University of Alberta

# Ternary Content Addressable Memory Directed Redundancy for Semiconductor Memory Yield Enhancement

by

Craig Joly ©

A thesis submitted to the Faculty of Graduate Studies and Research in partial fulfillment of the requirements for the degree of **Master of Science**.

Department of Electrical and Computer Engineering

Edmonton, Alberta
Spring 2004

Library and
Archives Canada

Published Heritage
Branch

395 Wellington Street
Ottawa ON K1A 0N4
Canada

Bibliothèque et
Archives Canada

Direction du
Patrimoine de l'édition

395, rue Wellington
Ottawa ON K1A 0N4
Canada

*Your file Votre référence*
*ISBN: 0-612-96496-5*
*Our file Notre référence*
*ISBN: 0-612-96496-5*

# Abstract

I describe a redundancy mechanism, known as associative ternary CAM redundancy, that is able to repair single cell, row, column or cluster faults by mapping a common pool of redundant memory to replace these defective regions. This is achieved by accessing a ternary content addressable memory, containing addresses of faulty regions, in parallel with the main memory.

Associative redundancy is completely transparent to the DRAM, allowing full regular operation and no reduction in frequency. Also, there is no increase in area over conventional row and column redundancy or extra fabrication steps.

Yield modeling shows that, in a 1-Gbit DRAM, associative redundancy performs better than conventional redundancies in most cases. A fault model comprised of single cell, row, column and cluster failures shows that associative redundancy can handle over 20 times greater fault densities at 50% equivalent yield.

*In the Kamigata area they have a sort of tiered lunchbox*
*they use for a single day when flower viewing.*
*Upon returning, they throw them away, trampling them underfoot.*
*The end is important in all things.*

Yamamoto Tsunetome, Hagakure

To Keiko.

# Acknowledgements

This research was supported by Micronet R&D, MOSAID Technologies Inc, the Canadian Microelectronics Corporation and the University of Alberta. I would like to give special thanks to my supervisor, Duncan Elliott for the help and direction. I would also like to thank Bruce Cockburn, Tyler Brandon, Daniel Leder, John Koob, Kris Breen and Curtis and Jennifer Wickman for listening, making suggestions and letting me bounce ideas off of them.

# Table of Contents

# List of Figures

# List of Tables

# List of Acronyms

**ASIC** Application Specific Integrated Circuit

**CAM** Content Addressable Memory

**/CAS** Column Access Strobe (active low)

**CAROM** Content Addressable Read Only Memory

**CMOS** Complementary Metal-Oxide Semiconductor

**DDR SDRAM** Double Data Rate Synchronous Dynamic Random Access Memory

**DRAM** Dynamic Random Access Memory

**ECC** Error Correcting Code

**EEPROM** See E$^2$PROM

**EPROM** Electrically Programmable Read Only Memory

**E$^2$PROM** Electrically Erasable and Programmable Read Only Memory

**FeRAM** Ferroelectric Random Access Memory

**IC** Integrated Circuit

**MOS** Metal-Oxide Semiconductor

**MRAM** Magnetic Random Access Memory

**NOR** Not OR (logical operation)

**NVRAM** Non-Volatile Random Access Memory

**PCM** Phase Change Memory

**RAM** Random Access Memory

**/RAS** Row Access Strobe (active low)

**ROM** Read Only Memory

**SEC** Single Error Correcting

**SEC-DED** Single Error Correcting, Double Error Detecting

**SDRAM** Synchronous Dynamic Random Access Memory

**SRAM** Static Random Access Memory

**TCAM** Ternary Content Addressable Memory

**TSMC** Taiwan Semiconductor Manufacturing Company Ltd.

**XOR** Exclusive OR (logical operation)

# List of Terms

**Associative Memory** A type of memory that will return data associated with the input data (such as a hash)

**Book** Area of memory array that column redundancy is applied to

**Content Addressable Memory** A compare array which will return a "match" or "no-match" for any input data

**Codeword** Grouping of bits along a word-line containing data and check bits for error correcting coding

**Complexity** A model's accuracy in accounting for an actual event's mechanisms

**Critical Path** The path within a design that dictates the fastest time at which the entire design can run

**Defect** Physical imperfection causing a variation from the design

**Dimensionality** The degrees of freedom in a model

**Equivalent Yield** Yield normalized to the area of a device without redundancy

**Fault** Failure attributable to some defect in the system

**Fault Model** Global abstraction level about what kinds of failures can occur

**Fidelity** Accuracy of a model's prediction compared to the actual event

**Functional Fault** Faults affecting a small area that cause a circuit failure

**Gray code** A binary code in which consecutive decimal numbers are represented by binary numbers that differ in the state of one bit (also Synonym reflected code)

**Hamming code** Common type of error correcting code

**lambda** In yield models, $\lambda$ is commonly used to denote the fault density. Note that this is contrary to the typical VLSI usage of $\lambda$ to denote one half the minimum feature size. This thesis also uses $\rho$, $\chi$ and $\psi$ to denote fault densities.

**Memory** A semiconductor device that stores data

**Page** A single row of memory cells in a book

**Parametric Fault** Faults affecting large areas resulting in functional devices that do not meet all specifications (timing, electrical, etc)

**Redundancy** Duplication of design elements for preventing failure of an entire system upon failure of a single component

**Section** Area of memory array to which row redundancy is applied

**Synergy** A result greater than the sum of its parts

**Word** Smallest addressable portion of a memory. A word can also refer to all cells on a word line, but the first definition is used in the context of this thesis.

**Yield** The ratio of fully operating integrated circuit dies to the total number of dies fabricated

**Yield Model** A mathematical model for predicting yield, characterized according to fidelity, complexity and dimensionality

# Chapter 1

# Introduction

Dynamic Random Access Memories (DRAM) accounted for 11% of the entire semiconductor market in 2002 [11]. This is about $15.5 billion dollars. In order for a DRAM to be sold, it must be operating at 100% of its specified capacity. All of the storage cells must be functioning correctly. With millions of storage cells on a single DRAM die, this is an impressive feat. In order to lower the number of nearly perfect memory ICs that will be thrown out and to raise yields, memory manufacturers employ redundancy and, occasionally, error correction.

In the RAM world, redundancy almost always means row and column redundancy. In this thesis, I propose an alternate redundancy method using a ternary content addressable memory (CAM) to watch for and redirect addresses of bad storage cells. The ternary CAM allows single cells, columns, rows and rectangular groupings of cells to be marked as bad and electrically replaced. This gives a large amount of flexibility in marking bad areas, either reducing the area required for redundancy, thereby shrinking die size and reducing cost [12] or permitting earlier production of fully functional chips while defect rates are still high.

This redundancy scheme can be applied to all semiconductor memories with binary addressing. I develop this scheme and analyze its performance applied to DRAM. I have chosen DRAM since it is a common and representative memory type. Also, the benefits of new redundancy methods should be especially significant in the highest density memories, which are DRAMs.

Chapter 2 discusses pertinent background on DRAMs, yield, redundancy schemes and CAMs. In Chapter 3, I outline the redundancy scheme, propose two implementations and discuss considerations for integrating the scheme into conventional DRAM fabrication processes. In Chapter 4, die yields for various classes of faults are estimated. I compare the two proposed ternary CAM redundancy methods' handling of the faults with previous redundancy methods using effective yield as a metric. Critical path timing of the two proposed methods are investigated in Chapter 5. Chapter 6 concludes with a summary of results and suggestions for future work.

# Chapter 2

# Background

## 2.1 Semiconductor Memories

Semiconductor memories can be divided into two main classes: volatile and non-volatile, as shown in Figure 2.1. Volatile memories, such as SRAMs and DRAMs lose their contents when power is lost. Conversely, non-volatile memories, such as flash, electrically erasable and programmable read only memories ($E^2$PROM), electrically programmable read only memories (EPROM), read only memoriess (ROM), ferroelectric RAM (FeRAM), magnetic RAM (MRAM) and phase-change memories (PCM) retain their contents, even when power is not applied.

All of the solid-state memories shown in Figure 2.1 are random access. This means that any bit can be accessed in any order with the same access time. Of the memories, DRAM has the lowest cost per bit because of its density. Although flash

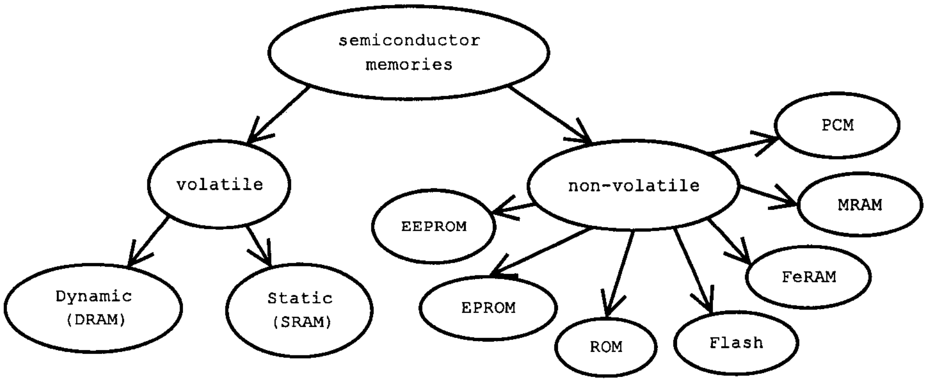


Figure 2.1: Taxonomy of solid-state memories

approaches and sometimes exceeds DRAM in bit density, it requires extra processing steps and more comprehensive testing requirements, both of which increase its cost above that of DRAM.

## 2.2 The DRAM Market

The DRAM market is a high volume, low profit margin, commodity business where every cent saved is important. The goal of this thesis is to match or improve yields at less cost, thereby improving manufacturers' profit.

DRAM manufacturers have to be able to supply the market at the right time with the right devices [8]. At this time, the highest volume device is the 256-Mbit DRAM, with the 1-Gbit DRAM quickly gaining market acceptance. The cost of the DRAM IC is determined by several factors:

**Design Effort** Due to the high volume of DRAM ICs shipped, these costs are less important as they are a one-time cost amortized over all ICs shipped.

**Silicon Area** Based on the technology and production, a price per area of processed silicon can be calculated [8].

**Production Yield** Because DRAMs tend to be the first devices fabricated in a smaller feature size, production yield is a significant contributor to the cost. They also use their own process, different from the more common logic processes.

**Packaging Cost** The costs of the package itself and the packaging process are relatively high. Minimizing the number of defective DRAMs being packaged will reduce costs [8].

**Test Costs** DRAMs must be 100% functionally correct to be sold. With larger storage capacities, a proportionally larger time is required to test the dies. Test related costs are increasing.

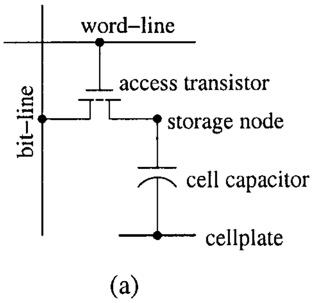


Figure 2.2: Basic DRAM cell

## 2.3 DRAM Architecture

A random access memory must provide a continuous address space of fully functioning cells that can be read and written. Every cell must also meet all timing constraints. In order to provide this, DRAM manufacturers use redundant bit-lines and word-lines to replace any faults within the array, so that the sold DRAM chips appear to provide a full-capacity continuous space of functioning cells [7].

The 1T1C (one transistor, one capacitor) DRAM has become ubiquitous because it can be implemented in a smaller area than any other memory cell type. The cell uses a single capacitor to store a certain amount of electrical charge that represents its bit. The charge representing the bit can be sensed by comparing the voltage difference between the storage node voltage and a mid-range reference voltage. A single MOS transistor is used to connect and disconnect the storage capacitor from other circuits. A typical cell is shown in Figure 2.2a.

In order to read the data stored in a cell, the bit-line is pre-charged to $\frac{V_{DD}}{2}$, then the word-line is asserted to turn on the access transistor. The charge on the cell capacitor is shared with the charge of the bit-line, causing a slight change in bit-line voltage. This slight change (up or down) is sensed by a sense amplifier to determine what bit value was stored.

Because the charge on the capacitor is shared, the voltage signal on the storage node is greatly attenuated, and so, the value must be rewritten after it is read. Also, due to leakage currents, the data must be periodically refreshed. Refresh is provided

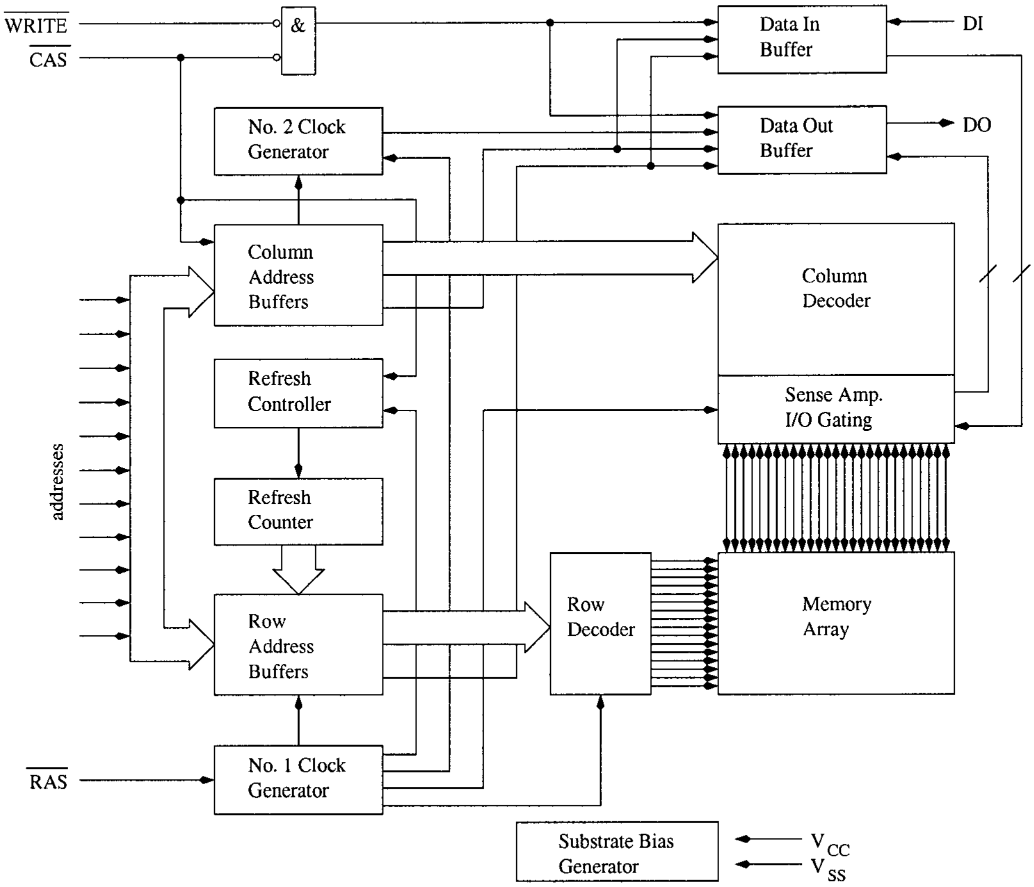


Figure 2.3: Block diagram of a typical DRAM [1]

by sense and write amplifiers attached to each pair of bit-lines.

A typical DRAM architecture, such as the one shown in Figure 2.3, consists of a refresh controller and counter, row and column address buffers, input and output data buffers, clock generators and other circuits.

## 2.4 Row and Column Redundancy

Redundancy has been used in DRAM designs since the 256-Kbit generation to improve yield by providing spare components that can be used to replace faulty ones [13]. In the case of semiconductor memories, redundancy means providing rows and columns of extra memory cells on the die that can be electrically swapped for bad ones [14]. Unfortunately, redundancy increases access and cycle times, power dissipation, IC area and requires design modifications [15]. These downsides are justified because redundancy reduces the cost per bit in large capacity memories, increases the memory bit capacity in immature processes and aids in providing fully functional parts in low volume productions. Smaller die sizes also reduce cost per bit, so the first point can sometimes be at cross-purposes. Excessive redundancy can also become a yield limiting factor in mature processes [15] (beyond a limit, where the reliability increase is balanced by the reliability loss due to the inflated number of elements in the memory, a reliability decrease (nuisance) appears).

Redundancy is achieved by having extra columns and rows of memory cells on the die. Originally, if a row or column was not 100% operational, it could be swapped out by way of a selection mechanism, such as a laser-blown fuse. In the abstract, the laser acts as, and can be replaced by, a non-volatile programmable memory. As feature sizes shrunk, the size of conventional fuses became prohibitively large. They could not be sufficiently shrunk and still enable a laser to be focused on them. This led to redundant sections of rows or columns, usually all rows or columns attached to one address decoder, still controlled by fuses. Fuses have their own reliability problems. Openings blown by lasers in the passivation layer can cause moisture contamination, relocation of blown fuse material can cause stresses in other layers of the die and partially blown fuses can cause poor reliability [14]. An alternative is to use programmable non-volatile memory in place of fuses.

In [7], the yield of an 16-Mbit and a 1-Gbit DRAM are discussed. The average number of defects at 50% yield and 50% equivalent yield (normalized yield to the area of a DRAM without redundancy) are used to compare redundancy effectiveness in terms of the number of recoverable defects. These values are shown in Table 2.1.

The data shows that row and column redundancy work together synergistically. That is, the effects of row and column redundancy together are more than the sum of their effects individually. This is because multiple errors along one row or column can be repaired with a single replacement.

Up to 90% of memory failures are single-bit failures [16]. This implies that,

| Memory | Redundancy | 50% Yield | 50% Equivalent Yield | Percent Area Overhead |
|---|---|---|---|---|
| 16-Mbit | Column Only | 28 | 28 | 1.54 |
| | Row Only | 79 | 79 | 0.58 |
| | Row and Column | 234 | 233 | 2.10 |
| 1-Gbit | Column Only | 28.5 | 28.5 | 0.098 |
| | Row Only | 430 | 430 | 0.78 |
| | Row and Column | 705 | 705 | 0.82 |

Table 2.1: Average number of sustainable memory cell defects at a 50% Yield/Equivalent Yield with Row and Column Redundancy [7]

without error correction, a 99.9% good row or column of memory can, and will be, swapped out because of one bad cell.

## 2.5 Error Correcting Codes

Error correction employs redundancy of a different type. It is usually employed to reduce the effects of soft errors, however, it can be used to eliminate the effects of a single faulty cell (hard error) in a word. When a word is written to the memory, check bits are calculated and stored along with it. The check bits can be as simple as a parity code, which can detect, but not repair, a single bit error in the stored word, however, more often it is a Hamming code. The most common type of Hamming code in semiconductor memories can detect two bit errors and correct a single bit error [7].

All error correcting codes work by recalculating the check bits when the word is read from the memory array. The newly calculated check bits are compared with the stored check bits, usually by taking the bitwise XOR [7], and syndrome bits are obtained. The syndrome bits may indicate whether an error has occurred and possibly where.

Table 2.2, when compared to Table 2.1, shows the average number of recoverable defects at 50% yield and 50% equivalent yield (yield normalized to the area of a memory without redundancy) for error correction codes in 16-Mbit and 1-Gbit DRAMs. These are 128 data bit, 9 check bit and 512 data bit, 11 check bit Hamming

| Memory | Redundancy | 50% Yield | 50% Equivalent Yield | Percent Area Overhead |
|---|---|---|---|---|
| 16-Mbit | ECC Only | 428 | 407 | 6.57 |
| 1-Gbit | ECC Only | 1707 | 1680 | 2.1 |

Table 2.2: Average number of sustainable memory cell defects at a 50% Yield/Equivalent Yield with Error Correcting Codes [7]

| Memory | Redundancy | 50% Yield | 50% Equivalent Yield | Percent Area Overhead |
|---|---|---|---|---|
| 16-Mbit | ECC, Row and Column | 8027 | 7977 | 10.41 |
| 1-Gbit | ECC, Row and Column | 54931 | 54751 | 4.94 |

Table 2.3: Average number of sustainable defects at a 50% Yield/Equivalent Yield with multiple redundancy [7]

codes, respectively. Error correction is much more effective than row and column redundancy in this example, but uses more area. This effectiveness is in part because up to 90% of memory failures are single-bit [16]. This implies that with column and row redundancy, most of the replaced rows and columns contain only a single error. Error correction is much more effective in dealing with random, single errors. This also implies that the co-ordinated use of column and row redundancy and error correction will have a very high synergy. Column and row redundancy can be used to address the 10% of clustered, column and row errors, while error correction is used to address the 90% of single bit errors. The data in Table 2.3 shows that this is indeed the case.

## 2.6 Defects, Faults and Yield

Yield is the ratio of fully operating integrated circuit dies to the total number of dies fabricated. Yield loss is caused by faults, which are in turn caused by defects. This relationship is shown in Figure 2.4. The solid lines indicate the faults which are more likely caused by specific defect types. Dashed-lines indicate less likely possibilities. Defects can be broken up into five classes [8]:

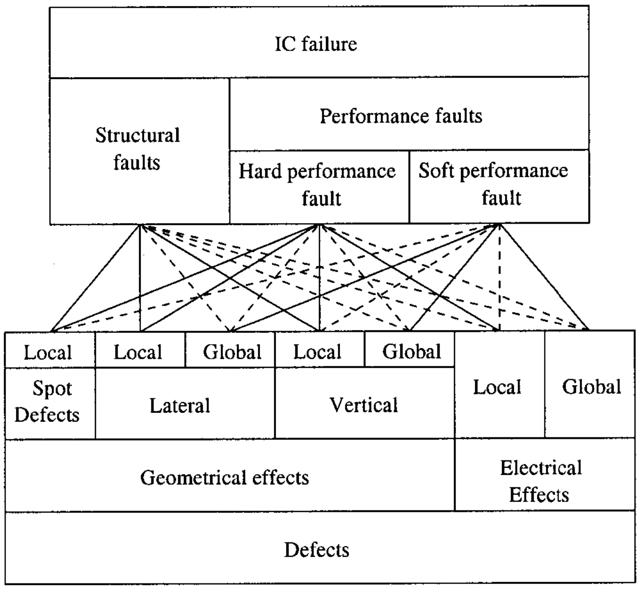


Figure 2.4: IC manufacturing process defects and relationships with IC faults [2]

**Wafer Defects** These are usually the result of contaminants and micro-cracks.

**Human Errors** Although automation is reducing the human factor, scratches, pollutants and process mistakes still do happen.

**Equipment Failure** Equipment failure is the main defect mechanism in modern manufacturing [8]. For the most part, these defect mechanisms are similar to human errors, however, incorrectly tuned equipment also often leads to systematic defect behaviour.

**Environmental Impact** This is typically caused by airborne contaminants such as dust.

**Process Instabilities** Temperature and pressure gradients can cause uneven deposition, etching or oxidation.

**Statistical Variations** Similar effect as process instabilities, resulting in variations in dopant atom concentrations, oxide thicknesses, etc.

| | Relative Defect Density | |
|---|---|---|
| Layer | Extra Conductive Material | Missing Conductive Material |
| Metal | 100 | 1 |
| Polysilicon | 50 | 1 |
| Thick oxide | 2 | 5 |
| Gate oxide | 20 | - |
| Doped area | 1 | 1 |

Table 2.4: Typical relative defect densities for a CMOS process [8]

These defect mechanisms can lead to global or local defects. Global defects occur over a large area and tend to be caused by more systematic mechanisms such as mask misalignment, line registration errors or differing implant levels. They tend to result more often in parametric faults. Local defects influence only a small area and tend to be caused by dust, contaminants, scratches, cracks or pinholes in the thin gate oxides of transistors and capacitors. In other words, a local defect is a certain amount of extra or missing material that may cause a difference between the designed and implemented electrical structures [8]. They tend to result in functional faults.

In a DRAM, global defects affect the entire IC, resulting in parametric faults such as timing failures. For the most part, these effects are not repairable by redundancy. These chips must either be rejected or sold at lower speeds. Hence, global defects are not considered in this thesis. Local defects result in specific functional unit failures, many of which can be mitigated using redundancy.

Local defects have three main characteristics: density, size distribution and clustering behaviour. These parameters change between layers and process steps. Manufacturers closely guard this data; however, Table 2.4 shows approximate relative defect densities between CMOS layers.

The size of the defects will have a significant impact on their final effects. The Ferris-Prabhu distribution [17] is often used to describe the probability density function for defect sizes. This probability function is shown in Figure 2.5. The independent variable is multiples of $\lambda$, where $2\lambda$ is the minimum feature size of the

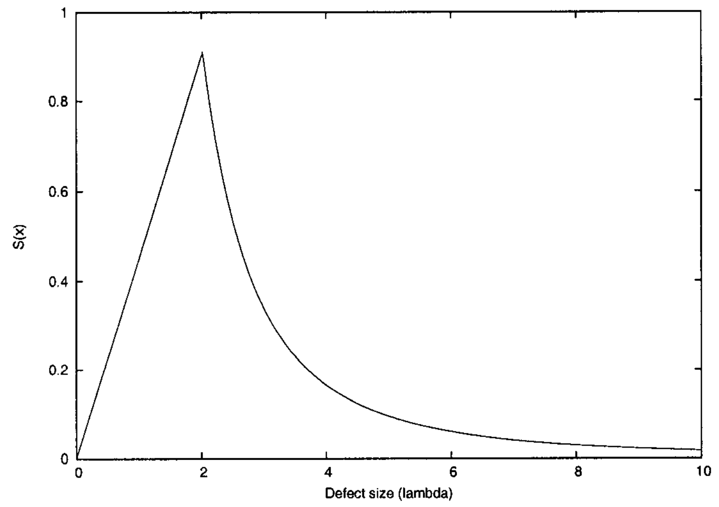


Figure 2.5: Typical defect size distribution function [3]

technology. There are two reasons for the sharp increase of smaller defects. First, any mechanism that creates large defects is likely to create many small defects at the same time. Second, the ambient air is continuously filtered, so most particles larger than a specific threshold will be removed.

Defects also tend to cluster instead of having a completely random spatial distribution on a wafer. Quantitative explanations are unknown, however, many defect mechanisms, such as edge handling sensitivity, micro-cracks and equipment disturbance, give an intuitive explanation for the existence of clustering [8]. Clustering does not cause a significant electrical effect, but can be significant when calculating yield based on defect data.

The importance of a defect is determined by whether or not it affects the behaviour of an IC [3]. If it affects the IC's behaviour, it causes a fault. Basically, a fault is the measurable effect of a defect. Several abstraction levels are defined for faults. More common ones are shown in Table 2.5. In this thesis, the system level fault model is focused on, specifically faults of the memory core or faults that can be mapped onto memory core faults. These would be cell, row, column and cell cluster faults, plus address decoder, word-line driver, sense-amplifier or other faults that manifest themselves as faults of groups of cells and can be eliminated by

| Abstraction | Fault | Types |
|---|---|---|
| Engineering | Architectural | Blocks |
| System | Behavioral | Modules |
| Logic | Functional | Gates |
| Circuit | Electrical | Devices |
| Symbolic | Geometrical | Primitives |
| Physical | Process | Incongruities |

Table 2.5: Fault Modeling Abstraction

redundancy.

The presence of a fault decreases yield. Yield models are used to estimate yield and to characterize a design's sensitivity to defects. Yield models can be characterized according to their fidelity, complexity and dimensionality [18]. Fidelity is the accuracy with which yield can be predicted compared to the actual fabricated yield; complexity is the accuracy in predicting yield loss mechanisms, and dimensionality is the number of features used in the model. These features are [3]:

- IC area
- Defect density
- Spatial distribution of defects on wafer (clustering)
- Defect size distributions
- Global disturbances on wafers
- Layout information
- Technological process information

I use the binomial model for all yield calculations [8]. This model takes into account the IC area, defect density and defect size distribution. It is a simplistic model that tends to give pessimistic yield estimates [8]. For the purpose of this thesis, fidelity and complexity are not very relevant since the yield model is used to compare redundancy schemes assuming they are manufactured with the same technology process under the same conditions. As mentioned earlier, global disturbances are considered "die-killers" and will not be considered. The binomial yield model has the basic form:

$$Y = e^{-A_c D_o} \tag{2.1}$$

where $A_c$ is the critical area and $D_o$ is the defect density. The critical area is the area of a design susceptible to defects of a certain size. The critical area depends on the size of the defect. For example, a defect could be some extra metal of a certain size that could cause a bridge fault. Only those areas where wires or other components are close enough together to be bridged are critical.

## 2.7 Content Addressable Memories

This thesis takes a narrower view of what is traditionally considered a content addressable memory, or CAM. For the purpose of this work, a CAM refers to a compare array. The compare array along with its associated data array is referred to as an "associative memory."

An associative memory determines its addressing based on already stored data, rather than an address location [4]. For example, an associative memory storing colours would return "blue" for "sky" or "purple" for "eggplant." To perform this association, an associative memory is composed of two parts: a compare array (the CAM) and a data array, as shown in Figure 2.6. The compare array compares incoming data with all of its entries in parallel. If a match is found, a "hit" occurs and a word-line in the data array is activated. The data array operates identically to a conventional random access memory, however, the word-lines are controlled by the compare array instead of address decoders. In some cases, more than one hit in the compare array may be possible and arbitration logic is typically included to select which one word-line of the data array to activate. Alternatively, logic can be added to prevent multiple identical entries from being written, thus preventing multiple matches.

