can start executing diagnosis procedures stored in ROM. When we must consider the allocation of spare bit lines and spare word lines, then we need more temporary storage than the scratch-pad registers in the DR-Unit provide. The solution is to use a fault-free module of the memory array to store a complete fault map of another module, which is being diagnosed. It is significant that *no* extra area overhead is needed to store the location of faulty cells, since with high probability there will always be some fault-free "Home Module" available to store the fault maps of other modules.

5.8 Global control algorithm

The DR-Unit's ROM contains an algorithm to control the entire diagnosis and repair process, as shown in Figure 5.3. The DRU tests itself in two passes. In

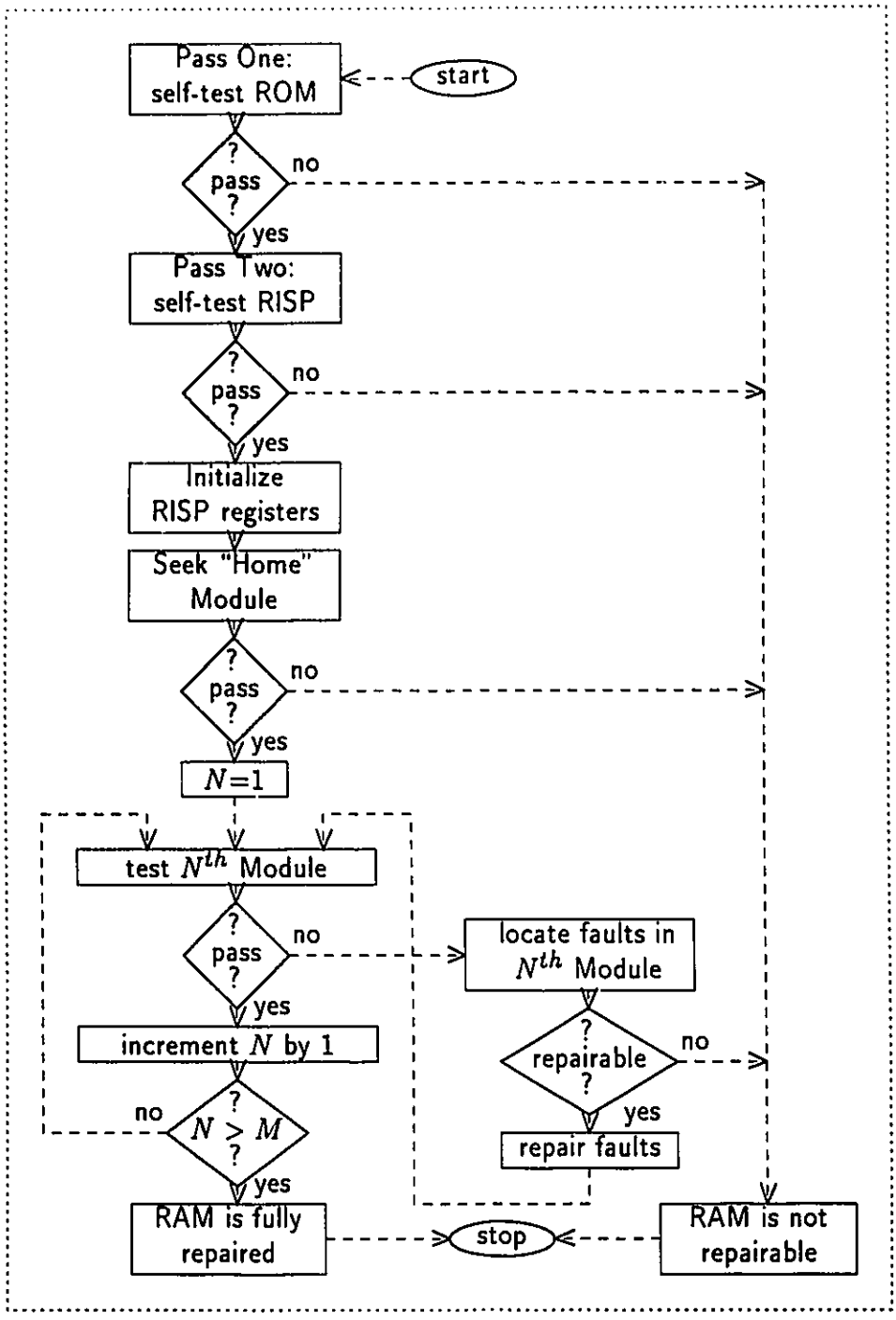

Fig. 5.3 Flow-chart of global Diagnosis and Repair procedure

the first pass, the Instruction Registers are configured into a multi-input linear-feedback shift register (MISR), while the Program Counter is incremented by one

at every clock cycle (i.e., no Branch instructions are used). This results in the entire contents of the ROM being read into the MISR, which at the end of Pass One will contain a Signature. If the Signature matches the Expected Signature, which may be stored as the very first or the very last word in the ROM, then we assume that there are no faults in the Program Counter, Instruction Register, the ROM's address decoder, and the entire contents of the ROM, and we go on to Pass Two.

In Pass Two, we run a self-test program also stored in a part of the ROM, which exercises all the remaining registers in the Data Path and all the instructions implemented by the RISP. If the DRU successfully completes Pass Two, then we assume it to be fault-free and we can start testing the embedded RAM itself.

6 Description of Hardware Design:

6.1 General overview of test hardware.

Test algorithms require certain hardware circuits:

1. An address generator which counts upward from 0 to $N-1$ and downward from $N-1$ to 0, is best implemented as a binary up/down counter with a pipelined carry signal to reduce the delay of the ripple-carry signal. When *march tests* are used, then address generators based on LFSRs are quite suitable.
2. An optional wait counter, which is connected with the address generator, to count down the wait time for the optional data-retention test.
3. A data pattern generator which generates the required data word patterns and their complementary values.
4. A data receiver that compares the expected value with the value actually obtained from a static RAM during a Read operation. A data receiver can behave in one of two ways:

   (a.) During every read operation, a comparison is made between the actual and the expected values of data.

   (b.) The data from the read operations are compressed using polynomial division in a parallel signature analyzer, and only the final

contents of the parallel signature analyzer (called the signature) are compared with the expected signature. This is only suitable when the location of faults is not required.

5. A control circuit which implements the test algorithm by controlling the various self-test circuits and the memory itself. The self-test controller is designed starting from a state diagram, and can be implemented as a finite-state machine, using a programmable logic array with flip-flops to store the internal state values.

The self-testing RAM designs mentionned previously, only consider the problem of fault detection. In order to provide instructions for the repair of a particular embedded RAM, we must consider the problem of fault *location* in addition to fault detection. The ability to locate faults calls for an "intelligent" self-diagnosis circuit, which we call a Diagnosis and Repair Unit (DR-Unit), to be added to the repairable embedded RAM. This thesis hypothesizes that the best way to design the DR-Unit is to combine a small reduced-instruction-set processor (RISP) with a small ROM, because such a design best satisfies the following constraints:

(1.) it can be easily adapted to various memory designs,

(2.) it requires minimal redesign of the memory itself, and

(3.) it has minimal impact on normal memory performance.

Here is a more detailed explanation of each of these constraints:

(1.) The easy adaptability to a variety of memory designs is due to the fact that the instructions stored in the DR-Unit's ROM can be easily changed to suit every kind of memory organization and can be adapted to detect various types of faults associated with different fabrication technologies.

(2.) Since we are assuming that an embedded RAM cannot be directly accessed through the chip's I/O pins, this means that such a RAM is almost never connected to data and address busses running through the chip because

such busses almost always lead to I/O pins. Instead, an embedded RAM is connected directly to neighboring circuits without busses. This means that the DR-Unit's RISP will be connected to an embedded RAM through an extra Bus that is used only for the purposes of diagnosis and repair. Hence the only redesign of the RAM that may be required is the incorporation of this Diagnosis and Repair Bus (DR-Bus), which is a relatively simple task.

(3.) The only changes in RAM performance will be caused by the DR-Bus, since all other parts of the DR-Unit are completely outside the RAM. The only other "foreign" circuits inside the RAM are the spare rows, spare columns and the programmable fuses; but these extra circuits are commonly found in full-chip RAMs, where their impact on performance is acceptable, hence their impact on embedded RAM performance is likely to be acceptable also.

An added advantage of using a DR-Bus to do diagnosis is that if the DR-Bus is connected directly to the memory cell array's sense amplifiers and write drivers without going through the address decoders, then any faults in the decoders can be isolated from any faults in the cell array, thereby simplifying the problems of fault detection and location.

6.1.1 A Comparison of BIST Architectures

The most widely used classification scheme for BIST designs is based on:

a) the number of bits accessed during a read or write operation (i.e. single bit or multiple bit access),

b) the number of cell sub-arrays accessed during a read or write operation (i.e. only a single sub-array is active, or multiple sub-arrays are active).

Hence there are four general classes of BIST architectures:

SASB: single sub-array, single bit;

SAMB: single sub-array, multiple bit;

MASB: multiple sub-array, single bit;

MAMB: multiple sub-array, multiple bit.

6.1.1.1 *Single sub-array, single bit*

The SASB architecture is the only architecture which could be used with *unmodified*, standard (non-BIST) algorithms that are also used by external testers to apply the test inputs and evaluate the resultant outputs. Some of the coupling fault models can be fully tested only with the SASB architecture.

6.1.1.2 *Single sub-array, multiple bit*

The SAMB architecture is also referred to as *line-mode testing* when there exists an internal mechanism capable of accessing a complete word line to:

1. write the same data into multiple cells in a single access, and
2. read the data from multiple cells and evaluate this data in a single access by using compression techniques.

Another kind of SAMB architecture, much less common than *line-mode testing*, involves making internal modifications to both the column and row decoders. The *modified decoders mechanism* allows r rows and c columns to be selected at the same time during a write operation, which implies that the same data value can be written to $r \cdot c$ cells. A read operation can only access c columns within one row.

Given a memory with n cells, with $\sqrt{n}$ rows and $\sqrt{n}$ columns, then *line-mode testing* can reduce the test time of most algorithms by a factor of $\sqrt{n}$. Some of the coupling faults will require a mixed architecture approach because of the difficulty of detecting certain types of coupling faults between cells belonging to the same row. Using the *modified decoders mechanism* most algorithms can be accelerated even more, however coupling faults will again pose a slight problem.

6.1.1.3 *Multiple sub-array, single bit*

The MASB architecture is only applicable to larger memories that contain several sub-arrays. If these sub-arrays can be operated independently of each other, then they can be tested in parallel — where a single bit is accessed from each sub-array. The major implementation advantages of the MASB architecture are:

a) the circuitry required for generating addresses and test patterns can be shared among s sub-arrays,

b) the response data can be verified by mutually comparing the s responses of the sub-arrays, and

c) all the coupling fault models are completely testable.

6.1.1.4 *Multiple sub-array, multiple bit*

The MAMB architecture combines most of the characteristics of both the SAMB and MASB architectures. This architecture can accelerate most algorithms by a factor of $\sqrt{n} \cdot s$. As with the SAMB architecture, there exists some difficulty in detecting coupling faults between cells belonging to the same row.

6.1.1.5 *Other architectural parameters*

The architectural classification system described above presents a viewpoint which is entirely appropriate for Built-In Self-*Test*, but is not adequate for Built-In Self-*Diagnosis* and Self-*Repair*. The most important additional parameter to be added to this classification scheme involves the organization of the spare rows and spare columns. If the spare lines are placed in individual sub-arrays, then we have *local redundancy*, and if the spare lines belong to the entire cell array as whole, then we have *global redundancy*.

6.1.2 BIST Address Generators

This section discusses the selection of address generators. In comparison with test data generators and response data evaluators, the address generators almost always require the most layout area. There are two possibilities for the implementation of an address generator: (1) a counter, or (2) a pseudo-random pattern generator (PRPG). Counters have the disadvantage that they require more hardware than PRPGs because of the carry propagation logic, and counters are also difficult to make self-testable.

When *march tests* are used, then address generators based on PRPGs are quite suitable. It is a straightforward matter to construct a PRPG using an LFSR, that has been augmented to produce the *all-zeroes pattern*, and that has been further modified so that the same pseudo-random sequence of addresses can be generated in both the forward and reverse directions. PRPGs built out of LFSRs have the following property: the probability that a particular address bit changes during any given clock cycle is equal for all address bits — this makes possible the detection of *some* write recovery faults (discussed in chapter 3), which are due to address dependent delays in the decoder circuit, that would not be detected if counters were used as address generators.

6.1.3 Pseudo-random pattern generators

"Pseudo-random pattern generators" (PRPG) are implemented as modified "linear feedback shift registers". The instructions for constructing a suitable PRPG are as follows:

1. select a primitive polynomial (in the Galois Field of order 2) of the desired bit-length, and a maximum length sequence (excluding only the all-zeroes pattern) will be produced from an LFSR whose feedback paths correspond to the primitive polynomial, by a very well known result from coding theory,

2. to add the all-zeroes pattern to the maximum length sequence, the required extra circuitry comprises a high fan-in NOR-gate (or its functional equivalent) and one extra EXOR-gate — the NOR-gate takes as input the state of every single cell, except for the last one, and the extra EXOR-gate takes as input the state of the last cell and the output of the NOR-gate; the result of this modification is that the state 000...001 does not immediately go to the next state 100...000 but instead passes through the state 000...000,

3. in order to construct a PRPG which generates exactly the reverse sequence of a given PRPG, we use the same cells which must be able to shift in the reverse direction, and we implement the *reciprocal primitive polynomial*; then we add the high fan-in NOR-gate and extra EXOR-gate, and a multiplexer at each end of the LFSR.

6.2 Descriptions of the Cells, and their Operation

The list below enumerates the hardware components of a "soft switching" design for embedded RAMs, using *local* redundancy. We assume that the RAM is divided into M sub-arrays (also called modules), each containing $(B/M + 1)$ bit lines, such that the single spare bit line in a given sub-array can only replace one of the B/M non-spare bit lines in the same sub-array.

6.2.1 "withRepair" (Fig. 6.1)

This is the top-level cell of the proposed design which incorporates the ability of the hardware to do both the diagnosis and repair of faults entirely by itself, without any help from outside circuitry. The signal START specifies when a self-diagnosis and self-repair session should take place, with the circuit operating in normal-mode at other times. The signals STOP(1:2) specify whether the self-repair successfully produced an operational circuit, or if there are some remaining unrepairable faults.

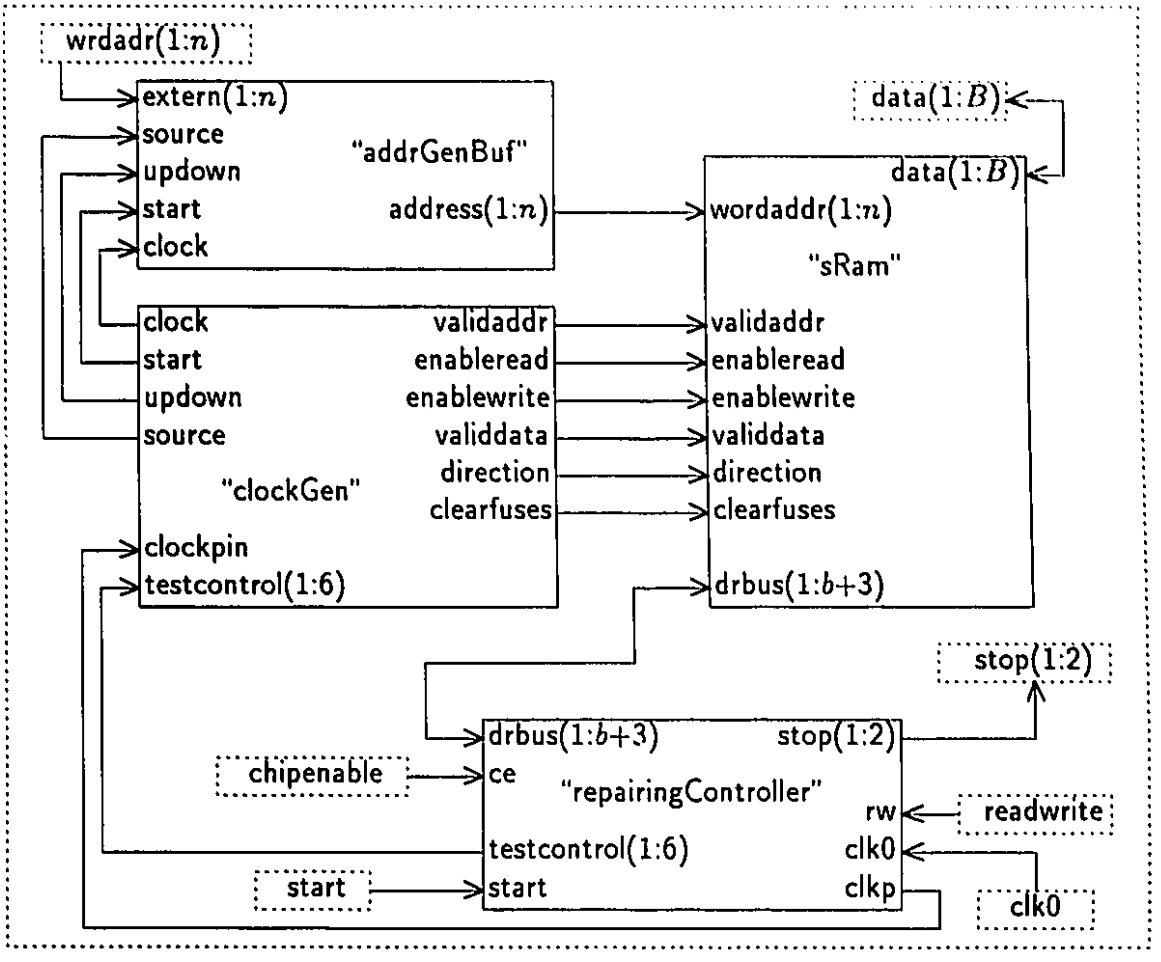

Fig. 6.1 "withRepair"

The signals WRDADR(1:n) carry the n bits of the word address, and hence we have $W = 2^n$ words in the memory. The signals DATA(1:B) show that each word in the memory contains B data bits. Therefore, the total capacity of this memory is $BW = B2^n$ bits.

6.2.1.1 "addrGenBuf" (Fig. 6.2)

This cell provides memory addresses in three ways: (1) by allowing the externally-supplied address from EXTERN(1:n) to pass through, during the normal mode of operation; (2) by generating a monotonically increasing sequence of addresses starting at 000 (all zeroes); (3) by generating a monotonically decreasing

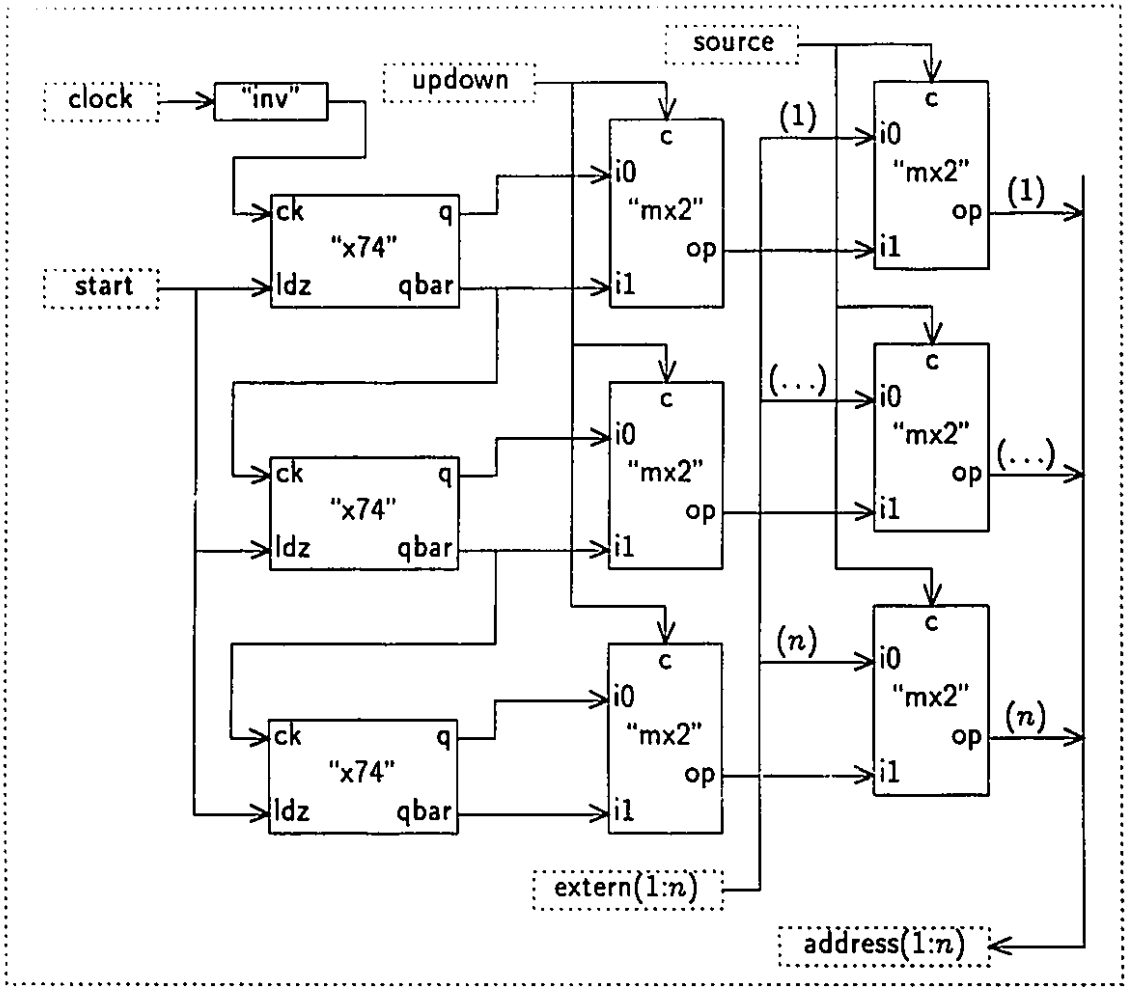

Fig. 6.2 "addrGenBuf"

sequence of addresses starting at 111 (all ones). These last two addressing sequences are created by an internal counter, and are used during the test mode of operation.