The CAM array operates by comparing, in parallel, incoming data with every

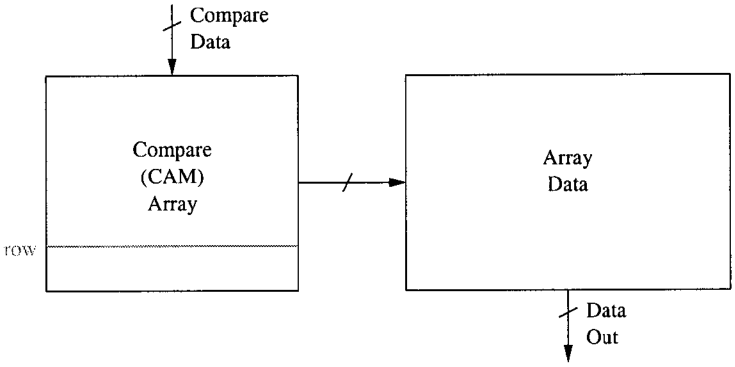


Figure 2.6: Associative memory CAM and data arrays [4]

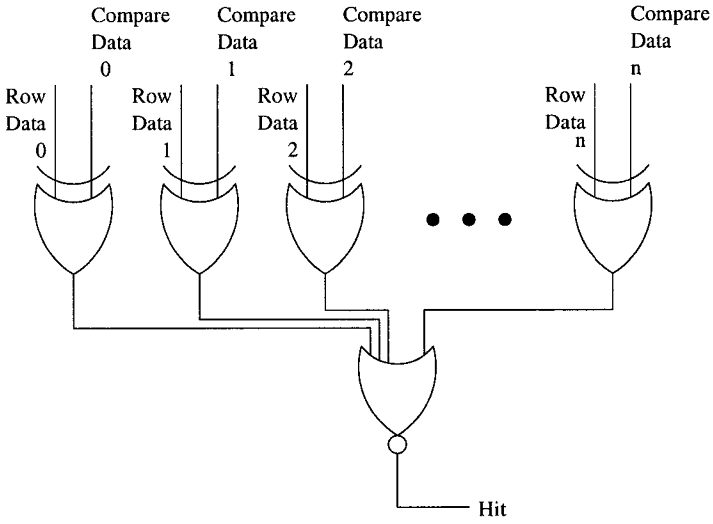


Figure 2.7: Compare function for each CAM row entry [4]

entry it has stored. Functionally, this occurs by performing an XOR operation for each bit and feeding the output into a wide NOR gate. If any bit does not match, the XOR output goes to "1" and the NOR output goes to "0" indicating a mismatch for that entry. This is shown in Figure 2.7.

The XOR operation is built as an integral part of each cell and the wide NOR gate makes use of dynamic logic. Figure 2.8 shows a static CAM cell with the built-in XOR operation. Any type of random access memory cell can be modified for compare operations. The match line connects to all cells in the entry and, in most implementations, is pre-charged high before a compare operation. If any bit in the

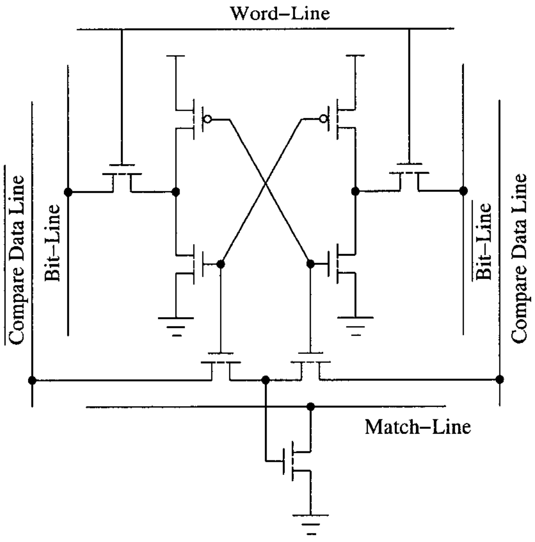


Figure 2.8: SRAM-based CAM cell

entry does not match, it is pulled low, providing the functionality of the wide NOR gate from Figure 2.7.

A CAM cell may also be dynamic. Like a dynamic RAM cell, the dynamic CAM cell must be periodically refreshed to retain its stored value. Also, like the dynamic RAM, it requires many fewer transistors than its static variant, and is therefore, much more dense and cost effective. A schematic is shown in Figure 2.9.

A ternary CAM (tCAM) operates like a regular binary CAM, but can also store a *don't care* value. If a cell contains a *don't care*, it will return a match for either compared value. The tCAM operates by using two connected 1-bit storage cells, as shown in Figure 2.9. The CAM cell has three states, stored on its two storage nodes as follows:

| | |
|---|---|
| 01 | (data = 0) |
| 10 | (data = 1) |
| 00 | (*don't care*) |

It also is possible to store a "11" value, however, then that cell will always return a non-match. This may be useful for marking that there is no data in that specific CAM word.

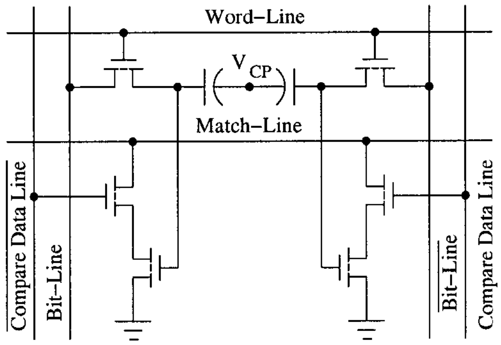


Figure 2.9: DRAM-based ternary CAM cell [5]

## 2.8 Associative Repair

Associative repair operates by accessing a CAM in parallel with the memory array. If the addressed location contains a fault, the faulty address is matched in the CAM and the data from the associated data array is placed onto the data pins in place of the data from the regular memory array.

Associative repair has several advantages over other repair mechanisms. It has completely transparent operation because the replacement memory is much smaller and has an access time on the order of a tenth of the main memory [15]. Also, all entries in the CAM are compared in parallel, so these two added together do not add any significant overhead to the DRAM access time. The CAM and replacement memory are small so have little power consumption compared to the main memory and together have approximately a 5% area overhead [15].

Associative repair operates as shown in Figure 2.10. An incoming memory address is sent to the CAM and the main memory in parallel. If the CAM matches an address, it may shut off (depending on the implementation) access to the main memory while the secondary memory is accessed.

Associative redundancies seem to have first appeared in radiation hardened SRAMs for space applications [19, 20]. The first occurrence was an iterative approach where the memory array was split into equal size blocks [19]. If a block contains a fault, any accesses within that block are redirected to another memory.

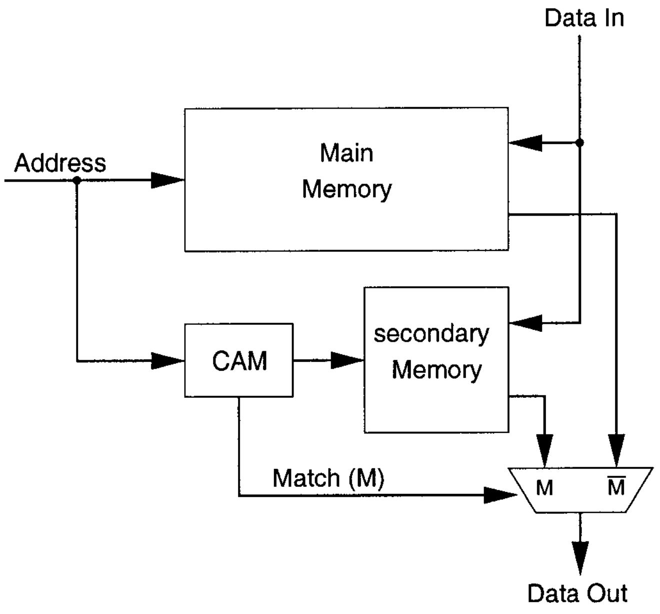


Figure 2.10: Associative repair memory block diagram

The bits that address the block are replaced, using the associative memory, with bits to address an equal size block in the redundant memory. This redundant memory can also be split into smaller blocks that are replaced, and so forth.

Another scheme for radiation hardened SRAMs, known as associative cache redundancy, was proposed independently several years later. This scheme uses a CAM to store the entire address and replaces single words [20]. The associated data array contains the replaced words.

Two researchers, Jien-Chung Lo and Jung H. Kim, have discussed the possible cache memory mapping schemes: fully associative, direct-mapped, associative set and associative multiple, in [21]. Unlike the iterative approach mentioned above, their approach uses triple modular redundancy (TMR) to guarantee that the compare array matches correctly. Each scheme operates as would be expected from memory hierarchies and cache design. It was found that the fully associative approach is the most expensive due to the wider match array, but does allow for less CAM entries because any entry is free to map any location in the primary memory address space. Direct-mapped and associative multiple (without TMR) require approximately the same amount of hardware complexity, however, direct-mapping offers no flexibility for replacement because once a tag address in fixed in the CAM, no more repair is possible for the memory locations with the same tag. Associative multiple (multiple copies of direct-mapped) yields better spare memory utilization for cluster memory cell failures. They determined that the set associative scheme is the most cost-effective approach.

# Chapter 3

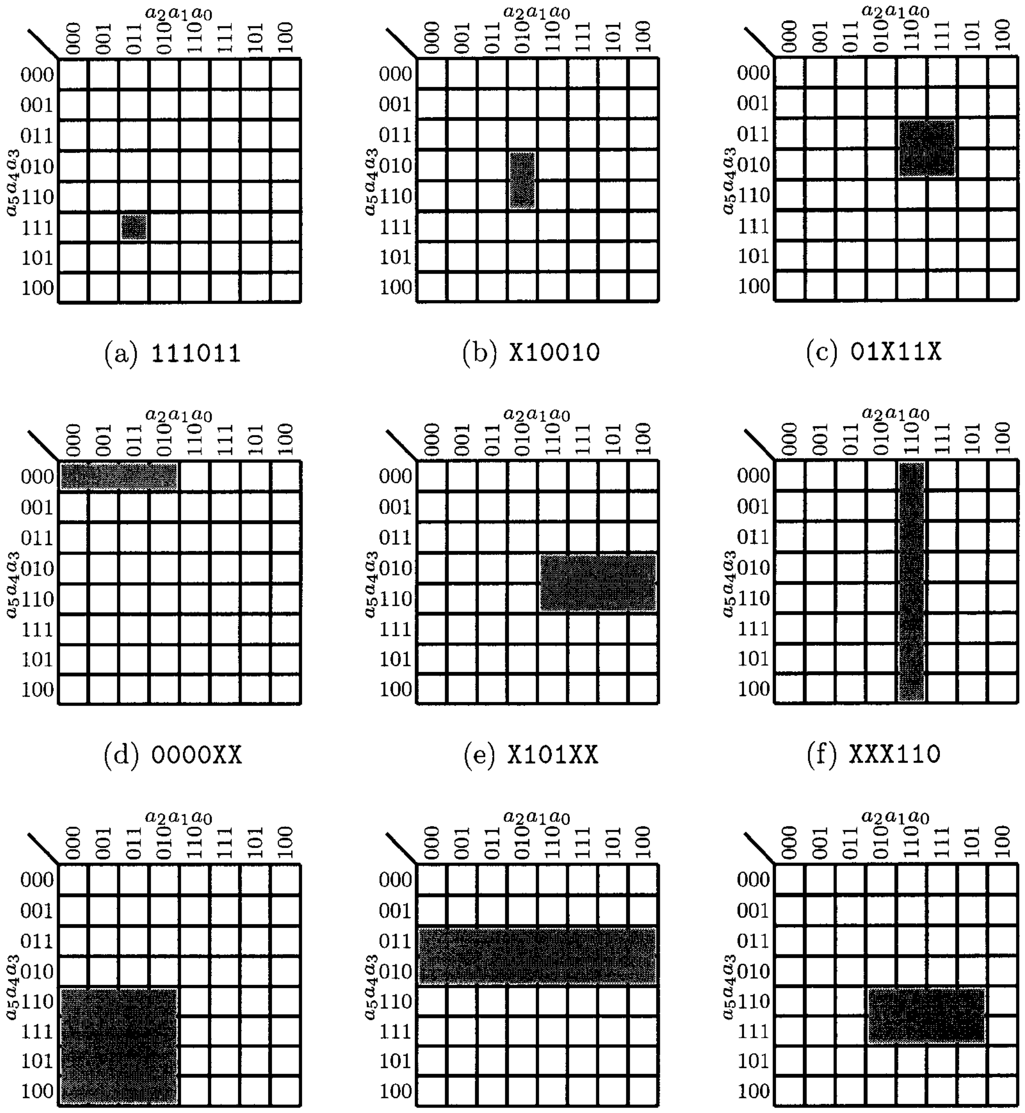
# Ternary CAM Redundancy

## 3.1 Overview

Ternary CAM redundancy/repair operates in much the same way as binary associative repair, as discussed in Section 2.8; however, the addition of the ternary CAM's *don't care* state enables compression of certain groups of words with bad cells. As with associative redundancy, words are replaced because they are the smallest addressable quanta of data in a memory. Consider the 8 row by 8 word memories shown in Figure 3.1. Words with bad bits are shown in grey. CAM entries for marking the bad bits are shown below the memories.

As can be seen, a single word in the CAM can be used to mark large, regular groupings of words with faulty cells in the main memory. It is even possible, in rare cases, that bad, non-contiguous groupings of words could be marked. Although groupings of bad words will not often fall nicely along easily marked addresses, using Gray code to address words does mitigate this problem somewhat. The row and column parts of the address each increase in Gray code order instead of a linear progression. This can be achieved by simply re-ordering the lines output from the row and column decoders. Notice that the entire address is not considered to be a Gray code; the row and column components in the address form a set of two Gray code values.

The Gray code property of having only a single bit differing between adjacent addresses ensures that any area that is two steps wide can be marked with a single



(a) 111011

(b) X10010

(c) 01X11X

(d) 0000XX

(e) X101XX

(f) XXX110

(g) 1XX0XX

(h) 01XXXX

(i) 11X—

Figure 3.1: 8 row by 8 word memory organization with bad words

entry using a *don't care.* If the grids in Figure 3.1 were numbered using regular binary addresses, cases (b) and (e) would require either two CAM entries or the marking of a larger area than necessary. However, bad areas larger than two steps wide, such as grouping (i), can still cause problems.

Modifying the column and row decoders in the DRAM so that addresses increase using a Gray code should not impact any functionality, such as the ability to do page mode. Instead of words being transferred in order (in terms of location), they will instead be pulled in a more "random" order. Since, addresses are sent by the memory controller, even during page transfers and other high speed operations, the location of subsequent addresses should not be a concern. An alternative is to leave the primary addresses as usual, but Gray encode incoming addresses as they enter the redundant path. Now the redundancy mechanism will have a "virtual" address that it matches on to replace words in physical primary memory addresses.

In [12], I discussed the possibility of adding XOR terms for marking groups of faulty cells that do not fall along conveniently markable addresses. Most of the usefulness of employing XOR terms is eliminated by the single-bit difference in Gray code. The elimination of XOR terms simplifies the logic and shrinks the area required for the redundancy scheme, so the trade-off is justified. Large groupings of bad words, such as case (i) will have to be split across multiple CAM entries.

Two possible methods of storing data from bad words are discussed in the following sections. I call these *associative indirect* and *associative direct* ternary CAM redundancy.

## 3.2 Associative Indirect

The associative indirect variant is the slower of the two designs, but it is more flexible in terms of fault coverage and requires fewer entries in the CAM array for the same number of replacements.

This variant is composed of four main components, plus several multiplexers. The components are the associative memory, bit selector, adder and secondary (re-

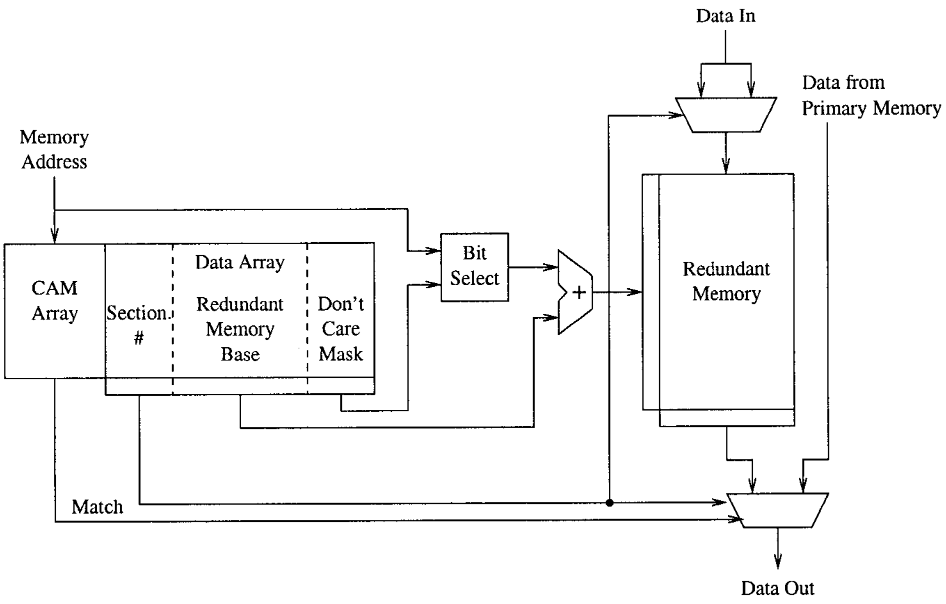
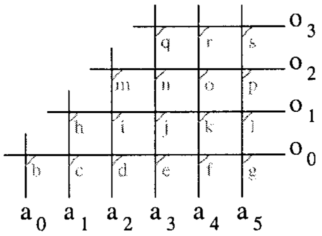


Figure 3.2: Associative indirect tCAM redundancy block diagram

dundant) memory. A block diagram is shown in Figure 3.2.

The associative memory is non-volatile or loaded from non-volatile storage and programmed based on manufacturing test data. The compare array is ternary and contains addresses of faulty cells in the main memory core. Whenever possible, *don't care* bits are used to reduce the number of CAM entries. The associated data array contains two or three values. The first is the optional section number. Modern DRAMs tend to split the storage of words into multiple sections. The section number allows only faulty sub-words to be stored, instead of the entire word, reducing memory requirements. For example, the 1-Gbit DRAM discussed later is a 32Mb×32 device. It is composed of four banks with two sections each. Each section in the bank stores half of the 32 bit word. The chance of both halves being faulty is minuscule. Therefore, only the 16 bits from one section need be replaced. Note that this introduces the restriction that, unless there is a redundant path for each section, all parts of a word cannot be replaced. The multiplexer at the top of the diagram selects which portion of the word is stored in the redundant memory array. A similar set of multiplexers at the bottom reconstruct the data



$b : m_0$<br>
$c : \overline{m_0} \cdot m_1$<br>
$d : \overline{m_0} \cdot \overline{m_1} \cdot m_2$<br>
$e : \overline{m_0} \cdot \overline{m_1} \cdot \overline{m_2} \cdot m_3$<br>
$f : \overline{m_0} \cdot \overline{m_1} \cdot \overline{m_2} \cdot \overline{m_3} \cdot m_4$<br>
$g : \overline{m_0} \cdot \overline{m_1} \cdot \overline{m_2} \cdot \overline{m_3} \cdot \overline{m_4} \cdot m_5$<br>
$h : m_0 \cdot m_1$<br>
$i : m_0 \cdot \overline{m_1} \cdot m_2$<br>
$j : m_0 \cdot \overline{m_1} \cdot \overline{m_2} \cdot m_3$<br>
$k : m_0 \cdot \overline{m_1} \cdot \overline{m_2} \cdot \overline{m_3} \cdot m_4$<br>
$l : m_0 \cdot \overline{m_1} \cdot \overline{m_2} \cdot \overline{m_3} \cdot \overline{m_4} \cdot m_5$<br>
$m : m_0 \cdot m_1 \cdot m_2$<br>
$n : m_0 \cdot m_1 \cdot \overline{m_2} \cdot m_3$<br>
$o : m_0 \cdot m_1 \cdot \overline{m_2} \cdot \overline{m_3} \cdot m_4$<br>
$p : m_0 \cdot m_1 \cdot \overline{m_2} \cdot \overline{m_3} \cdot \overline{m_4} \cdot m_5$<br>
$q : m_0 \cdot m_1 \cdot m_2 \cdot m_3$<br>
$r : m_0 \cdot m_1 \cdot m_2 \cdot \overline{m_3} \cdot m_4$<br>
$s : m_0 \cdot m_1 \cdot m_2 \cdot \overline{m_3} \cdot \overline{m_4} \cdot m_5$

Figure 3.3: Cross-bar shifter with switch equations (Associative Indirect).

word from the stored and working main memory portions.

The second value in the associated data array is a base memory address for the redundant memory. The third value is a mask of the *don't care* bits (if the address stored in the CAM is 01X1X0, the mask will contain 001010).

The next component is a bit selector. The selector's job is to extract the bits marked as *don't care* from an incoming memory address. It does this by ANDing the *don't care* mask with the incoming address then moving the masked bits to the lowest significant bits. For example, assume that the address 011100 is sent in. It matches with the 01X1X0 CAM entry. The mask value of 001010 is sent to the selector which outputs an offset of 000010. The least significant '10' are the input address bits corresponding to the two *don't care* bits in the mask.

The simplest, and fastest selector is a cross-bar. A cross-bar with a six-bit address (main memory) and a 4-bit offset is shown in Figure 3.3. Each of the letters '$b$' through '$s$' represents a switch controlled by the equation on the right side of the figure. The '$m$' values represent the mask bits stored in the associative memory's data array.

The third component adds the produced offset with the base address stored in the associated data array. This produces the address of the replaced word in the redundant memory. Although, at first glance, the requirement for an adder may seem to make the associative indirect method very slow, in fact, the adder can be as simple as an OR gate for each shifter output bit. The CAM programmer has full control of where in the secondary memory to place redundant words. Careful placement of redundant groupings will greatly reduce the required adder logic. The base for single word failures can be placed anywhere in the secondary memory array. The base for double word failures can be placed at any even address. The base for four-word failure clusters should point to any address that is a multiple of four. And so on. This ensures that there will never be a carry-out signal as at least one of each pair of bits entering the adder will be a zero.

The **match** signal indicates whether an incoming address matches an entry in the CAM array. If so, it can turn off access to the main memory and set the output multiplexer so that data is sent from the redundant memory instead of the main memory. The section value will place the correct portion of a word in the correct place (if sub-words are being stored), or determine which portion of the word should be stored in the redundant memory.

Associative indirect redundancy has the advantage of having a dense redundant memory with all stored words consolidated together. This makes it easy for the redundant memory to be shrunk as the DRAM process matures and defect densities decrease and still keep the increased yield for a small number of defects.

## 3.3 Associative Direct

I developed the associative direct variant before I decided to use a no-carry adder in the associative indirect method. I believed there was a possibility that associative indirect would not be fast enough to complete its operations in the necessary DRAM access time. Chapter 5 will show that my concern was unfounded.

Associative direct ternary CAM redundancy sacrifices flexibility and some fault coverage for greater speed by removing the adder and redundant DRAM access.

An associative direct redundancy block diagram is shown in Figure 3.4. It is composed of an associative memory, a bit selector and several multiplexers.

The compare array of the associative memory is identical to the associative indirect case. It is ternary and matches on the addresses of faulty cells. The one difference is that the number of *don't care bits* is limited. The design in Figure 3.4

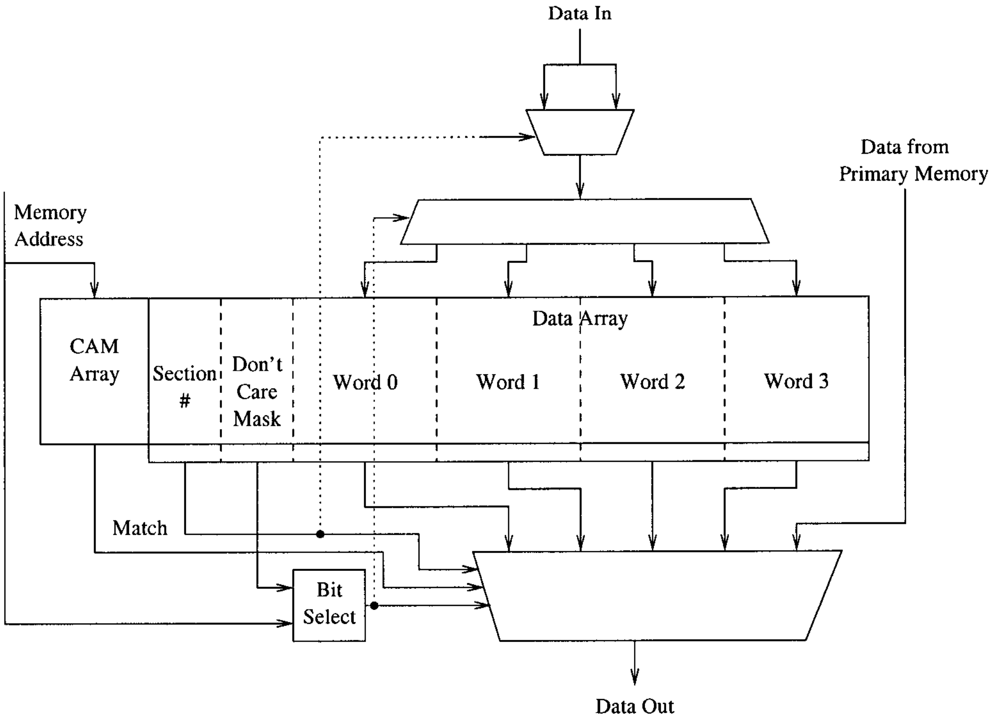
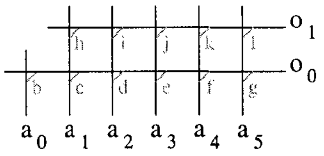


Figure 3.4: Associative direct tCAM redundancy block diagram

is limited to two *don't cares.* However, the associated data array is quite different. Each word line has two parts, a non-volatile part and a volatile part. The non-volatile part (programmed along the the match array entries during manufacturing test) holds the section number and *don't care* mask. Both of these are identical to the associative indirect case. The difference is the volatile part. This contains a certain number of spaces for storing data words; four in the design I analyzed. This is why the number of *don't care* bits is limited to two. If there were to be eight places for words, up to three *don't care* bits would be allowed.

The bit selector operates exactly as before; however, it only outputs one bit for each allowed *don't care* bit. These bits select which of the word locations are written to or read from.

The top multiplexer in the diagram selects the sub-word to be stored. The large demultiplexer underneath it places it in the correct word. The bottom multiplexer selects which word (or sub-word) is read.



$b : m_0$
$c : \overline{m_0} \cdot m_1$
$d : \overline{m_0} \cdot \overline{m_1} \cdot m_2$
$e : \overline{m_0} \cdot \overline{m_1} \cdot \overline{m_2} \cdot m_3$
$f : \overline{m_0} \cdot \overline{m_1} \cdot \overline{m_2} \cdot \overline{m_3} \cdot m_4$
$g : \overline{m_0} \cdot \overline{m_1} \cdot \overline{m_2} \cdot \overline{m_3} \cdot \overline{m_4} \cdot m_5$

$h : m_0 \cdot m_1$
$i : m_0 \cdot \overline{m_1} \cdot m_2$
$j : m_0 \cdot \overline{m_1} \cdot \overline{m_2} \cdot m_3$
$k : m_0 \cdot \overline{m_1} \cdot \overline{m_2} \cdot \overline{m_3} \cdot m_4$
$l : m_0 \cdot \overline{m_1} \cdot \overline{m_2} \cdot \overline{m_3} \cdot \overline{m_4} \cdot m_5$

Figure 3.5: Cross-bar shifter with switch equations (Associative Direct).

For small faulty clusters, the associative direct method works well with Gray code addressing, especially for faults that render a 2×2 grouping of cells inoperable. The need for adder circuitry is also removed. Unfortunately, this method can be very inefficient in terms of secondary method usage because, as mentioned earlier, up to 90% of memory failures are single bit failures [16]. Only the first word storage area in the data array will be used in most cases, leading to a much larger CAM than in the associative indirect case.

## 3.4 Multiple Column Access

A common trait of modern DRAMs is that row and column addresses share the same pins. This reduces the pin count and allows high speed operations such as page mode where multiple columns are accessed per row access. In this case, each column is held open until a subsequent column address appears [13].

This offers a challenge and an opportunity to associative redundancy schemes. The simple solution is to have the CAM treat each new column address as an individual memory access. This is fairly expensive in terms of power dissipation because each match line has to be pre-charged for each new input address, and most, if not all, will be discharged without being used.