6.2.1.2 "clockGen"

This cell generates 10 control signals, that are properly synchronized with each other, from the six bits of TESTCONTROL(1:6) and the CLOCKPIN.

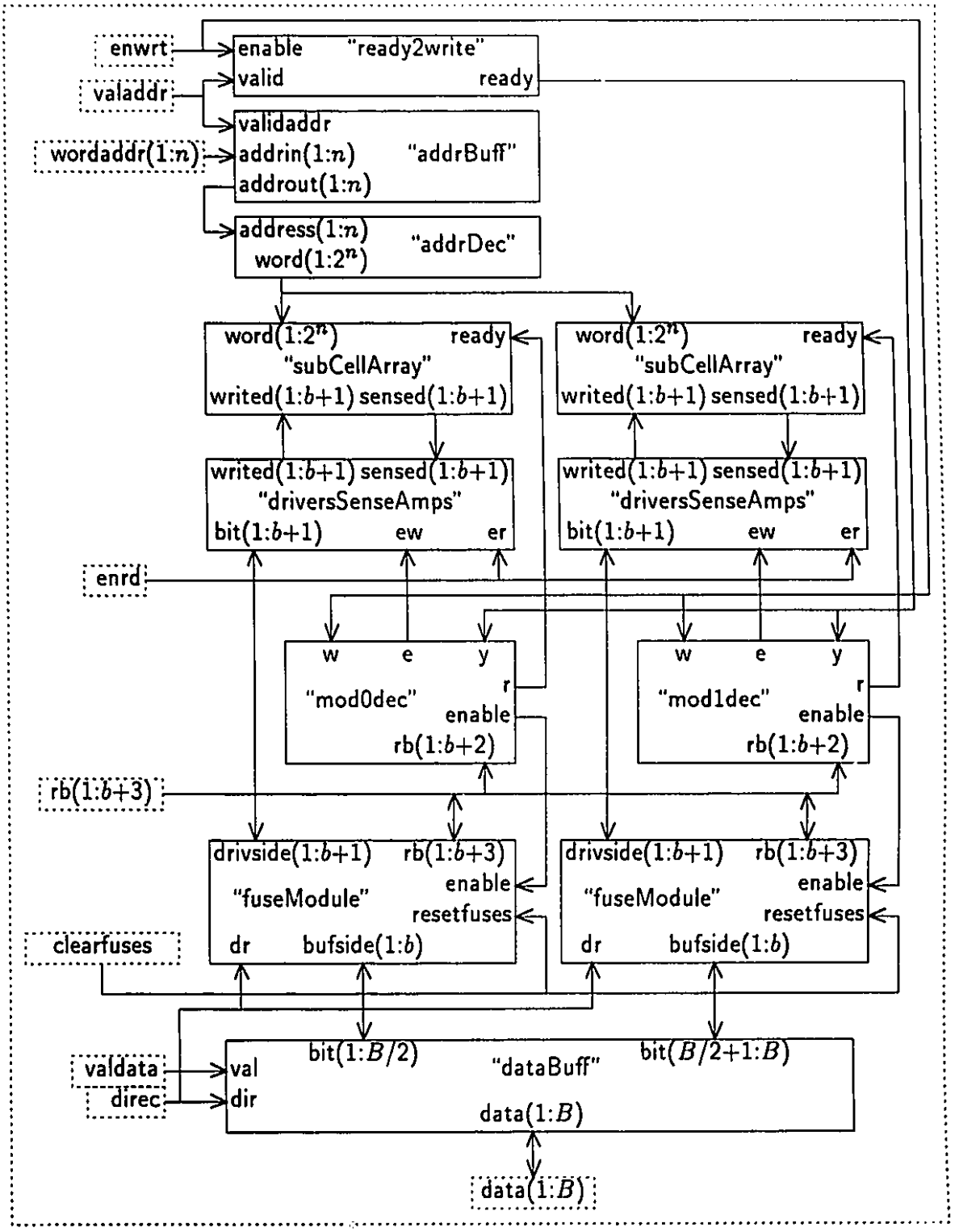

Fig. 6.3 "sRam" (repairable Static RAM)

6.2.1.3 *"sRam" (Fig. 6.3)*

This cell embodies a static RAM which has been augmented with circuitry that enables the repair of some faults. Note that the memory array itself is partitioned into M modules, each module containing the cells "subCellArray", "driversSenseAmps", "mod#dec", and "fuseModule". Each module contains one spare bit line which provides the redundancy required for the repair of faults.

6.2.1.4 *"addrDec"*

This cell is a standard design of an address decoder circuit.

6.2.1.5 *"addrBuff"*

This cell constitutes a buffer for the address, using a column of D-type flip-flops with a rising (positive) edge-triggered clock.

6.2.1.6 *"ready2write"*

This cell generates a synchronized pulse which indicates when a "write to memory operation" is valid.

6.2.1.7 *"subCellArray"*

Each memory sub-array (also called module) is composed of b+1 columns: $b = B/M$ basic columns and one spare column. As a result of limitations in the CAD software used to create this hardware design, some non-standard elements were used, such as two separate sets of bit lines: SENSED(0:4) for reading from the memory, and WRITED(0:4) for writing to the memory. The CAD software was supplied by Bell-Northern Research, and it did not contain standard-cells for SRAMs, and its simulator did not allow the use of bidirectional lines in many circumstances. The resulting hardware design does not invalidate the BISD technique itself, in spite of the occasional use of a non-standard feature.

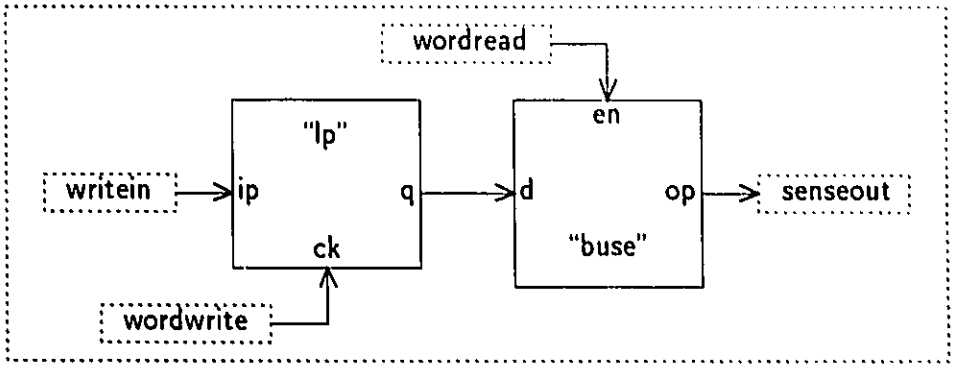

Fig. 6.4 "singleCell"

6.2.1.8 "singleColumn"

Each column is composed of W individual memory cells.

6.2.1.9 "singleCell" (Fig. 6.4)

Figure 6.4 shows a non-standard SRAM cell required by the BNR functional simulator.

6.2.1.10 "driversSenseAmps"

This is an design of write-drivers and sense-amplifiers with storage provided by edge-triggered flip-flops. The signal SENSED(1:b+1) is a unidirectional line from the memory array, WRITED(1:b+1) is a unidirectional line to the memory array, and BIT(1:b+1) is a bidirectional line to the data buffer.

6.2.1.11 "mod#dec" (Fig. 6.5)

This cell is an address decoder for module number #. The address is carried by RB(1:b), which is a valid address when RB(b+1)'s edge rises. RB(b+2) specifies whether we are reading/writing to all the modules in the memory together, or only to one selected module. In general there can be any number of modules, each one requiring a unique module decoder.

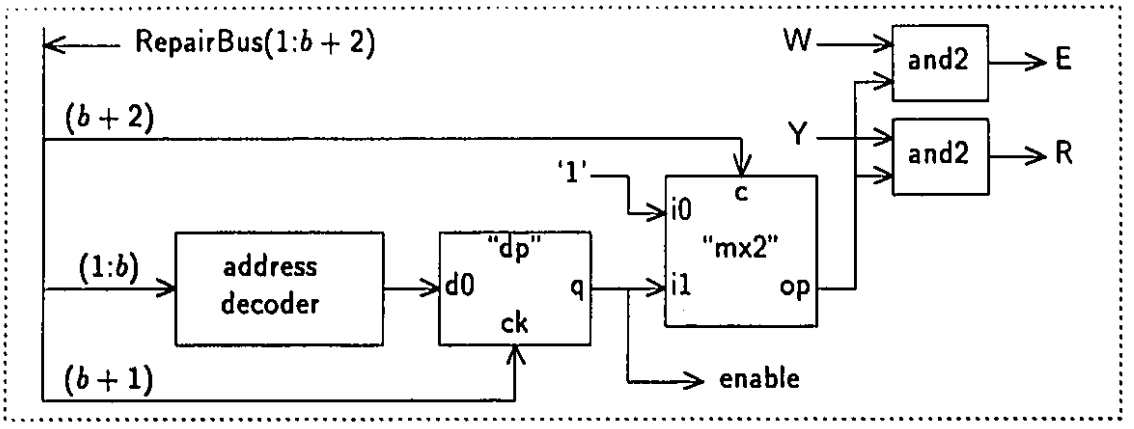

Fig. 6.5 "mod#dec" (module #0 decoder)

6.2.1.12 *"fuseModule" (Fig. 6.6)*

This cell provides a programmable interface between the b bits of the data buffer, and the b+1 bits (one of which is the spare) of the memory module. The data-carrying signals on both sides of this cell are bidirectional, hence the signal DR selects the direction of data-flow. The signal RESETFUSES forces the fuses into a predetermined default setting. The signal ENABLE comes from one of the module address decoders, to specify which module is being tested or being reconfigured using its fuses.

6.2.1.13 *"softFuse" (Fig. 6.7)*

The soft fuse is really a resettable, edge-triggered D-type flip-flop ("drsp"). The signal DC specifies the direction of data-flow: from the buffer-side (BUFS) to the drivers-side (DRIVS/SPARE), or vice versa. The state of "drsp" determines whether the signal SPARE replaces DRIVS.

6.2.1.14 *"bidirecMux" (Fig. 6.8)*

This cell is a bidirectional 1-of-2 multiplexer. The signal DR specifies the direction of data-flow: from the one-bit-side (W) to the two-bit-side (V0/V1),

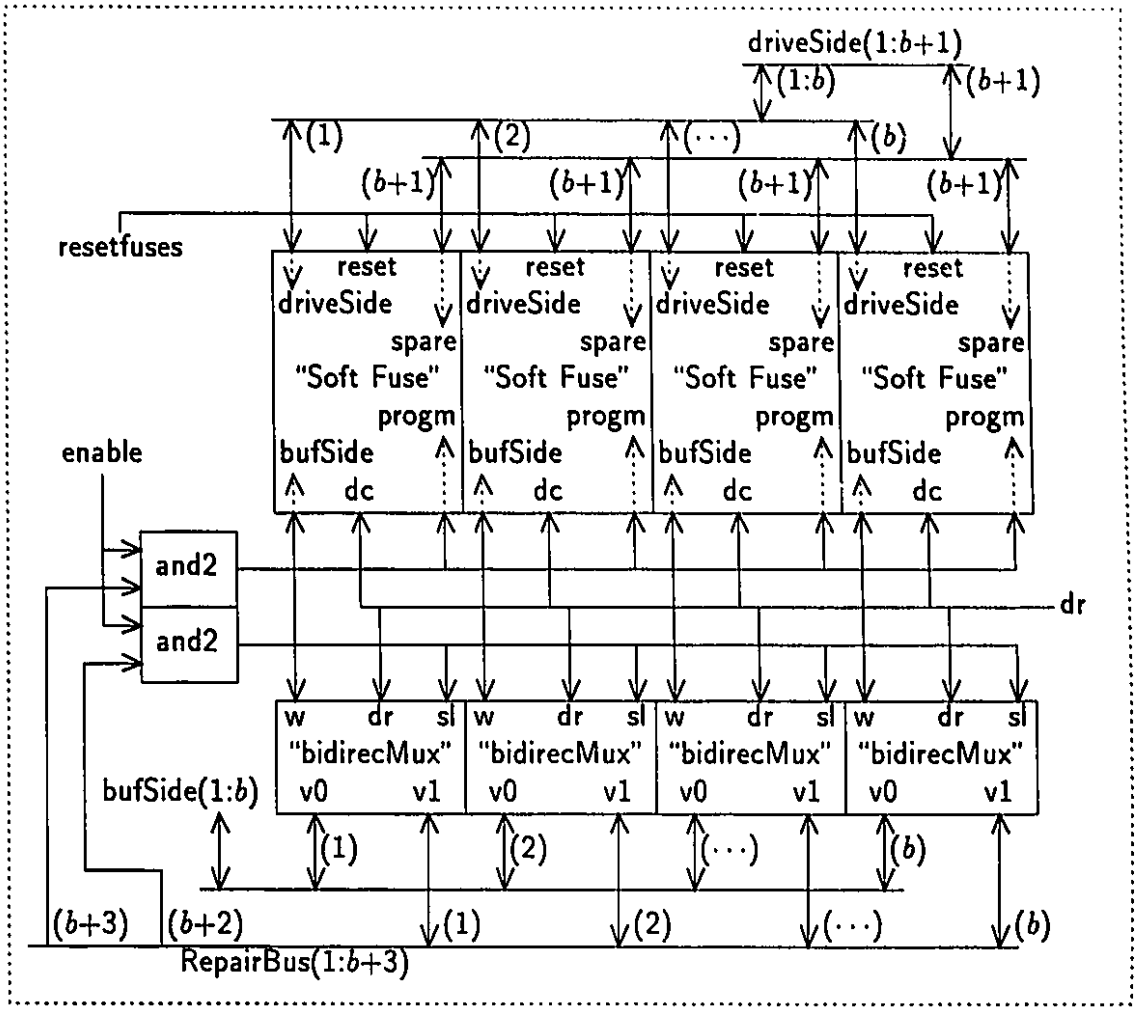

Fig. 6.6 "fuseModule"

or vice versa. The signal SL selects which of V0 or V1 will be connected to W. Ordinarily, this cell would be laid out quite simply as two transmission gates in parallel, but since transmission gates are not available in this design environment, this more complex design is used instead.

6.2.1.15 "dataBuffer" (Fig. 6.9)

This cell implements an B-bit bidirectional data buffer, using B instances of a D-type flip-flop ("dp"), B instances of a unidirectional multiplexer ("mx2"), and $2B$ instances of a three-state buffer ("buse").

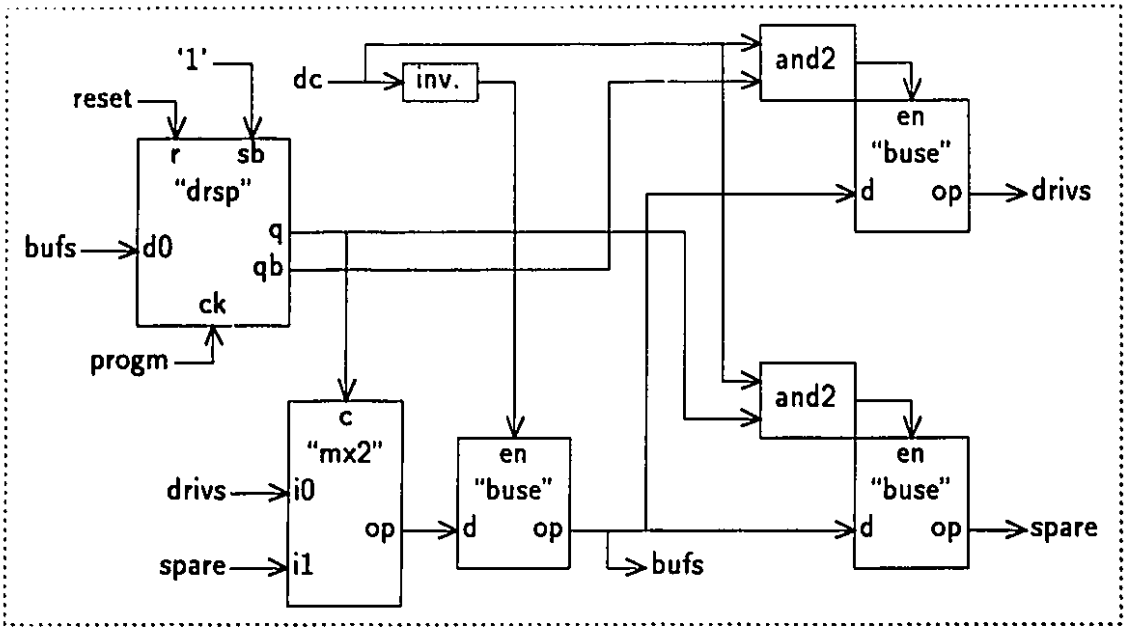

Fig. 6.7 "softFuse"

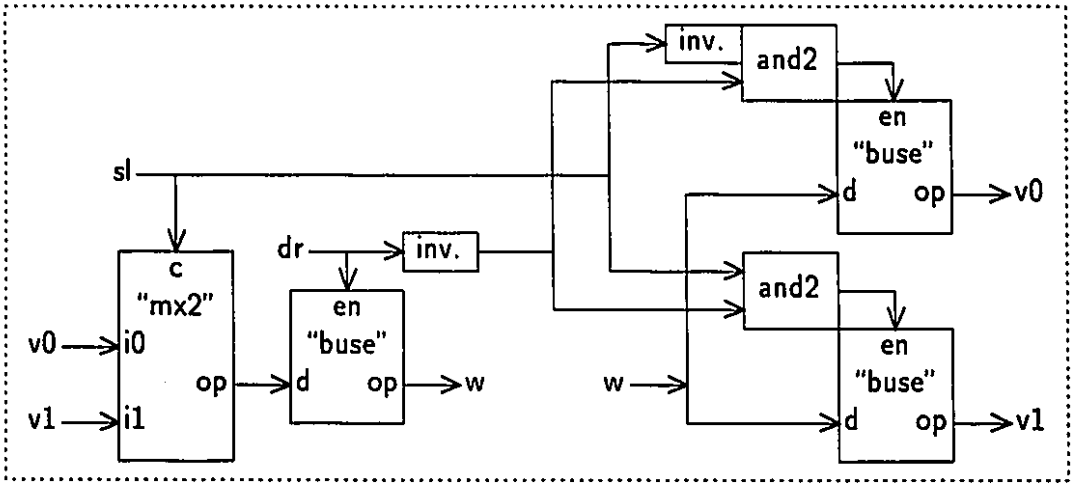

Fig. 6.8 "bidirecMux" (bidirectional multiplexer)

6.2.1.16 "repairingController" (Fig. 6.10)

This cell represents a programmable controller which carries out the functions of testing the memory array, locating the faults, and then programming the soft fuses to effect repairs. The only means of data transfer between the controller

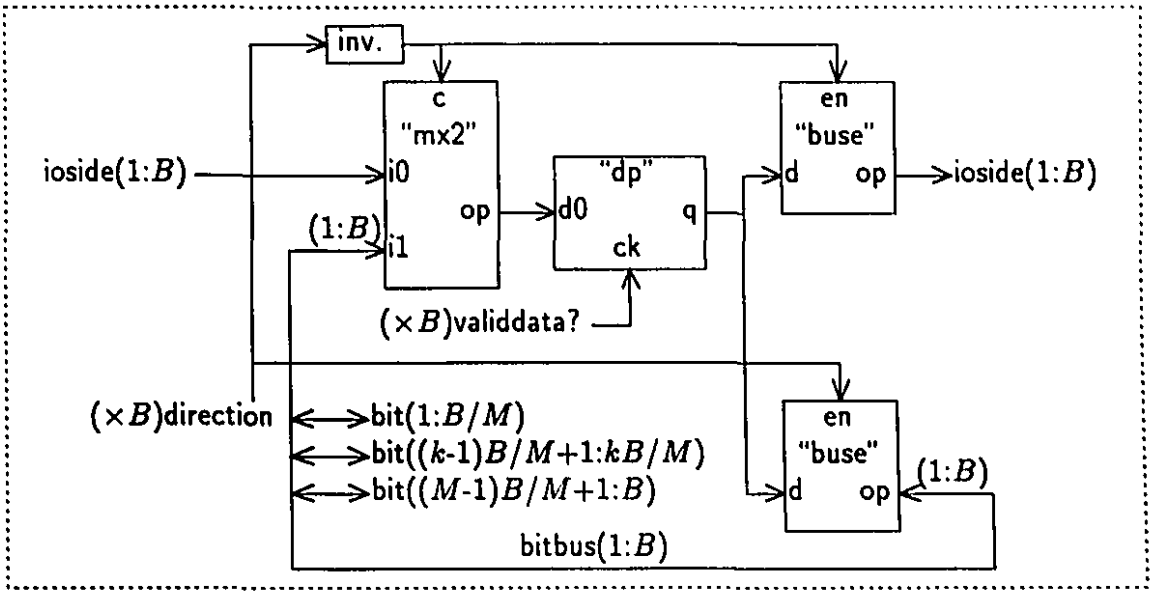

Fig. 6.9 "dataBuffer"

and the memory array is the bidirectional part of the DR-Bus (DRBUS(1:b)). Unidirectional control lines from the controller are three lines in the DR-Bus (DRBUS(b+1:b+3)), and six more lines (TESTCONTROL(1:6)). The controller contains a finite-state machine combined with an instruction register ("instrRegCntl"), software stored in a ROM ("Rom"), a program counter for the ROM combined with a stack ("pCstack"), decoding circuits which convert the current instruction into 44 control signals ("instrDecod" and "clkGen2"), and a data path which creates test vectors for storage into the memory, examines the vectors retrieved from the memory, and programs the soft fuses to reconfigure the memory ("alu", "readOnlyRegs", "readWriteRegs", and "busPort").