I propose a more effective solution by splitting the match array into two, as shown in Figure 3.6. After the /RAS signal is asserted , the first match array compares the row address. After the /CAS signal is asserted, only those entries which matched after the /RAS signal will be pre-charged and used in the comparison. The figure shows three of the row entries matching (grey match lines). Therefore, only these

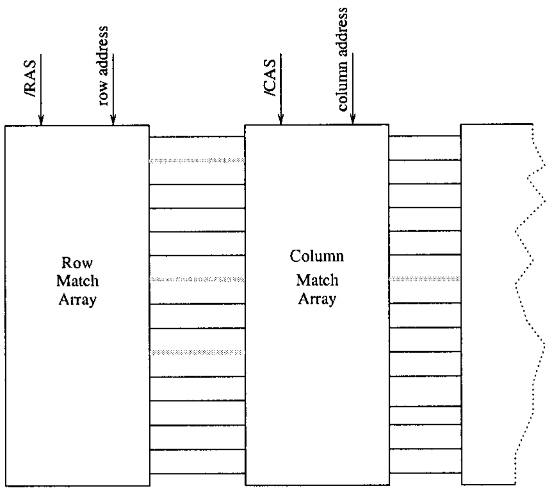


Figure 3.6: Splitting the match array to handle page mode and reduce power consumption

three lines are pre-charged in the column array.

Splitting the match array allows multiple column accesses per row access for page mode and avoids the wasteful power consumption of charging then discharging all of the match lines every cycle. If, for example, the row address is changed once for every four column addresses, on average, the power dissipation will be nearly quartered, even before including the effects of halving the number of cells connected to each match line.

## 3.5 DDR Concerns and Set Associativity

A significant portion of commodity synchronous DRAMs are now DDR. DDR stands for "Double Data Rate," meaning that data is transmitted to and from the DRAM on both the rising and falling edges of the clock signal.

As described, it is difficult for ternary CAM redundancy to operate quickly enough for more than a conventional, single data-rate DRAM (see Chapter 5 for timing simulations). The solution is to move to a set associative design. This gives a separate redundant path for even and odd column address acceses, allowing DDR frequencies to be reached. It is also possible to share the row match array between

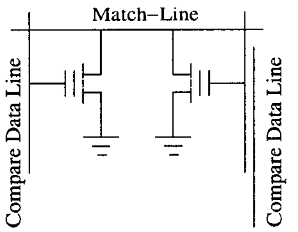


Figure 3.7: Ternary flash CAM cell configuration

sets, leaving the column match component to be set associative.

## 3.6 Implementation Technology

Any implementations chosen for the components of the ternary CAM redundant mechanism must be compatible with DRAM processing technology. There are three reasons for this; adding processing will require additional masks, additional masks will require production changes, and, as noted in Section 2.6, increasing the number of processing steps can increase defect densities. The first two increase manufacturing complexity, increasing cost, and the last decreases yield, also increasing cost.

The redundancy scheme implementation should be as compact as possible to reduce area overhead, once again, leading to reduced cost. The CAM array and, in the case of the associative indirect method, the associated data array should be non-volatile.

Achieving process compatibility and compactness is trivial for the redundant memory array. Non-volatility is not a concern, so a DRAM array can be used. If access speed is an issue, SRAM technology may be employed as the redundant (secondary) memory instead; however, this should not be an issue as the shorter bit-lines and word-lines in the small DRAM array lead to much faster access.

A compact and non-volatile ternary CAM cell is demonstrated in [22]. The cell is composed of two floating gate transistors as shown in Figure 3.7. The threshold voltage of each transistor is programed to be either under $\frac{1}{2}V_{DD}$ or much over $V_{DD}$. For a compare operation, each compare data line is set at either $V_{SS}$ or $V_{DD}$,

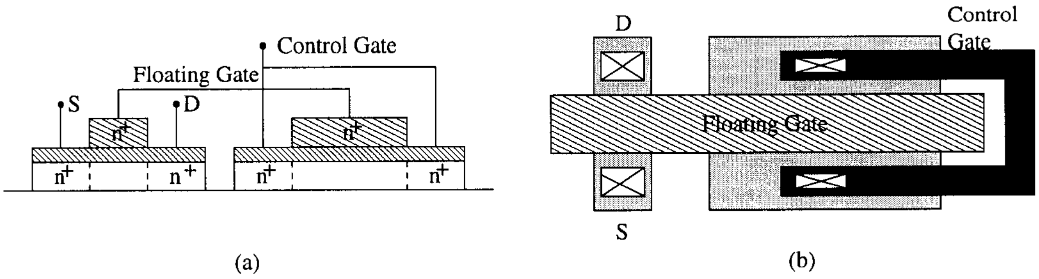


Figure 3.8: Twin ploy-silicon thin-film transistor memory cell [6] a. Cross section b. Top view

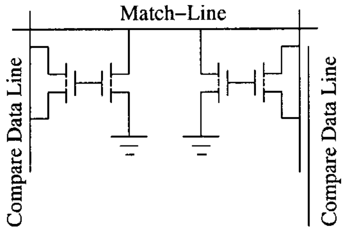


Figure 3.9: Planar non-volatile ternary CAM cell

depending on the value of the data bit being searched for.

Although combined flash-DRAM processes are available, it is unlikely that they are available in commodity DRAMs. Hence, the extra processing steps to produce the required floating gate are undesirable. Instead, a floating gate planar device, as in Figure 3.8 can be constructed. It is composed of two transistors, where the gates of the two transistors are connected to form the floating gate, while the source and drain of the larger transistor are connected to form the control gate [6]. The cell is programmed and erased by Fowler-Nordheim tunneling through the gate oxide of the small transistor. This happens through the smaller transistor because the smaller gate leads to less capacitance and a higher voltage difference. The planar devices can be connected to form a non-volatile ternary CAM as in Figure 3.9.

# Chapter 4

# Yield Comparison

## 4.1 The Yield Model

As mentioned in Section 2.6, the binomial yield model will be used for all calculations in this thesis. This has the basic form:

$$Y = e^{-A_c D_o} \tag{4.1}$$

where $A_c$ is the critical area (area susceptible to defects of a certain size) and $D_o$ is the defect density. The quantity $A_c D_o$ can be simplified to $\lambda$, the average number of faults, leading to:

$$Y = e^{-\lambda} \tag{4.2}$$

This yield model will be combined with various fault models to predict yields and compare redundancy schemes. Single cell, row and column and cluster fault models will be discussed in depth. Faults in the periphery and in redundancy circuitry will be covered briefly.

Although this is a simplistic model, and there are much more accurate models, this model is used because it does not account for clustering. Clustering is usually considered a benefit because it more accurately models the behaviour of single cell faults. This behaviour probably also extends to cluster faults, but it is unclear if and how this mechanism occurs with row and column failures. In order to keep the treatment of all considered fault types equal, the simplistic binomial model, which does not include clustering of faults, is used.

Single cell faults are faults that affect individual memory cells. These can be due to defects in the access transistors, substrate defects affecting the capacitor structure, or even problems with the address decoders not allowing access to certain cells. Row and column faults can be caused by defects in the bit or word-lines or problems with the sense amplifiers, word-line drivers or address decoders. Since these faults should be independent, row and column faults will be treated separately. Clusters may be caused by large defects affecting several cells, defects in the bit or word lines (causing clusters at the far ends), defects in sub-bit-lines and others.

All yield equations and plots for the four fault models can be found in Appendix C.

## 4.2 Equivalent Yield

There is more to the cost equation than just the effectiveness of the redundant system. After all, triple modular redundancy (TMR) is an extremely effective yield mechanism used in many mission critical systems. Unfortunately, TMR has at least a 200% area overhead. This is simply not cost effective for conventional, everyday use. Silicon and packaging costs are also very relevant at the commodity volumes of DRAMs.

Equivalent yield is the yield of a DRAM normalized to the area of a DRAM with no redundancy. Estimates for overheads of conventional redundancy methods are calculated in Appendix B. The overhead of associative direct ternary CAM redundancy depends on the number of entries, while the overhead of associative indirect ternary CAM redundancy depends on the number of entries and the size of the secondary memory array.

A normalized average fault density, expressed as faults per die containing no redundancy, is used as the independent variable for all plots in the following sections. The number of cells, rows and columns in a DRAM change depending on the redundancy mechanism. Therefore, for a given fault density, the absolute number of faults per die will be different. To simplify the presentation of the data, the average number of faults per die on the $x$-axis is normalized to a DRAM with no redundancy.

### 4.2.1 Overhead of Ternary CAM Redundancy

The planar EEPROM devices discussed in Section 3.6 and [6] do not have area information available. However, a similar single-poly EPROM device is discussed in [23]. This EPROM cell is 22 $\mu m$ by 54 $\mu m$ in a 3-$\mu m$ process.

The IBM 16-Mbit DRAM was produced in a 0.5-$\mu m$ process [9]. By scaling the EPROM cell directly, it can be assumed that in a 0.5-$\mu m$ process, a flash cell will be approximately eight times the size of a DRAM cell, while a ternary CAM cell will be approximately 16 times the size of a DRAM cell. Scaling to the 0.16-$\mu m$ process used in the Samsung 1-Gbit DRAM [10], a flash cell is approximately 10 times the area of that of a DRAM cell while a ternary CAM cell is approximately double that. Values of 8 and 16 times, for flash and ternary CAM cells, respectively, will be used for both DRAMs studied.

The associative direct redundant path contains the CAM entry, the section number and *don't care* mask in flash and, in this case, four words in conventional DRAM. There are also the bit selector and multiplexers. There must be one multiplexer for each section. Each must multiplex five words down to one. A five-to-one pass-transistor multiplexer can be constructed from eight transistors, an area of approximately 8 cells. This loads to a redundancy area of:

$$\begin{aligned} A_{ovr} &= v \cdot N_{CAM} \times [b_{addr} \cdot (A_{CAM} + A_{flash}) + b_{sec} \cdot A_{flash} + 4b_{word} \cdot A_{cell}] \\ &\quad + A_{selector} + A_{mux} \\ &= A_{cell}\,[4v \cdot N_{CAM}(6b_{addr} + 2b_{sec} + b_{word}) + T + 8N_s \cdot b_{word}] \end{aligned} \tag{4.3}$$

where $v$ is the wiring overhead, $A_{CAM}$, $A_{flash}$, $A_{cell}$, $A_{selector}$ and $A_{mux}$ are the areas of a CAM cell, flash cell, DRAM cell, the bit selector and the multiplexers, respectively. $N_{CAM}$ is the number of CAM entries, $b_{addr}$, $b_{sec}$ and $b_{word}$ are the numbers of bits in an address, the number of bits to specify the section and the number of bits in a word, respectively. $N_s$ is the number of sections a word is split into and T is the number of transistors in the selector and multiplexers. For the 16-Mbit DRAM, $N_s$ is four while $T$ is 69, while for the 1-Gbit DRAM, $N_s$ is two while $T$ is 273. This assumes that a cross-bar shifter takes up the area of one cell for each cross-bar.

The associative indirect path contains the CAM entry, section number, secondary memory pointer and *don't care* mask in flash, plus the secondary memory array. There is also a bit selector, adder and multiplexer. Assuming that the maximum replaceable area is 64 × 64 words, then the selector will have twelve outputs. This is 198 transistors for the 16-Mbit DRAM and 234 transistors for the 1-Gbit DRAM. Twelve adders (OR gates) are required, resulting in 48 transistors. The multiplexer's are two-to-one, requiring two pass transistors each. This totals to 262 transistors for the 16-Mbit DRAM and 378 transistors for the 1-Gbit DRAM. The area of the redundancy mechanism is:

$$\begin{aligned}
A_{ovr} &= \upsilon \left\{ N_{CAM} \left[ b_{addr} \cdot (A_{CAM} + A_{flash}) + (b_{sec} + b_{red}) A_{flash} \right] \right. \\
&\quad \left. + \frac{N_{red} \cdot b_{word} \cdot A_{cell}}{E_{red}} \right\} + A_{selector} + A_{adder} + A_{mux} \\
&= A_{cell} \cdot \upsilon \left[ 8N_{CAM}(3b_{addr} + b_{sec} + b_{red}) + \frac{N_{red} \cdot b_{word}}{E_{red}} \right. \\
&\quad \left. + T + 2N_s \cdot b_{word} \right] \qquad (4.4)
\end{aligned}$$

where $N_{red}$ is the size of the redundant memory in words and $E_{red}$ is for the overhead of the sense amplifiers and other periphery circuitry in the redundant memory. This is assumed to be 0.67, an approximate average of the cell efficiencies calculated in Appendix B.

Both of the area equations make the conservative assumption that the area required for the periphery is proportional to the number of memory cells on the die. It is more likely that the periphery area is fixed.

For both DRAMs, three amounts of CAM entries for each of the associative repair mechanisms will be used to compare effective yield. In all cases, the wiring overhead, $\upsilon$, is assumed to be 1.02 (2%). This is the same wiring overhead as

| | Host memory size (bits) | CAM entries | Redundant Address bits | Red. mem. size (bits) | overhead (%) |
|---|---|---|---|---|---|
| Associative Direct | 16M | 2K | - | - | 3.91 |
| | | 6K | - | - | 11.73 |
| | | 8.5K | - | - | 16.62 |
| | 1G | 73K | - | - | 3.10 |
| | | 100K | - | - | 4.24 |
| | | 124K | - | - | 5.26 |
| Associative Indirect | 16M | 1.6K | 15 | 32K | 3.96 |
| | | 4.5K | 17 | 128K | 11.60 |
| | | 6.9K | 15 | 32K | 16.51 |
| | 1G | 66K | 15 | 512K | 3.08 |
| | | 72K | 19 | 8M | 4.23 |
| | | 76K | 20 | 16M | 5.23 |

Table 4.1: Selected sizes of associative redundancy schemes for comparison

assumed in [7]. The overheads for these will be approximately equal to the overheads of the conventional memory schemes (see Appendix B). For associative indirect redundancy, I chose to have the number of bits in the redundant memory always be a power of two. This is not a necessity, but it does constrain the possible choices of the number of CAM entries. The chosen amounts are shown in Table 4.1.

### 4.2.2 IBM 16-Mbit DRAM

The three amounts of CAM entries for associative direct ternary CAM redundancies were chosen so that the area overhead came out to approximately the same as the three conventional redundancies. This works out to 2K, 6K and 8.5K CAM entries, giving the same overheads as row and column, ECC and row and column with ECC redundancy, respectively. For associative indirect ternary CAM redundancies, the amounts of CAM entries and the size of the redundant memory array were chosen in an attempt to maximize yield under all four fault models, giving 1.6K CAM entries with 15 bits, 4.5K with 17 bits and 6.9K with 15 bits for row and column, ECC and row and column with ECC redundancies, respectively. Plots for equivalent yield versus fault density for the four fault models are shown in Figures 4.1 through 4.4.

The associative direct redundancy method performs very well for single cell and cluster faults. With 2K CAM entries, associative direct redundancy can handle 8.5 times more single cell faults than row and column redundancy and 5.4 times more single cell faults than ECC redundancy for 50% equivalent yield. With 8.5K CAM entries, 1.05 times more faults per cell can be handled than row and column with ECC redundancy.

An interesting effect is seen for ternary CAM redundancies when dealing with failing columns. All of the other fault types show constant yield/equivalent yield until a threshold is crossed, followed by a quick drop to zero (or not so quick in the cluster fault case). With column faults, the yield/equivalent yield drops linearly until the threshold, where it drops off quickly as expected. The reason for this is that failures in the redundancy mechanism are included in all of the fault models. This effect is only seen with column failures because they are the only fault type that seriously affect the performance of the redundancy method. Other types of failures can be handled gracefully, but with the current design, a column failure may render the entire redundancy mechanism inoperable. The probability of a column failure rendering the entire redundancy mechanism inoperable is proportional to the fault density, leading to the linear decrease in yield, as shown in Figure 4.3.

For cluster faults, associative direct with 2K entries can handle 2.4 times more faults than row and column redundancy, 930 times more faults than ECC and 1.9 times more faults than row and column redundancy with ECC to achieve a 50% equivalent yield.

Unfortunately, associative direct ternary CAM redundancy does not perform as well for row and column faults. Only with prohibitive area overhead can associative direct redundancy improve on the performance of row and column redundancy for row or column failures. On the other hand, associative direct redundancy does perform satisfactorily well compared to ECC.

For single cell faults, the number of faults that can be handled by associative redundancies is proportional to the number of CAM entries. Hence, it is not surprising that the chosen sizes for associative indirect redundancy do not perform as well as the associative direct redundancies of the same area overhead. The graphs would be very different if the size of the redundant memory array had been sacrificed to make room for more CAM entries. The 1.6K CAM entry associative indirect redundancy with 32K of redundant memory still outperforms row and column and ECC redundancies. Unfortunately, 6.9K entries is not enough to outperform row and column redundancy with ECC.

For cluster faults, the relationship between the fault density and the number of CAM entries is more nebulous. The size of the redundant memory array also plays an important part. Both the 1.6K and 4.5K associative indirect redundancies outperform the associative direct redundancies of the same area. However, the 6.9K associative indirect redundancy is starved by its comparatively small redundant memory. Conversely, the large 128-Kbit redundant memory coupled with 4.6K of entries performs extremely well.

Associative indirect redundancy does not struggle as much as associative direct with the row or column failures. Contrary to single cell faults, the number of faults that can be handled is more affected by the size of the redundant memory. In this model, most of the CAM entries are vacant. Unsurprisingly then, the 4.5K case with the large 128-Kbit redundant memory performs extremely well. Even the smaller 1.6K entries with 32 Kbit case outperforms all of the conventional redundancies. For a 50% equivalent yield, the 1.6K CAM case can handle about 1.6 times more faults per row or 1.8 times more faults per column than both row and column and row and column redundancy with ECC.

### 4.2.3 Samsung 1-Gbit DRAM

To match the overheads of conventional redundancies, associative direct redundancies with 73K, 100K and 124K CAM entries and associative indirect redundancies with 66K entries and 15 address bits, 72K entries and 19 bits and 76K and 20 bits are studied. These correspond to row and column, ECC and row and column with ECC redundancy overheads.

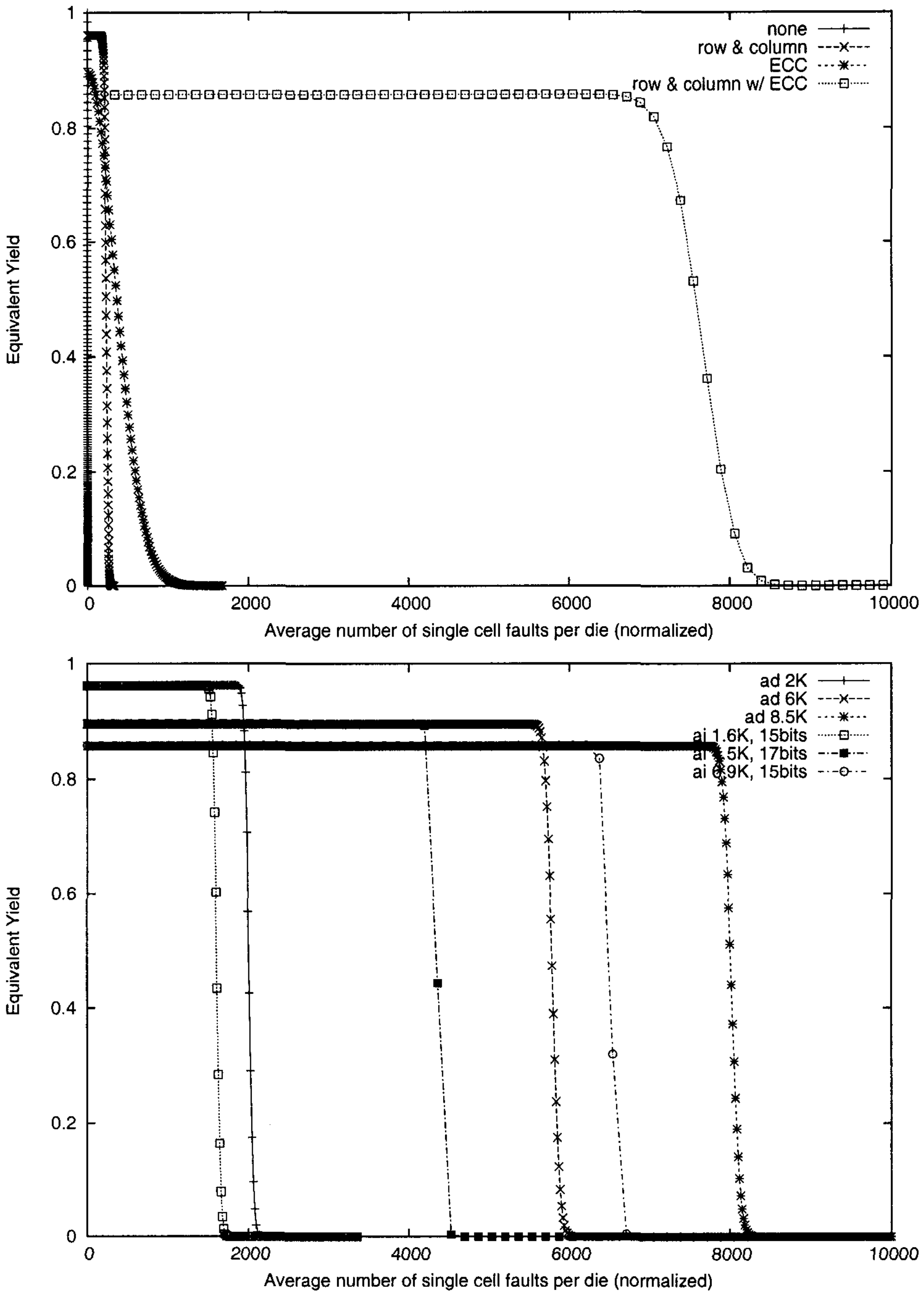


Figure 4.1: Equivalent yield of a 16-Mbit DRAM for a single cell fault model for conventional and associative redundancies.

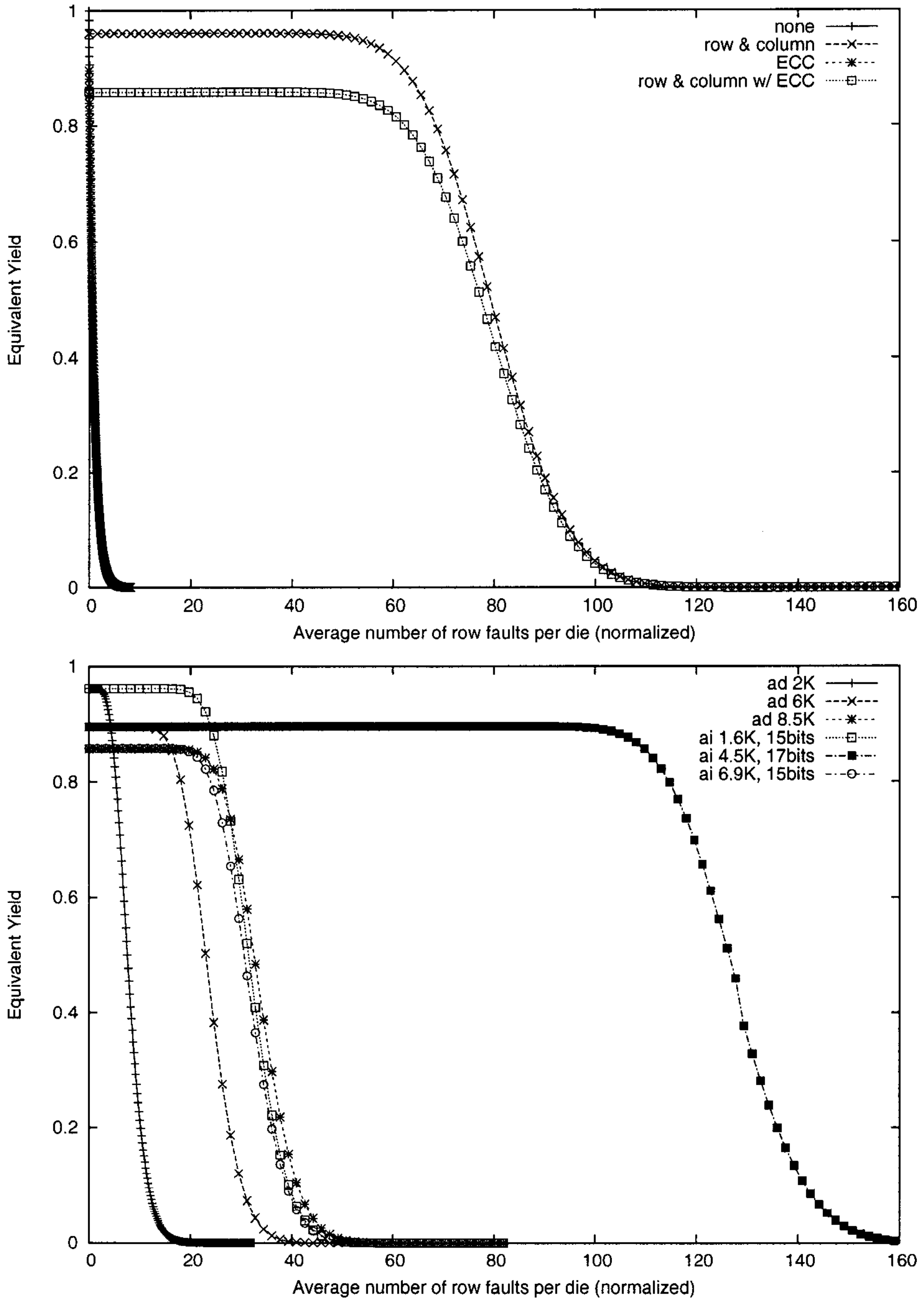


Figure 4.2: Equivalent yield of a 16-Mbit DRAM for a row fault model for conventional and associative redundancies.

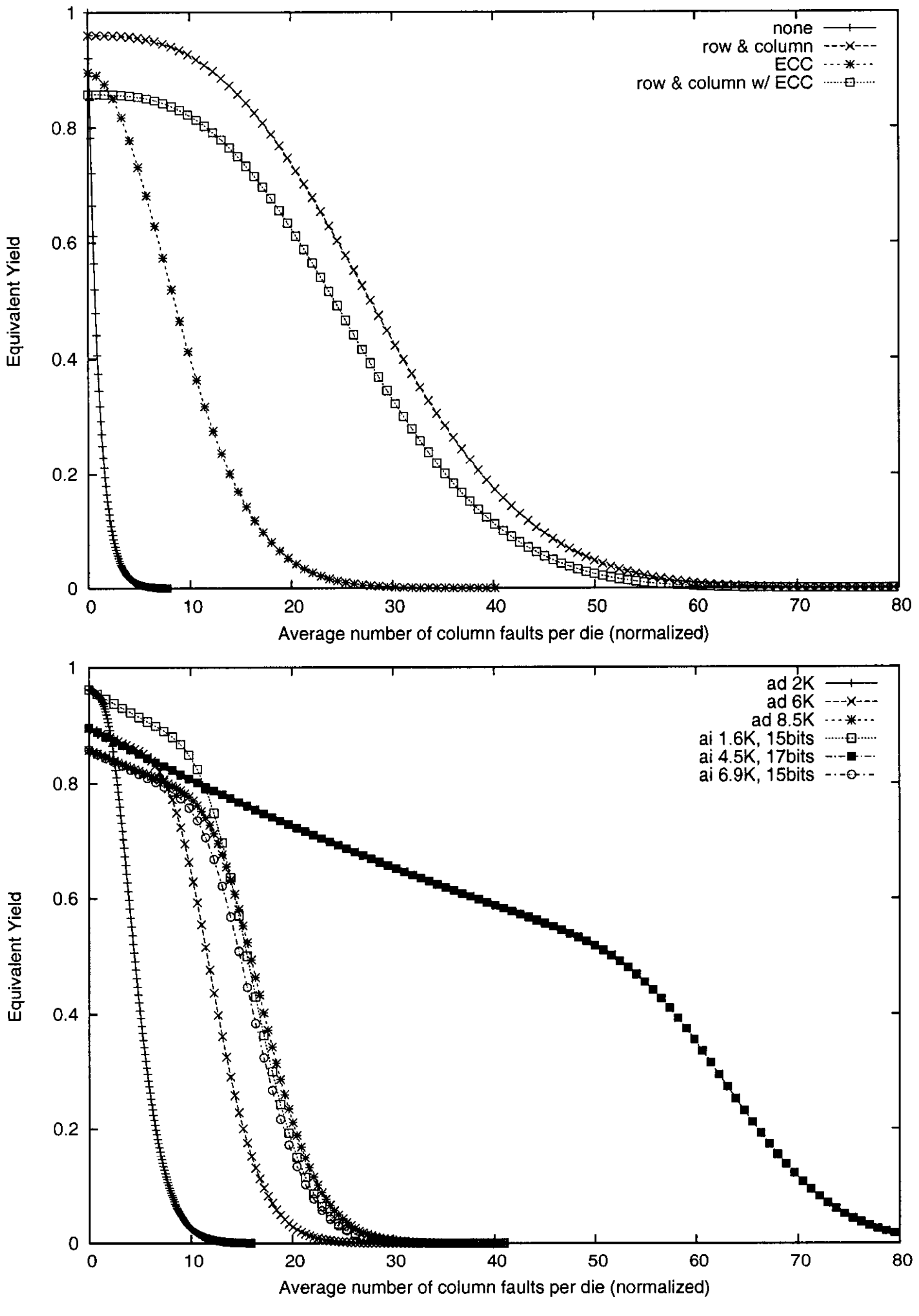


Figure 4.3: Equivalent yield of a 16-Mbit DRAM for a column fault model for conventional and associative redundancies.