6.2.1.17 "alu" (Fig. 6.11)

This cell is a b-bit arithmetic and logic unit, which contains the input registers X and Y that hold the operands, and the output register Z that holds the result of the operation. Three flags are also generated which identify when the operands: add up to 0 (FLAG(1)), are identical (FLAG(2)), or cause an overflow condition

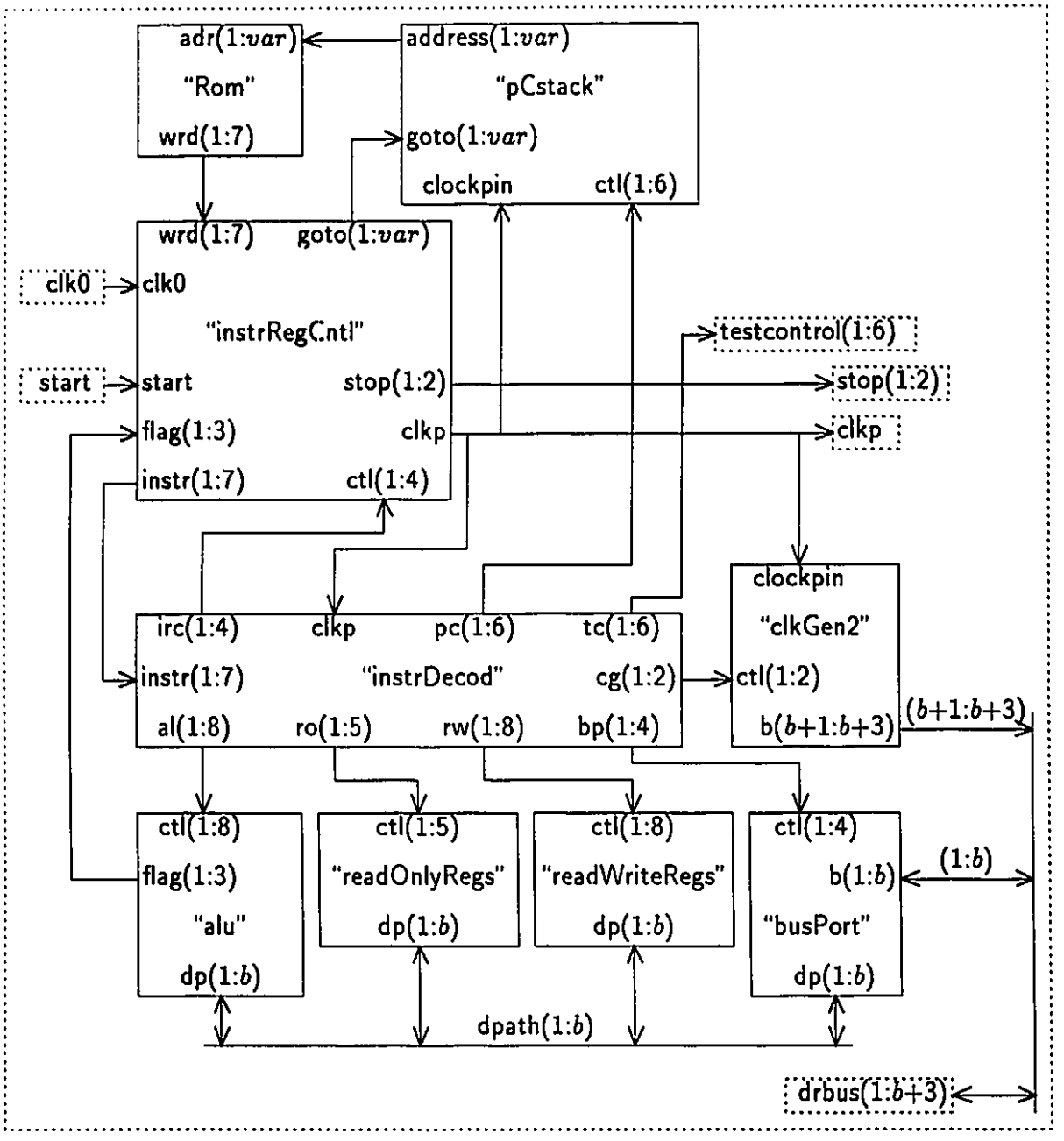

Fig. 6.10 "repairingController"

when they are added together (FLAG(3)).

6.2.1.18 "readOnlyRegs"

This cell contains 5 numerical constants which can be fetched by "alu", or

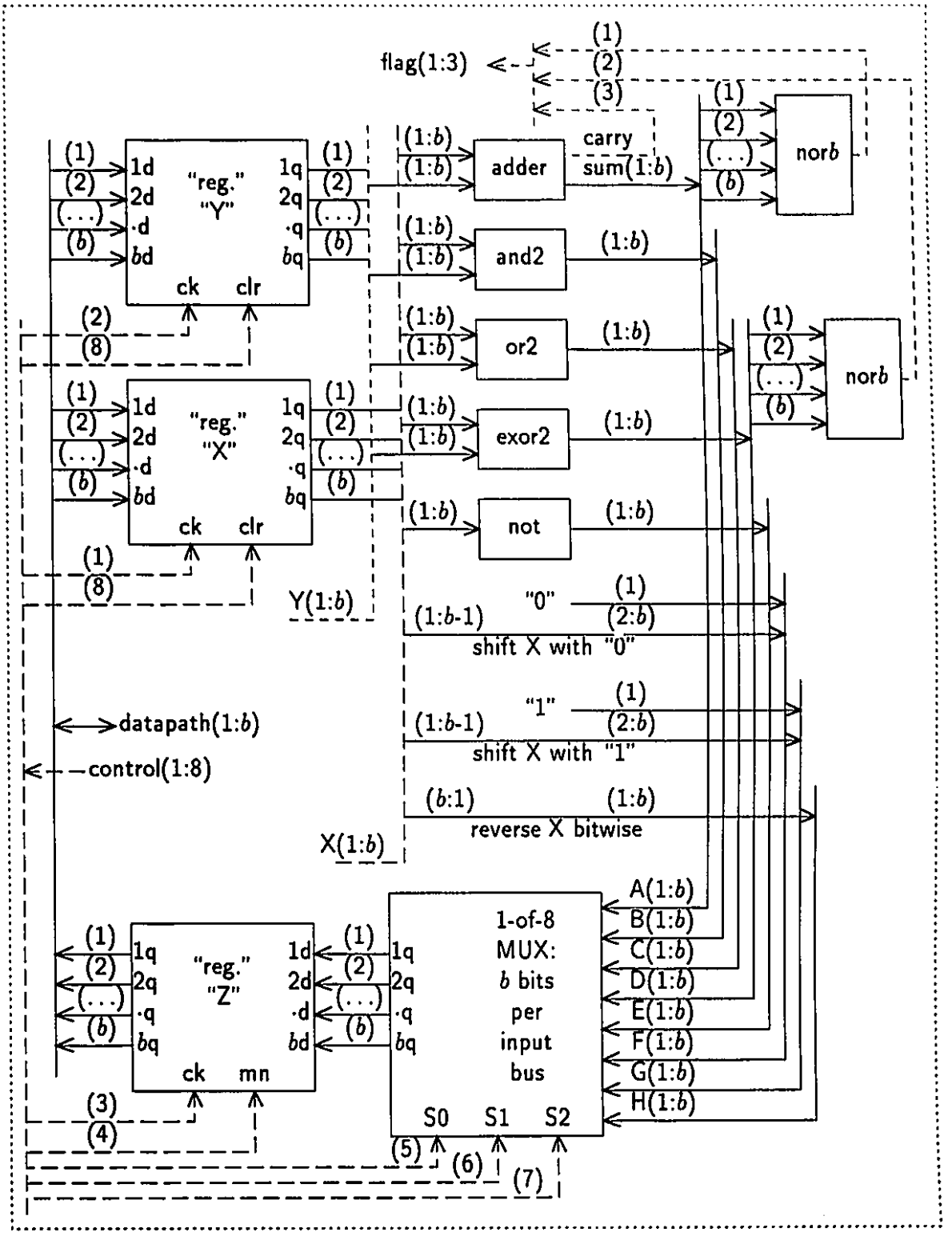

Fig. 6.11 "alu" (b-bit arithmetic and logic unit)

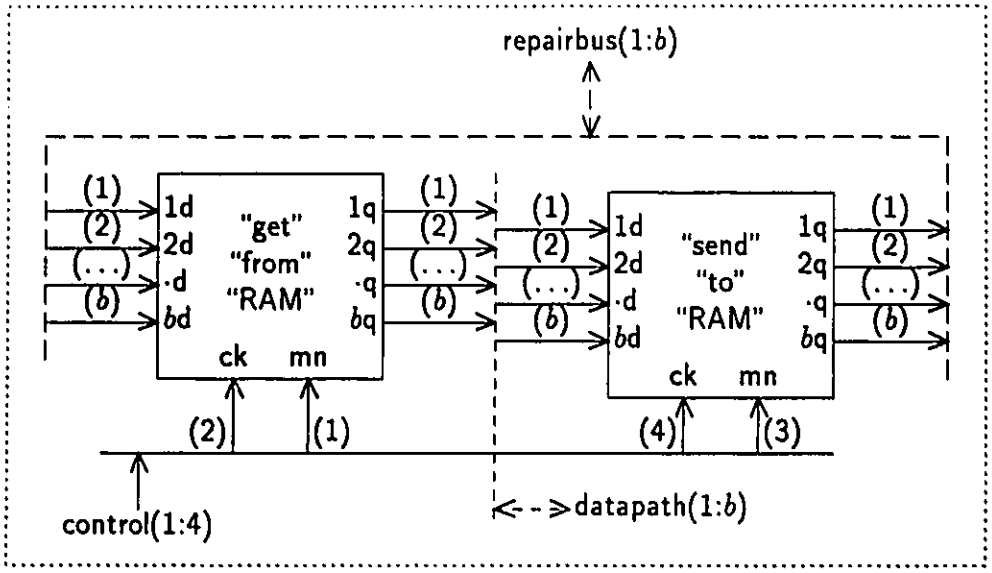

Fig. 6.12 "busPort"

which can be sent directly to the "busPort". We have these commonly used constants available in this type of read-only register arrangement, because the alternative would be to store them in the Rom, where they would likely be repeated over and over again, occupying a lot of expensive space in the Rom. One possible choice for the constants is 0001, 0000, 1111, 0101, 1010.

6.2.1.19 "readWriteRegs"

This cell contains 4 read/write registers which can be accessed by "alu" and by the "busPort", in order to temporarily store variables or test-data.

6.2.1.20 "busPort" (Fig. 6.12)

This cell provides an interface between the DR-Bus and the "Data Path". If the "busPort" did not exist, then "readWriteRegs", "alu" and "readOnlyRegs" would, in the first place, have to be able to drive the long DR-Bus lines when writing to the memory and be able to accurately sense the DR-Bus lines when reading from the memory. Such an arrangement would require "readWriteRegs",

"readOnlyRegs" and "alu" to contain significantly larger circuitry, that must be carefully calibrated in its analog behavior. In the current design, the "busPort" is the only place where such circuitry with carefully calibrated analog behavior exists, except for the write drivers and sense amplifiers of the memory array itself. In the second place (if "busPort" did not exist), "readWriteRegs", "readOnlyRegs" and "alu" would each have to be connected to two busses (instead of only one): to the "Data Path" to communicate among themselves, and to the DR-Bus to communicate with the memory array.

6.2.1.21 "clkGen2"

This cell generates 3 control signals, namely R(b+1:b+3), which form part of the DR-Bus, that are properly synchronized with each other, from the two bits of CTL(1:2) and the CLOCKPIN.

6.2.1.22 "instrDecod"

This cell decodes the instructions stored in the Rom, INSTR(1:7), to provide 43 control signals, namely: AL(1:8), RO(1:5), RW(1:8), BP(1:4), CG(1:2), TC(1:6), PC(1:6), IRC(1:4). Some of these control signals are also synchronized with respect to the clock signal CLKP. The decoding can be accomplished by using ordinary combinational logic, or by using programmable logic arrays.

6.2.1.23 "instrRegCntl" (Fig. 6.13, 6.14, 6.15, 6.16)

This cell is the most complex component of the "repairingController". It incorporates an instruction register for the current instruction (most recently fetched from the Rom), together with a finite-state machine control circuit which decides whether to transmit the currently stored instruction on to the instruction decoder ("instrDecod"), or to transmit some other instruction generated by the finite-state machine. This cell is also the destination of the START signal which specifies when the self-test-and-repair mode should begin, and the source of the

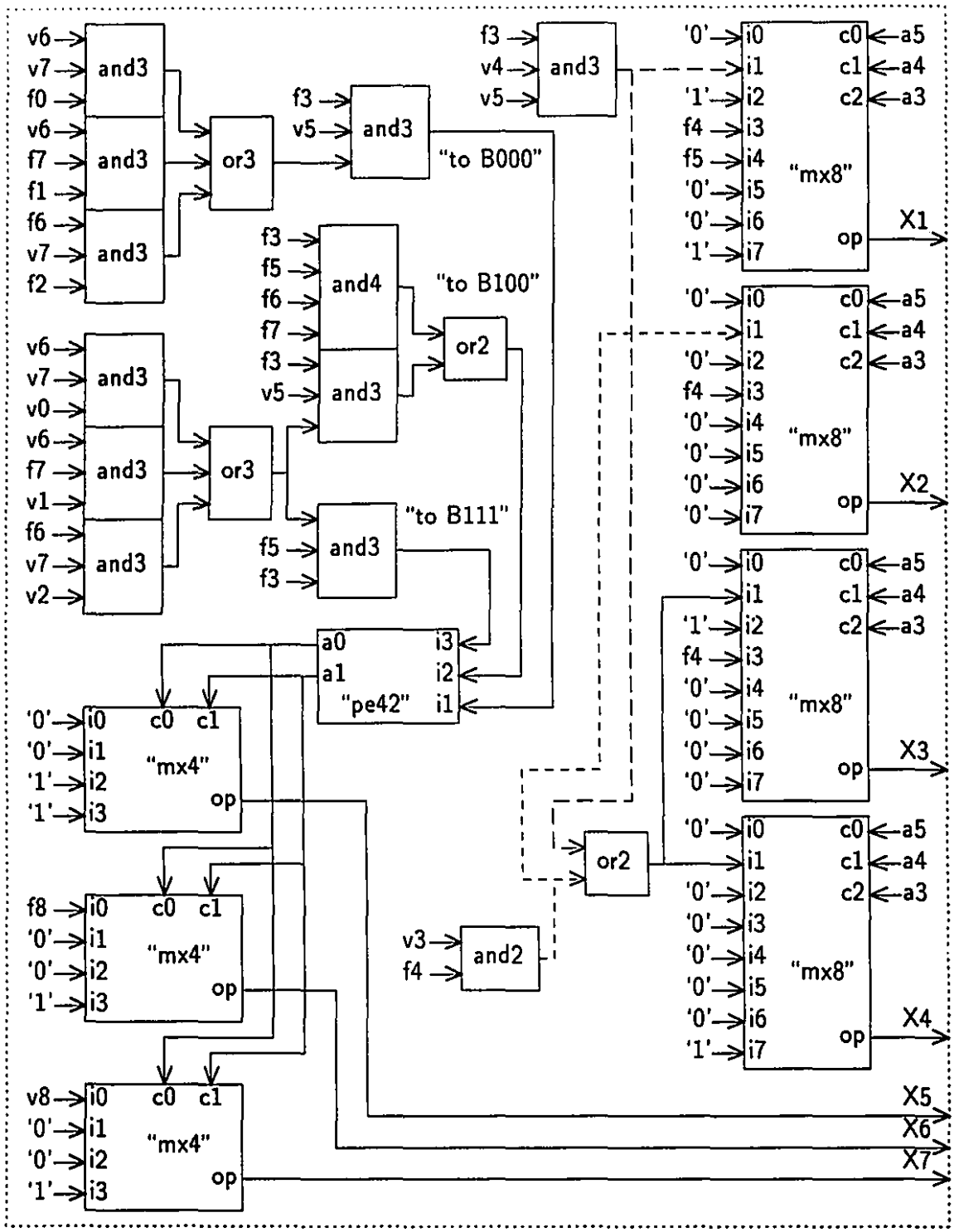

Fig. 6.13 "instrRegCntl" (part 1)

STOP(1:2) signals which signify when the self-test mode has ended and whether

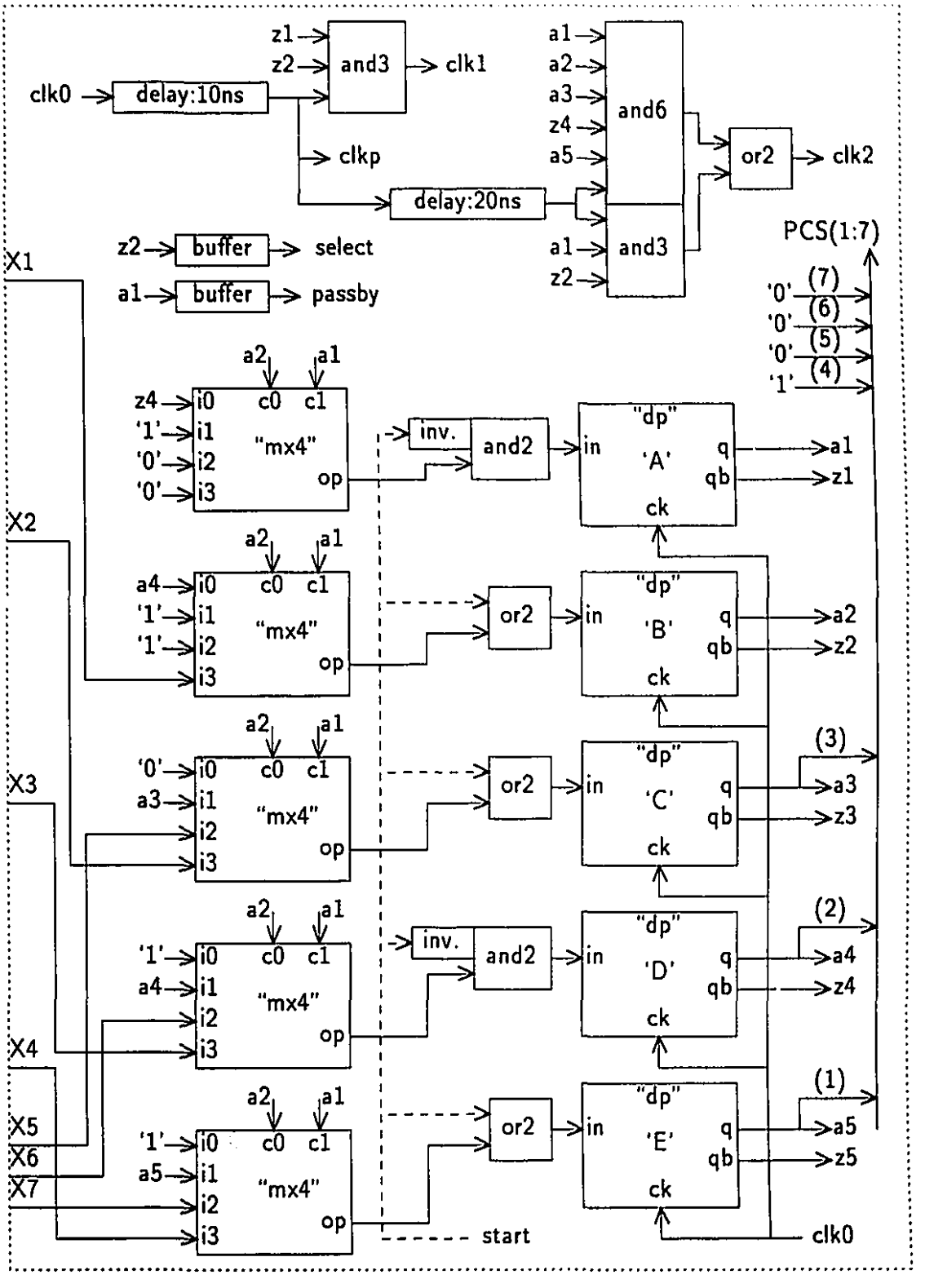

Fig. 6.14 "instrRegCntl" (part 2)

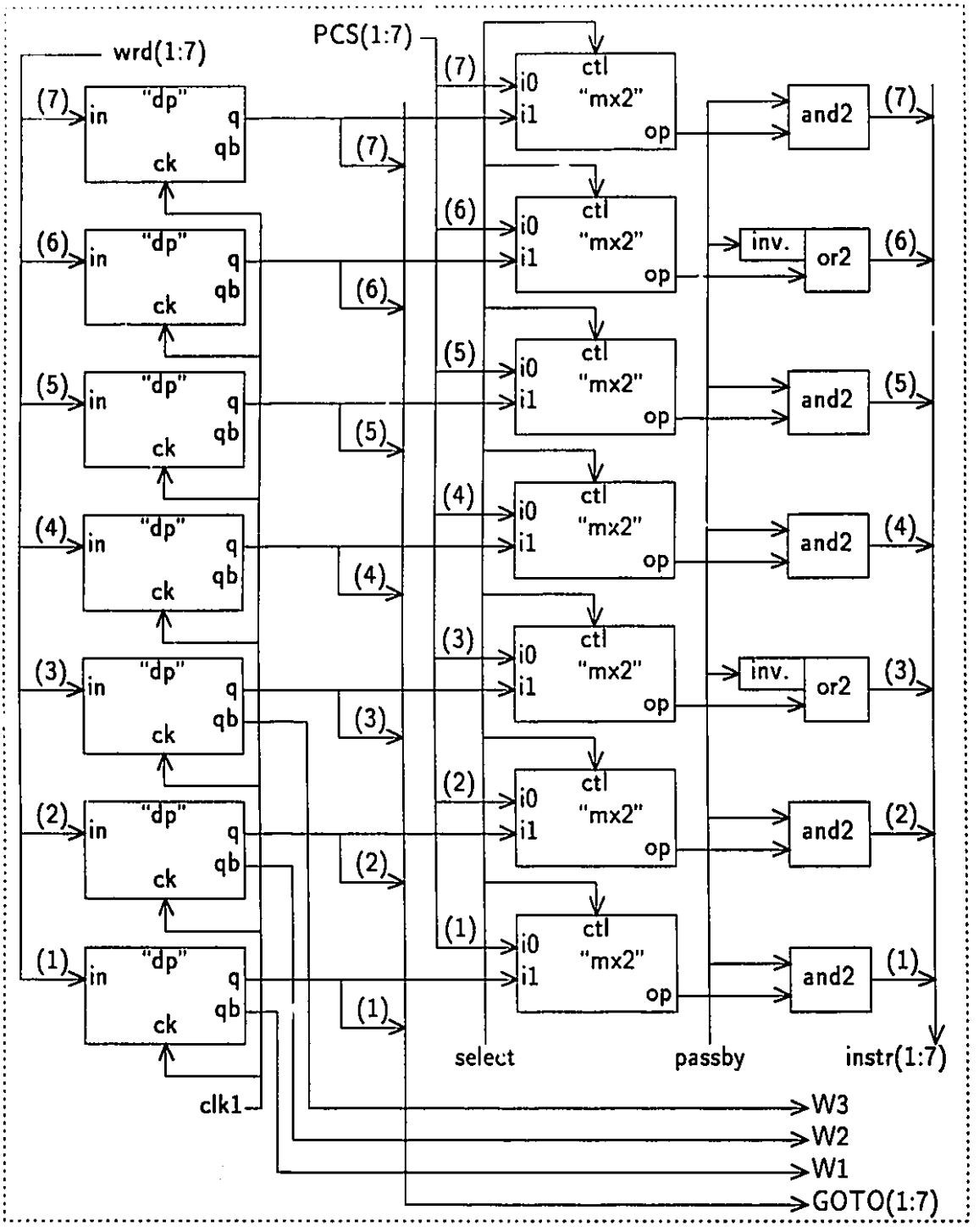

Fig. 6.15 "instrRegCntl" (part 3)

the self-repair procedure was successful or not.