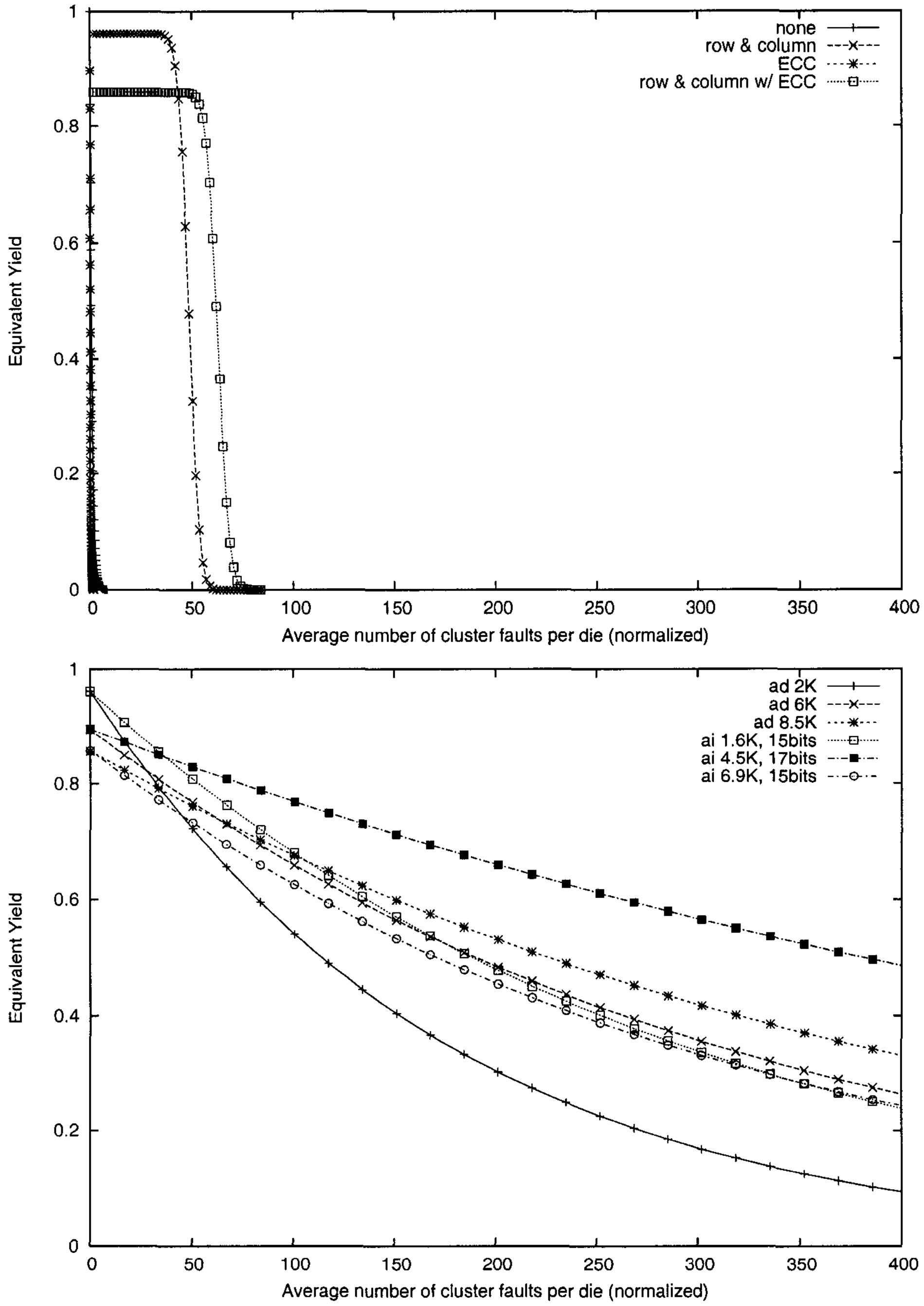


Figure 4.4: Equivalent yield of a 16-Mbit DRAM for a cluster fault model for conventional and associative redundancies.

All three associative direct cases surpass the performance of conventional redundancies for single cell and cluster faults. The smallest, with 73K entries achieves 50% equivalent yield with 53, 45 and 0.96 times more single cell faults than row and column, ECC and row and column with ECC redundancies, respectively. Although row and column redundancy with ECC slightly outperforms a 73K entry associative direct design, it does with 2.1% more area overhead. With cluster faults, the performance of associative redundancies depends on whether the DRAM employs contiguous or interleaved words. For contiguous words, the 50% equivalent yield point is achieved with 10, 19230 and 8 times more cluster faults. For interleaved words, 4.6, 8641, 3.6 times more cluster faults for row and column, ECC and row and column with ECC redundancies, respectively, can be handled and still achieve 50% equivalent yield.

For row and column faults, the outcome is similar to that of the 16-Mbit DRAM. Associative direct ternary CAM redundancy cannot achieve the performance of row and column redundancy. All three associative direct cases achieve greater than 75% of the 50% equivalent yield point of row and column redundancy for row faults, but only about one third of the column faults. The effect of column failures on the redundancy mechanism is still visible, but much less prominent.

The smallest associative indirect case, with 66K CAM entries, is still large enough to perform comparably to all conventional redundancies for single cell and cluster faults. For single cell faults, it reaches 50% equivalent yield with 47, 41 and 0.87 times more faults than the three conventional redundancies. For cluster faults in both, contiguous and interleaved word DRAMs, these numbers are 0.84, 1538 and 0.66 times more faults.

With row faults, the 512-Kbit redundant memory of the 66K entry case is not large enough to pose a challenge for conventional row and column or row and column with ECC redundancies. Increasing the redundant memory to 8 Mbits solves this. The 72K entry with 19 address bits case reaches 50% equivalent yield with 1.2 times more row faults than both row and column and row and column with ECC redundancies.

None of the associative indirect ternary CAM redundancies offer much competition to row and column and row and column with ECC redundancies for row or column failures. The cases studied simply do not have large enough redundant memories. A case with 22 address bits, would drastically change this picture.

## 4.3 Inferences

From this the previous section, several conclusions can be reached.

ECC is only effective for single cell and column faults. It cannot deal with more than one fault in a codeword. It should also be slightly more effective in interleaved DRAMs. For ECC, Only DRAMs with contiguous words were modeled. A contiguous DRAM stores all of the bits of a word together in adjacent cells along a word-line. An interleaved DRAM stores all of the bits from the same bit position in a word in adjacent cells. Because of this, it is much less likely that a cluster fault will affect two cells from the same codeword, thereby improving the effectiveness of ECC repair.

The overhead of ECC when coupled with row and column redundancy is not worth it for most fault types, however, the large performance increase offered for single cell faults makes this option very attractive.

When dealing with cluster faults, the effectiveness of the associative redundancies is somewhat reduced for interleaved DRAMs. This is because associative redundancies are a word-replacement scheme. With interleaving, words are split up, causing a cluster to affect a greater number of words, meaning more words require replacement.

Associative direct ternary CAM redundancy performs very well for single cell and cluster faults. Unfortunately, redundant memory is wasted when replacing single words. This wasted memory cannot be used for another fault. When dealing with row failures, associative direct redundancy performs nearly as well as a conventional redundancy of a comparable area, but this redundancy method is not well suited to

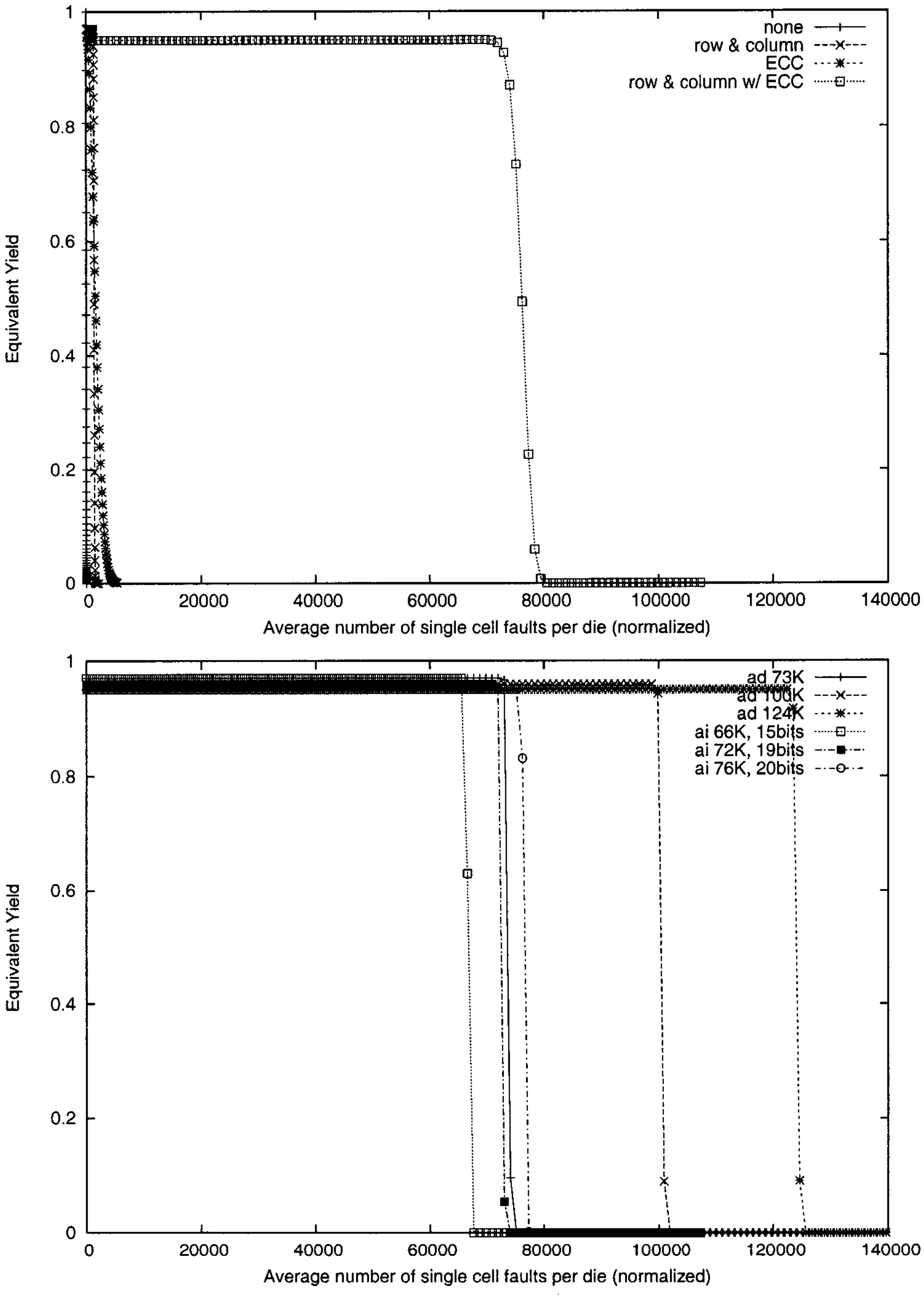


Figure 4.5: Equivalent yield of a 1-Gbit DRAM for a single cell fault model for conventional and associative redundancies.

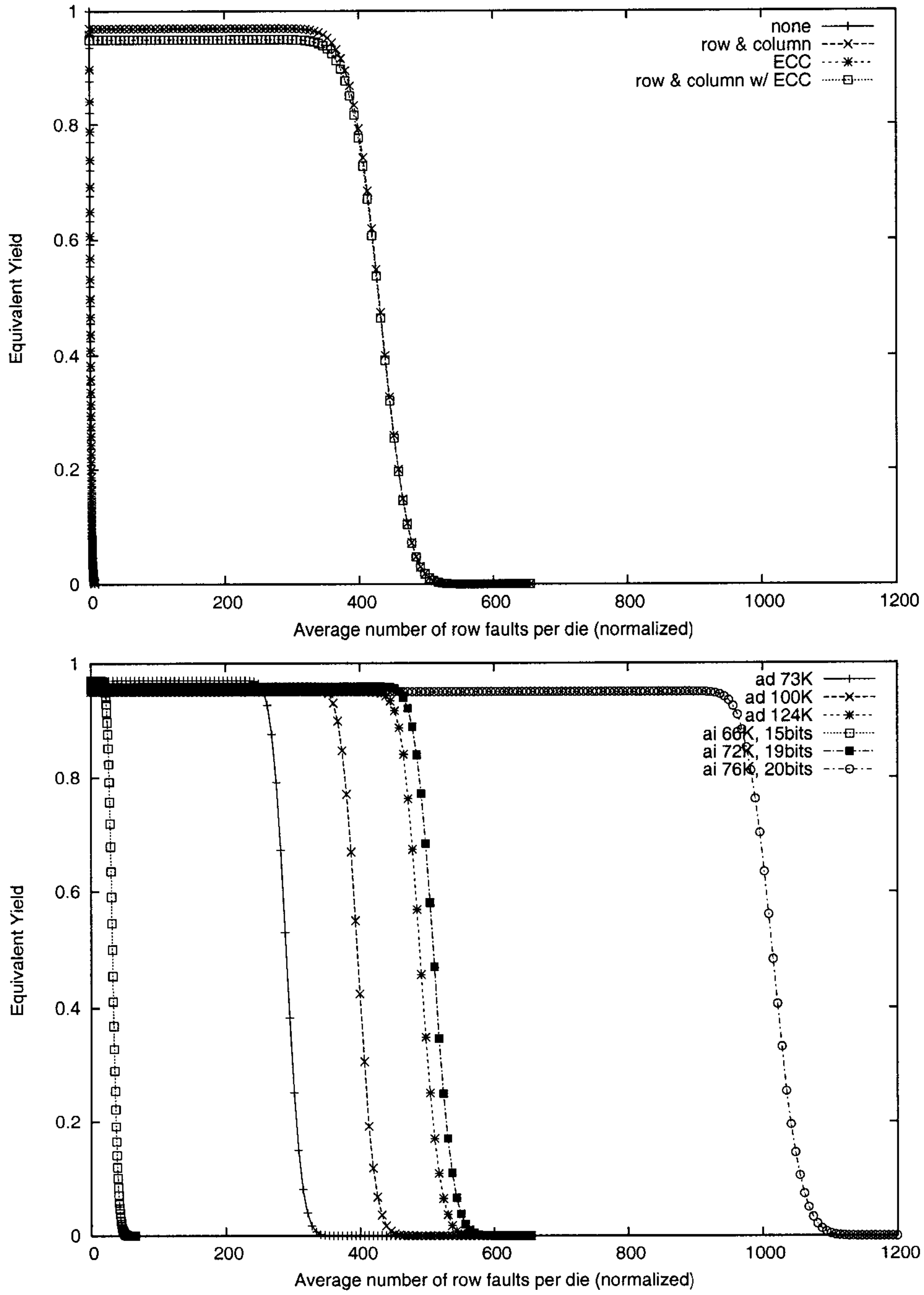


Figure 4.6: Equivalent yield of a 1-Gbit DRAM for a row fault model for conventional and associative redundancies.

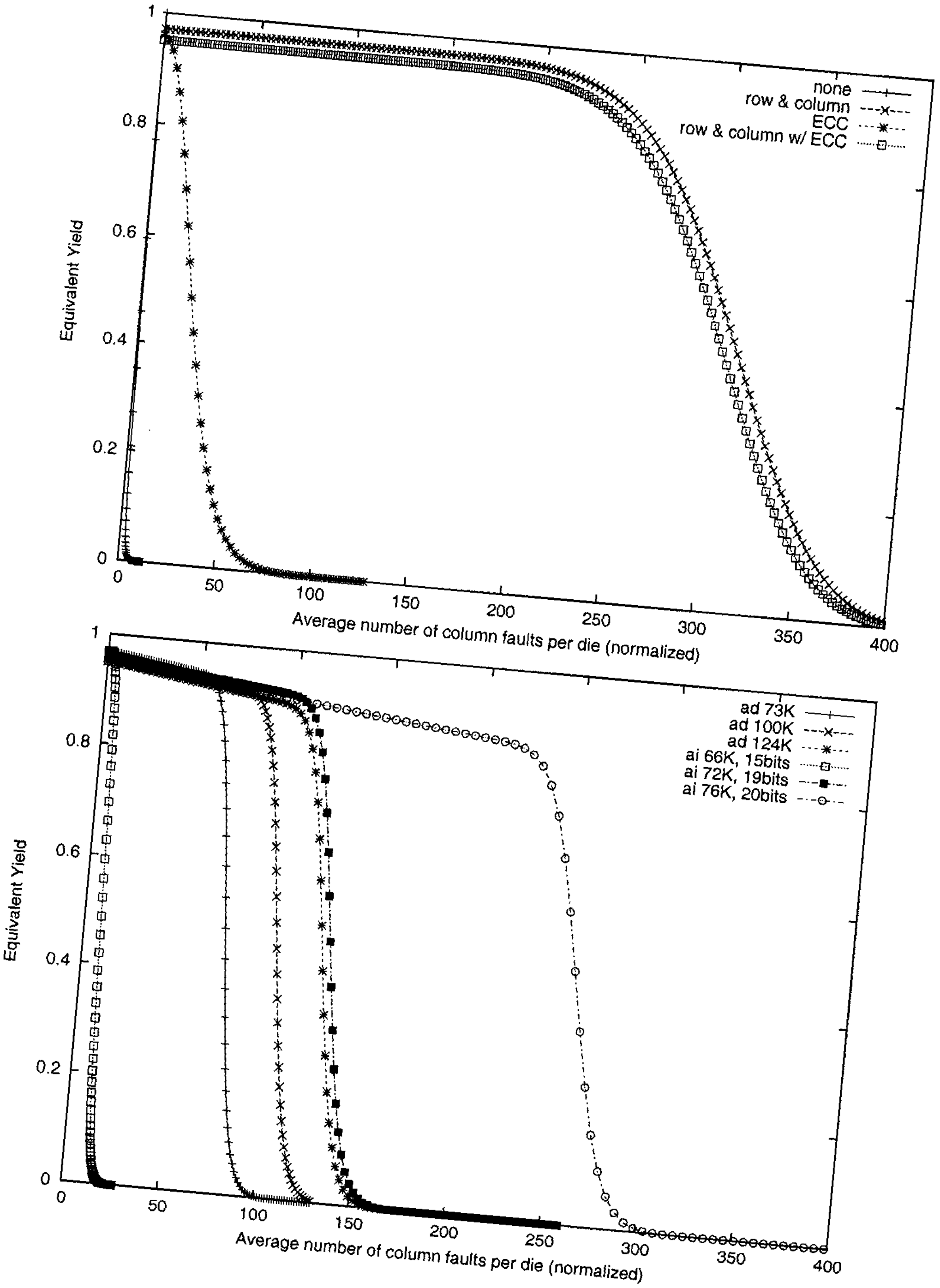


Figure 4.7: Equivalent yield of a 1-Gbit DRAM for a column fault model for conventional and associative redundancies.

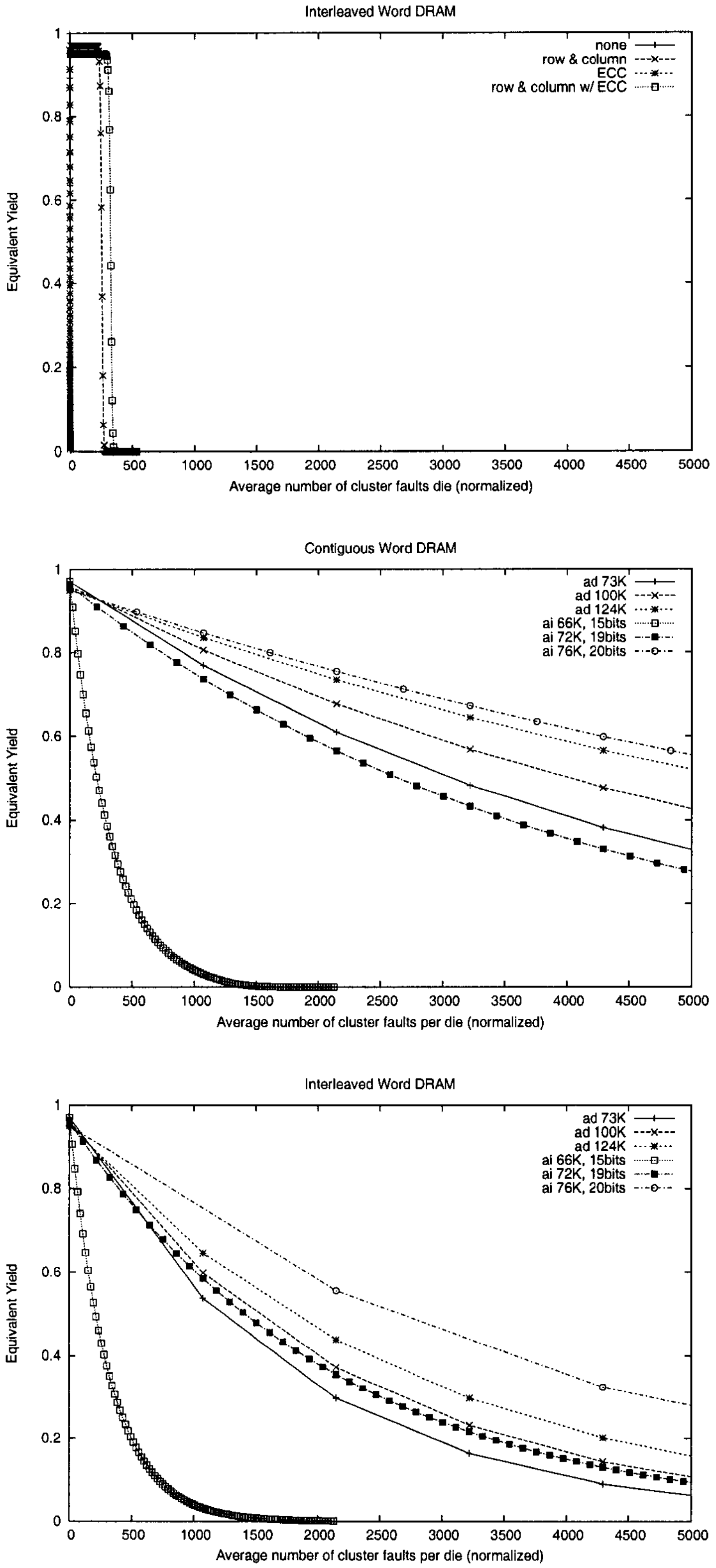


Figure 4.8: Equivalent yield of a 1-Gbit DRAM for a cluster fault model for conventional and associative redundancies.

dealing with column failures.

Associative indirect ternary CAM redundancy offers a large amount of flexibility in tailoring the redundancy to the predominant failure type. In addition, only the amount of redundant memory needed to replace failing word portions is used. The rest can be used for another failure, theoretically, reducing the area overhead.

The equivalent yield plots show that, for the most part, it is advantageous to maximize the size of the redundant memory at the cost of CAM entries.

This would indicate that the best choice for a 16-Mbit DRAM is to have 1.1K CAM entries and 18 address bits. This has an area overhead of 4.06%, less than row and column redundancy, and allows more faults to 50% equivalent yield than row and column redundancy for all fault types. It is only out-performed by row and column with ECC redundancy for single cell faults, a solution that occupies 12.43% more area.

For the 1-Gbit DRAM, such a trade-off is more difficult to find. An associative indirect redundancy with 2K entries and 22 bits will exceed the performance of row and column redundancy in all cases. Unfortunately, it has an overhead of 6.28%, even greater than that of row and column with ECC redundancy. The best trade-off at a comparable overhead to row and column redundancy is 32K CAM entries with 20 bits. At 3.10% overhead, the number of faults that can be handled by all conventional redundancies at 50% equivalent yield is exceeded for row and cluster failures. The number of faults handled by row and column redundancy is also exceeded for single cell faults. This associative indirect redundancy size can also handle approximately five sixths of the column faults that row and column redundancy can.

## 4.4 A Combined Fault Model

In Section 4.2, I compared the performance of the three conventional redundancies and my two associative redundancies using four different fault models. Unfortunately, these faults do not occur independently. In a real fabrication process, all four types of faults will occur on the die. In order to investigate this, performance will be compared with a combined fault model. Using the inferences from Section 4.3, I have chosen the best all-round sizes at the same overhead as row and column redundancy for my two redundancy methods in the two DRAMs. For the 16-Mbit DRAM, this is a 2K entry associative direct tCAM redundancy and a 1.1K entry with a 256-Kbit redundant DRAM associative indirect tCAM redundancy. The 1-Gbit DRAM will have 73K entry associative direct and 32K entry with 16-Mbit redundant DRAM associative indirect tCAM redundancies.

The four fault types are independent, requiring four independent variables and one dependent variable for the yield. Not only is this computationally prohibitive: it is difficult to display the data in a meaningful manner. Instead, I have chosen to fix the ratios of the four fault densities. The number of single cell and cluster faults per cell are equal. Individual fault densities are calculated as follows:

$$\lambda = \frac{2F}{4b + R + C} \tag{4.5}$$

$$\rho = \frac{F}{4b + R + C} \tag{4.6}$$

$$\chi = \frac{F}{4b + R + C} \tag{4.7}$$

$$\psi = \frac{2F}{4b + R + C} \tag{4.8}$$

where $\lambda$ is the number of single cell faults per cell, $\rho$ is the number of row failures per row, $\chi$ is the number of column failures per column and $\psi$ is the number of cluster faults per cell. The values $b$, $R$ and $C$ are the number of bits, row and columns, respectively, in a die without redundancy. The value $F$ is the number of normalized faults per die. The ratios of the four fault types can be trivially adjusted in the equations in Appendix C.5.

This ratio of fault densities does not correspond to a real process. In fact, it is unlikely that the ratios of fault types will remain fixed as the fault density increases. It may be likely that there is a correlation between single cell and cluster faults, but no such relationship exists for row and column failures. The following data is for a completely arbitrary combination of faults.

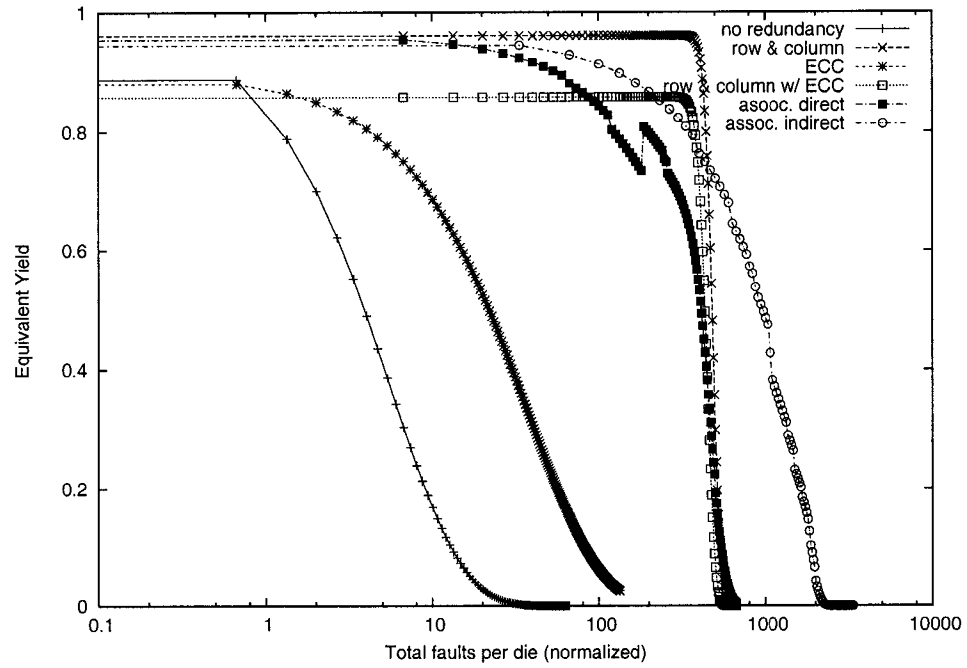


Figure 4.9: Equivalent yield of a 16-Mbit DRAM with various redundancy methods for a combined fault model

### 4.4.1 IBM 16-Mbit DRAM

The 2K entry associative direct scheme has an overhead of 4.73 $\mu$m, or 3.91%. The 1.1K entry with 256-Kbit redundant memory associative indirect scheme has an overhead of 4.91 $\mu$m, or 4.06%. Equivalent yields are shown in Figure 4.9.

The plot shows that for this ratio of faults, when there are less than about 80 faults (40 single cell, 40 cluster, 0.019 row and 0.0096 column faults per die), row and column redundancy offers the best performance. Above this amount, associative indirect is a better choice.

The ternary CAM redundancy plot lines are not smooth. This is because the portions of the CAM entries used for each fault type are calculated by computer algorithm. Non-linearities and jumps are caused by the algorithm shifting CAM entries between fault types.

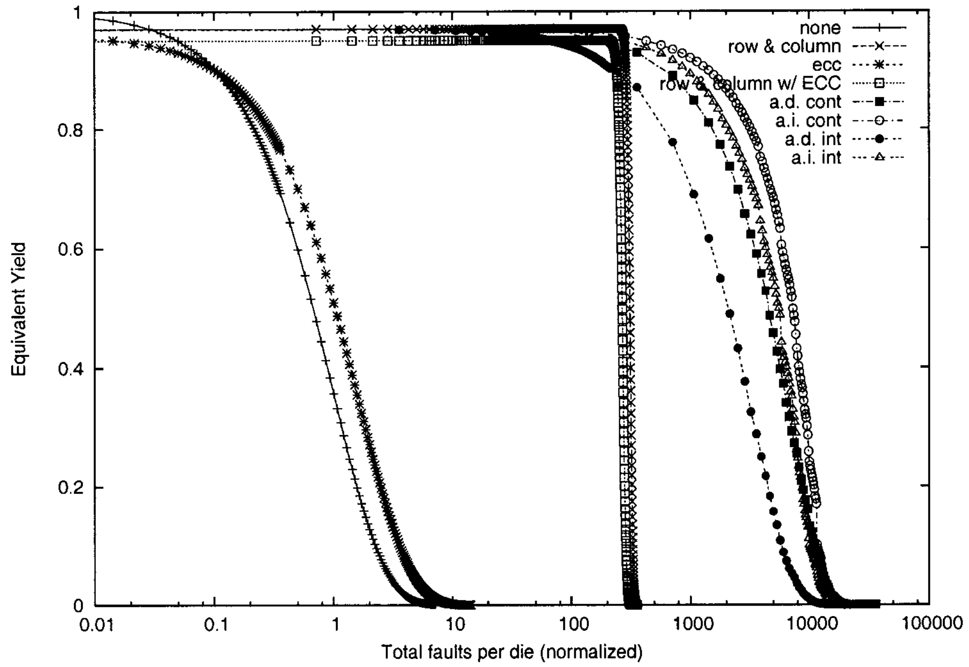


Figure 4.10: Equivalent yield of a 1-Gbit DRAM with various redundancy methods for a combined fault model

### 4.4.2 Samsung 1-Gbit DRAM

Both the 73K entry associative direct scheme and the 32K entry with 16-Mbit redundant DRAM associative indirect scheme have an overhead of 17.11 $mm^2$, or 3.10%. Equivalent yields are shown in Figure 4.10.

The equivalent yield plot is quite different for a 1-Gbit DRAM where there is greater area for entries and redundant words. Except for a very narrow range, both associative redundant schemes are superior to conventional redundancies for this mix of faults.

The plot lines for the ternary CAM associative redundancy methods are much smoother in the 1-Gbit DRAM than the 16-Mbit one. This is because the yield calculations were run several times with adjustments to the algorithm coefficients. Data from each run was then pieced together to give the best yield at every fault density, resulting in a much smoother curve.

# Chapter 5

# System Modelling, Design & Simulation

## 5.1 The Critical Path

In this chapter, I show that ternary CAM redundancy can operate at DRAM page speeds. The critical path of a design is the path within the design that dictates the smallest cycle time at which an entire design can run. It can also be thought of as the worst-case delay over all possible input patterns [24]. In DRAM, the critical path is the time from the column-access strobe signal lowering to the time when the data appears on the output pins or when the data is written into the cells.

Synchronous DRAM latencies are specified with three numbers of the format $x$-$y$-$z$. For instance 3-2-2 is a common specification. The numbers signify the number of cycles for CAS latency, the RAS-to-CAS delay and the RAS pre-charge time [25]. The CAS latency is the number of cycles necessary for data to appear on the output pins after the /CAS signal pin is activated. RAS-to-CAS delay is the number of cycles that must be waited after activation of the /RAS pin before activating the /CAS pin. RAS delay is the number of cycles that must be waited after /RAS pin deactivations and activations.

With 3-2-2, there is a five cycle (3+2) delay from the activation of the /RAS signal. With the most relaxed timing, during a read operation, ternary CAM redundancy must have data available within five cycles. However, multiple /CAS activations may occur for every /RAS activation. This means that data must be available within three cycles. Current SDRAMs operate at speeds up to 200 MHz. To be employed in these DRAMs, ternary CAM redundancy must have a critical path of less than 15 ns.

In order to determine whether reaching this critical path delay is possible, I created two models of the associative indirect method. The first, a VHDL model, was created to determine the design's functionality. A VHDL test-bench wrapper was written. The second model is a schematic in the Cadence tool-set. This was used to determine timing. Unfortunately, an extracted layout is not possible due to the unavailability of a DRAM process.

## 5.2 Modeling and Simulation

I created a VHDL model in order to simulate the functionality of my associative indirect design. A full simulation, using the CAM and redundant memory sizes determined at the end of the last chapter, is not necessary for this purpose and would take a prohibitively long time for a full simulation. Instead, I assumed a 16×4 bit memory in two banks with half of the memory word stored in each bank. The CAMs were given 8 entries and the redundant memory 16 entries (16×2 bits). Although this is excessively large compared to the size of the main memory, it allows for a very high percentage of faults, giving a more complete functional simulation. The offset (see Section 3.2) was set to 3 bits, allowing a maximum fault size of eight memory cells.

The VHDL model was created with 10 files as shown in Figure 5.1. The most prevalent objects denote regular VHDL files, while the other two denote VHDL packages. Standard packages are not included in this figure. The components all correspond to components in Figure 3.2. The file nc_adder.vhdl corresponds to the adder while decode.vhdl and red_array.vhdl form the secondary memory.

The three files dealing with memory were written behaviorally. This is because flash and DRAM devices are difficult to create in VHDL and are not synthesizable. In fact, the flash devices are described as ROMs. This is valid because the flash

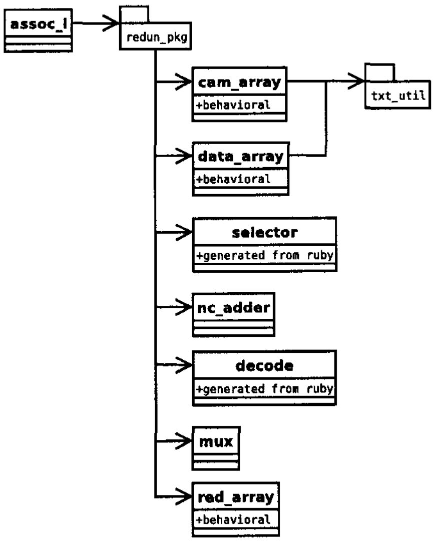


Figure 5.1: VHDL model source file organization

devices are expected to be programmed during manufacturing test, and not by the user. The txt_util.vhdl package is a small collection of functions used to convert between VHDL types for reading and writing data files. In this case, the functions are used to load the ROMs (flash).

Two of the components, the bit selector and decoder, are very regular components that depend on the sizes of other components. Unfortunately, this regularity is not expressible with the VHDL `generate` iterator due to the many combinations of the input signals. Instead, I wrote a Ruby script to produce the VHDL descriptions [26]. The VHDL and Ruby files can be found in Appendix E.

The test-bench is composed of two files, plus the associative indirect model files, as shown in Figure 5.2. The lfsr_generic.vhdl file is from a university course in ASIC design [27]. In this case, the txt_util.vhdl package is used to record the generated input to and output from the model. The VHDL source can be found in Appendix F.

The VHDL simulation showed that my design is functional and can handle page operations.

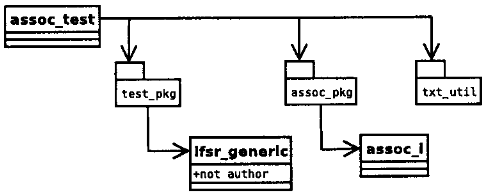


Figure 5.2: VHDL testbench source file organization

## 5.3 Schematic and Simulation

In order to determine if my redundancy method is fast enough to include in commodity DRAM, I performed timing simulations based on schematics in Cadence. Preparation of the schematics was done in three stages. The first was a direct port of the functional VHDL mini-design to a schematic. The second stage consisted of scaling the schematic up to 32K CAM entries with 25 address bits per entry and 20 bits of secondary address space. I estimated capacitances and resistances of the long metal lines in the memory arrays and added these loads to the design for the third stage.

Simulation was done with TSMC's CMOSp18 180-nm technology. This is a logic process, therefore, comparisons with DRAMs implemented in memory processes is somewhat suspect. Memory processes are optimized for density while logic processes are optimized for speed. The Samsung 1-Gbit DRAM I've used for comparison is implemented in a 0.16-$\mu$m technology [10]. The 2002 International Technology Roadmap for Semiconductors states that current DRAM production is at 0.10-$\mu$m [28]. The smaller feature sizes should give the DRAMs a speed advantage, therefore, if my design performs favorably at 0.18 $\mu$m, it should also perform well enough after a process shrink and switch to a DRAM process.

I automated the conversion from VHDL to schematic as much as possible. The two RTL components, the bit selector and non-carry adder, where synthesized in Synopsys and imported into Cadence. This could not be done with the behavioral memory elements. However, the memory elements are very regular, making it easy for me to create SKILL code that would generate the match and data arrays from input files [29]. The SKILL code, after capacitive and resistive loads are added, is included in Appendix G.

Timing and control circuitry was created manually. These do not have equivalents in VHDL. This circuitry is for pre-charge signals and timing signals, ensuring that pre-charging lines are not shorted to ground and that operations occur in the correct order.

In Section 3.4, I explain how and why only column match-lines whose associated row match-line already matched are pre-charged on the /CAS signal. Unfortunately, due to timing constraints, the associated data array cannot be pre-charged only if a column matched. Instead, the data array is pre-charged along with the column-match array on /CAS if there was a match in the row-match array. This is undesirable due to power consumption, but a satisfactory compromise since it is not pre-charged on every DRAM access and only discharged if necessary.

The direct port of the mini functional design was carried out in order to determine if my schematics were functional. Although the timing data is not very useful, it shows an upper bound of 825 MHz with a CAS latency of 2 (provided a 2×16 bit DRAM with an access time of 1.21 ns can be found).

Scaling the mini-design to the full 34200 entries was obviously not practical. Instead, the design was kept at eight entries and the capacitive load of the other 34192 entries was calculated and added. Because I was only scaling the design, resistances were not estimated at this time. Simulations showed that with this many entries, the match-line pre-charge transistors could not be turned on fast enough. Therefore, I split the design into 32 1K entry components. Even so, I had to lengthen the duration of the pre-charge signal. Row match-line pre-charge now occurs in 400 ps and the row-match operation takes 1.92 ns. Due to the higher capacitance of the OR gates enabling column match-line pre-charge, this pre-charge is slightly slower at 450 ps.

Although the associated data array could be pre-charged more than quickly enough, with 1024 cells attached to each bit-line, the small cell transistors could not pull down the bit-lines fast enough. I split each 1024 entry data array into four 256 entry arrays. Now the data lines could be pulled down fast enough, plus this offers greater control over pre-charging the data arrays and an easy match-line arbitration logic. Only data arrays attached to rows that matched need to be pre-charged, thus saving power.

Match-line arbitration logic is frequently used in CAMs to ensure that, if more than one entry matches the incoming data, only one associated data is accessed. In associative redundancy, this can occur in cases where a single cell belongs to two fault types, such as a faulty row crossing a faulty column. My original design assumed this would be taken care of by splitting up groups of faulty cells, ensuring that no two match-lines would ever match, during programming of the flash devices after manufacturing test. Instead, the encoding of the four match signals from each of the data arrays into a two bit multiplexer select ensures that only one data signal will be chosen, provided that the two or more matched signals are in the same group of 1024 CAM entries, but in different groups of 256. As before, this constraint, along with a cell being present in not more than four faults (an extremely rare, if not impossible, case) can be ensured by the programming equipment.

I made one more major design decision. Samsung's 1-Gbit DRAM consists of four banks of 256 Mbits [10]. One of the advantages of associative redundancy is that it is a pooled redundancy, able to replace any faulty cells anywhere on the die. Although a significant area can be recovered by removing conventional redundancy methods from a DRAM, it is dubious whether or not this area can be collected into a single area large enough for 32K CAM entries and a 16-Mbit dynamic memory. Instead, I opted for a four-way set associative design with one set per bank. This better reflects the way associative redundancy would be used in DDR or DDR-II memory and allows for the use of four smaller, and therefore faster, redundant memories instead of one larger one at the cost of some yield. I still performed the simulations with 25 address bits and 20 secondary memory address bits instead of

| | Length<br>($\mu$m) | Resistance/□<br>($\Omega$/□) | Resistance/cell<br>($\Omega$) |
|---|---|---|---|
| Compare Line<br>(metal 1) | 1.08 | 0.078 | 0.366 |
| Match Line<br>(metal 2) | 5.21 | 0.076 | 1.41 |

| | Length over<br>active<br>($\mu$m) | Length over<br>poly<br>($\mu$m) | Capacitance<br>to active<br>(fF) | Capacitance<br>to poly<br>(fF) | Capacitance<br>/cell<br>(fF) |
|---|---|---|---|---|---|
| Compare Line<br>(metal 1) | 0.89 | 0.19 | 0.246 | 0.249 | 0.266 |
| Match Line<br>(metal 2) | 3.84 | 1.37 | 0.218 | 0.219 | 1.14 |

Table 5.1: Capacitive and resistive loads for memory arrays

the 23 and 18 that should have been used for a four-way design.

The simulation showed that this design has a RAS-to-CAS latency of 1.9 ns and a maximum frequency of 268 MHz. A 256K×16 bit redundant DRAM with an access time of 3.5 ns would allow for a CAS latency of 2 cycles.

Adding the estimated capacitive and resistive loads slowed the RAS-to-CAS latency to 3.27 ns. These loads are shown in Table 5.1. This latency can be seen in Figure 5.3. The top plot shows the (inverted) pre-charge and discharge of the the match lines. This takes approximately 1.5 ns from the row-access-strobe lowering. The bottom plot shows the row-access-strobe signal and the four (inverted) match-occurred signals for each group of 256 match lines. These are pre-charge enable signals for the associated data array. The charging and discharging of these lines limits the RAS-to-CAS latency.

The frequency was slowed to 222 MHz at a CAS latency of two cycles after adding the capacitive and resistive loads. As show in in Figure 5.4, the column match, associated data access and calculation of the redundant memory address take 4.5 ns. Only the data line that takes the longest to discharge is shown.

The Samsung DRAM has a cycle time of 143 MHz meaning that at CAS-2, the redundant memory's access time would only have to be about 9 ns (with 0.5 ns for

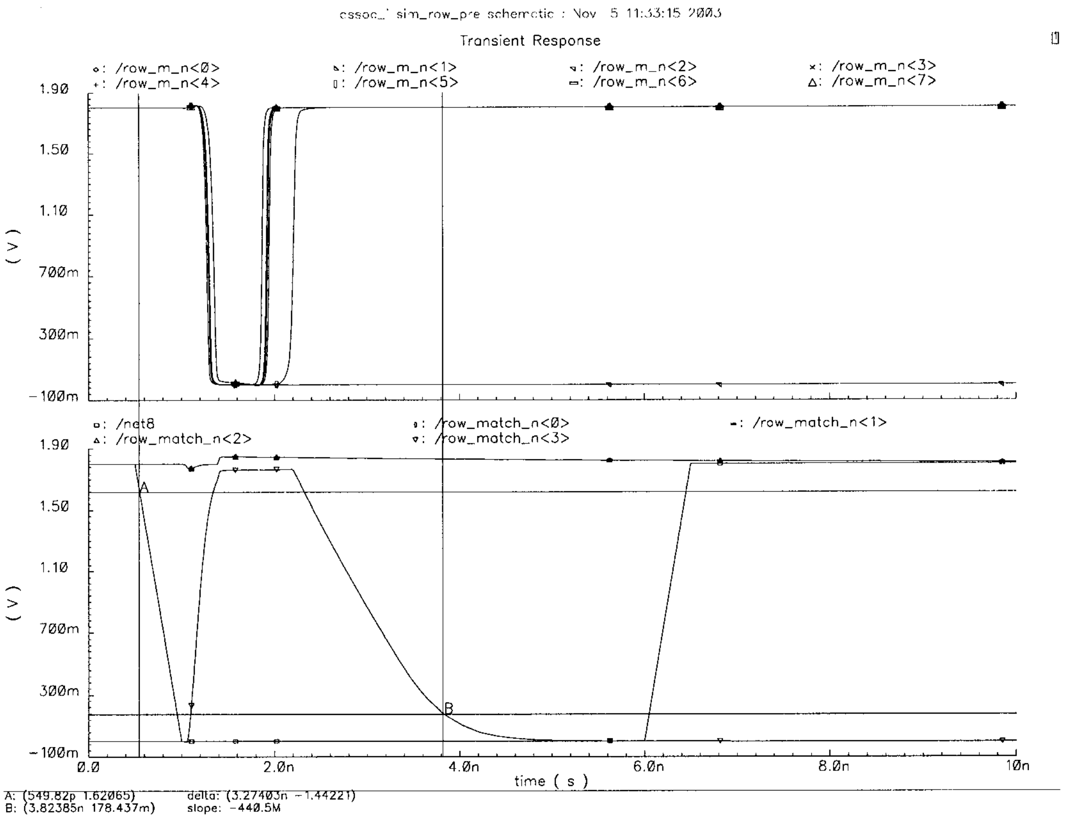
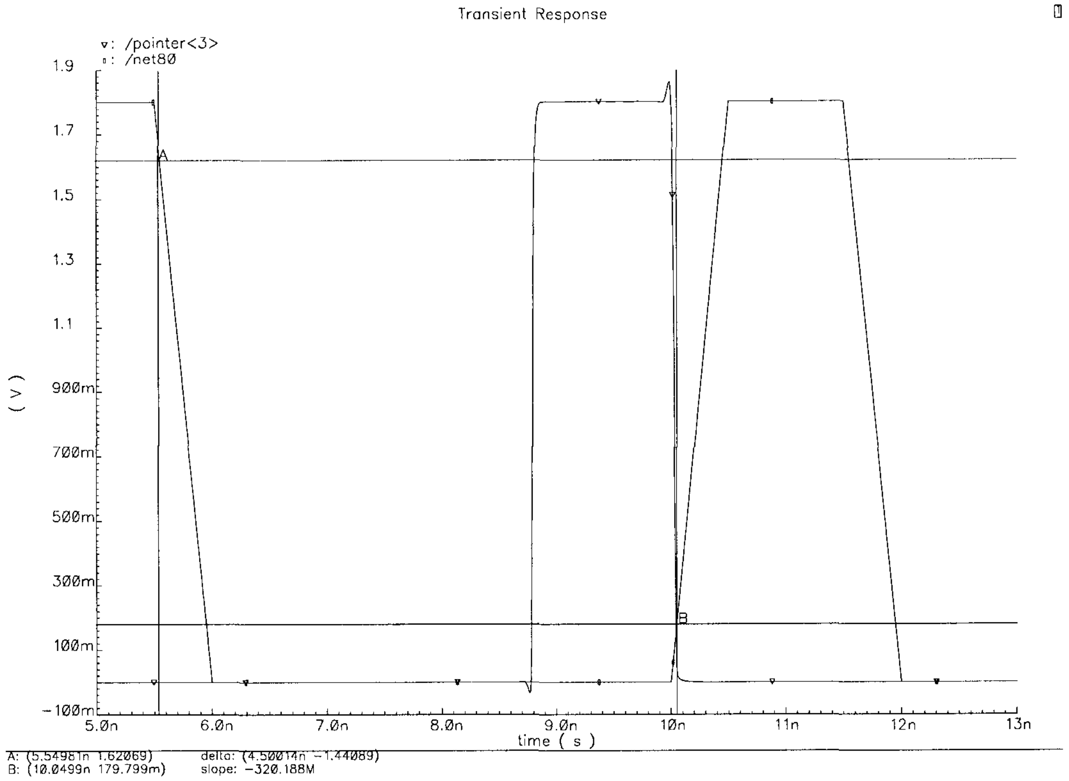


Figure 5.3: Timing waveforms showing row access latency.



```
A: (5.54981n 1.62069)      delta: (4.50014n −1.44089)
B: (10.0499n 179.799m)     slope: −320.188M
```

Figure 5.4: Timing waveforms showing column access latency.

the required multiplexing). Current SDRAMs at 200 MHz are also not a problem. The lowest latency DRAM currently on the market (216 MHz - 433 MHz DDR) has a latency of 9.26 ns, and is presumably manufactured at 0.1 $\mu$m. The highest frequency DRAM operates at 250 MHz (500 MHz DDR) and has a CAS latency of 3 (12 ns). My associative redundancy design operates more that fast enough for inclusion into these designs, even when modelled in an IC process with a much larger minimum feature size.

# Chapter 6

# Conclusions

## 6.1 Synopsis

In this work, I have presented two designs for associative redundancy using ternary content addressable memory in DRAMs. The first of these, which I call associative direct redundancy, uses the compare array of the CAM to match on faulty cell addresses and stores the replacement data in the associated data array. The second, associative indirect redundancy, again uses the compare array to match on faulty cell addresses, but it uses the associated data array to store a base memory address for a secondary memory array. It calculates the memory address of the replacement data cells in the secondary memory by adding the base memory address to the shifted *don't care* bits. This is a slightly slower, but much more flexible approach.

The designs were given four constraints: they must be implementable in a standard DRAM process; occupy the same or less area than typically used redundancies; increase DRAM yield and not reduce the operating frequency or functionality of the DRAM in which they are included.

I have made suggestions on how my designs can be implemented in typical DRAM processes with planar flash cells for non-volatile memory. Both designs allow easy tailoring of the amount of entries and, hence, the area they occupy.

Using a binomial yield model and four fault models: single cell, row, column and cluster faults, I compared the yields and equivalent yields (yield normalized to area overhead) of DRAMs using my designs with DRAMs using three conventional redundancy schemes: row and column redundancy, error correction coding and row and column redundancy with error correction coding. As expected, my indirect associative design outperformed the associative direct design. The associative indirect design, using comparable area, also outperformed all conventional redundancy types, except when dealing with column faults in some cases. Dealing with a column fault requires a large number of CAM entries and a large portion of the secondary memory. In a design where column faults are a limiting factor, the designer may want to consider a hybrid column/associative redundancy design.

I performed the timing simulation in Cadence with a 0.18-$\mu$m logic process using the slower and more flexible associative indirect design. My simulations showed that a DRAM including associative indirect redundancy should be able to operate at a frequency of 222 MHz and a /CAS latency of two cycles. This is more than fast enough for the 143 MHz, 0.16-$\mu$m 1-Gbit DRAM the schematic design was geared for. It also shows much promise for inclusion into currently shipping DRAMs topping out at 250 MHz (500 MHz DDR) with a /CAS latency of three cycles or 216 MHz (433 MHz DDR) with a /CAS latency of two cycles.

## 6.2 Advantages

Ternary CAM associative redundancy's primary advantage is in its flexibility. Diverse types of faults can be replaced, all with the identical CAM entries. Pooled redundancy resources mean that the number of entries or the amount of redundant cells can be scaled down as yield ramps up. The amount of associativity can be set, allowing redundancy on a per chip, bank, section, book or other basis. Set associativity should allow for inclusion into DDR and other high-speed DRAMs. Section-level associativity allows more than one sub-word to be replaced in DRAMs that split data words across several sections. At lower associativity, the designer can choose whether to replace entire words, or sub-words. For embedded DRAMs, the two-stage match array can be removed to allow SRAM-style access.

## 6.3 Liabilities

Associative redundancy's two functional liabilities both deal with column failures. A memory column failing due to a fault can be replaced, but it takes a large number of redundant memory cells since every word (or sub-word depending on the design) that the column traverses must be replaced. A failing column of cells is treated like a failing column of words. As I mentioned before, a manufacturer encountering a large amount of failing columns may want to consider a hybrid column/associative redundancy approach.

In a completely pooled design, a column failing in a redundancy system array or a fault in the control logic renders the entire redundancy mechanism non-functional. This could be seen from the linear decrease in yield/equivalent yield in the column failure models. In the original design, such a failure was catastrophic; however, the need to break up the match and data arrays into several parts due to capacitance greatly mitigates this problem. A column failure or control system (timing and control circuitry) fault in a redundancy mechanism greatly reduces the amount of redundancy, but does not, in most cases, result in a non-fully-functional DRAM.

From a design perspective, the design choices necessary for conventional redundancies are fairly fixed and the extra components are regular and periodic in the data array. This makes for easy scripting and automation, reducing design time and cost. Associative redundancy's flexibility make scripting and automation more difficult because it is not simply extra rows and column with their associated bit steering. The content addressable memories require timing circuitry, signals must be multiplexed onto and off of the data and address buses in the DRAM, and refresh of the redundant memory array must be considered. All of these add complexity that is not found in conventional row and column redundancy. The point at which the lowering recurring costs due to greater yield offsets these greater upfront costs cannot easily be determined but are likely insignificant due to amortization of design cost over the huge number of manufactured DRAM dies.

## 6.4 Future Work

Future work with ternary CAM associative redundancy should focus on several areas. First, the fault models should be refined to match fabrication data, if such information can be obtained. This could include more accurate yield models, such as the negative binomial model, a physically accurate size distribution for cluster faults and a more realistic mix of fault densities for the combined fault model. Also, the yield models assumed a monolithic associative redundancy path. Modifying this to closer approximate the simulated device (32 1K entry CAMs instead of one 32K entry CAM) would make a large difference in modelling column failures.

A comparison of yields and equivalent yields for various levels of associativity and replacement of words or sub-words should be performed. Intuitively, higher associativity should lower yield, but the advantages may out-way the costs. Also along this line of investigation are modifications necessary for DDR and other high-speed DRAM operation.

A cost analysis for the non-recurring design time and recurring manufacturing yield increase should be performed. This will determine the necessary yield improvement to make associative redundancy cost effective.

If the redundant memory array is implemented in DRAM, the stored data must be periodically refreshed. How this is to be accomplished was not considered. Along a similar line of enquiry, I did not consider how to program the non-volatile CAM and flash cells after manufacturing test. The necessary circuitry and wiring must be added and the method to get the programming data onto the die should be determined.

A more accurate simulation with a system that can model DRAM fabrication processes would be beneficial. This would allow for an estimate of power consumption.

Finally, an associative design should be manufactured in a DRAM process. This will enable a direct investigation of overhead, yield, speed and power consumption.

## 6.5 Coda

Ternary CAM associative redundancy is a redundancy method for a DRAM manufacturing process where all cell failures do not fall into the nice categories of single cell faults or row and column failures. With minimum feature sizes shrinking, the likelihood of a contiguous grouping of DRAM cells failing is increasing. The traditional redundancy methods are ill-equipped to deal with these arbitrary-shaped faults. Ternary CAM associative redundancies, specifically the associative indirect variant, offer the flexibility and speed, with no increase in area, to handle intractable faults in today's and future dynamic random access memories.

# Bibliography

[1] Graham Allan. *MOSAID In-House DRAM Design Course.* MOSAID Technologies Inc., July 1996.

[2] W. Maly, A.J. Strojwas and S.W. Director. VLSI yield prediction and estimation: A unified framework. *IEEE Transactions on Computer Aided Design*, CAD-5(1):114–130, Jan 1986.

[3] Josè Pineda de Gyvez. *Integrated Circuit Defect Sensitivity: Theory and Computational Models.* Kluwer Academic Publishers, Boston, 1993.

[4] R. Dean Adams. *High Performanec Memory Testing: Design Principles, Fault Modeling and Self-Test*, volume 22 of *Frontiers in Electronic Testing.* Kluwer Academic Publishers, Boston, Sept 2002.

[5] V. Lines et.al. 66MHz 2.3M ternary dynamic content addressable memory. In *Proc. Intl. Workshop on Memory Technology, Design and Testing*, pages 101–105, Aug 2000.

[6] M. Cao et.al. A simple EEPROM cell using twin polysilicon thin film transistors. *IEEE Electron Device Letters*, 15(8):304–306, Aug 1994.

[7] Curtis Wickman. File store memories. Master's thesis, University of Alberta, Edmonton, Alberta, November 2000.

[8] J.P. de Gyvez and D.K. Pradhan, editor. *Integrated Circuit Manufacturability: The Art of Process and Design Integration.* IEEE Press, Piscataway, 1999.

[9] H.L. Kalter et.al. A 50-ns 16-Mb DRAM with a 10-ns data rate and on-chip ECC. *IEEE Journal of Solid-State Circuits*, 25(5):1118–1128, Oct 1990.

[10] K. Lee et.al. A 1 Gbit synchronous dynamic random access memory with an independent subarray-controlled scheme and a hierarchical decoding scheme. *IEEE Journal of Solid-State Circuits*, 33(5):779–786, May 1998.

[11] J.-P. Dauvin. State of the semiconductor market. Online, January 2003. `http://www.st.com/stonline/company/investor/prescast/document/q4fy2002.pdf`.

[12] Craig Joly. Semiconductor memory redundacy through ternary content addressable memory: Single cell fault model. University of Alberta, EE 652: Semiconductor Memories, Project Report, September 2002.

[13] B. Keeth and R.J. Baker. *DRAM Circuit Design: A Tutorial.* Microelectronic Systems. IEEE Press, Piscataway, 2001.

[14] J. Jex and A. Baker. Content addressable memory for flash redundancy. In *IEEE Pacific Rim Conf. on Communications, Computers and Signal Processing*, pages 741–744, May 1991.

[15] Tegze P. Haraszti. *CMOS Memory Circuits.* Kluwer Academic Publishers, Boston, 2000.