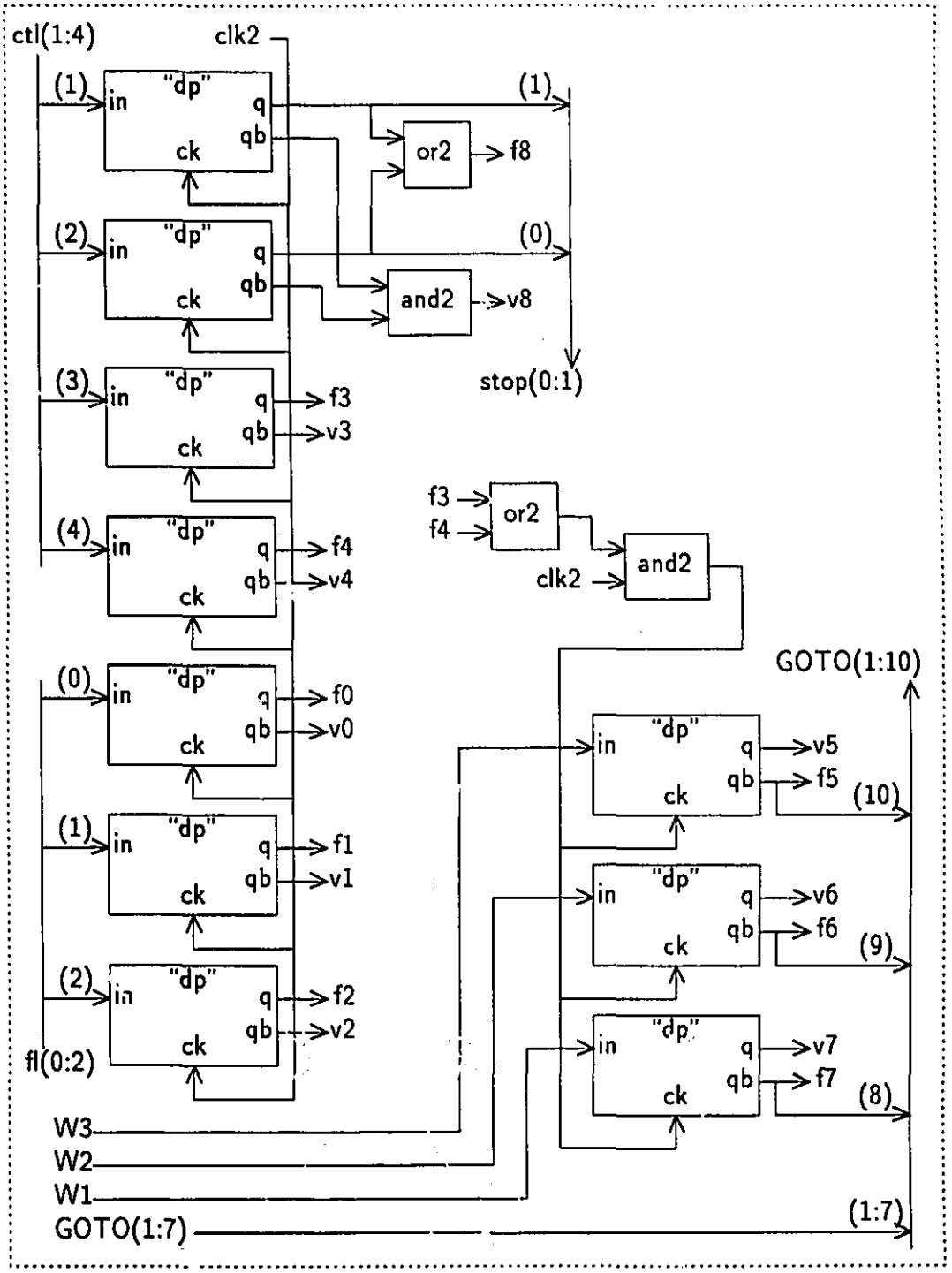

Fig. 6.16 "instrRegCntl" (part 4)

Approximately one half of the cell, as shown in Fig. 6.13 and 6.14, comprises a

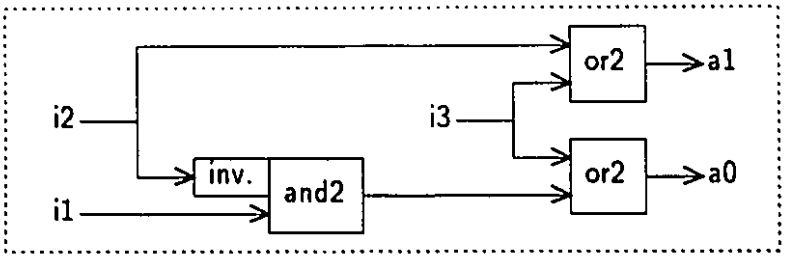

Fig. 6.17 "pe42" (priority encoder)

five-bit finite-state machine (FSM). The five-bit state is stored in the five D-type flip-flops (Fig. 6.14) labelled 'A', 'B', 'C', 'D', 'E' and which generate the signals: *a1, a2, a3, a4, a5* and their inverted values: *z1, z2, z3, z4, z5*. The three lower bits, namely: *a3, a4, a5*, are combined with four constant valued bits, namely: '0', '0', '0' and '1', to form a seven-bit *instruction* which is sent via bus PCS(1:7) to another part of the cell, in Fig. 6.15. This generated instruction, carried by PCS(1:7), is used to control the operation of the Program Counter with Stack (the cell "pCstack", Fig. 6.18).

Inputs			Outputs	
I1	I2	I3	A1	A0
0	0	0	0	0
1	0	0	0	1
x	1	0	1	0
x	x	1	1	1

Table 6.1 Priority Encoder truth-table

6.2.1.24 "pe42" (Fig. 6.17)

This cell embodies a 1-of-3 Priority Encoder, which functions as shown in Table 6.1.

Inputs		Output
C1	C0	OP
0	0	I0
0	1	I1
1	0	I2
1	1	I3

Table 6.2 1-of-4 Multiplexer truth-table

6.2.1.25 "mx4"

This cell embodies a 1-of-4 multiplexer, which functions as shown in Table 6.2.

Inputs			Output
C2	C1	C0	OP
0	0	0	I0
0	0	1	I1
0	1	0	I2
0	1	1	I3
1	0	0	I4
1	0	1	I5
1	1	0	I6
1	1	1	I7

Table 6.3 1-of-8 Multiplexer truth-table

6.2.1.26 "mx8"

This cell embodies a 1-of-8 multiplexer, which functions as shown in Table 6.3.

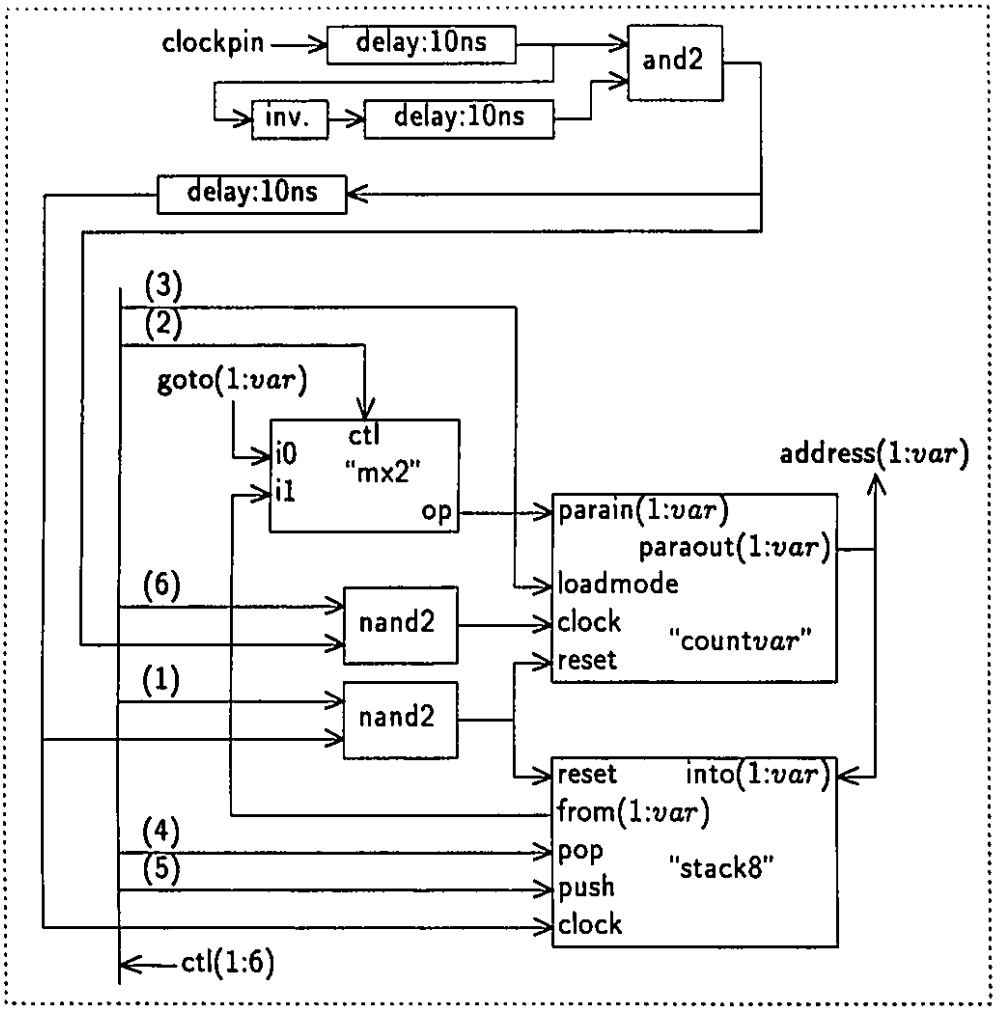

Fig. 6.18 "pCstack" (program counter stack)

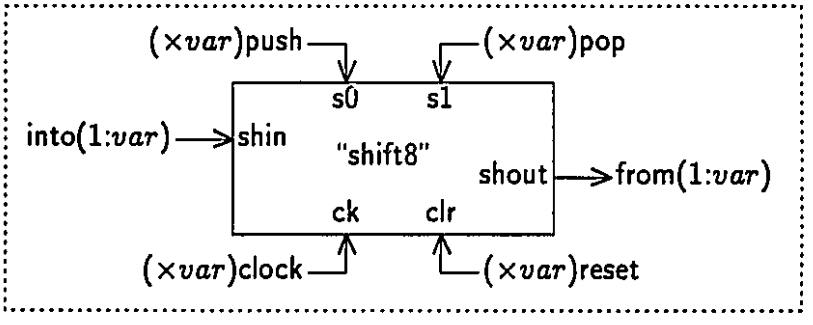

Fig. 6.19 "stack8" (8-item stack)

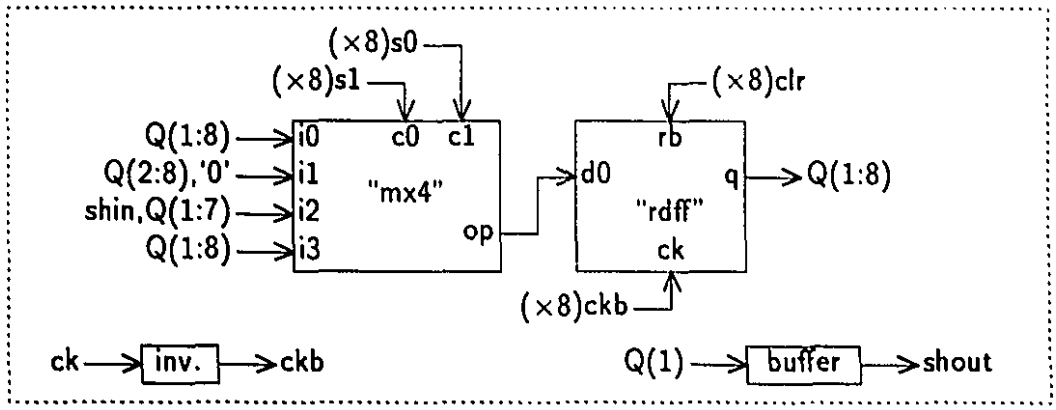

Fig. 6.20 "shift8" (8-bit shift register)

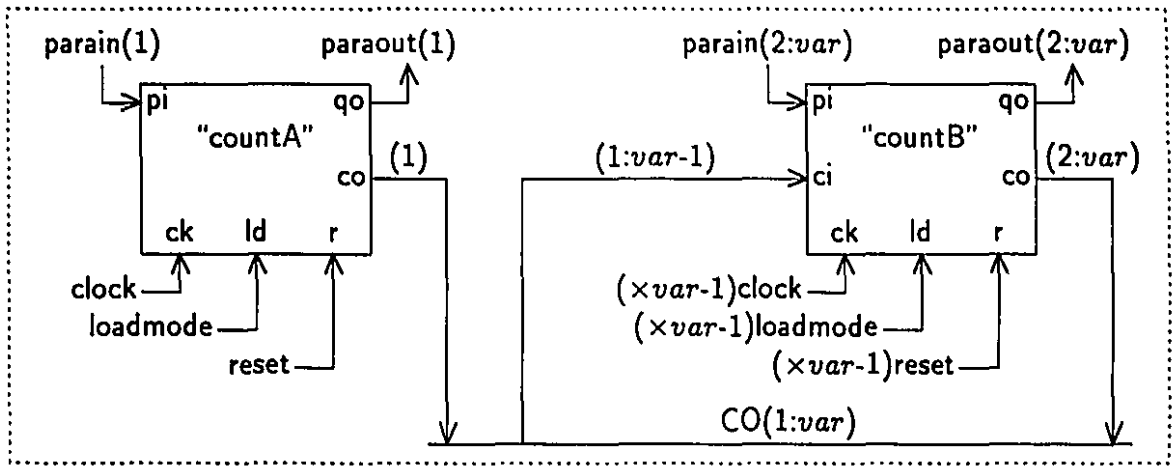

Fig. 6.21 "count*var*" (*var*-bit counter)

6.2.1.27 "pCstack" (Fig. 6.18)

This cell is a Program Counter with Stack. The stack (see Fig. 6.19) has the capacity to store upto eight *var*-bit addresses. The stack is made up of *var* bidirectional shift registers, where each bit of an address is stored in a different shift register (see Fig. 6.20). The length of the shift registers determines the depth of the stack, which in this case is 8. The *var*-bit counter is unidirectional (i.e. it only increments its present value, and never decrements) which can also be loaded with an arbitrary *var*-bit value, or can be reset to the all zeroes starting value (see

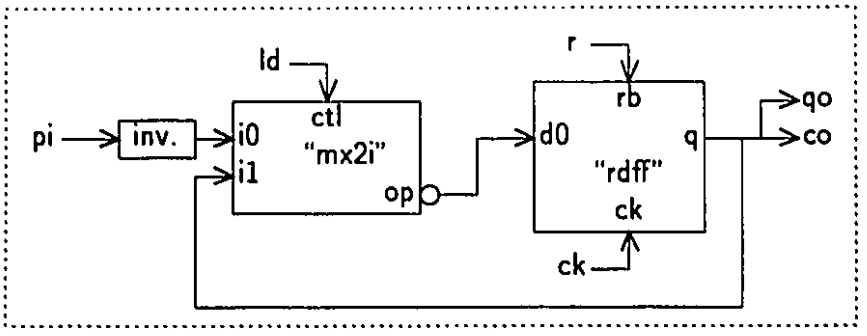

Fig. 6.22 "countA"

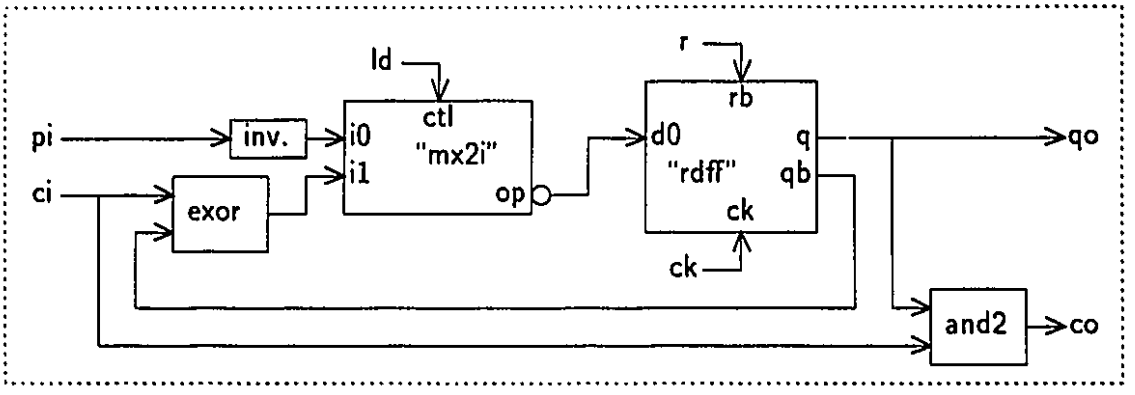

Fig. 6.23 "countB"

Fig. 6.21, 6.22, 6.23).

6.2.2 "diagnosisOnly" (Fig. 6.24)

This is the top-level cell of the proposed design which incorporates the ability of the hardware to do only the diagnosis of faults entirely by itself, and the repair is performed with help from outside circuitry. The signal START specifies when a self-diagnosis session should take place, with the circuit operating in normal-mode at other times. The signals STOP(1:2) specify whether the externally controlled repair successfully produced an operational circuit, or if there are some remaining unrepairable faults.

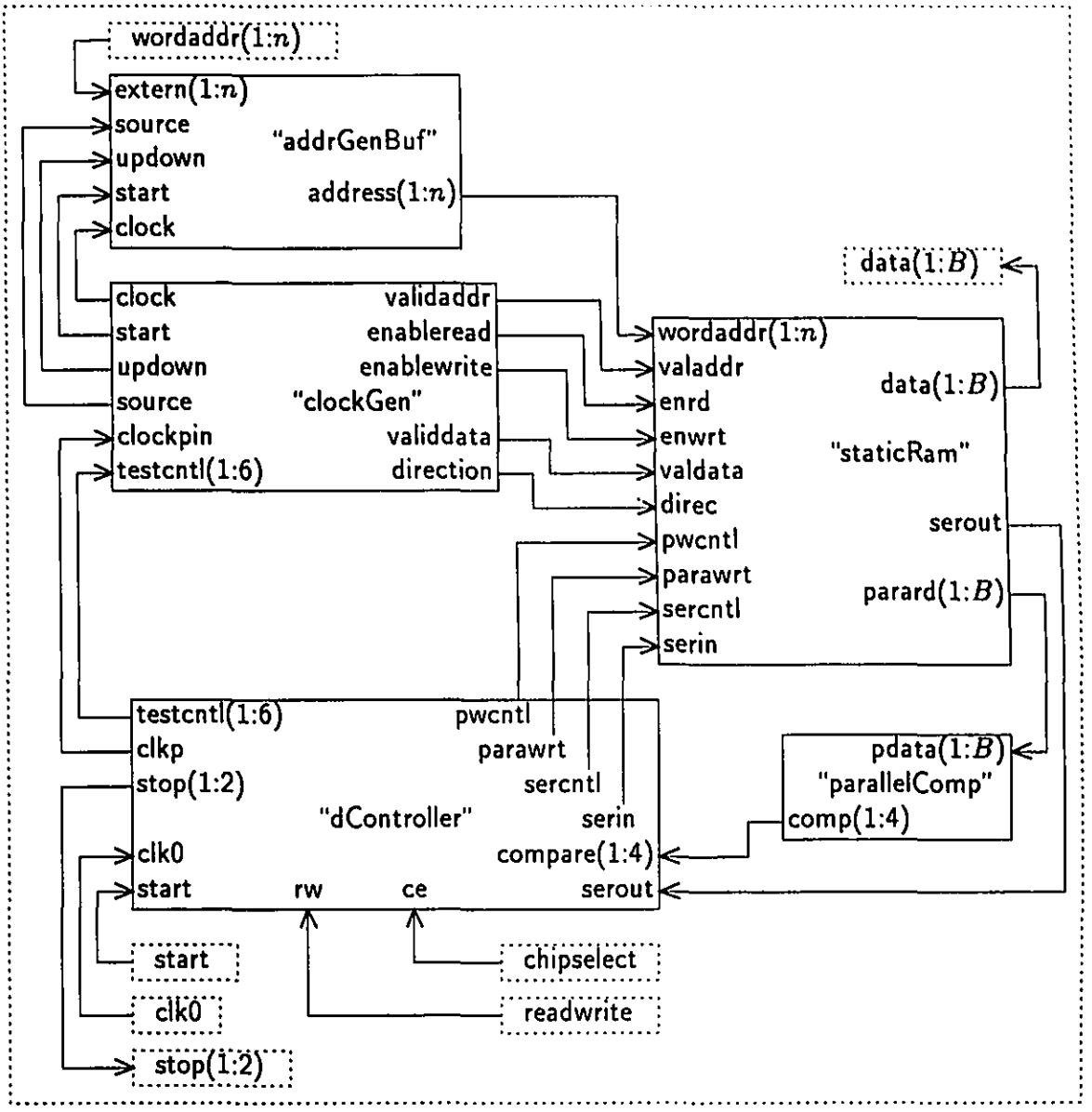

Fig. 6.24 "diagnosisOnly"

6.2.2.1 "clockGen"

This cell generates 9 control signals, that are properly synchronized with each other, from the six bits of TESTCONTROL(1:6) and the CLOCKPIN.

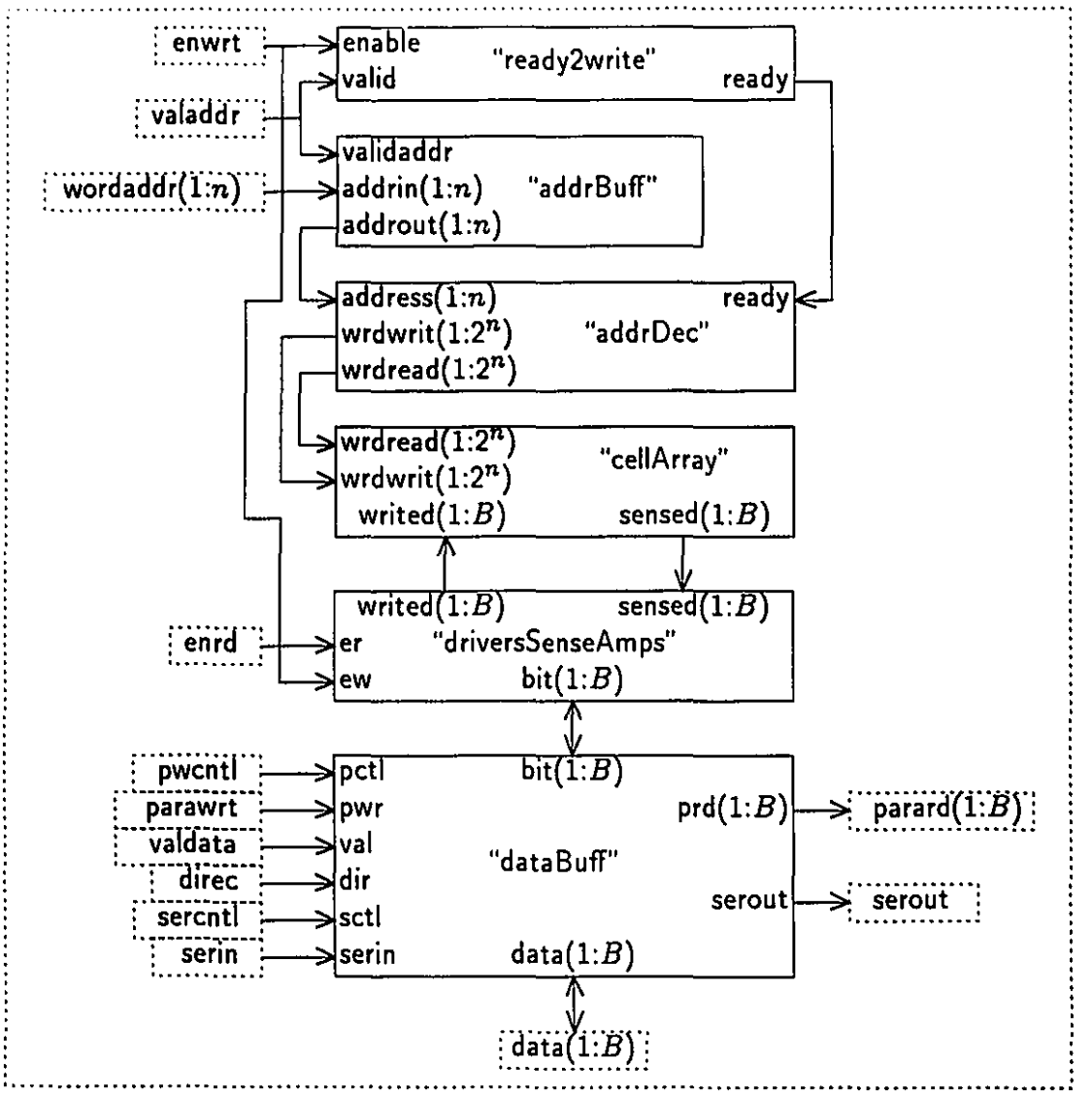

Fig. 6.25 "staticRam"

6.2.2.2 *"parallelComp"*

This cell reads all the data lines in parallel, and determines whether all the even lines simultaneously carry the value zero or the value one, and whether all the odd lines simultaneously carry the value zero or the value one.

6.2.2.3 *"staticRam" (Fig. 6.25)*

This cell embodies a typical static RAM which contains some spare lines which

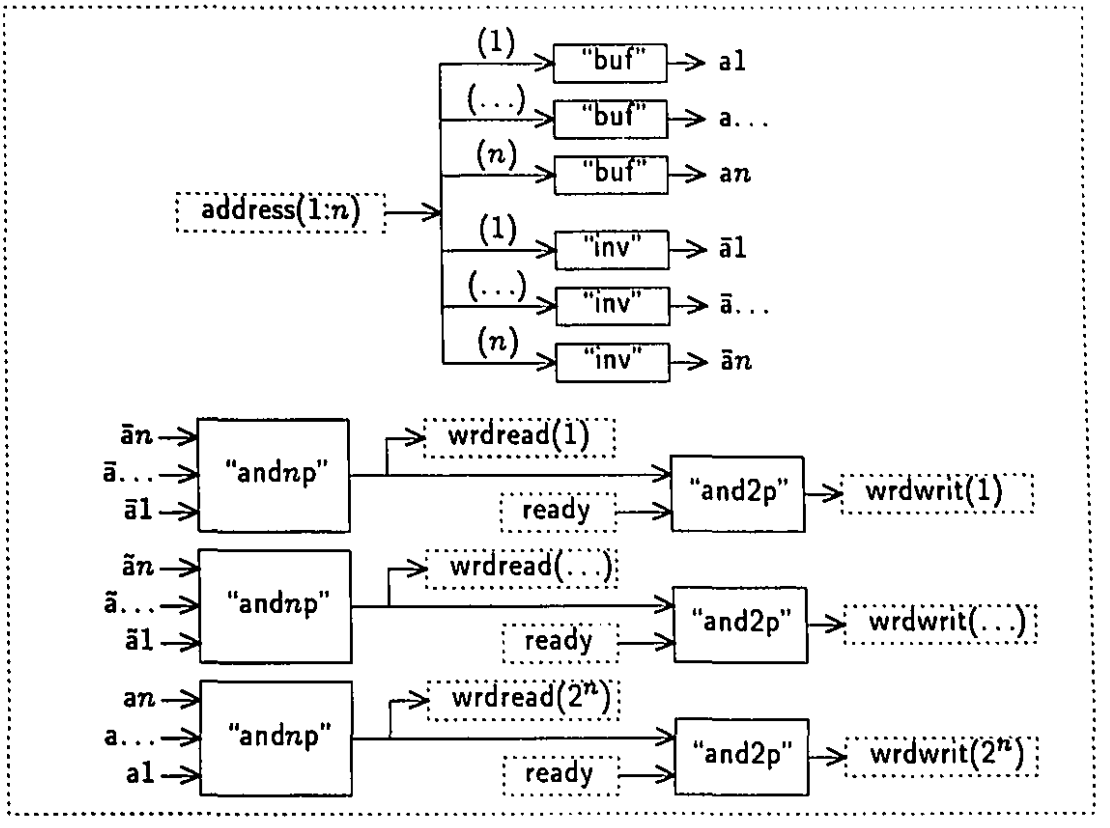

Fig. 6.26 "addrDec"

can only be activated by off-chip means. As a result, from the point of view of the self-diagnosis circuitry, the redundancy is completely transparent, and is therefore not displayed in Fig. 6.25.

6.2.2.4 "addrDec" (Fig. 6.26)

This cell implements a typical address decoder.

6.2.2.5 "cellArray" (Fig. 6.27)

As before, the cell array uses non-standard bit lines: one for writing, and one for reading. In reality, a pair of complementary bit lines would be used for both reading and writing. Note that the area occupied by both arrangements is the same, because both use 2 lines per column.

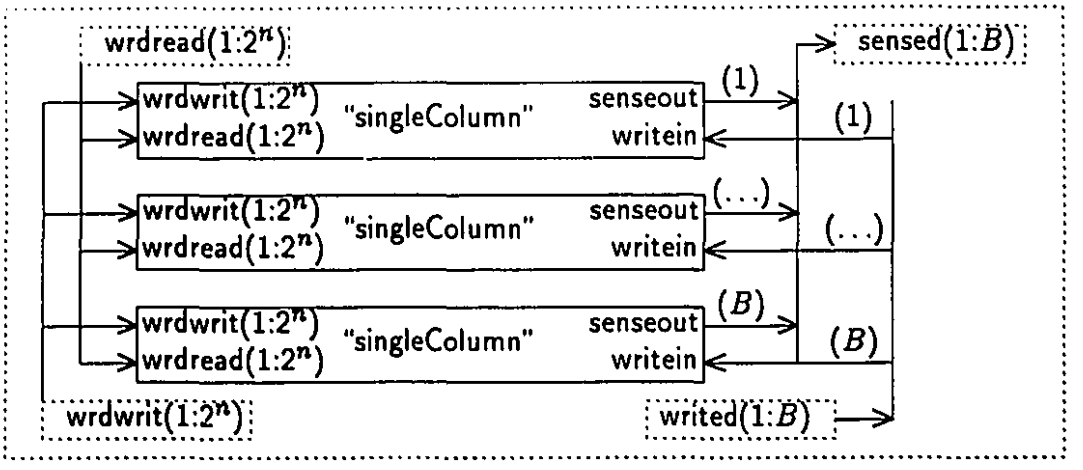

Fig. 6.27 "cellArray"

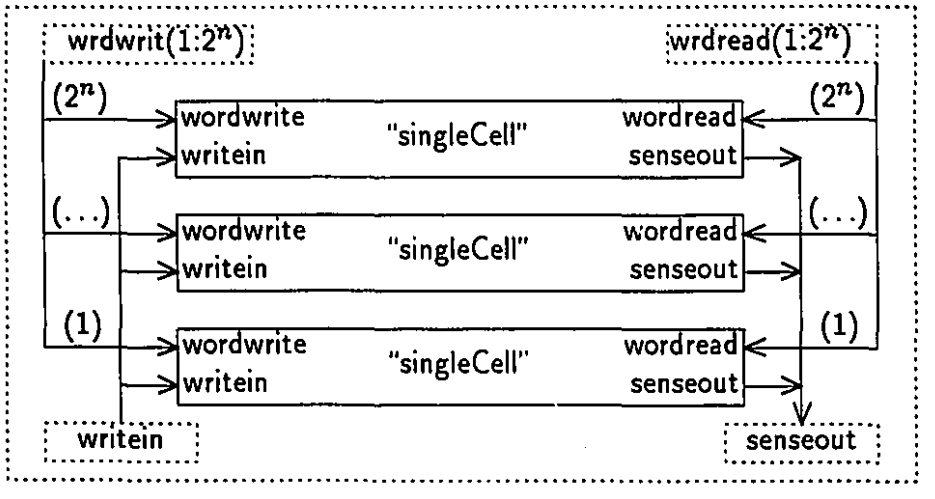

Fig. 6.28 "singleColumn"

The signal WRITEIN is a non-standard item required by the CAD software.

6.2.2.6 "singleColumn" (Fig. 6.28)

6.2.2.7 "driversSenseAmps" (Fig. 6.29)

The figure shows a read/write circuit equipped with flip-flops.

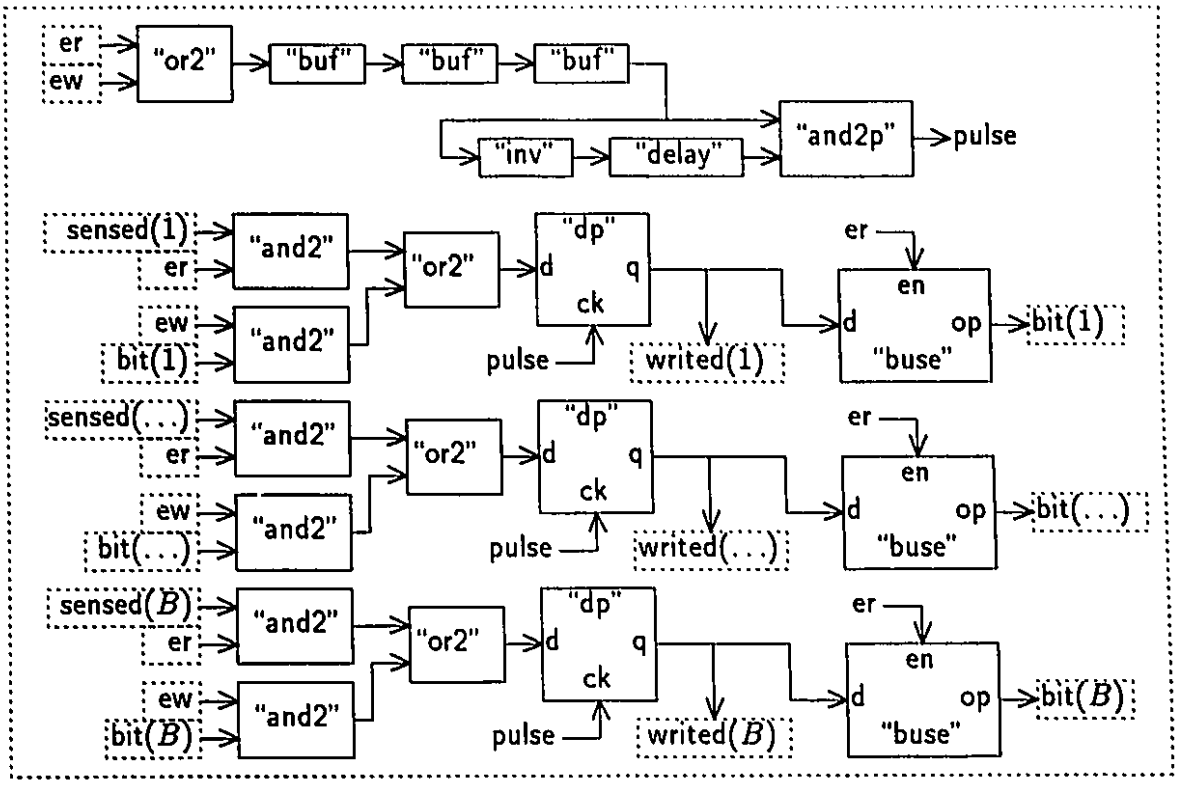

Fig. 6.29 "driversSenseAmps"

6.2.2.8 *"dataBuff" (Fig. 6.30)*

This cell shows a bi-directional data buffer that has been augmented with circuits to allow for serial input and output, and parallel writing and parallel reading of an entire B-bit word.

6.2.2.9 *"buffer" (Fig. 6.31)*

This cell is a flip-flop with multiplexed inputs and dual outputs, as required by the cell "dataBuff".

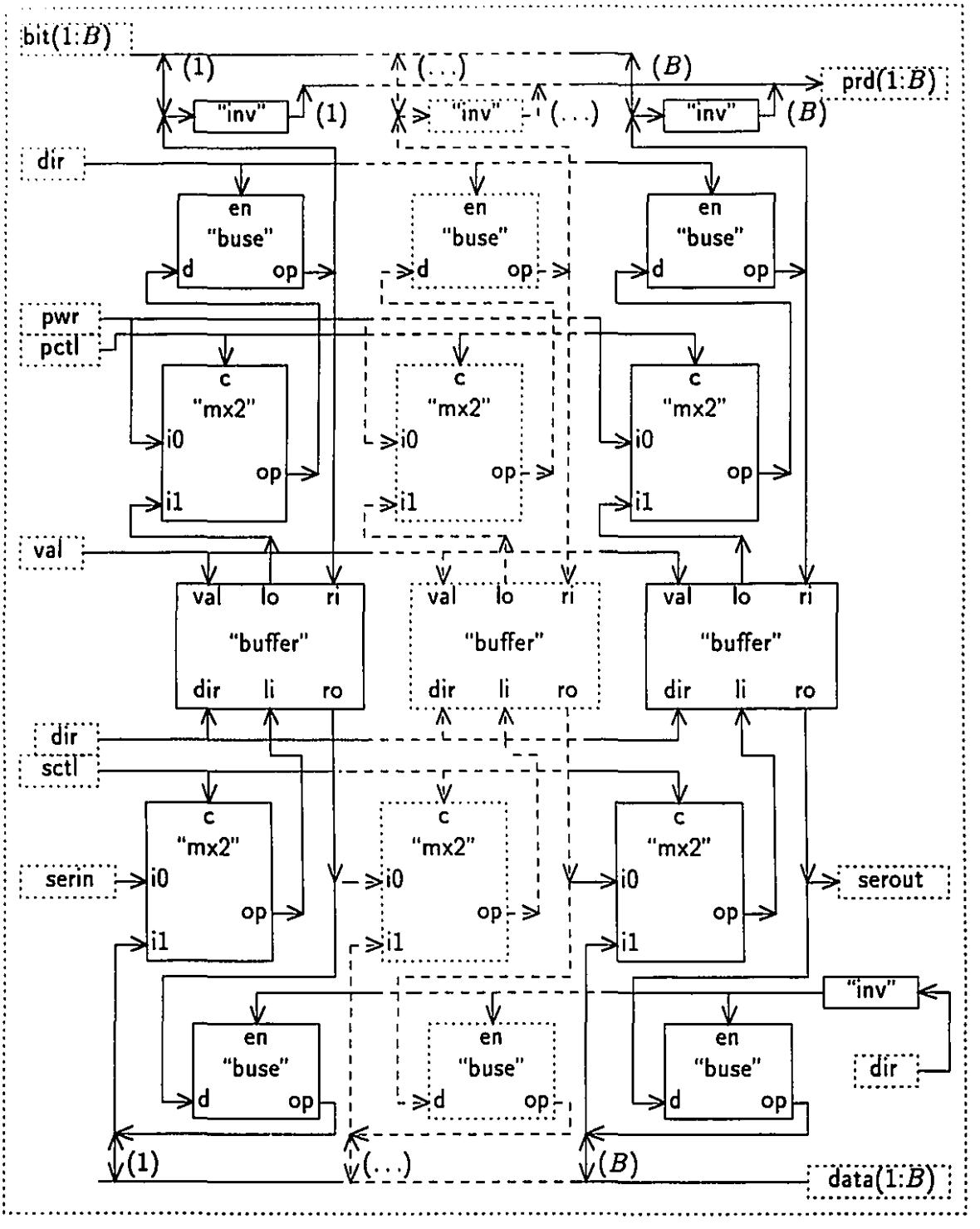

Fig. 6.30 "dataBuff"

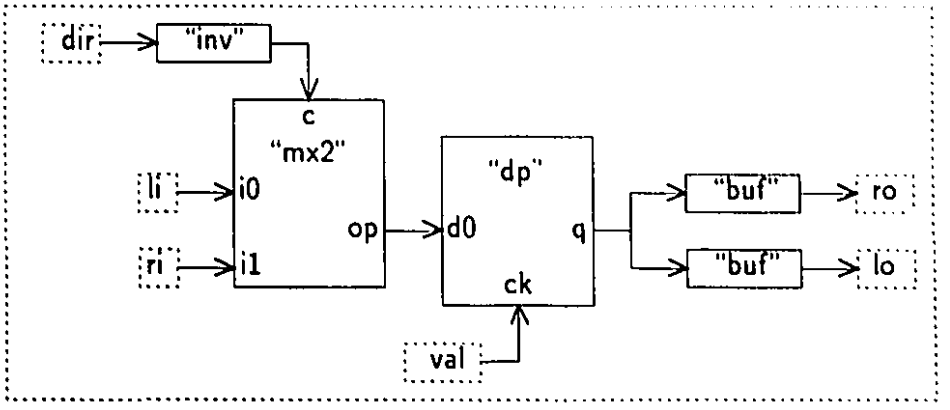

Fig. 6.31 "buffer"

6.3 Design costs and tradeoffs

6.3.1 Area Overhead

Area overheads were estimated, based on the circuits described in this chapter, for a BISD-with-self-repair version of a Static RAM with a single metal layer and a single polysilicon layer, using 6-transistor CMOS cells. Array sizes from 2k (2048 bits) upto 32M (33554432 bits) were used with several different internal arrangements for each size — the constraint WB=(one of the array sizes) was imposed, but otherwise the values of W, B, M, and other parameters, were varied widely. The average overheads are tabulated in Table 6.4, and graphed in Figure 6.32. By inspection of the graph, the overhead appears to be inversely proportional to the size. A statistical power regression analysis shows that the relationship is actually

$$overhead = 221.27 \times size^{-0.95832}$$

with a correlation coefficient of $r = -0.99956$, which is very close to the ideal value of -1.