[16] D. Siewiorek and R. Swarz. *The Theory and Practice of Reliable System Design*, pages 216, 226, 298. Digital Equipment Corp., Hudson, MA, 1982.

[17] A.V. Ferris-Prabhu. Defect size variations and their effect on the critical areas of VLSI devices. *IEEE Journal of Solid-State Circuits*, 20(4):878–880, Aug 1985.

[18] W. Maly. Yield simulation - a comparative study -. In *Proc. Intl. Workshop on Defect and Fault Tolerance in VLSI Systems*, Oct. 1989.

[19] Tegze P. Haraszti. A novel associative approach for fault-tolerant MOS RAMs. *IEEE Journal of Solid-State Circuits*, 17(3):539–546, June 1982.

[20] M.A. Lucente, C.H. Harris and R.M. Muir. Memory system reliability improvement through associative cache redundancy. *IEEE Journal of Solid-State Circuits*, 26(3):404–409, March 1991.

[21] J.-C. Lo and J.H. Kim. Highly available memory systems with fault tolerant associative repair mechanisms. *Journal of Microelectronic Systems Integration*, 3(3):205–216, 1995.

[22] T. Miwa et.al. A 1-Mb 2-Tr/b nonvolatile CAM based on flash memory technologies. *IEEE Journal of Solid-State Circuits*, 31(11):1601–1609, Nov 1996.

[23] D.H.K. Hoe et.al. Cell and circuit design of single-poly EPROM. *IEEE Journal of Solid-State Circuits*, 24(4):1153–1157, Aug 1989.

[24] J.M. Rabaey. *Digital Integrated Circuits: A Design Perspective.* Prentice Hall, New Jersey, 1996.

[25] J. Stokes. Ars technica ram guide part II: Asynchronous and synchronous dram. Online, July 2000. `http://arstechnica.com/paedia/r/ram_guide/ram_guide.part2-1.html`.

[26] Ruby, the object-oriented scripting language. Online, August 2003. `http://www.ruby-lang.org/en/`.

[27] R. Sung and J. Koob. Autonomous linear feedback shift register. Online, Oct 2000. EE 552 Student Application Note. `http://www.ece.ualberta.ca/~elliott/ee552/labs/lab5/lfsr_generic.vhd`.

[28] ITRS. International technology roadmap for semiconductors 2002 update. Technical report, Semiconductor Insdustry Association, 2002.

[29] Cadence Design Systems, Inc. *Cadence SKILL Language User Guide*, 06.00 edition, June 2000.

[30] C.H. Stapper and H.-S. Lee. Synergistic fault-tolerance for memory chips. *IEEE Transactions on Computers*, 41(9):1078–1087, Sep 1992.

# Appendix A

# DRAM Data

## A.1 IBM 16-Mbit DRAM

The IBM 16-Mbit DRAM has on-chip error-correcting code (ECC), supports either 11/11 or 12/10 RAS/CAS addressing, can be packaged as a 2Mb×8, 4Mb×4, 8Mb×2 or 16Mb×1 DRAM, and is capable of operating in fast page mode, static column mode or toggle mode [9]. The DRAM has four quadrants, four array blocks per quadrant and two segments per quadrant. For the purpose of this work, the DRAM is assumed to be a 4 Mb×4 DRAM with 11/11 RAS/CAS addressing.

Each quadrant (section) is an independent 4 Mb×1 DRAM [30]. The quadrants contain 4096 word lines and 1024 bit lines. For bit-line redundancy, the quadrant is subdivided into books. Eight books are formed, each containing 2048×128 bits. Two additional redundant bit-lines per book can be used for replacements of any bit-lines within the book. A book has 2048 pages, each addressed independently. There are also 24 redundant word-lines available for substitution of any of the 4096 word-lines in a quadrant [30].

Error correcting codes add an extra nine bits for every 128 bits [9]. This increases the page and book widths by 9 bits, making them 137 bits wide. The section width is increased to 1096 bits.

For yield calculations, the 16-Mbit DRAM is treated as four 4Mb×1 DRAMs because the words are spread across all four sections. Each 4Mb×1 DRAM has one section with 16 books. The section is 1024 (1096 for ECC) bits by 4096 bits. It has

| | |
|---|---|
| Bits (nominal) | 16×1024×1024 |
| Address bits | 22 |
| Words | 4×1024×1024 |
| Word size (bits) | 4 |
| Banks | 1 |
| Sections | 4 |
| Word size per section | 1 |
| Books per section | 16 |
| Pages per book | 2048 |
| Pagewidth (Bits per page) | 128 |
| Redundant columns per book | 2 |
| Redundant rows per section | 24 |
| Bits (w/ ECC) | 17956864 |
| Codewords per book | 2048 |
| Bits per codeword | 137 |
| Pagewidth (w/ ECC) | 137 |

Table A.1: Data for the IBM 16-Mbit DRAM [9].

24 redundant rows. Each book is 128 (137 for ECC) bits by 2048 bits and has two redundant columns. Since each section stores a single bit word, bit-line interleaving can safely be ignored.

## A.2 Samsung 1-Gbit DRAM

The Samsung 1-Gbit DRAM contains eight independent 128-Mb blocks (sections) and a hierarchical decoding scheme [10]. Externally, it is a 32Mb×32 DRAM. Internally, it is eight 8Mb×16 DRAMs. The blocks are paired into four banks, each bank comprising an 8Mb×32 DRAM.

The DRAM has 64 redundant rows per 128 Mbits and 16 redundant columns per 32 Mbits [10]. Therefore, a section is a 128-Mbit block and there are four books per section. This assumption allows for the greatest flexibility in row and column redundancy, increasing the calculated yield. There are 256 memory cells connected to each bit-line and 512 cells connected to each sub-word-line. The entire memory is constructed of 256-Kbit cell arrays. The cell arrays are aligned along word-lines, therefore, they are composed of two sub-word-lines. From the die photo, the blocks

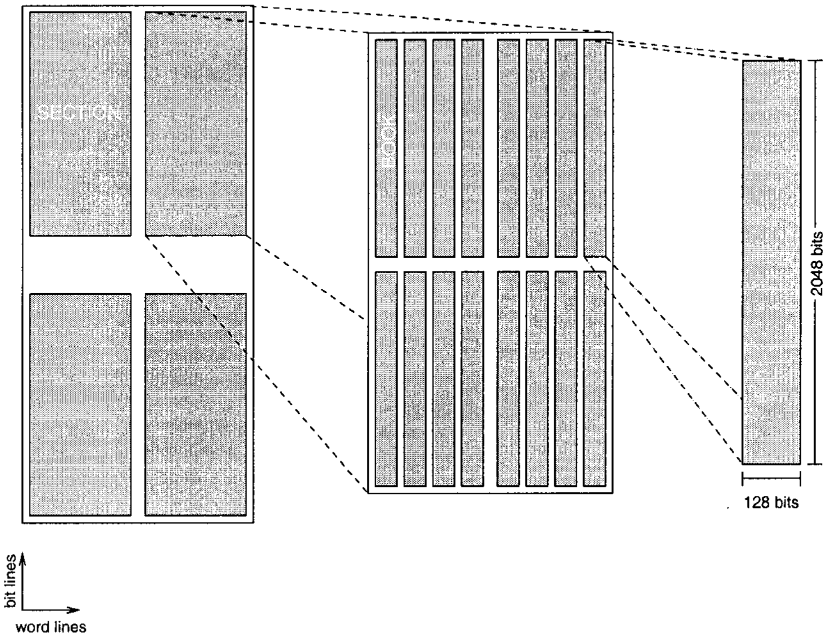


Figure A.1: Simplified block diagram of the IBM 16-Mbit DRAM without redundancy

(sections) are roughly square. The cells have approximately a 2:1 aspect ratio. Assuming the larger dimension is along the bit-line, the cell array also has a 2:1 aspect ratio, with the larger dimension aligned with the word-lines. Assuming the section is split into four nearly square books, there must be 8×16 cell arrays per book. This leads to a pagewidth of 8 Kbits and 4K pages per book.

Following the same assumption as [7], the ECC codeword is set to 512 data bits plus 11 check bits.

For yield calculations, the 1-Gbit DRAM is treated as eight 8Mb×16 DRAMs. Each 8Mb×16 DRAM has one section with 4 books. The section is 16 kbits (16736 for ECC) by 8 Kbits. It has 64 redundant rows. Each book is 8 Kbits (8368 for ECC) by 2 Kbits and has 26 redundant columns. Bit-line interleaving must be considered.

| | |
|---|---|
| Bits (nominal) | 1024×1024×1024 |
| Address bits | 25 |
| Words | 32×1024×1024 |
| Word size (bits) | 32 |
| Banks | 4 |
| Sections | 8 |
| Word size per section | 16 |
| Books per section | 4 |
| Pages per book | 4K |
| Pagewidth (Bits per page) | 8K |
| Redundant columns per book | 16 |
| Redundant rows per section | 64 |
| Bits (w/ ECC) | 1046×1024×1024 |
| Codewords per book | 64K |
| Bits per codeword | 523 |
| Pagewidth (w/ ECC) | 8368 |

Table A.2: Data for the Samsung 1-Gbit DRAM [10].

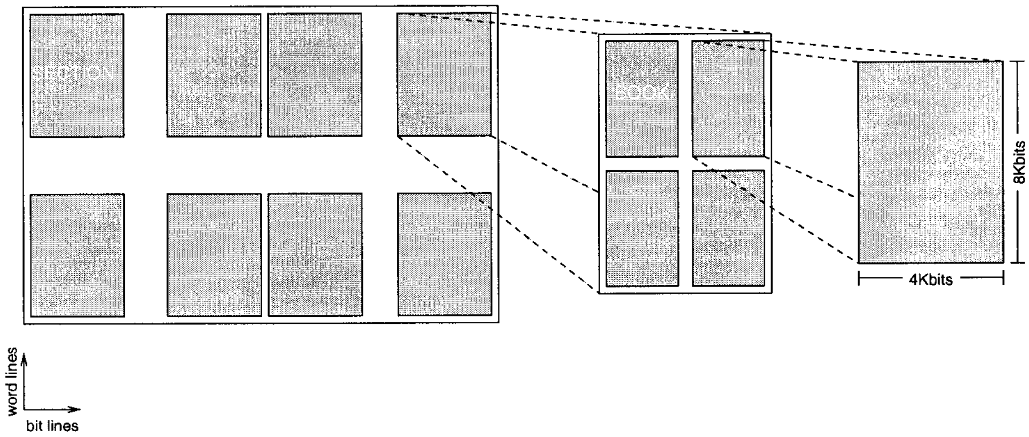


Figure A.2: Simplified block diagram of the Samsung 1-Gbit DRAM without redundancy

# Appendix B

# Redundancy Overhead

In [7], Wickman makes the assumption that the area required to implement bit steering for ECC and row and column redundancy is 2%. This has been extended to assume that any redundancy requires 2% overhead. From this data, the cell efficency for row and column and row and column and ECC redundancies has been calculated.

The calculated overheads are considerably different from those calculated by Wickman and also used in [12]. However, Wickman's numbers do not match with the 11% overhead of error correction in the IBM DRAM, as stated in [9].

## B.1 IBM 16-Mbit DRAM

The IBM 16-Mbit DRAM has a chip size of 140.86 $mm^2$ [9] with 18325760 cells at a size of 4.13 $\mu m^2$ per cell. It has ECC and row and column redundancy. Error correction, including all cells, sense amplifiers and wiring increased the the chip area by 15 $mm^2$. Without ECC there are 17139200 cells in a chip of 125.86 $mm^2$. The cell efficiencies for no redundancy and ECC are simple interpolations. The calculated values are presented in Table B.1. Data stated in [9] is shown in bold.

## B.2 Samsung 1-Gbit DRAM

The Samsung 1-Gbit DRAM has a die size of 569.7 $mm^2$ with 1084243968 cells at a size of 0.344 $\mu m^2$ per cell. As above, after subtracting 2% for bit steering, the cell

| | none | row & column | ECC | r & c & ECC |
|---|---|---|---|---|
| Chip area ($mm^2$) | 120.92 | **125.86** | 134.89 | **140.86** |
| # of cells | 16777216 | 17139200 | 17956864 | 18325760 |
| Cell area ($mm^2$) | 69.29 | 70.78 | 74.16 | 75.69 |
| Periphery area ($mm^2$) | 48.75 | 55.08 | 60.73 | 65.17 |
| Cell Efficiency (%) | 57.3 | 56.2 | 55.0 | 53.7 |
| Area Overhead (%) | 0 | 4.09 | 11.55 | 16.49 |

Table B.1: Areas and overheads for the IBM 16-Mbit DRAM.

| | none | row & column | ECC | r & c & ECC |
|---|---|---|---|---|
| Chip area ($mm^2$) | 552.59 | **569.7** | 575.97 | 581.59 |
| # of cells | 1073741824 | 1084243968 | 1096810496 | 1107492864 |
| Cell area ($mm^2$) | 358.63 | 362.14 | 366.33 | 369.90 |
| Periphery area ($mm^2$) | 193.9 | 207.46 | 209.64 | 211.69 |
| Cell Efficiency (%) | 64.9 | 63.6 | 63.6 | 63.6 |
| Area Overhead (%) | 0 | 3.10 | 4.23 | 5.24 |

Table B.2: Areas and overheads for the Samsung 1-Gbit DRAM.

efficiency is 64.9%. This value is kept constant for all redundancies.

# Appendix C

# Yield Calculations

## C.1 Single Cell Fault Model

### C.1.1 No Redundancy

The negative binomial yield model for DRAM begins with the yield of a single cell:

$$Y_{sc} = e^{-\lambda} \tag{C.1}$$

where $Y_{sc}$ is the yield of a single DRAM bit and $\lambda$ represents the average number of faults per bit. The yield of a DRAM bit is the probability any given cell will be functional. The yield of a DRAM, $Y_{DRAM}$ is then:

$$Y_{DRAM} = Y_{sc}^{b_{DRAM}} \tag{C.2}$$

where $b_{DRAM}$ is the number of bits in the DRAM. Yields for 16-Mbit and 1-Gbit DRAMs are shown in Figure C.1. From the graph, for both DRAMs to achieve better that 50% yield, there can be a less than one fault per die. This is an extremely small fault density and serves as a telling example of why redundancy is used in nearly all commercial memories.

### C.1.2 Row and Column Redundancy

Some terms must first be defined for row and column redundancy techniques. A *book* is the area of memory to which column redundancy is applied. A *section* is the area of memory to which row redundancy is applied. In DRAMs, a section tends to be larger than a book [7]. The number of columns in a book is the *pagesize*.

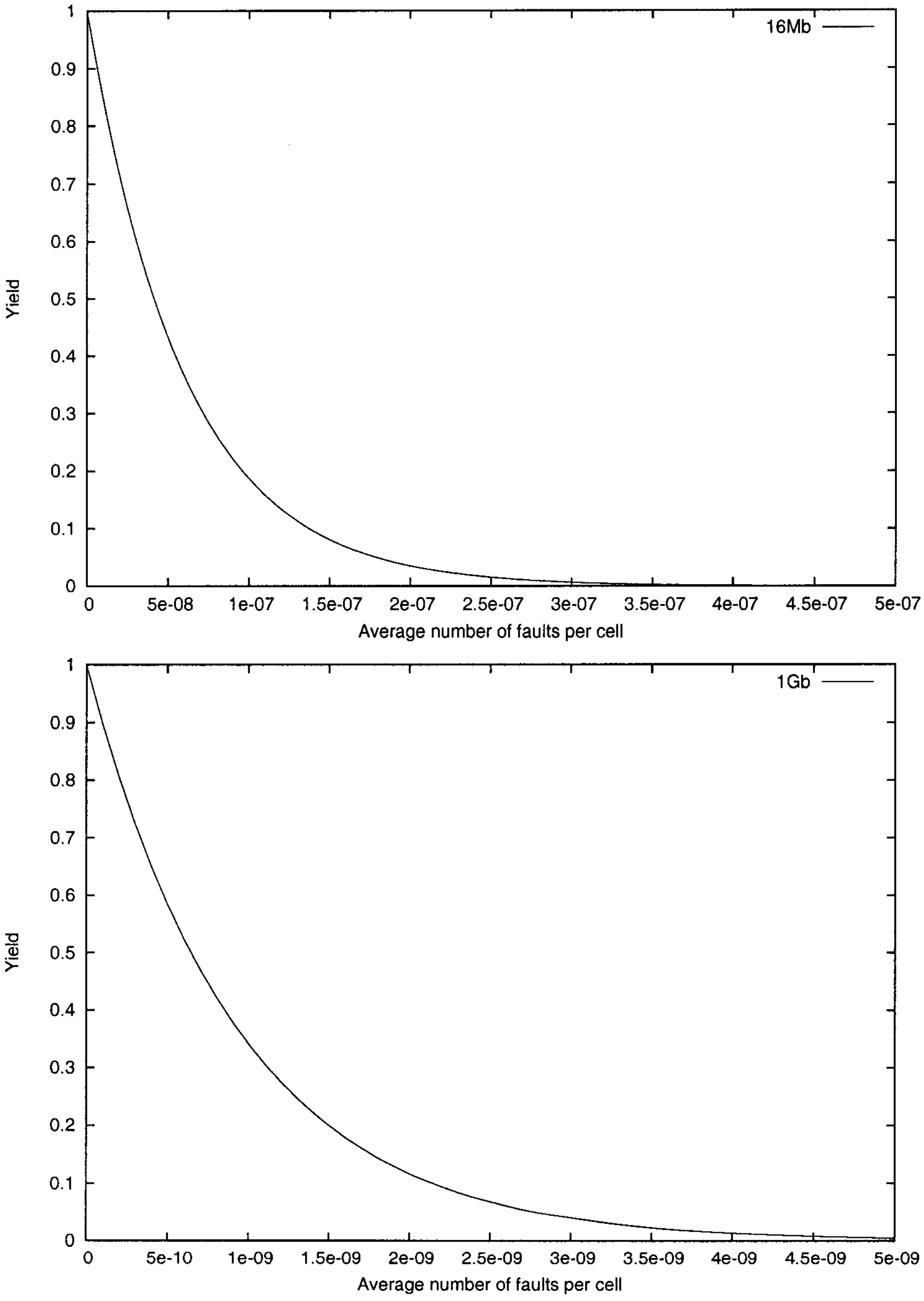


Figure C.1: Single cell fault model yield of a 16-Mbit and a 1-Gbit DRAM without redundancy

The yield of a column inside a DRAM is given by:

$$Y_{column} = Y_{sc}^{b_{column}} \tag{C.3}$$

where $Y_c$ is the yield of a column and $b_{column}$ is the number of bits in a column. The yield of a book is then calculated by summing over all contributions to the yield distributions from zero errors to the number of redundant columns:

$$Y_{book} = \sum_{i=0}^{r_c} \begin{pmatrix} p + r_c \\ i \end{pmatrix} \cdot Y_{column}^{p+r_c-i} \cdot (1 - Y_{column})^i \tag{C.4}$$

where $Y_{book}$ is the yield of a book, $r_c$ is the number of redundant columns per book and $p$ is the page size. The first term of the equation chooses which columns are bad, the second is the yield of the good columns and the third is the yield of the bad columns.

From the yield of a book, the effective yield of a single cell when column redundancy is determined:

$$Y_{sc_{book}}^{b_{book}} = Y_{book} \tag{C.5}$$

where $Y_{sc_{book}}$ is an estimate for the effective yield of a single cell inside the book. This effective yield is used to determine the yield of a row:

$$Y_{row} = Y_{sc_{book}}^{b_{row}} \tag{C.6}$$

where $Y_{row}$ is the yield of a row and $b_{row}$ is the number of bits in a row. The yield of a section is then calculated:

$$Y_{section} = \sum_{i=0}^{r_r} \begin{pmatrix} s + r_r \\ i \end{pmatrix} \cdot Y_{row}^{s+r_r-i} \cdot (1 - Y_{row})^i \tag{C.7}$$

where $Y_{section}$ is the yield of a section, $r_r$ is the number of redundant rows per section and $s$ is the number of rows in a section. The yield of the DRAM with redundant rows and columns is then given by:

$$Y_{DRAM_{RC}} = Y_{section}^{N_{sections}} \tag{C.8}$$

where $N_{sections}$ is the number of sections in the DRAM.

Yields for 16-Mbit and 1-Gbit DRAMs with row and column redundancy are shown in Figure C.2. To achieve a 50% yield, the 16-Mbit DRAM can handle up to approximately 240 faults per die while the 1-Gbit DRAM can handle about 1400 faults.

### C.1.3 ECC Redundancy

An *ECC codeword* is the complete ECC word stored in memory including the data and check bits. The yield of a SEC (single error correcting) code word can be calculated as follows:

$$Y_{cw} = Y_{sc}^{b_{cw}} + \begin{pmatrix} b_{cw} \\ 1 \end{pmatrix} \cdot Y_{sc}^{b_{cw}-1} \cdot (1 - Y_{sc}) \tag{C.9}$$

where $Y_{cw}$ is the yield of an ECC code word and $b_{cw}$ with the number of bits in the code word. The yield of a book is then calculated as:

$$Y_{book_{ECC}} = Y_{cw}^{N_{cw}} \tag{C.10}$$

where $Y_{book_{ECC}}$ is the yield of a book with ECC and $N_{cw}$ is the number of codewords in a book. The yield of a DRAM with ECC is then:

$$Y_{DRAM_{ECC}} = Y_{book_{ECC}}^{N_{books}} \tag{C.11}$$

where $N_{books}$ is the number of books in the DRAM.

Figure C.3 shows DRAM yield when ECC redundancy is employed. For a 50% yield, the 16-Mbit DRAM can handle less than approximately 400 faults while the 1-Gbit DRAM can handle less than approximately 1665 faults.

### C.1.4 Row and Column Redundancy with ECC

Equation C.9 gives the yield of an ECC codeword. From this, the effective yield of a single cell when using ECC is calculated as follows:

$$Y_{sc_{ECC}}^{b_{cw}} = Y_{cw} \tag{C.12}$$

The effective yield of a column is then given by:

$$Y_{column_{ECC}} = Y_{sc_{ECC}}^{b_{column}} \tag{C.13}$$

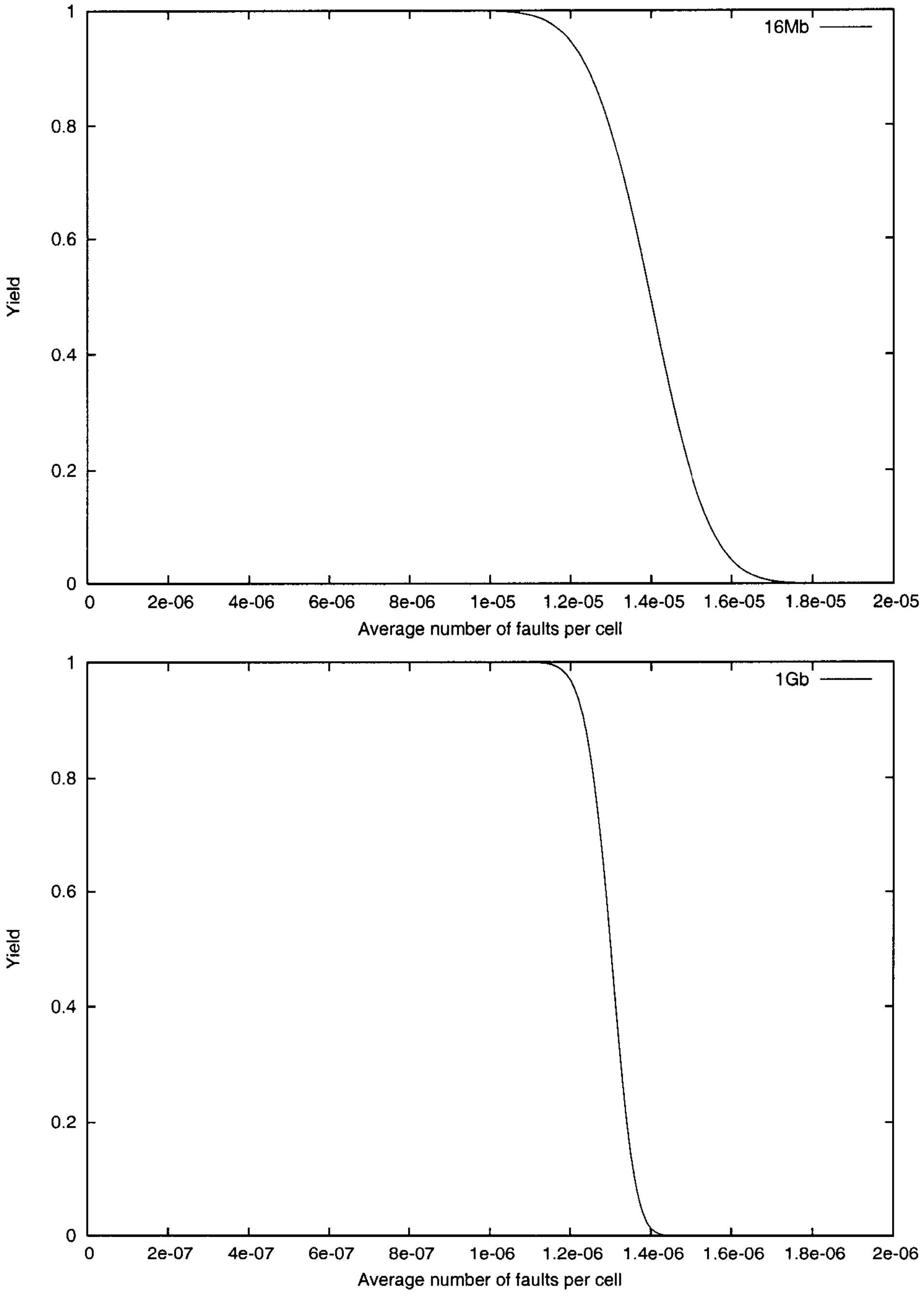


Figure C.2: Single cell fault model yield of a 16-Mbit and a 1-Gbit DRAM with row and column redundancy

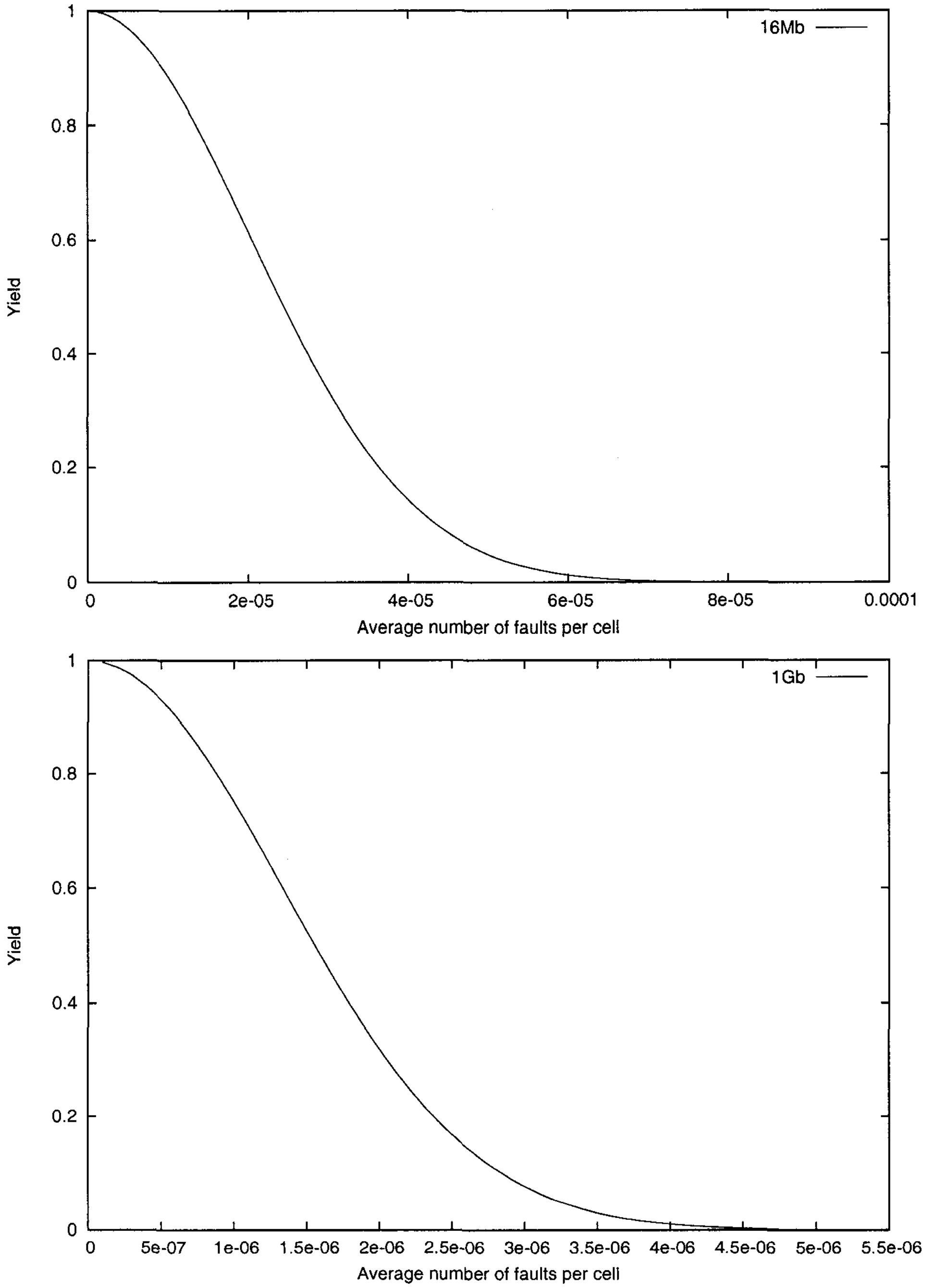


Figure C.3: Single cell fault model yield of a 16-Mbit and a 1-Gbit DRAM with ECC redundancy

and the effective yield of a book is:

$$Y_{book_{col\&ECC}} = \sum_{i=0}^{r_c} \begin{pmatrix} p + r_c \\ i \end{pmatrix} \cdot Y_{column_{ECC}}^{p+r_c-i} \cdot (1 - Y_{column_{ECC}})^i \tag{C.14}$$

The effective yield of a single cell when using ECC and column redundancy is estimated as:

$$Y_{sc_{col\&ECC}}^{b_{book}} = Y_{book_{col\&ECC}} \tag{C.15}$$

The yield of a DRAM with ECC, row and column redundancy is calculated as in Equations C.6 through C.8.