Additional estimates of area overhead were calculated, based on a 4-transistor CMOS Static RAM with two metal layers and two polysilicon layers, in order to

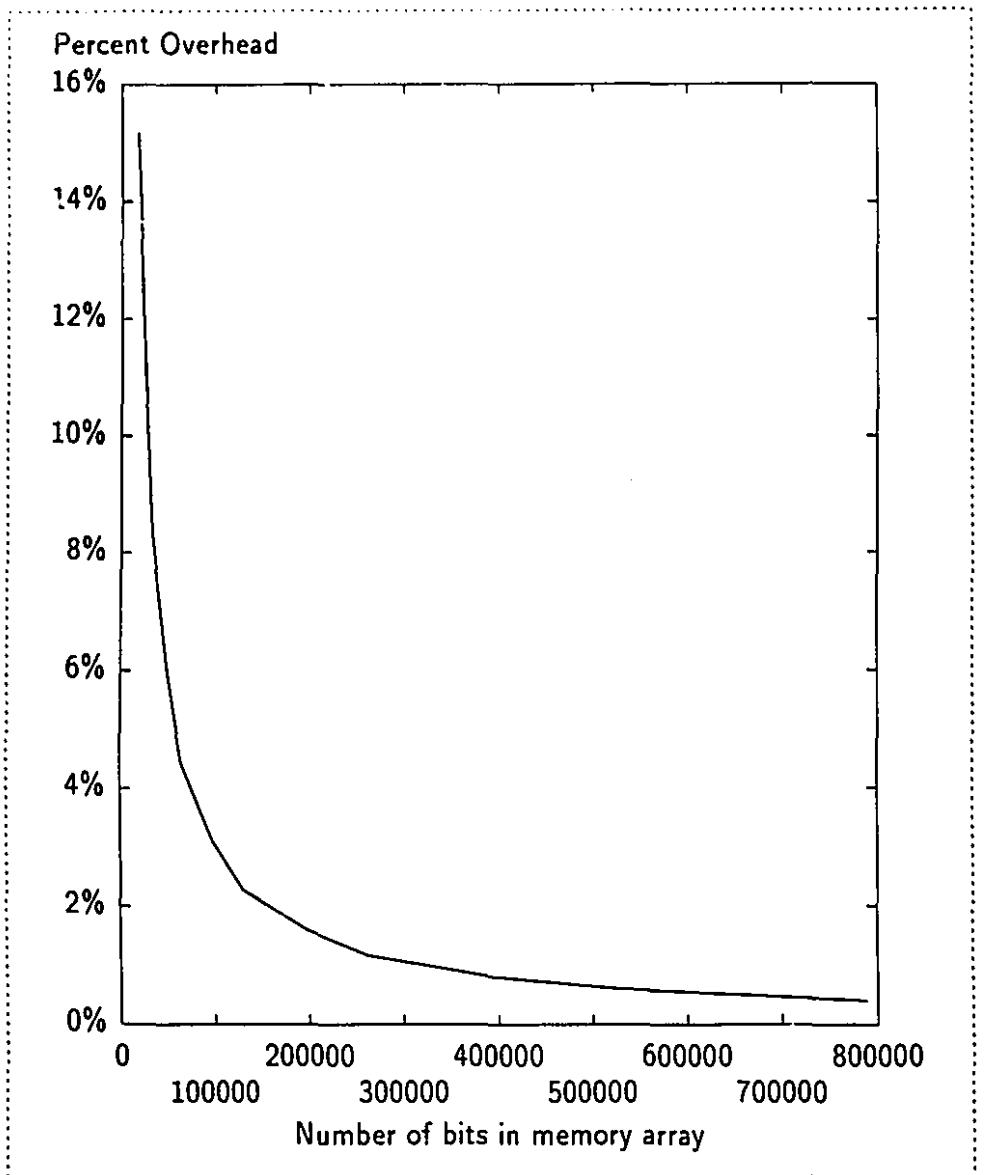

Fig. 6.32 Average area overheads for 6-T arrays

properly compare the BISD-with-self-repair overheads with pure-BIST overheads, using overhead values reported in [Dekker *et al.* 89]. Dekker *et al.* used such a 4-T SRAM in their implementation. The overheads are compared in Table 6.5, and it can be seen that the BISD scheme requires approximately 6 to 8 times more surface area than the march-test-based pure BIST scheme.

Array Size	Overhead
2k	49.5%
4k	36.9%
8k	25.5%
16k	15.1%
32k	8.38%
64k	4.43%
128k	2.28%
256k	1.16%
512k	0.586%
1M	0.294%
2M	0.148%
4M	0.0741%
8M	0.0371%
16M	0.0194%
32M	0.0103%

Table 6.4 BISD overhead for 6-T arrays

Array Size	BISD overhead	Dekker's overhead
2k	79.2%	15%
4k	59.0%	10%
8k	40.8%	5%
16k	24.2%	3%
32k	13.4%	2%
64k	7.09%	1%

Table 6.5 Overhead comparison for 4-T arrays

6.3.2 *Repair Choices*

When a fault is located, we need to store the location in a *fault map*, and only when the fault map is complete, do we apply a repair algorithm, which seeks to assign the spare rows and spare columns in as nearly an optimal allocation as

possible. A fault-free module of the memory cell array is used to store a complete fault map of another module which is being diagnosed. There is a high probability there will always be at least one fault-free module available to store the fault maps of other modules, and this implies that no extra area overhead within the D&R-Unit is needed to store fault maps.

There are two possible ways to apply self-repair algorithms: (1) by programming an appropriate heuristic algorithm, such as branch-and-bound or best-first-search, into the ROM of the D&R-Unit, or (2) by using an electronic neural network implementation of a gradient-descent algorithm [Mazumder and Yih 90]. The neural network technique has the advantage of significantly higher percentages of successful repair allocations, in comparison with the programmed heuristic technique (i.e. the neural network will find repair-plans in most borderline cases where the programmed heuristic technique gives-up and declares the memory to be unrepairable). However, the neural network technique may require more area overhead than reported in Table 6.4. In addition, the implementation of a neural network may require significant changes to the fabrication processes being used.

The effect on fabrication yield of the BISD circuit and possible self-repair techniques will not be calculated here; instead, the reader is referred to the literature [Mazumder and Yih 90], [Stapper *et al.* 80], [Stapper 86].

6.3.3 Hierarchical Redundancy

A hierarchical self-diagnosis scheme is practical, when global redundancy is used for repair. Use the self-diagnosis with external repair scheme, but send all of the diagnosis information to another chip in the computer. This other chip will send repair orders to the globally redundant spare lines controlled by soft fuses that are really Shadow flip-flops (i.e. each flip-flop is paired with an EEPROM cell). Global redundancy provides a better yield improvement, and can be laid out with shorter wires than local redundancy, but cannot provide "free" access to the memory cell

array. In the case of very fast SRAMs, global redundancy is impossible because of the requirement for "hierarchical word decoding" (HWD) which reduces power consumption and significantly improves access time [Murakami *et al.* 91]. In the currently popular "divided word line" architecture, the global word decoder drives the global word line. The local word decoder placed in every "block" receives the global word line and some block selection signals, and selects one local word line. As SRAMs become larger, they will be divided into more "blocks" to keep the array current small. This trend results in a larger stray capacitance on the global word line since the number of local word decoders also increases. Therefore, the charging/discharging current in the word-decoding circuit significantly increases even in the "divided word line" architecture.

The "hierarchical word decoding" architecture is proposed as a further method to reduce the power consumption and increase the speed of the word-decoding circuitry. In the "hierarchical word decoding" architecture, the word-decoding circuit is divided into (at least) three stages, which consist of a global word decoder, a subglobal word decoder, and a local word decoder. The subglobal word decoders are inserted as buffering stages to efficiently distribute the stray capacitance on the word-decoding path. In this type of architecture there can be more stages along the word-decoding path, with the additional buffering stages named subsubglobal word decoders and/or superlocal word decoders. The "divided word line" architecture is obviously equivalent to a two-stage "hierarchical word decoding" architecture. For every memory capacity there exists an optimum number of stages, which can be calculated by determining the total capacitance on the decoding path. For example, for CMOS SRAMs operating on 3-V power supplies, we get the following results: for 256 Kb and smaller SRAMs, two stages are optimal, for 1Mb SRAMs both two and three stages are equally good, for 4Mb and 16Mb SRAMs three stages are optimal, for 64Mb SRAMs both three and four stages are equally good, and for 256Mb and larger SRAMs four stages are optimal.

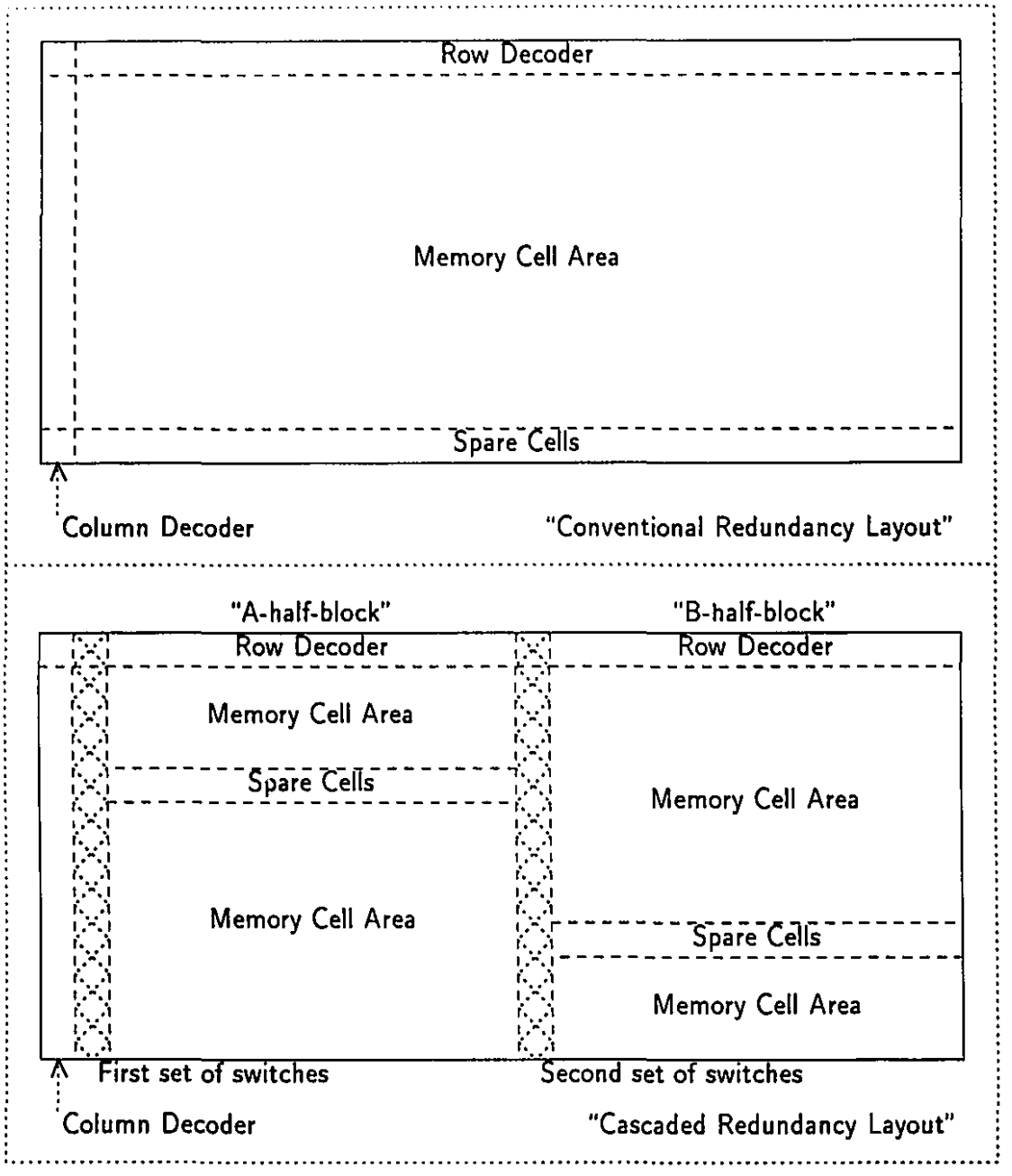

Fig. 6.33 "Two Redundancy Techniques Compared"

6.3.4 Segmentation of Spares

One of the major concerns with using spare columns to repair faulty cells is the large number of good cells that are wasted. The paper [Mori *et al.* 91] describes a 64Mb DRAM with a "cascaded redundancy" scheme (see Fig. 6.33). Each

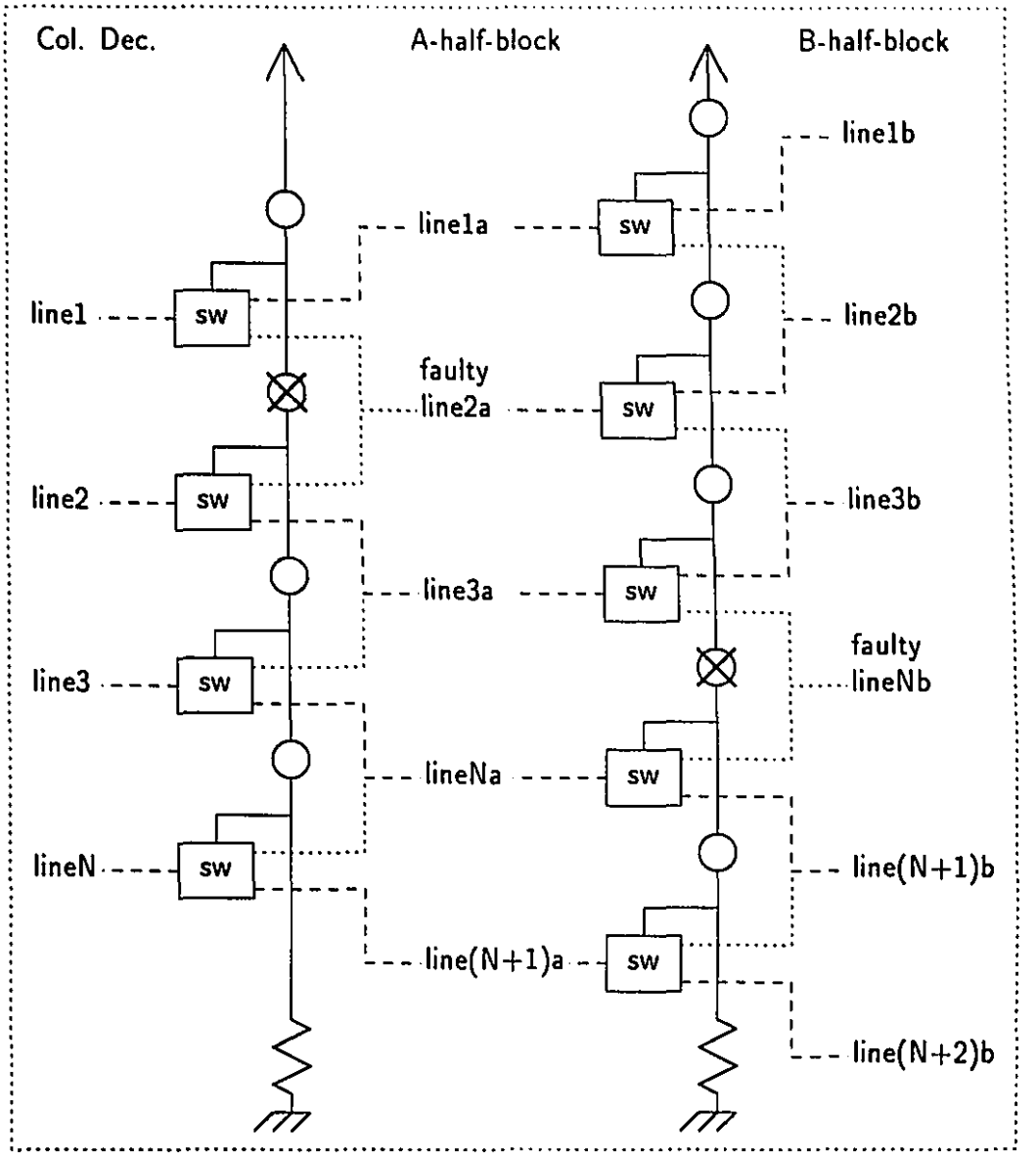

Fig. 6.34 "Cascaded Redundancy"

"block" contains N column decoders, $N+1$ column select lines in the "A-half-block" with a first set of switches, and $N+2$ column select lines in the "B-half-block" with a second set of switches (see Fig. 6.34). The first set of switches, which are controlled by fuse links, selects paths between the column decoders and the "A-half-block". Similarly, the second set of switches selects paths between the "A-half-block" and the "B-half-block". Each of the switches is essentially a

1-of-2 multiplexer, with the control signal coming from a line seeded with fuses. If there are no faulty lines to avoid, then the entire control line is at V_{DD}, and all of the switches connect their input lines with their *upper* output lines. If there is a faulty line to avoid, then one of the fuses is blown, causing a break in the control line; the portion of the control line above the blown fuse is still at V_{DD}, and thus the "upper" switches still connect their input lines with their *upper* output lines, but the portion of the control line below the blown fuse is now at ground, and thus the "lower" switches now connect their input lines with their *lower* output lines. Each half-block is only able to tolerate one faulty column line, even though the "B-half-block" contains two extra column lines—because one of these extra column lines is rendered inaccessible by the single spare column line in the "A-half-block". A higher replacement efficiency results from this "cascaded redundancy" scheme than from the conventional single spare line per block scheme, since the replacement of a faulty half-line in each half-block is done independently.

6.3.5 Redundancy Choices

The paper [Kikuda *et al.* 91] develops a "failure-related yield model" to forecast the yield of very large DRAM chips. The yield varies depending on the selection of two design parameters, namely: the number of "blocks" into which the memory cell array is divided, and the number of spare elements for each "block". The selection of these design parameters to maximize the yield is strongly dependent on the "defect density coefficient" determined by the manufacturing process. There are seven typical failure modes which can be caused by a single physical defect:

1. single memory cell failure
2. single word line failure
3. adjacent word line pair failure
4. single bit line failure
5. adjacent bit line pair failure

6. word line and bit line intersection failure

7. memory "block" fatal failure

Items 1, 2 and 4 require only a single spare line for repair, items 3, 5 and 6 require two spare lines for repair, and item 7 cannot be repaired with spare lines at all.

The most significant conclusion from the yield analysis based on the failure model described above, deals with the selection of the "block" size and the resultant number of spares per block, when a given constant number of spare bits is specified. For example, four 256Kb "blocks" with one spare row and one spare column, have the same number of spare bits as one 1Mb "block" with two spare rows and two spare columns. Considering only the single memory cell failures, the one spare row and one spare column redundancy scheme can replace up to 8 faulty bits per 1Mb array, while the two spare rows and two spare columns redundancy scheme can replace only 4 faulty bits per 1Mb plane. However, the analysis shows that the two spare rows and two spare columns redundancy scheme is much more effective at improving yield because of its much greater flexibility. Hence, for very large DRAMs, a redundancy scheme with at least two spare rows and at least two spare columns per "block" provides the greatest yield improvement for a given amount of spare bits.

In the context of "soft switching" repair, when the spare lines are spread throughout the memory (i.e. local redundancy), then the extra area overhead needed for a complete self-diagnosis circuit is less than when the spare lines are grouped in only one part of the memory (i.e. global redundancy). Although the design with global redundancy has a higher area overhead than with local redundancy, it is able to repair a greater variety of faults because of the greater inherent flexibility of global over local redundancy. The difference in area requirements is explained below: All the possible versions of *repair* algorithms need essentially the same hardware (i.e., some kind of control unit, and a DR-Bus that

communicates with Programmable Modules), whereas all the possible versions of *diagnosis* algorithms can use a wide variety of extra hardware (i.e., there are many more hardware-design alternatives for implementing a diagnosis method than for a repair method). In the case of local redundancy, the repair hardware makes spare lines *spread throughout the RAM* accessible, and thus using the same DR-Bus and Programmable Modules, the non-spare lines spread throughout the RAM are also easily accessible for diagnosis purposes. In the case of global redundancy, the repair hardware makes the spare lines placed in *only one small part of the RAM* accessible, and therefore significant additions (e.g., increasing the length of the DR-Bus, and adding more lines to the Bus to cope with a more complex addressing situation) to the repair hardware are needed before the non-spare lines placed in the rest of the RAM are accessible for diagnosis purposes.

If we use both spare columns and spare rows, we need extra area to implement a more complex arrangement of Soft Fuses, and we need a more complex repair algorithm to make proper use of both types of spare lines. Since the *optimal* allocation of both types of spare is known to be an $\mathcal{NP}$-complete problem, an exact or heuristic algorithm for repair will need much temporary memory storage (e.g., a "home" module) to retain a fault map, which is required for any algorithm to make a reasonably good allocation of spares. All this added complexity may increase the area overhead to such a degree that any gains obtained by having fewer spare lines in total (i.e., since there would be spare lines in both directions, there would be greater flexibility in fault repair and thus fewer spares would be needed in total) may be outweighed by a repair scheme having more spare lines in total, but where all these spare lines are in only one direction.

In the design as depicted in the Figures, the DR-Unit uses a narrow DR-Bus with one set of Data lines, an Address Mode signal, and other Mode signals. We could have used a wider Bus with 2 sets of Data lines (where really only one set of lines would carry data, while the other would carry sub-array addresses only), no Address Mode signal, but with all other Mode signals remaining. The

narrower DR-Bus design uses less area, but increases the execution times of both diagnosis and repair algorithms, than would the wider Bus design. Furthermore, the wider Bus design is more fault tolerant because it does not need a volatile and potentially faulty flip-flop in every programmable module to record whether that Programmable Module is currently being accessed by the DR-Unit, since the sub-array address is continuously available on the wider Bus.

7 Low-level Algorithms:

7.1 Low-level notation for writing the algorithms.

Here are the assembly language instructions which are available from the DR-Units. The first set of instructions corresponds to the "diagnosisOnly" design, which uses the "parallel/serial" style of writing/reading data to/from the memory cell array. The second set of instructions corresponds to the "withRepair" design, which uses the "modular" style of writing/reading data to/from the memory cell array.

"Diagnosis Only" Instruction Set:
Hex-code, Description

08 Pop top stack-value and Increment counter
09 Increment counter
0A Pop top stack-value (not used)
0B Push onto stack
0C Load top stack-value into counter
0D Clear: reset counter, reset stack
0E Load Goto-value into counter
0F Increment counter (duplicates 09)

10 Upstart: reset address counter to 0000

11 Upcount: increment counter by 1

12 Downstart: reset address counter to 1111

13 Downcount: decrement counter by 1

18 Read from memory during Uppause (with Serial 0)

19 Write to memory during Uppause (with Serial 0)

1A Read from memory during Downpause (with Serial 0)

1B Write to memory during Downpause (with Serial 0)

1C Read from memory during Uppause (with Serial 1)

1D Write to memory during Uppause (with Serial 1)

1E Read from memory during Downpause (with Serial 1)

1F Write to memory during Downpause (with Serial 1)

20 Startup and Clear (in test-mode)

24 No-op (in test-mode)

30 Stop: report failure to repair

31 Stop: report successful repair

38 Startup and Clear (in normal-mode)

3C No-op (in normal-mode)

3E Read from memory (in normal-mode)

3F Write to memory (in normal-mode)

58 Read from memory during Uppause (with Parallel 0000)

59 Write to memory during Uppause (with Parallel 0000)

5A Read from memory during Downpause (with Parallel 0000)

5B Write to memory during Downpause (with Parallel 0000)

5C Read from memory during Uppause (with Parallel 1111)

5D Write to memory during Uppause (with Parallel 1111)

5E Read from memory during Downpause (with Parallel 1111)

5F Write to memory during Downpause (with Parallel 1111)

60 Until Flag(0) is true

61 Until Flag(1) is true

62 Until Flag(2) is true

63 Until Flag(3) is true

64 Until Flag(4) is true

67 Repeat ...

68 If Flag(0) is true

69 If Flag(1) is true

6A If Flag(2) is true

6B If Flag(3) is true

6C If Flag(4) is true

6F Return from MACRO

70, 71, 72, 73, 74, 75, 76, 77 GOTO MACRO (split 3 + 7 address)

"With Repair" Instruction Set:

Hex-code, Description

00 Until Flag(0) is true ($X + Y = 0, X = -Y$)

01 Until Flag(1) is true ($X = Y$)

02 Until Flag(2) is true (overflow)

03 Repeat ...

04 If Flag(0) is true ($X + Y = 0, X = -Y$)

05 If Flag(1) is true ($X = Y$)

06 If Flag(2) is true (overflow)

07 Return from MACRO

08 Pop top stack-value and Increment counter

09 Increment counter

0A Pop top stack-value (not used)

0B Push onto stack

0C Load top stack-value into counter

0D Clear: reset counter, reset stack, reset ALU registers X and Y

0E Load Goto-value into counter

0F Increment counter (duplicates 09)

10 Upstart: reset address counter to 0000

11 Upcount: increment counter by 1

12 Downstart: reset address counter to 1111

13 Downcount: decrement counter by 1

18 Read from memory during Uppause

19 Write to memory during Uppause

1A Read from memory during Downpause

1B Write to memory during Downpause

20 Startup and Clear (in test-mode)

21 Enable a selected Module Decoder

22 Program a selected Soft Fuse

24 No-op (in test-mode)

30 Stop: report failure to repair

31 Stop: report successful repair

38 Startup and Clear (in normal-mode)

3C No-op (in normal-mode)

3E Read from memory (in normal-mode)

3F Write to memory (in normal-mode)

40, 41, 42, 43 Load X with 0001, 1111, 0101, 1010

44, 45, 46, 47 Load Y with 0001, 1111, 0101, 1010

48 Send 0000 to busPort (precedes 19, 1B, 21, 22)

49 Send 1111 to busPort (precedes 19, 1B, 21, 22)

4A Send 0101 to busPort (precedes 19, 1B, 21, 22)

4B Send 1010 to busPort (precedes 19, 1B, 21, 22)

50, 51, 52, 53 Load X from register A, B, C, D

54, 55, 56, 57 Load Y from register A, B, C, D

58, 59, 5A, 5B Send Z to register A, B, C, D

60 Z = X PLUS Y

61 Z = X AND Y

62 Z = X OR Y

63 Z = X EXOR Y

64 Z = NOT(X)

65 Z = SHIFT(X) WITH 0

66 Z = SHIFT(X) WITH 1

67 Z = REVERSE(X)

6C Feedback Z to X

6D Feedback Z to Y

68 Load X from busPort (follows 18, 1A)

69 Load Y from busPort (follows 18, 1A)

6F Send Z to busPort (precedes 19, 1B, 21, 22)

70, 71, 72, 73 Load reg. A, B, C, D from busPort (follows 18,1A)

74, 75, 76, 77 Send reg. A, B, C, D to busPort (pre. 19, 1B, 21, 22)

78, 79, 7A, 7B, 7C, 7D, 7E, 7F GOTO MACRO (split 3 + 7 address)

The shifting instructions with hex-codes 65 and 66 refer to a bit-wise right-shifting operation, where the new bit (from the left side) can be either a 0 (hex-code 65) or a 1 (hex-code 66). In combination with two other instructions we can emulate a bit-wise left-shifting operation, using a sequence of five instructions. The reverse(x) instruction with hex-code 67, performs a bit-wise reversal — in other words: the first bit is exchanged with the last, the second is exchanged with the second-to-last, and so on. The feedback z-to-x instruction with hex-code 6C, simply sends the processed value of x, which is now present in z, back to x for further processing. The four ways to perform shifting are shown below:

"shift right with a new bit of 0" is performed as follows:

(65) Z = SHIFT(X) WITH 0

"shift right with a new bit of 1" is performed as follows:

(66) Z = SHIFT(X) WITH 1

"shift left with a new bit of 0" is performed as follows:

(67) Z = REVERSE(X)

(6C) Feedback Z to X

(65) Z = SHIFT(X) WITH 0

(6C) Feedback Z to X

(67) Z = REVERSE(X)

"shift left with a new bit of 1" is performed as follows:

(67) Z = REVERSE(X)

(6C) Feedback Z to X

(66) Z = SHIFT(X) WITH 1

(6C) Feedback Z to X

(67) Z = REVERSE(X)

7.2 Translation from High-level to Low-level notation

A sample diagnosis algorithm is shown below, and march elements 1 and 2 are expanded into the "withRepair" assembly language (hex-codes have been omitted).

Sample Diagnosis Algorithm:

march element 1: For i=0 to m-1, W1(i).

march element 2: For i=0 to m-1, W0(i), R0(i), W1(i), R1(i).

march element 3: For i=0 to m-1, WC(i).

march element 4: For i=0 to m-1, W1(i), R1(i), W0(i), R0(i).

The assembly language version of march elements 1 and 2 are given below.

* Status register: sends signals to BIST circuits not connected to Local Test Bus.

- ⋆ Interbus Port: specially buffered register used to interface the Local Test Bus with the Data Path Bus.
- ⋆ Input X: an input register for the ALU, with built-in shifter.
- ⋆ Input Y: an input register for the ALU, no shifting ability.
- ⋆ Output Z: the only output register from the ALU.
- ⋆ R/O register #1: contains all zeroes.
- ⋆ R/O register #2: contains all ones.
- ⋆ R/O register #3: contains the maximum row address plus one.
- ⋆ R/W register #1: keeps track of the current row being accessed.

step 1: For i=0 to m-1, W1(i).

1. Send R/O reg #1 to R/W reg #1. (this clears R/W reg #1 to all zeroes)
2. Send via Status reg, signal to initialize the Row decoder's counter to all zeroes.
3. Send R/O reg #2 to Interbus Port.
4. Drive the Interbus Port. (this writes all ones to the current word in the current memory segment)
5. Send R/W reg #1 to Input Y.
6. Increment Input Y operation.
7. Send Output Z to Input Y.
8. Send R/O reg #3 to Input X.
9. Exclusive OR operation.
10. Send Input Y to R/W reg #1.
11. Send via Status reg, signal to increment Row decoder's counter by one.
12. If non-zero Output Z, goto label 3.

step 2: For i=0 to m-1, W0(i), R0(i), W1(i), R1(i).

1. Send R/O reg #1 to R/W reg #1. (this clears R/W reg #1 to all zeroes)
2. Send via Status reg, signal to initialize the Row decoder's counter to all zeroes.
3. Send R/O reg #1 to Interbus Port.
4. Drive the Interbus Port. (this writes all zeroes to the current word in the current memory segment)
5. Sense the Interbus Port. (this reads the contents of the current word)
6. Send Interbus Port to Input Y.
7. Send R/O reg #1 to Input X.
8. Exclusive OR operation.
9. If non-zero Output Z, goto "Repair Info Storage"(R/W reg #1).
10. Send R/O reg #2 to Interbus Port.
11. Drive the Interbus Port. (this writes all ones to the current word in the current memory segment)
12. Sense the Interbus Port. (this reads the contents of the current word)
13. Send Interbus Port to Input Y.
14. Send R/O reg #2 to Input X.
15. Exclusive OR operation.
16. If non-zero Output Z, goto "Repair Info Storage"(R/W reg #1).
17. Send R/W reg #1 to Input Y.
18. Increment Input Y operation.
19. Send Output Z to Input Y.

20. Send R/O reg #3 to Input X.

21. Exclusive OR operation.

22. Send Input Y to R/W reg #1.

23. Send via Status reg, signal to increment Row decoder's counter by one.

24. If non-zero Output Z, goto label 3.

8 Conclusion

This thesis describes the many details of how to design a repairable embedded RAM with built-in self-diagnosis and self-repair capabilities. Calculated estimates show that such a design can be implemented with an acceptably small amount of extra area overhead. The design methodology is very flexible, since all that is required to change the many different fault types being diagnosed is to reprogram a ROM. When combined with a "soft switching" built-in self-repair technique, this self-diagnosis methodology can completely repair an embedded RAM without any external intervention or assistance.

Furthermore, the diagnosis and repair of several embedded RAMs contained in the same computer system can be carried out by a single *shared* DR-Unit. The only extra circuits that must be replicated in every RAM are: the DR-Bus and some address generators. In cases where the RAM is embedded on a chip with a general purpose processor, that processor could substitute for the DR-Unit, and thereby reduce the area overhead quite dramatically. The BISD scheme can also be conveniently incorporated into regular system maintenance procedures. The BISD methodology also has the original feature of being able to use a fault-free portion of the RAM-under-test in order to store "fault maps" of other faulty sections of the memory, as part of the self-repair algorithms.

8.1 Short Summary

The thesis began in chapter 1 with an introduction to the topic and the reasons why new results on this topic are needed. It is clearly not cost-efficient for semiconductor manufacturers to discard chips with only 1 or 2 faulty bits in their embedded RAMs. Hence, there is a requirement for fault location, but since embedded RAMs do not allow access from the I/O pins, the standard testing procedures for full-chip RAMs cannot be applied here. The resulting problem of "how to achieve yield improvement for embedded RAMs" is a special instance of the more general engineering challenge of "how to apply architectural techniques to solve the manufacturing yield problem."

Chapter 2 described the physical structures commonly used in embedded RAMs. Chapter 3 classified the physical failures that occur within these RAM circuit structures into formal "fault models." Chapter 4 provided a survey of research related to this thesis. Of particular interest is the formalization, in section 4.2, of a result pertaining to *sequential sense amplifier behavior.*

Chapter 5 starts with a preview of test and diagnosis algorithms. Section 5.3 explains original rules for the transformation of single-bit march tests into multi-bit march tests. This is followed, in section 5.4, by a discussion of the *hardware implications* of the three types of algorithms that must be built-in, namely:

(1) Test algorithms, which detect faults,

(2) Diagnosis algorithms, which locate faults, and

(3) Repair algorithms, which replace faults.

The limited, built-in data-access to the embedded RAM for diagnostic purposes can take only one of four forms, namely: (a) purely serial access, (b) purely parallel access, (c) hybrid serial and parallel access, and (d) modular access. Architectures (c) and (d) are the most powerful, and they are used in the two designs presented in chapter 6. Section 5.5 summarizes some previously published results about

necessary and sufficient algorithmic conditions to detect (but not locate) faults that are *linked* with one another.

Section 5.6 contains all of the original algorithms proposed in this thesis. This section starts with new notation that extends the march notation to include algorithms with two levels of FOR-loops, since such algorithms are essential for the location of coupling faults and stuck-open faults in address decoders. This section also shows new algorithms using the "serial shifting notation" introduced in section 5.4. This chapter concludes with brief discussions of repair algorithms, and with the global control algorithm that allows the circuits in chapter 6 to apply the algorithms of chapter 5.

Section 6.1 is the sequel to section 5.4, with its circuit-level description of the various architectural options (i.e. SASB, SAMB, MASB, and MAMB) available to designers, and a summary of various built-in address generator circuits (i.e. up/down binary counter, bidirectional and all-zeroes-augmented linear feedback shift register, Hamming-unit-distance and orthogonal-address EXORs-and-shift-register-array) to choose from.

Section 6.2 describes the operation of, and includes circuit schematics of, the two new designs: subsection 6.2.1 shows the "built-in self-diagnosis with self-repair" design that uses "modular data-access", and subsection 6.2.2 shows the "built-in self-diagnosis without self-repair" design that uses "hybrid serial and parallel data-access."

Section 6.3 discusses the various costs associated with the design (such as the area overhead estimates), and the tradeoffs considered during the design process (such as the hierarchical architecture and local/global placement of redundant lines, and the possible use of neural networks to implement the repair algorithms).

Chapter 7 lists the assembly languages of both designs described in section 6.2. This is followed by a sample of how to translate the high-level algorithms from chapter 5 into these low-level instructions, which are stored in the ROM mentionned in chapter 6.

8.2 Significance of Original Results

The two most significant challenges that were resolved by the new results reported in this thesis are:

(1) the challenge of *developing original algorithms* that can locate a wide variety of fault types, and that are suitable for implementation in either of the two most powerful built-in self-diagnosis architectures: namely (a) the hybrid serial and parallel data-accessing architecture, and (b) the modular data-accessing architecture. Chapter 5 describes numerous original algorithms, and most of these algorithms are expressed using both a graphical representation (that makes them easy to understand) and using a newly-developed *compact* mathematical notation (which displays the inherent symmetry).

(2) the challenge of *designing performance-optimized and area-optimized hardware* which can execute the algorithms from chapter 5. Chapter 6 actually describes *two distinct* novel hardware designs: one for each of the two BISD architectures mentionned above. Given current trends in the reduction of feature sizes achieved by semiconductor manufacturers, it is reasonable to predict that such BISD hardware designs will be commercially implemented before the end of this decade.

Although this thesis has focussed upon RAMs that are embedded at the chip-level, the algorithms and hardware designs can also be applied to memories that are embedded at the board-level, or even at the system-level.

8.3 Future Work

In order to justify the theoretical estimates of area overhead, and to validate the claim that the speed performance penalty of the proposed hardware designs is negligible, the BISD circuits should be implemented in a semiconductor RAM chip.

Although the BISD circuits were designed using professional quality CAD software, and were simulated at the logical-functionality and transistor-delay-timing levels using a state-of-the-art proprietary simulator, an implementation in silicon could provide additional opportunities for evaluating the circuits, and thereby lead to improvements and insights which a simulator cannot give.

As new fault models are proposed to explain the behavior of physical defects in emerging fabrication technologies, new algorithms will obviously be required. It is not currently known how to systematically enumerate, in a practical fashion, all possible test and diagnosis algorithms for RAMs. However, the new compact notations introduced in chapter 5 (such as: the Galloping FOR-loop, the Hamming FOR-loop, the Serial read/write shifting operation with several variations, the rules for transforming single-bit march tests to multi-bit march tests, etc.) are built on top of the well-established march-element notation, and they give us a new overview of the symmetries and shared structures among the algorithms. This suggests the possibility that there may exist a general classification scheme for all practical test and diagnosis algorithms (including future algorithms which apply to as-yet-unknown fault models). To date, researchers in the field of memory testing have been content to describe new algorithms in any format, so long as the algorithms could be applied in a practical fashion. Perhaps a formal, mathematical notation — that forbids the use of pseudo-code and tabular-listings — could lead to new insights about memory testing.

9 Bibliography.

[Abadir and Reghbati 83] M.S. Abadir, and H.K. Reghbati, "Functional Testing of Semiconductor Random Access Memories," *ACM Computing Surveys*, vol. 15, no. 3, Sept. 1983, pp. 175–198.

[Aichelmann 84] F.J. Aichelmann, "Fault Tolerant Design Techniques for Semiconductor Memory Applications," *IBM Jour. Res. Develop.*, vol. 28, no. 2, March 1984, pp. 177–183.

[Aizaki *et al.* 90] S. Aizaki, T. Shimizu, M. Ohkawa, K. Abe, A. Aizaki, M. Ando, O. Kudoh, and I. Sasaki, "A 15-ns 4-Mb CMOS SRAM," *IEEE Jour. Solid-State Circuits*, vol. 25, no. 5, Oct. 1990, pp. 1063–1066.

[Asai 86] S. Asai, "Semiconductor Memory Trends," *Proceedings of the IEEE*, vol. 74, no. 12, Dec. 1986, pp. 1623–1635.

[Awaya *et al.* 87] T. Awaya, K. Toyoda, O. Nomura, Y. Nakaya, K. Tanaka, and H. Sugawara, "A 5ns Access Time 64Kb ECL SRAM," *1987 IEEE Internat. Solid-State Circuits Conf.*, pp. 130–131.

[Bardell and McAnney 85] P.H. Bardell, and W.H. McAnney, "Self-Test of Random Access Memories," *1985 IEEE Internat. Test Conf.*, pp. 352–355.

[Bardell and McAnney 88] P.H. Bardell, and W.H. McAnney, "Built-in Test for RAMs," *IEEE Design and Test of Computers*, vol. 5, no. 4, August 1988, pp. 29–36.

[Blalock and Jaeger 91] T.N. Blalock, and R.C. Jaeger, "A High-Speed Clamped Bit-Line Current-Mode Sense Amplifier," *IEEE Jour. Solid-State Circuits*, vol. 26, no. 4, April 1991, pp. 542–548.

[Blough 91] D.M. Blough, "On the Reconfiguration of Memory Arrays containing Clustered Faults," *FTCS-21*, 1991, pp. 444–451.

[Breuer and Friedman 76] M.A. Breuer, and A.D. Friedman, *Diagnosis and Reliable Design of Digital Systems*, Computer Science Press, 1976, pp. 139–160.

[Cenker *et al.* 79] R.P. Cenker, D.G. Clemons, W.R. Huber, J.B. Petrizzi, F.J. Procyk, and G.M. Trout, "A Fault-Tolerant 64K Dynamic RAM," *1979 IEEE Internat. Solid-State Circuits Conf.*, pp. 150–151.

[Chappell *et al.* 91] T.I. Chappell, B.A. Chappell, S.E. Schuster, J.W. Allan, S.P. Klepner, R.V. Joshi, and R.L. Franch, "A 2-ns Cycle, 3.8-ns Access 512-kb CMOS ECL SRAM with a Fully Pipelined Architecture," *IEEE Jour. Solid-State Circuits*, vol. 26, no. 11, Nov. 1991, pp. 1577–1584.

[Chen and Hsiao 84] C.L. Chen, and M.Y. Hsiao, "Error-Correcting Codes for Semiconductor Memory Applications: A State-of-the-Art Review," *IBM Jour. Res. Develop.*, vol. 28, no. 2, March 1984, pp. 124–134.

[Day 85] J.R. Day, "A Fault-Driven, Comprehensive Redundancy Program," *IEEE Design and Test of Computers*, vol. 2, no. 3, June 1985, pp. 35–44.

[de Jong and van de Goor 88] P. de Jong, and A.J. van de Goor, "Test Pattern Generation for API Faults in RAM," *IEEE Trans. Computers*, vol. 37, no. 11, Nov. 1988, pp. 1426–1428.

[Dekker *et al.* 88] R. Dekker, F. Beenker, and L. Thijssen, "Fault Modelling and Test Algorithm Development for Static Random Access Memories," *1988 IEEE Internat. Test Conf.*, pp. 343–351.

[Dekker *et al.* 89] R. Dekker, F. Beenker, and L. Thijssen, "Realistic Built-in Self-Test for Static RAMs," *IEEE Design and Test of Computers*, vol. 6, no. 1, Feb. 1989, pp. 26–34.

[Fitzgerald and Thoma 80] B.F. Fitzgerald, and E.P. Thoma, "Circuit Implementation of Fusible Redundant Addresses on RAMs for Productivity Enhancement," *IBM Jour. Res. Develop.*, vol. 24, no. 3, May 1980, pp. 291–297.