As mentioned, in Section 2.5 error correcting code redundancy and row and column redundancy have very high synergies. This is shown in Figure C.4 where the 16-Mbit DRAM is shown to be able to achieve a 50% yield ratio with up to about 7650 faults while the 1-Gbit DRAM can achieve the same with up to about 76200 faults.

### C.1.5 Associative Direct Ternary CAM Redundancy

Since ternary CAM redundancies are based on replacing bad words, the yield of a word must first be calculated:

$$Y_{word} = Y_{sc}^{b_{word}} \tag{C.16}$$

where $b_{word}$ is the number of bits in a word. The effective number of CAM entries must also be calculated:

$$c = Y_{CAM_{sc}}^{b_{addr}} \cdot Y_{flash_{sc}}^{b_{addr}+b_{sec}} \cdot Y_{am_{sc}}^{b_{word}} \cdot N_{CAM} \tag{C.17}$$

where $Y_{CAM_{sc}}$ is the yield of a single CAM cell. It is calculated as in Equation C.1 with $\lambda_{CAM}$, the average number of faults per CAM cell. $Y_{am_{sc}}$ is the yield of a single cell of the associated memory and $Y_{flash_{sc}}$ is the yield of a single cell of the *don't care* mask. Unless the associated memory is created in a different technology, $Y_{am_{sc}}$ will be the same as $Y_{sc}$. The *don't care* mask will most likely be stored in non-volatile memory. $b_{addr}$ and $b_{word}$ are the number of address bits and the number of bits in

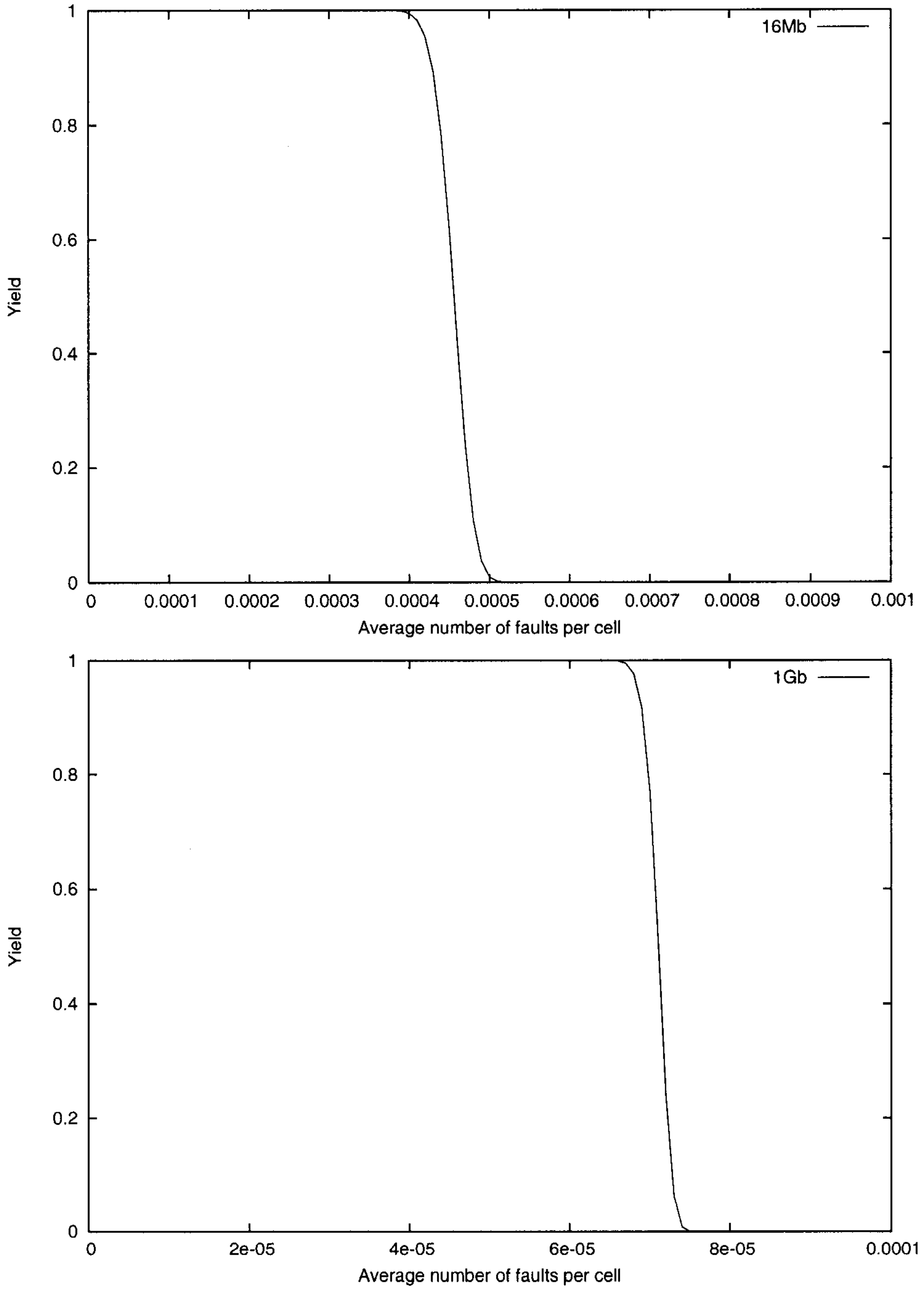


Figure C.4: Single cell fault model yield of a 16-Mbit and a 1-Gbit DRAM with row and column redundancy and ECC

a word, respectively. $b_{sec}$ is the number of bits to mark which section should be replaced. Although each CAM entry has several associated words, in the single cell case, only the first word contains valid data, therefore, it is the only one included in the above calculation.

Assuming the redundant memory is the same technology as the primary memory, the yield of a DRAM using associative direct redundancy is:

$$Y_{DRAM} = \sum_{i=0}^{c} \left( \begin{array}{c} W \\ i \end{array} \right) \cdot Y_{word}^{W-i} \cdot (1 - Y_{word})^{i} \tag{C.18}$$

where $W$ is the number of words in the DRAM. The number of CAM entries is not included in the equation (except for the sum), because that yield is accounted for in the above equation for $c$.

Figure C.5 shows the 50% yield lines for DRAMs using associative direct ternary CAM redundancy with variable amounts of CAM entries. This is different from the graphs for conventional redundancy methods. A single cell fault density of five times that of the DRAM cell has been chosen for the CAM cell. The fault density of non-volatile flash cell has been set to half of that of the CAM cell. Although the CAM and flash cells are much larger than a DRAM cell, they do not have the large capacitor structure, so this number should be conservative for the yield calculations. With these numbers, the graphs show that with a CAM of 256 entries, this redundancy method achieves better yield than row and column redundancies for the 16-Mbit DRAM. The 1-Gbit DRAM will require a CAM with 1024 entries. Associative direct redundancy with 512 CAM entries can also beat ECC redundancy in the 16-Mbit DRAM. The 1-Gbit DRAM will require 2K CAM entries to better ECC redundancy. To beat ECC, row and column redundancy, 8.5K and 65K CAM entries will be required for the 16-Mbit and 1-Gbit DRAMs, respectively.

### C.1.6 Associative Indirect Ternary CAM Redundancy

As in the associative direct case, the effective number of CAM entries must be found:

$$c = Y_{CAM_{sc}}^{b_{addr}} \cdot Y_{flash_{sc}}^{b_{sec}+b_{addr}+b_{red}} \cdot N_{CAM} \tag{C.19}$$

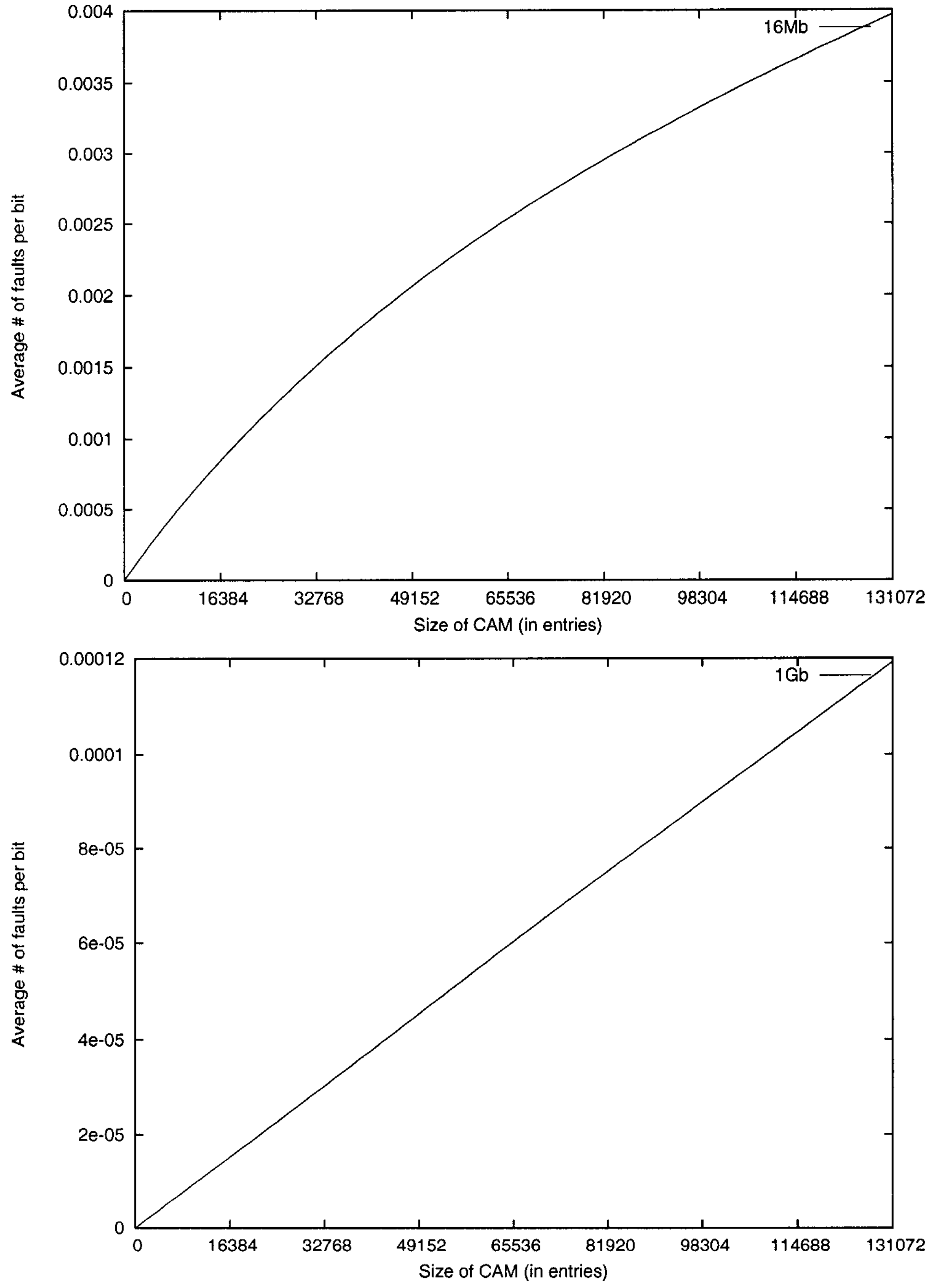


Figure C.5: 50% yield line of a 16-Mbit and a 1-Gbit DRAM with associative direct ternary CAM redundancy using a single cell fault model

where $b_{red}$ is the number of address bits for the redundant memory.

The yield of a DRAM is the same as with the associative direct method. The difference is in the effective number of CAM entries, $c$.

This yield equation will hold true if the secondary memory is of sufficient size. The minimum size is given by:

$$N_{sec} \geq \frac{c}{Y_{red_{sc}}^{b_{word}}} \tag{C.20}$$

where $Y_{red_{sc}}$ is the yield of a single cell of the redundant memory array. In most cases this will be the yield of a single DRAM cell.

The 50% yield lines for 16-Mbit and 1-Gbit DRAMs using associative indirect ternary CAM redundancy are shown in Figure C.6. The fault densities of the CAM and flash cells have been set to 5 times and 2.5 times the fault density of a DRAM cell, respectively, as in the associative direct case.

As with the associative direct case, with 256 CAM entries and 9 redundant memory address bits, associative indirect will beat row and column redundancy, 512 entries and 10 address bits will beat ECC redundancy and 8.5K entries with 13 address bits will beat row and column redundancy with ECC for a 16-Mbit DRAM. For the 1-Gbit DRAM, 1K, 1.7K and 65K CAM entries with 11, 11 and 17 address bits, respectively, will be required. Notice that the yield lines of these plots are very similar to those of the associative direct method.

## C.2 Row Fault Model

### C.2.1 No Redundancy

Analogous to the single-cell fault model, the yield of a DRAM row is:

$$Y_{row} = e^{-\rho} \tag{C.21}$$

The row fault density is represented by $\rho$. The yield of the DRAM is:

$$Y_{DRAM} = Y_{row}^{N_{rows}} \tag{C.22}$$

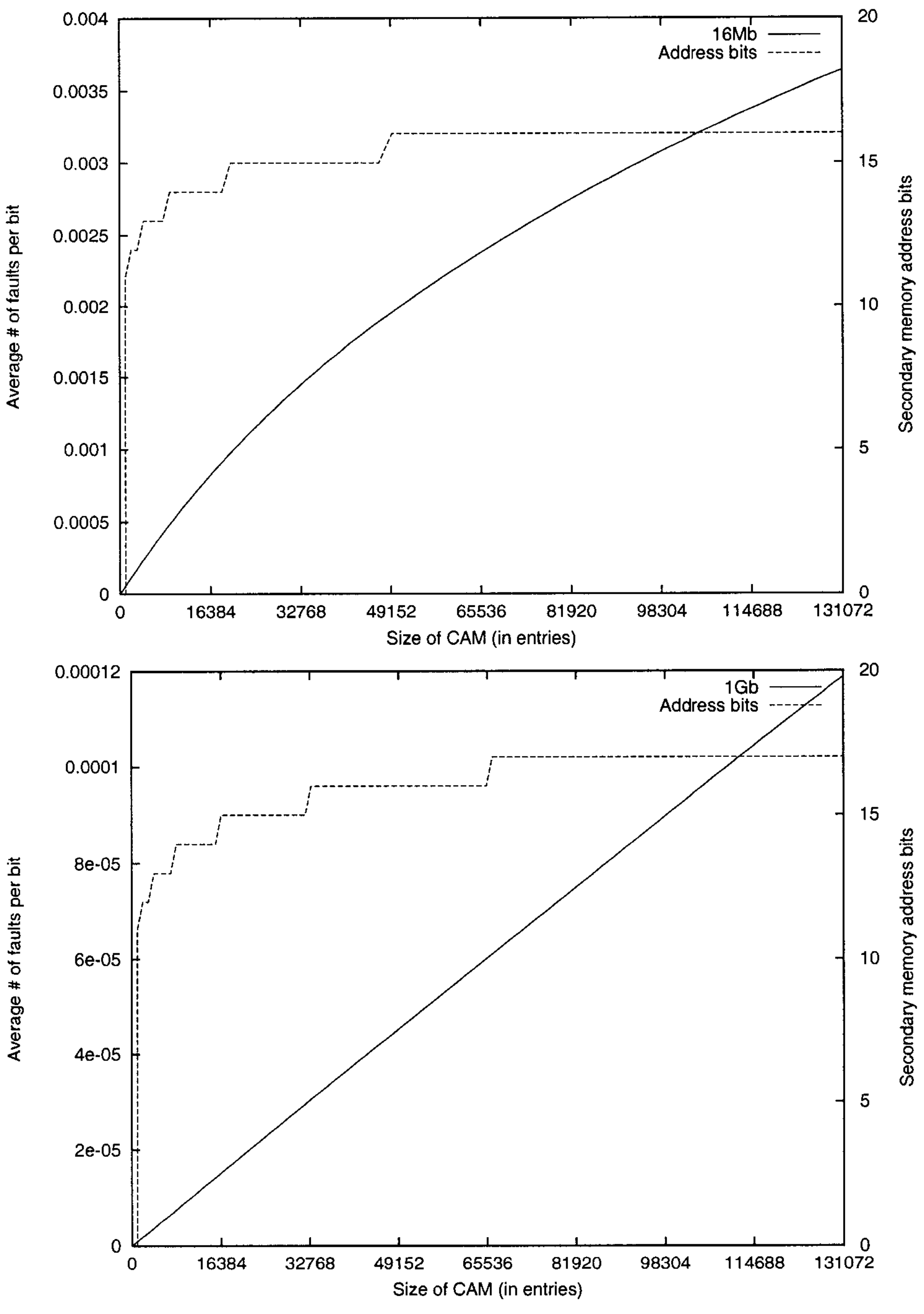


Figure C.6: 50% yield line of a 16-Mbit and a 1-Gbit DRAM with associative indirect ternary CAM redundancy using a single cell fault model

Yields for 16-Mbit and 1-Gbit DRAMs without redundancy are graphed in Figure C.7. For a yield better than 50%, there must be less than one fault per die, a result very similar to the single cell fault model.

### C.2.2 Row and Column Redundancy

In this fault model, columns do not fail, therefore, only the yield of a section is necessary. The yield of a row is as in Equation C.21 while the yield of section is the same as in Equation C.7. The yield of the DRAM is calculated as follows:

$$Y_{DRAM_{row}} = Y_{section}^{N_{sections}} \tag{C.23}$$

As shown in Figure C.8, for a 50% yield, the 16-Mbit DRAM can average 80 row faults per die while the 1-Gbit DRAM averages 432. This is lower than the 96 and 512 redundant rows per die because the failures will not be evenly distributed over all of the sections.

### C.2.3 ECC Redundancy

ECC can only repair a single bit error in its codeword. Therefore, it can do nothing for a failing row. ECC redundancy gives no improvement and will confer the same results as no redundancy, shown in Figure C.7.

### C.2.4 Row and Column Redundancy with ECC

For the same reasons as the previous subsection, adding ECC will give no improvement over row and column redundancy. These results are shown in Figure C.8.

### C.2.5 Associative Direct Ternary CAM Redundancy

Assuming that associative direct redundancy holds four words per CAM, entry it is implied that $\frac{b_{row}}{4b_{word}}$ CAM entries will be required to replace an entire faulty row, where $b_{row}$ is the number of bits in a row (section width) and $b_{word}$ is the number of bits in a word. The effective number of CAM entries is given by:

$$c = Y_{CAM_{row}} \cdot N_{CAM} \tag{C.24}$$

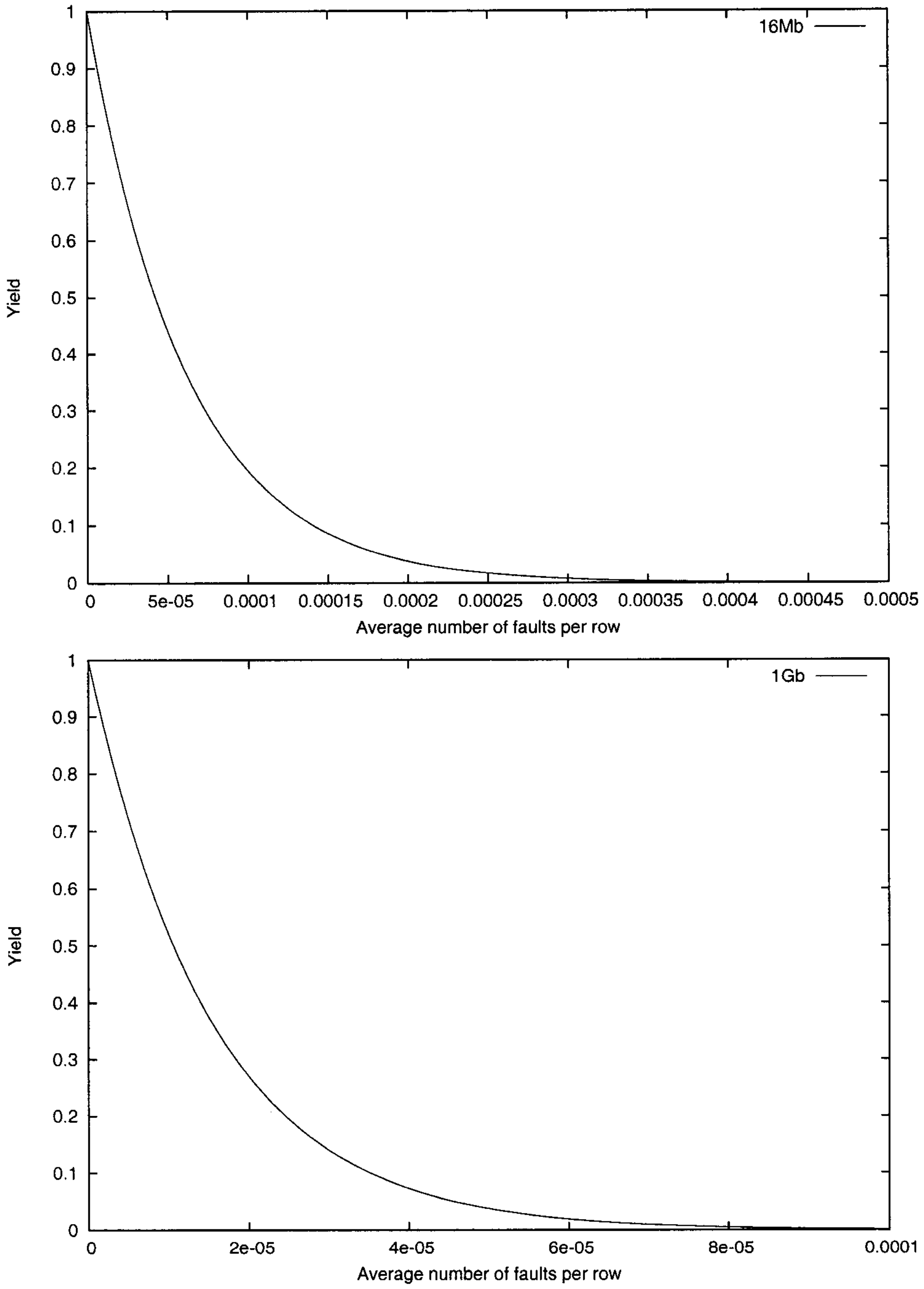


Figure C.7: Row fault model yield of a 16-Mbit and a 1-Gbit DRAM without redundancy

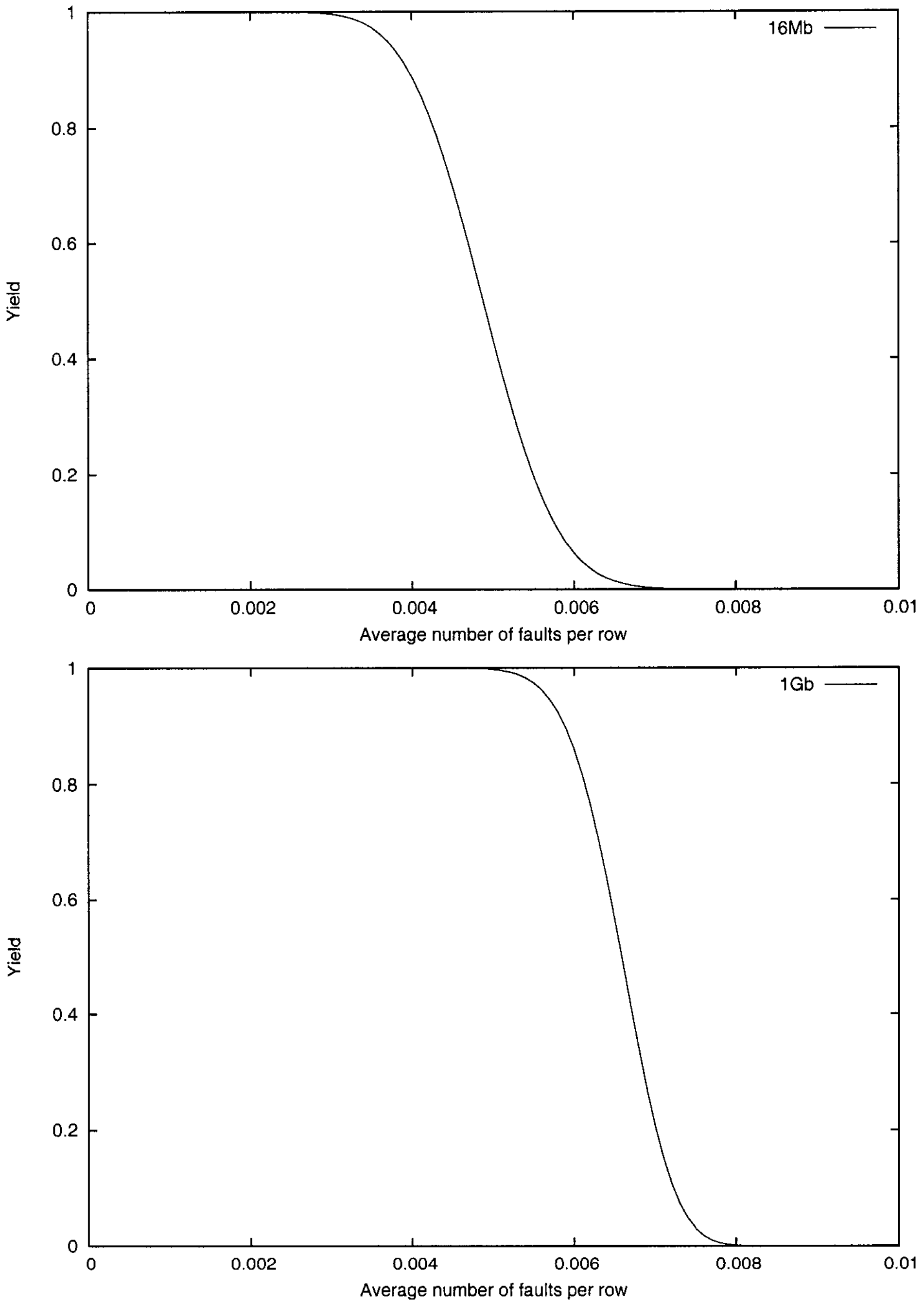


Figure C.8: Row fault model yield of a 16-Mbit and a 1-Gbit DRAM with row and column redundancy

where $Y_{CAM_{row}}$ is the yield of an entire row of the associative memory, including the CAM cells, flash cells and regular DRAM cells.

$$Y_{DRAM_{row}} = \sum_{i=0}^{\frac{4c \cdot b_{word}}{b_{row}}} \binom{N_{row}}{i} \cdot Y_{row}^{N_{row}-i} \cdot (1 - Y_{row})^{i} \tag{C.25}$$

A row fault density of double that of the DRAM row fault density has been chosen for the associative memory row. This should take into account any faults in the pre-charge circuitry and match-line.

As shown in Figure C.9, to match the 50% yield point of row and column, redundancies, a 16-Mbit DRAM using the associative direct method requires approximately 20K ternary CAM entries. The 1-Gbit DRAM will require nearly 110K ternary CAM entries.

### C.2.6 Associative Indirect Ternary CAM Redundancy

Unlike associative direct redundancy, associative indirect only requires one CAM entry per failed row or column. This allows for much smaller redundancy components.