[Flannagan *et al.* 90] S.T. Flannagan, P.H. Pelley, N. Herr, B.E. Engles, T. Feng, S.G. Nogle, J.W. Eagan, R.J. Dunnigan, L.J. Day, and R.I. Kung, "8-ns CMOS 64K×4 and 256K×1 SRAMs," *IEEE Jour. Solid-State Circuits*, vol. 25, no.5, Oct. 1990, pp. 1049–1054.

[Foss 79] R. Foss, "The Design of MOS Dynamic RAMs," *1979 IEEE Internat. Solid-State Circuits Conf.*, pp. 140–141.

[Foss and Harland 75] R. Foss, and R. Harland, "Simplified Peripheral Circuits for a Marginally Testable 4K RAM," *1975 IEEE Internat. Solid-State Circuits Conf.*, pp. 102–103.

[Franklin and Saluja 90] M. Franklin, and K.K. Saluja, "Built-in Self-Testing of Random-Access Memories," *Computer*, vol. 23, no. 10, Oct. 1990, pp. 45–56.

[Franklin *et al.* 90] M. Franklin, K.K. Saluja, and K. Kinoshita, "A Built-In Self-Test Algorithm for Row/Column Pattern Sensitive Faults in RAMs," *IEEE Jour. Solid-State Circuits*, vol. 25, no. 2, April 1990, pp. 514–523.

[Fuja and Heegard 86] T. Fuja, and C. Heegard, "Row/Column Replacement for the Control of Hard Defects in Semiconductor RAMs," *IEEE Trans. Computers*, vol. 35, no. 11, Nov. 1986, pp. 996-1000.

[Gongwer and Gudger 83] G.S. Gongwer, and K.H. Gudger, "A 16K EEPROM Using EE Element Redundancy," *IEEE Jour. Solid-State Circuits*, vol. 18, no. 5, Oct. 1983, pp. 550–553.

[Grosspietsch *et al.* 86] K.E. Grosspietsch, H. Huber, and A. Müller, "The Concept of a Fault Tolerant and Easily Testable Associative Memory," *FTCS-16*, 1986, pp. 34–39.

[Grosspietsch *et al.* 87] K.E. Grosspietsch, L. Muhlack, K.L. Paap, and G. Wagner, "A Memory Interface Chip Designed for Fault Tolerance," *VLSI Systems Design*, June 1987, pp. 112–118.

[Gupta and Jha 88] G. Gupta, and N.K. Jha, "A Universal Test Set for CMOS Circuits," *IEEE Trans. Computer-Aided Design*, vol. 7, no. 5, May 1988, pp. 590–597.

[Hasan and Liu 88] N. Hasan, and C.L. Liu, "Minimum Fault Coverage in Reconfigurable Arrays," *FTCS-18*, 1988, pp. 348–353.

[Hayes 80] J.P. Hayes, "Testing Memories for Single-Cell Pattern-Sensitive Faults," *IEEE Trans. Computers*, vol. 29, no. 3, March 1980, pp. 249–254.

[Haznedar 91] H. Haznedar, *Digital Microelectronics*, "Chapter 10: Semiconductor Memories," pp. 436–528, Benjamin/Cummings, 1991.

[Inoue *et al.* 87] J. Inoue, T. Matsumura, M. Tanno, and J. Yamada, "Parallel Testing Technology for VLSI Memories," *1987 IEEE Internat. Test Conf.*, pp. 1066–1071.

[Jain and Stroud 86] S.K. Jain, and C.E. Stroud, "Built-In Self Testing of Embedded Memories," *IEEE Design and Test of Computers*, vol. 3, no. 5, Oct. 1986, pp. 27–37.

[Jarwala and Pradhan 87] N. Jarwala, and D. Pradhan, "An Easily Testable Architecture for Multi-Megabit RAMs," *1987 IEEE International Test Conference*, pp. 750–758.

[Jarwala and Pradhan 88] N.T. Jarwala, and D.K. Pradhan, "TRAM: A Design Methodology for High-Performance, Easily Testable, Multimegabit RAMs," *IEEE Trans. Computers*, vol. 37, no. 10, Oct. 1988, pp. 1235–1250.

[Kalter *et al.* 90] H.L. Kalter, C.H. Stapper, J.E. Barth, J. DiLorenzo, C.E. Drake, J.A. Fifield, G.A. Kelley, S.C. Lewis, W.B. van der Hoeven, and J.A. Yankosky, "A 50-ns 16-Mb DRAM with a 10-ns Data Rate and On-Chip ECC," *IEEE Jour. Solid-State Circuits*, vol. 25, no.5, Oct. 1990, pp. 1118–1127.

[Kikuda *et al.* 91] S. Kikuda, H. Miyamoto, S. Mori, M. Niiro, and M. Yamada, "Optimized Redundancy Selection Based on Failure-Related Yield Model for

64-MB DRAM and Beyond," *IEEE Jour. Solid-State Circuits*, vol. 26, no. 11, Nov. 1991, pp. 1550–1555.

[Kinoshita and Saluja 86] K. Kinoshita, and K.K. Saluja, "Built-In Testing of Memory Using an On-Chip Compact Testing Scheme," *IEEE Trans. Computers*, vol. 35, no. 10, Oct. 1986, pp. 862–870.

[Knaizuk and Hartmann 77a] J. Knaizuk, and C.R.P. Hartmann, "An Algorithm for Testing Random Access Memories," *IEEE Trans. Computers*, vol. 26, no. 4, April 1977, pp. 414–416.

[Knaizuk and Hartmann 77b] J. Knaizuk, and C.R.P. Hartmann, "An Optimal Algorithm for Testing Stuck-at Faults in Random Access Memories," *IEEE Trans. Computers*, vol. 26, no. 11, Nov. 1977, pp. 1141–1144.

[Koeppe 87] S. Koeppe, "Optimal Layout to Avoid CMOS Stuck-Open Faults," *24th ACM/IEEE Design Automation Conference*, 1987, pp. 829–835.

[Kumanoya *et al.* 85] M. Kumanoya, K. Fujishima, H. Miyatake, Y. Nishimura, K. Saito, T. Matsukawa, T. Yoshihara, and T. Nakano, "A Reliable 1-Mbit DRAM with a Multi-Bit Test Mode," *IEEE Jour. Solid-State Circuits*, vol. 20, no. 5, Oct. 1985, pp. 909–912.

[Kuo and Fuchs 87] S.Y. Kuo, and W.K. Fuchs, "Efficient Spare Allocation in Reconfigurable Arrays," *IEEE Design and Test of Computers*, vol. 4, no. 1, Feb. 1987, pp. 24–31.

[Kuriyama *et al.* 91] H. Kuriyama, T. Hirose, S. Murakami, T. Wada, K. Fujita, Y. Nishimura, and K. Anami, "An 8-ns 4-Mb Serial Access Memory," *IEEE Jour. Solid-State Circuits*, vol. 26, no. 4, April 1991, pp. 502–506.

[Le and Saluja 86] K.T. Le, and K.K. Saluja, "A Novel Approach for Testing Memories Using a Built-In Self Testing Technique," *1986 IEEE Internat. Test Conf.*, pp. 830–839.

[Lombardi and Huang 88] F. Lombardi, and W.K. Huang, "Approaches For the Repair of VLSI/WSI RRAMs by Row/Column Deletion," *FTCS-18*, 1988, pp. 342–347.

[Mano *et al.* 82] T. Mano, M. Wada, N. Ieda, and M. Tanimoto, "A Redundancy Circuit for a Fault Tolerant 256K MOS RAM," *IEEE Jour. Solid-State Circuits*, vol. 17, no. 4, Aug. 1982, pp. 726–730.

[Marinescu 82] M. Marinescu, "Simple and Efficient Algorithms for Functional RAM Testing," *1982 IEEE Internat. Test Conf.*, pp. 236–239.

[Mazumder and Patel 87] P. Mazumder, and J.H. Patel, "An Efficient Built-In Self Testing for Random Access Memory," *1987 IEEE Internat. Test Conf.*, pp. 1072–1077.

[Mazumder and Patel 89] P. Mazumder, and J.H. Patel, "Parallel Testing for Pattern-Sensitive Faults in Semiconductor Random-Access Memories," *IEEE Trans. Computers*, vol. 38, no. 3, March 1989, pp. 394–407.

[Mazumder *et al.* 87] P. Mazumder, J.H. Patel, and W.K. Fuchs, "Design and Algorithms for Parallel Testing of Random Access and Content Addressable Memories," *24th ACM/IEEE Design Automation Conf.*, 1987, pp. 688–694.

[Mazumder and Yih 90] P. Mazumder, and J.S. Yih, "A Novel Built-in Self-Repair Approach to VLSI Memory Yield Enhancement," *1990 IEEE Internat. Test Conf.*, pp. 833–841.

[McAdams *et al.* 86] H. McAdams, J.H. Neal, B. Holland, S. Inoue, W.K. Loh, and K. Poteet, "A 1-Mbit CMOS Dynamic RAM with Design For Test Functions," *IEEE Jour. Solid-State Circuits*, vol. 21, no. 5, Oct. 1986, pp. 635–641.

[McAnney *et al.* 84] W.H. McAnney, P.H. Bardell, and V.P. Gupta, "Random Testing for Stuck-at Storage Cells in an Embedded Memory," *1984 IEEE Internat. Test Conf.*, pp. 157–166.

[McAnney *et al.* 85] W.H. McAnney, J. Savir, and S.R. Vecchio, "Random Pattern Testing for Data-Line Faults in an Embedded Multiport Memory," *1985 IEEE Internat. Test Conf.*, pp. 100–105.

[McAuley and Cotton 91] A.J. McAuley, and C.J. Cotton, "A Self-Testing Reconfigurable CAM," *IEEE Jour. Solid-State Circuits*, vol. 26, no. 3, March 1991, pp. 257–261.

[Miyaji *et al.* 90] F. Miyaji, T. Emori, Y. Matsuyama, Y. Kanaishi, K. Seno, and Y. Hagiwara, "A Multibit Test Trigger Circuit for Megabit SRAMs," *IEEE Jour. Solid-State Circuits,* vol. 25, no. 1, Feb. 1990, pp. 68–71.

[Moore 86] W.R. Moore, "A Review of Fault Tolerant Techniques for the Enhancement of Integrated Circuit Yield," *Proceedings of the IEEE,* vol. 74, no. 5, May 1986, pp. 684–698.

[Mori *et al.* 91] S. Mori, H. Miyamoto, Y. Morooka, S. Kikuda, M. Suwa, M. Kinoshita, A. Hachisuka, H. Arima, M. Yamada, T. Yoshihara, and S. Kayano, "A 45-ns 64-Mb DRAM with a Merged Match-Line Test Architecture," *IEEE Jour. Solid-State Circuits,* vol. 26, no. 11, Nov. 1991, pp. 1486–1491.

[Murakami *et al.* 91] S. Murakami, K. Fujita, M. Ukita, K. Tsutsumi, Y. Inoue, O. Sakamoto, M. Ashida, Y. Nishimura, Y. Kohno, T. Nishimura, and K. Anami, "A 21-mW 4-Mb CMOS SRAM for Battery Operation," *IEEE Jour. Solid-State Circuits,* vol. 26, no. 11, Nov. 1991, pp. 1563–1569.

[Nadeau-Dostie *et al.* 90] B. Nadeau-Dostie, A. Silburt, and V.K. Agarwal, "Serial Interfacing for Embedded-Memory Testing," *IEEE Design and Test of Computers,* vol. 7, no. 2, April 1990, pp. 52–63.

[Naidu and Mahapatra 88] R.V. Naidu, and S. Mahapatra, "Fault Tolerance in NMOS Random Access Memories with Dynamic Redundancy Methods," *Microelectron. Reliab.,* vol. 28, no. 2, 1988, pp. 193–200.

[Nair *et al.* 78] R. Nair, S.M. Thatte, and J.A. Abraham, "Efficient Algorithms for Testing Semiconductor Random-Access Memories," *IEEE Trans. Computers,* vol. 27, no. 6, June 1978, pp. 572–576.

[Nakagome *et al.* 88] Y. Nakagome, M. Aoki, S. Ikenaga, M. Horiguchi, S. Kimura, Y. Kawamoto, and K. Itoh, "The Impact of Data-Line Interference Noise on DRAM Scaling," *IEEE Jour. Solid-State Circuits,* vol. 23, no. 5, Oct. 1988, pp. 1120–1125.

[Nicolaidis 85] M. Nicolaidis, "An Efficient Built-In Self-Test Scheme for Functional Test of Embedded RAMs," *FTCS-15*, 1985, pp. 118–123.

[O'Connor 87] K.J. O'Connor, "Modular Embedded Cache Memories for a 32b Pipelined RISC Microprocessor," *1987 IEEE Internat. Solid-State Circuits Conf.*, pp. 256–257.

[Ohba *et al.* 91] A. Ohba, S. Ohbayashi, T. Shiomi, S. Takano, K. Anami, H. Honda, Y. Ishigaki, M. Hatanaka, S. Nagao, and S. Kayano, "A 7-ns 1-Mb BiCMOS ECL SRAM with Shift Redundancy," *IEEE Jour. Solid-State Circuits*, vol. 26, no. 4, April 1991, pp. 507–511.

[Ohsawa *et al.* 87] T. Ohsawa, T. Furuyama, Y. Watanabe, H. Tanaka, N. Kushiyama, K. Tsuchida, Y. Nagahama, S. Yamano, T. Tanaka, S. Shinozaki, and K. Natori, "A 60-ns 4-Mbit CMOS DRAM with Built-In Self-Test Function," *IEEE Journal of Solid-State Circuits*, vol. 22, no. 5, October 1987, pp. 663–667.

[Papachristou and Sahgal 85] C.A. Papachristou, and N.B. Sahgal, "An Improved Method for Detecting Functional Faults in Semiconductor Random Access Memories," *IEEE Trans. Computers*, vol. 34, no. 2, Feb. 1985, pp. 110–116.

[Prince 91] B. Prince, *Semiconductor Memories: A Handbook of Design, Manufacture and Application second edition*, John Wiley & Sons, 1991.

[Reddy and Reddy 85] M.K. Reddy, and S.M. Reddy, "On FET Stuck-Open Fault-Detectable CMOS Memory Elements," *FTCS-15*, 1985, pp. 424–429.

[Reddy and Reddy 86] M.K. Reddy, and S.M. Reddy, "Detecting FET Stuck-Open Faults in CMOS Latches and Flip-Flops," *IEEE Design and Test of Computers*, vol. 3, no. 5, Oct. 1986, pp. 17–26.

[Regener 88] E. Regener, "A Transition Sequence Generator for RAM Fault Detection," *IEEE Trans. Computers*, vol. 37, no. 3, March 1988, pp. 362–368.

[Ritter and Müller 87] H.C. Ritter, and B. Müller, "Built-In Test Processor for Self Testing Repairable Random Access Memories," *1987 IEEE Internat. Test Conf.*, pp. 1078–1084.

[Saluja and Kinoshita 85] K.K. Saluja, and K. Kinoshita, "Test Pattern Generation for API Faults in RAM," *IEEE Trans. Computers*, vol. 34, no. 3, March 1985, pp. 284–287.

[Saluja *et al.* 87] K.K. Saluja, S.H. Sng, and K. Kinoshita, "Built-In Self Testing RAM: A Practical Alternative," *IEEE Design and Test of Computers*, vol. 4, no. 1, Feb. 1987, pp. 42–51.

[Sarkany and Hart 87] E.F. Sarkany, and W.S. Hart, "Minimal Set of Patterns to Test RAM Components," *1987 IEEE Internat. Test Conf.*, pp. 759–764.

[Savir *et al.* 85] J. Savir, W.H. McAnney, and S.R. Vecchio, "Random Pattern Testing for Address-Line Faults in an Embedded Multiport Memory," *1985 IEEE Internat. Test Conf.*, pp. 106–114.

[Schroder 87] D.K. Schroder, *Advanced MOS Devices*, "Chapter 5: Semiconductor Memories," pp. 163–192, Addison-Wesley, 1987.

[Schroeder and Proebsting 77] P.R. Schroeder, and R.J. Proebsting, "A 16K×1–bit Dynamic RAM," *1977 IEEE Internat. Solid-State Circuits Conf.*, pp. 12–13.

[Seth and Narayanaswamy 81] S.C. Seth, and K. Narayanaswamy, "A Graph Model for Pattern-Sensitive Faults in Random Access Memories," *IEEE Trans. Computers*, vol. 30, no. 12, Dec. 1981, pp. 973–977.

[Smith *et al.* 81] R.T. Smith, J.D. Chlipala, J.F.M. Bindels, R.G. Nelson, F.H. Fischer, and T.F. Mantz, "Laser Programmable Redundancy and Yield Improvement in a 64K DRAM," *IEEE Jour. Solid-State Circuits*, vol. 16, no. 5, Oct. 1981, pp. 506–513.

[Sridhar 86] T. Sridhar, "A New Parallel Test Approach for Large Memories," *IEEE Design and Test of Computers*, vol. 3, no. 4, Aug. 1986, pp. 15–22.

[Srini 78] V.P. Srini, "Fault Location in a Semiconductor Random-Access Memory Unit," *IEEE Trans. Computers*, vol. 27, no. 4, April 1978, pp. 349–358.

[Stapper *et al.* 80] C.H. Stapper, A.N. McLaren, and M. Dreckmann, "Yield Model for Productivity Optimization of VLSI Memory Chips with Redundancy and

Partially Good Product," *IBM Jour. Res. Develop.*, vol. 24, no. 3, May 1980, pp. 398–409.

[Stapper 86] C.H. Stapper, "On Yield, Fault Distributions, and Clustering of Particles," *IBM Jour. Res. Develop.*, vol. 30, no. 3, May 1986, pp. 326–338.

[Stein *et al.* 72] K.U. Stein, A. Sihling, and E. Doering, "Storage Array and Sense/Refresh Circuit for Single-Transistor Cells," *1972 IEEE Internat. Solid-State Circuits Conf.*, pp. 56–57.

[Stevens 86] A.K. Stevens, *Introduction to Component Testing*, Addison-Wesley, 1986, pp. 39–62.

[Suk and Reddy 80] D.S. Suk, and S.M. Reddy, "Test Procedures for a Class of Pattern-Sensitive Faults in Semiconductor Random Access Memories," *IEEE Trans. Computers*, vol. 29, no. 6, June 1980, pp. 419–429.

[Suk and Reddy 81] D.S. Suk, and S.M. Reddy, "A March Test for Functional Faults in Semiconductor RAMs," *IEEE Trans. Computers*, vol. 30, no. 12, Dec. 1981, pp. 982–985.

[Sun and Wang 84] Z. Sun, and L.T. Wang, "Self Testing of Embedded RAMs," *1984 IEEE Internat. Test Conf.*, pp. 148–156.

[Suzuki *et al.* 91] M. Suzuki, S. Notomi, M. Ono, N. Kobayashi, E. Mitani, K. Odani, T. Mimura, and M. Abe, "A 1.2-ns HEMT 64-kb SRAM," *IEEE Jour. Solid-State Circuits*, vol. 26, no. 11, Nov. 1991, pp. 1571–1575.

[Takada *et al.* 90] M. Takada, K. Nakamura, T. Takeshima, K. Furuta, T. Yamazaki, K. Imai, S. Ohi, Y. Sekine, Y. Minato, and H. Kimoto, "A 5-ns 1-Mb ECL BiCMOS SRAM," *IEEE Jour. Solid-State Circuits*, vol. 25, no. 5, Oct. 1990, pp. 1057–1061.

[Takano *et al.* 87] S. Takano, H. Makino, N. Tanino, M. Noda, K. Nishitani, and S. Kayano, "A 16K GaAs SRAM," *1987 IEEE Internat. Solid-State Circuits Conf.*, pp. 140–141.

[Takeshima *et al.* 90] T. Takeshima, M. Takada, H. Koike, H. Watanabe, S. Koshimaru, K. Mitake, W. Kikuchi, T. Tanigawa, T. Murotani, K. Noda, K. Tasaka, K. Yamanaka, and K. Koyama, "A 55-ns 16-Mb DRAM with Built-in Self-Test Function Using Microprogram ROM," *IEEE Jour. Solid-State Circuits*, vol. 25, no. 4, Aug. 1990, pp. 903–909.

[Varma *et al.* 84] P. Varma, A.P. Ambler, and K. Baker, "On Chip Testing of Embedded RAMs," *IEEE Transactions on Computers*, vol. 33, no. 7, July 1984, pp. 286–290.

[van de Goor 91] A.J. van de Goor, *Testing Semiconductor Memories: Theory and Practice*, John Wiley & Sons, 1991.

[van de Goor and Verruijt 90] A.J. van de Goor, and C.A. Verruijt, "An Overview of Deterministic Functional RAM Chip Testing," *ACM Computing Surveys*, vol. 22, no. 1, March 1990, pp. 5–33.

[Wagner 88] K.D. Wagner, "Clock System Design," *IEEE Design and Test of Computers*, vol. 5, no. 5, Oct. 1988, pp. 9–27.

[Weste and Eshraghian 85] N.H.E. Weste, and K. Eshraghian, *Principles of CMOS VLSI Design: A Systems Perspective*, "Section 8.5: Random access memory," pp. 348–366, Addison-Wesley, 1985.

[Williams *et al.* 88] T. Williams, K. Beilstein, B. El-Kareh, R. Flaker, G. Gravenites, R. Lipa, H.S. Lee, J. Maslack, J. Pessetto, W.F. Pokorny, M. Roberge, and H. Zeller, "An Experimental 1-Mbit CMOS SRAM with Configurable Organization and Operation," *IEEE Jour. Solid-State Circuits*, vol. 23, no. 5, Oct. 1988, pp. 1085–1093.

[You and Hayes 85] Y. You, and J.P. Hayes, "A Self Testing Dynamic RAM Chip," *IEEE Jour. Solid-State Circuits*, vol. 20, no. 1, Feb. 1985, pp. 428–435.