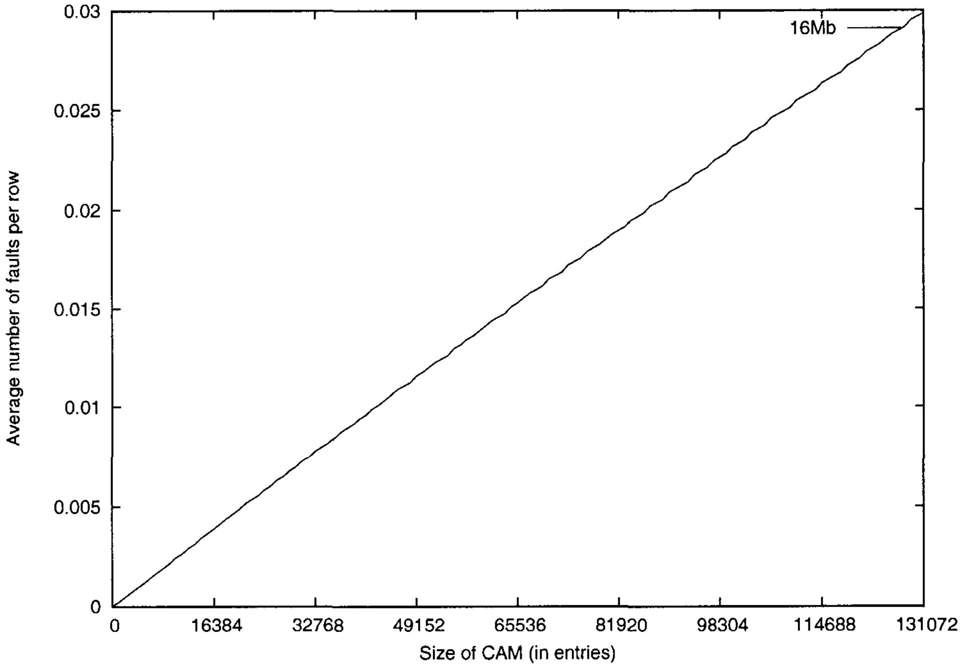
The effective number of CAM entries is as before in Equation C.24. Failing rows contribute to the DRAM yield as follows:

$$Y_{DRAM_{row}} = \sum_{i=0}^{c} \binom{N_{row}}{i} \cdot Y_{row}^{N_{row}-i} \cdot (1 - Y_{row})^{i} \tag{C.26}$$

As with the single cell fault model, the secondary memory must be large enough to contain the redundant rows:

$$N_{sec} \geq \left( \frac{c \cdot b_{row}}{b_{word} \cdot Y_{red_{row}}} \right) \tag{C.27}$$

Figure C.10 shows that in order to meet the 50% yield mark of row and column redundancy, a 16-Mbit DRAM using associative indirect redundancy will require 80 CAM entries with 17 secondary memory address bits. The 1-Gbit DRAM will require 448 ternary CAM entries with 19 secondary memory address bits. This model seems to break down after 10% of the rows are faulty.



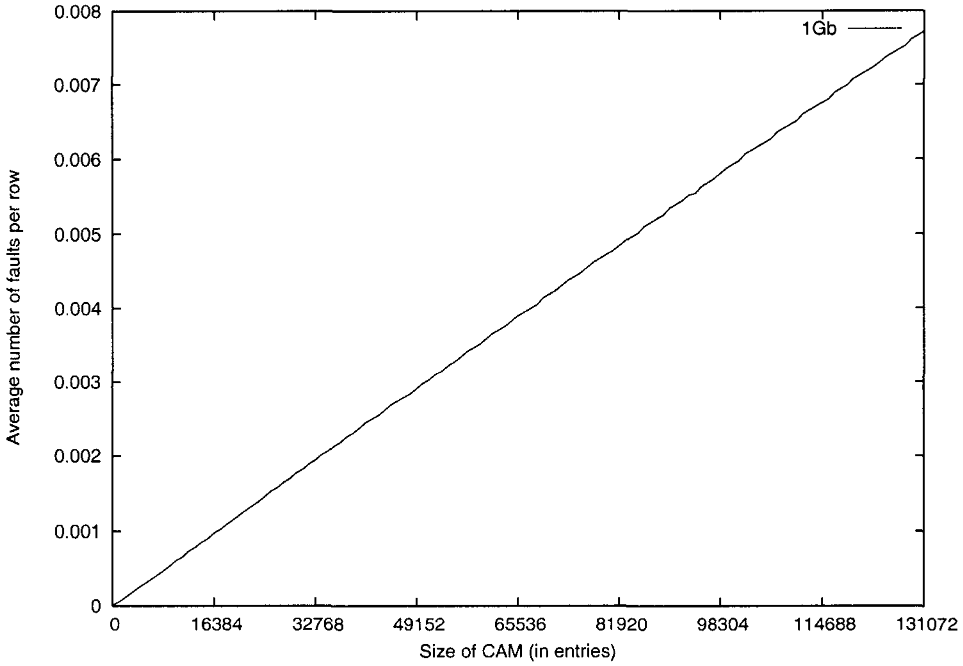


Figure C.9: 50% yield line of a 16-Mbit and a 1-Gbit DRAM with associative direct ternary CAM redundancy using a row fault model

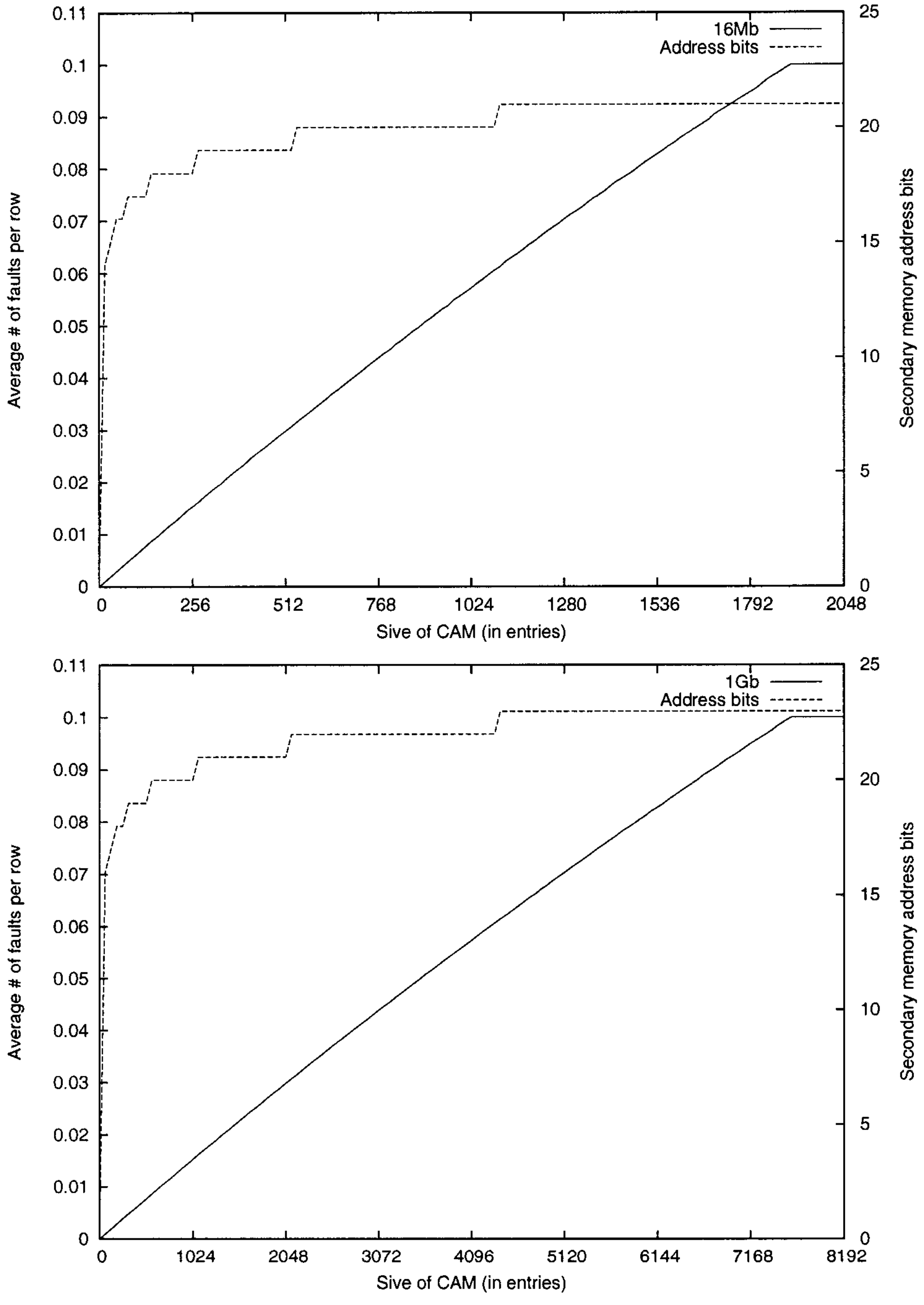


Figure C.10: 50% yield line of a 16-Mbit and a 1-Gbit DRAM with associative indirect ternary CAM redundancy using a row fault model

## C.3 Column Fault Model

### C.3.1 No Redundancy

The yield of a DRAM column is:

$$Y_{column} = e^{-\chi} \tag{C.28}$$

The fault density of columns is represented by $\chi$. The yield of the DRAM is:

$$Y_{DRAM} = Y_{column}^{N_{columns}} \tag{C.29}$$

Yields for 16-Mbit and 1-Gbit DRAMs without redundancy are graphed in Figure C.11. For a yield greater than 50%, there must be less than approximately one faults per three dies, a result very similar to the single cell fault model.

### C.3.2 Row and Column Redundancy

Opposite to the row fault model, in this fault model, columns do not fail and only the yield of a book is necessary. The yield of a column is as in Equation C.28 and the yield of a book is the same as in Equation C.4. The yield of the DRAM is:

$$Y_{DRAM_{column}} = Y_{book}^{N_{books}} \tag{C.30}$$

As shown in Figure C.12, for a 50% yield, the 16-Mbit DRAM can average 28 row or column faults per die while the 1-Gbit DRAM handles 310 faults. As with the row fault model, this is less than the 128 and 512 redundant columns available. This number of faults for the 16-Mbit DRAM is low because there are only two redundant columns per book, while the 1-Gbit DRAM has 16 redundant columns per book, offering much more flexibility.

### C.3.3 ECC Redundancy

The yield of a book can be calculated by treating it as a stack of codewords:

$$Y_{book} = \left( Y_{col}^{b_{cw}} + \binom{b_{cw}}{1} \cdot Y_{col}^{b_{cw}-1} \cdot (1 - Y_{col}) \right)^{N_{cwp}} \tag{C.31}$$

where $N_{cwp}$ is the number of codewords per row in a page.

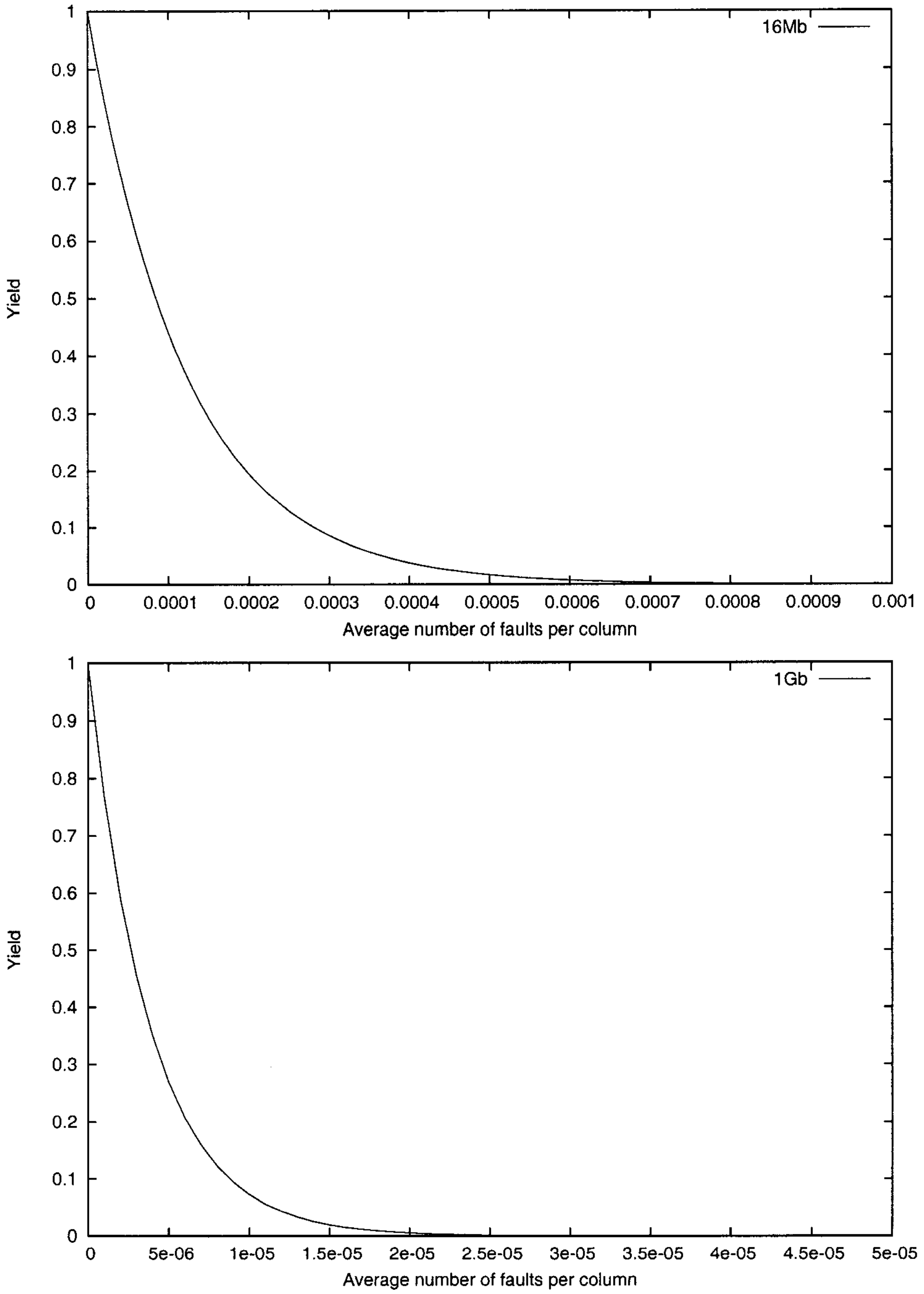


Figure C.11: Column fault model yield of a 16-Mbit and a 1-Gbit DRAM without redundancy

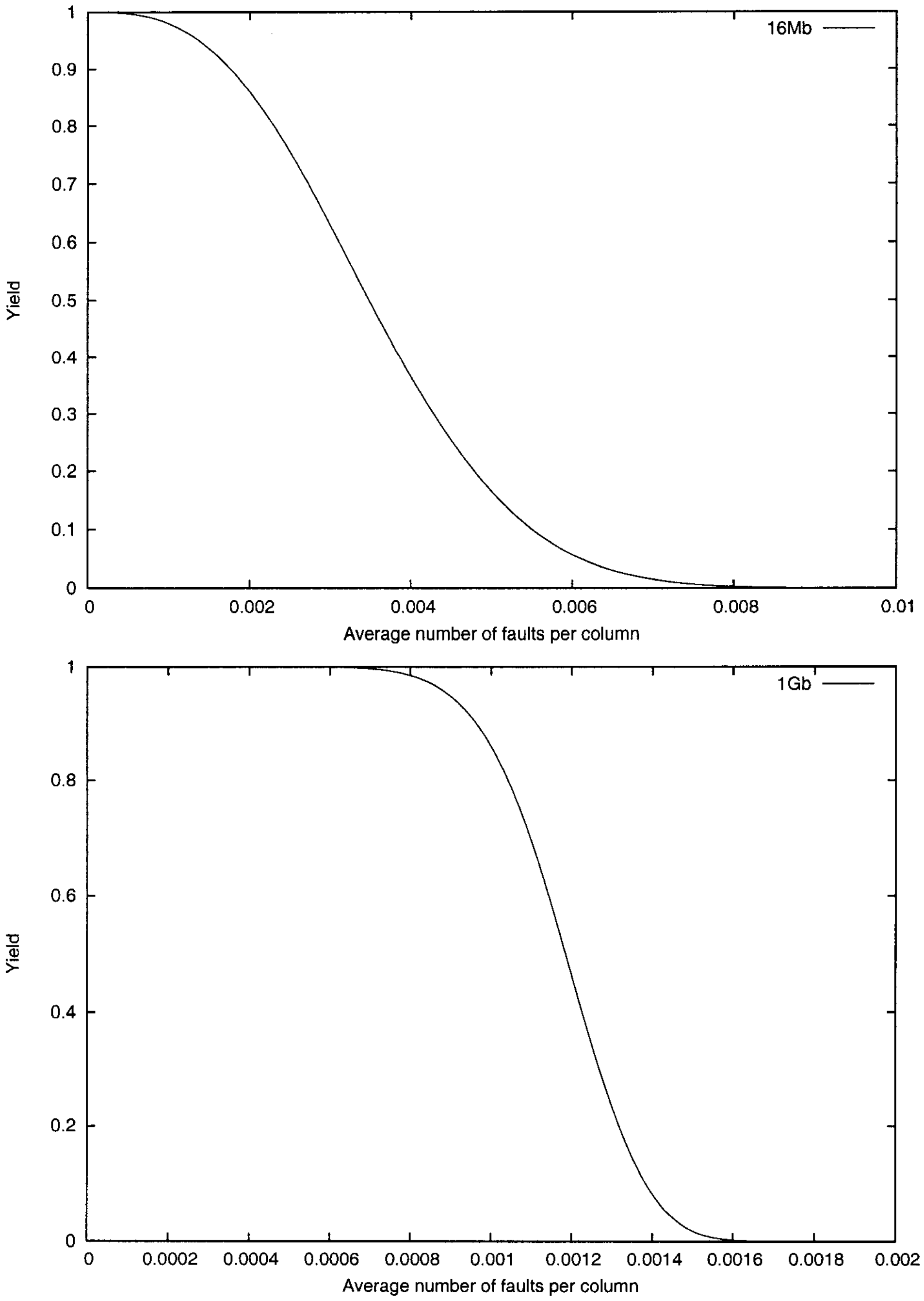


Figure C.12: Column fault model yield of a 16-Mbit and a 1-Gbit DRAM with row and column redundancy

The yield of a DRAM with ECC redundancy is:

$$Y_{DRAM_{ECC}} = Y_{book}^{N_{books}} \tag{C.32}$$

As shown in Figure C.13, ECC redundancy performs worse than row and column redundancy for the column fault model. The 16-Mbit DRAM can handle 9 faults while the 1-Gbit DRAM can handle 26 faults to achieve a 50% yield ratio. This is because, in the 16-Mbit DRAM, ECC can only replace one column per book. In the 1-Gbit DRAM, ECC can replace 16 columns per book, the same amount as row and column redundancy, but only one column per 523 bit codeword can be replaced, greatly reducing the flexibility of the replacement scheme.

### C.3.4 Row and Column Redundancy with ECC

The yield of a column of codewords is found:

$$Y_{col_{cw}} = Y_{col}^{b_{cw}} + \binom{b_{cw}}{1} \cdot Y_{col}^{b_{cw}-1} \cdot (1 - Y_{col}) \tag{C.33}$$

The effective yield of a column is then found:

$$Y_{col_{ECC}}^{b_{codeword}} = Y_{col_{cw}} \tag{C.34}$$

From this, the yields of a book with ECC and column redundancy are found as in Section C.3.2.

These equations lead to the graphs in Figure C.14. The 16-Mbit DRAM can handle 26 column faults for a 50% yield while the 1-Gbit DRAM can handle about 307 faults. This is slightly less than row and column redundancy only. This is most likely because ECC's ability to handle one extra column failure per codeword is offset by the extra columns required.

### C.3.5 Associative Direct Ternary CAM Redundancy

Using the same assumption as the row fault model, associative direct redundancy can hold four words per CAM entry. Therefore, $\frac{b_{col}}{4}$ CAM entries will be required to replace an entire faulty column, where $b_{col}$ is the number of bits in a column (book height).

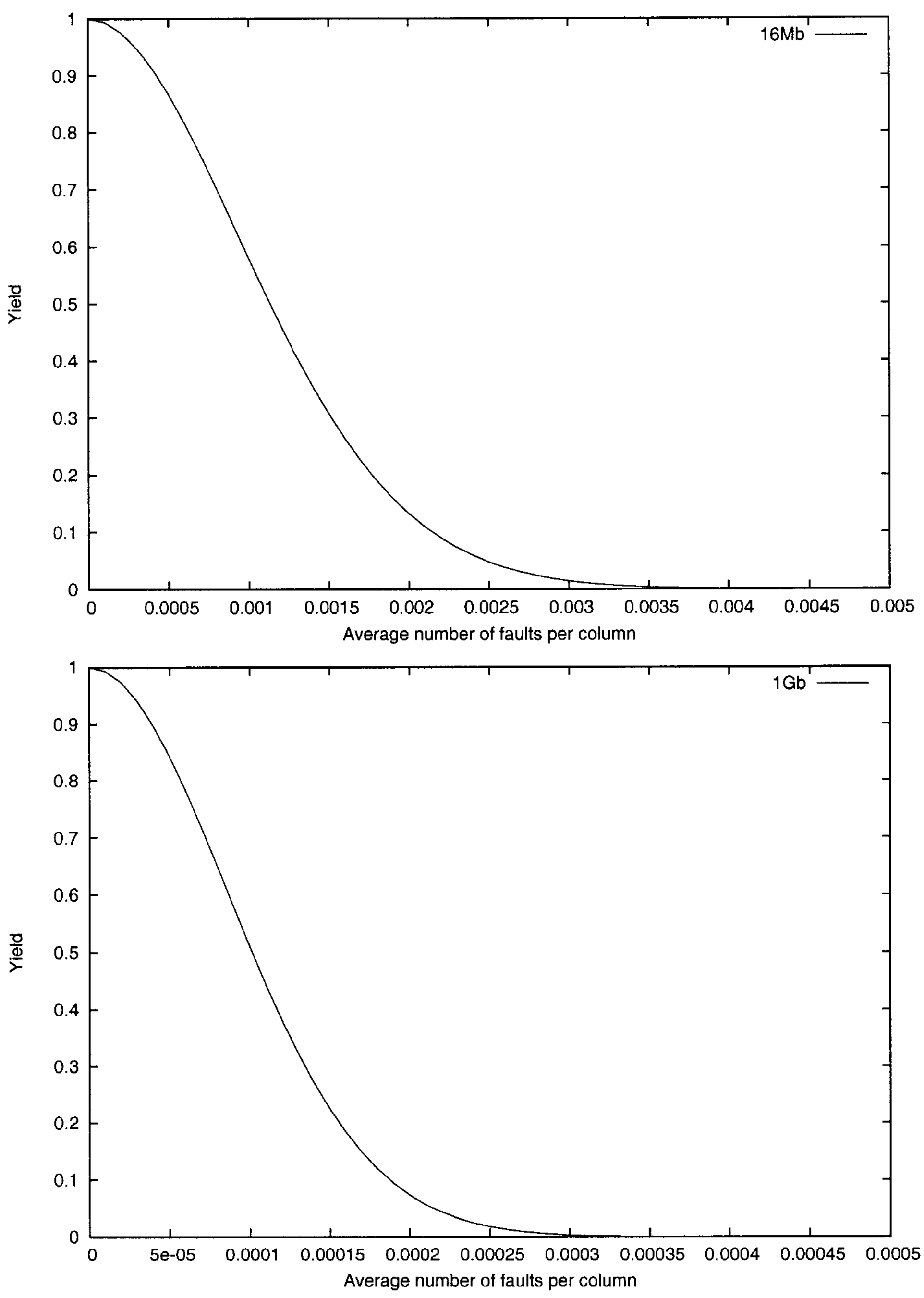


Figure C.13: Column fault model yield of a 16-Mbit and a 1-Gbit DRAM with ECC redundancy

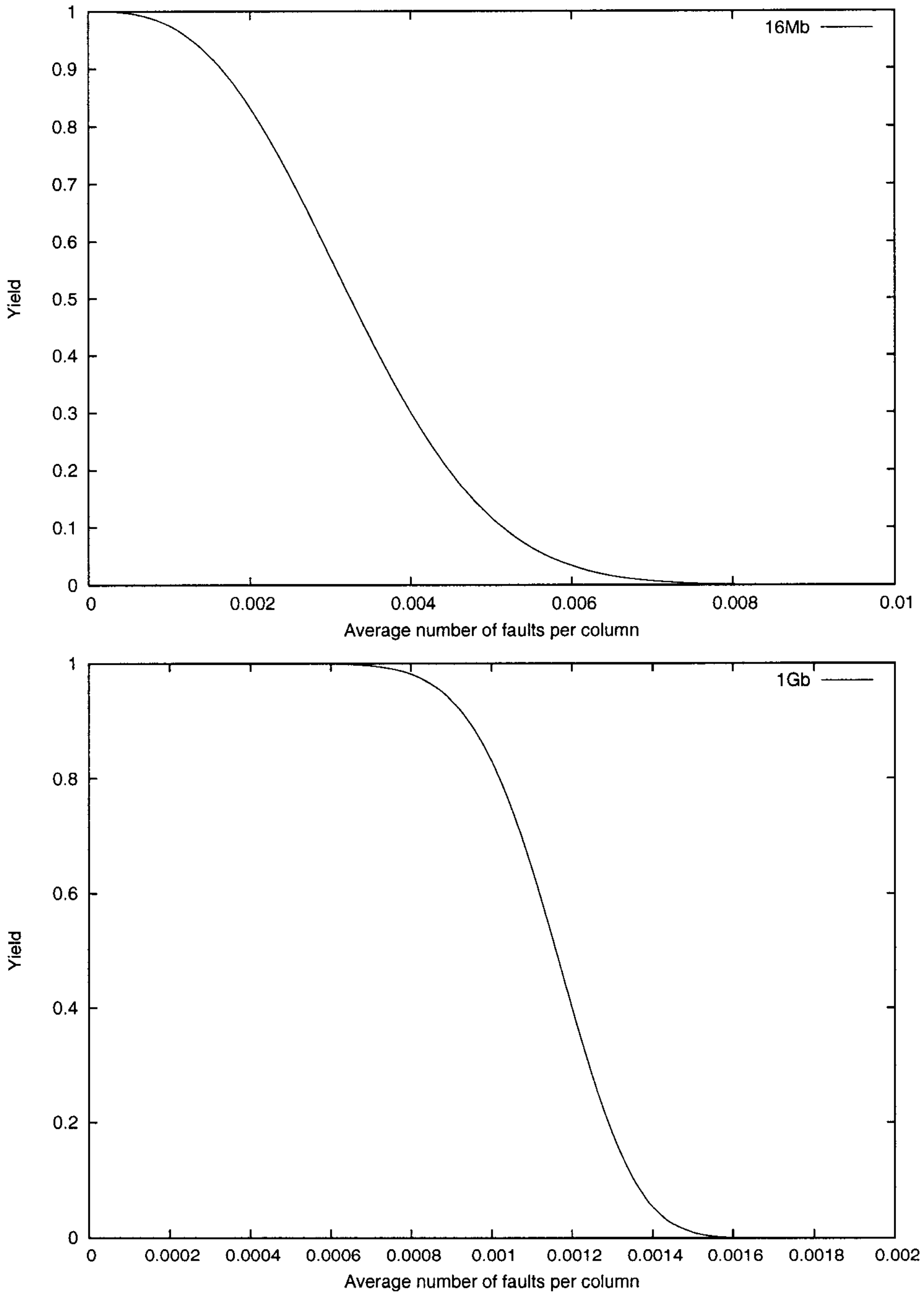


Figure C.14: Column fault model yield of a 16-Mbit and a 1-Gbit DRAM with row and column redundancy and ECC

Rows do not fail, therefore, all of the CAM entries will be operational. However, a column failure in the redundancy system will, in most cases, kill the die and must be considered (this problem is not as severe in the simulated design, due to the splitting of the compare array into several smaller arrays):

$$Y_{red_{col}} = Y_{CAM_{col}}^{2b_{addr}} \cdot Y_{flashcol}^{b_{addr}+b_{sec}} \cdot Y_{am_{col}}^{4b_{word}} \tag{C.35}$$

The $Y_{CAM_{col}}$ component is squared due to the double bit-line in the ternary CAM cell.

Since associative redundancy replaces words, the yield of a column of words must be considered instead of the individual columns. Otherwise, a word that has two faults in it may be replaced twice. The yield of a column of DRAM words is given by:

$$Y_{word_{col}} = Y_{col}^{b_{word}} \tag{C.36}$$

The contribution of failing columns to the DRAM yield is:

$$Y_{DRAM_{col}} = \sum_{i=0}^{\frac{4c}{b_{col}}} \binom{\frac{N_{col}}{b_{word}}}{i} \cdot Y_{word_{col}}^{\frac{N_{col}}{b_{word}}-i} \cdot (1 - Y_{word_{col}})^{i} \tag{C.37}$$

The $\frac{N_{col}}{b_{word}}$ component is due to this being a word-level redundancy mechanism. The entire group of columns in a word is always replaced.

The final yield is given by:

$$Y_{DRAM} = Y_{DRAM_{col}} \cdot Y_{red_{col}} \tag{C.38}$$

There are very few column failure mechanisms in the CAM, therefore the same column density as that of the DRAM array is used. As shown in Figure C.15, to match the 50% yield point of row and column, ECC and row and column redundancy with ECC, a 16-Mbit DRAM using the associative direct method requires approximately 15K, 4.5K and 14K ternary CAM entries, respectively. The 1-Gbit DRAM will require 310K, 26K and 304K ternary CAM entries, respectively.

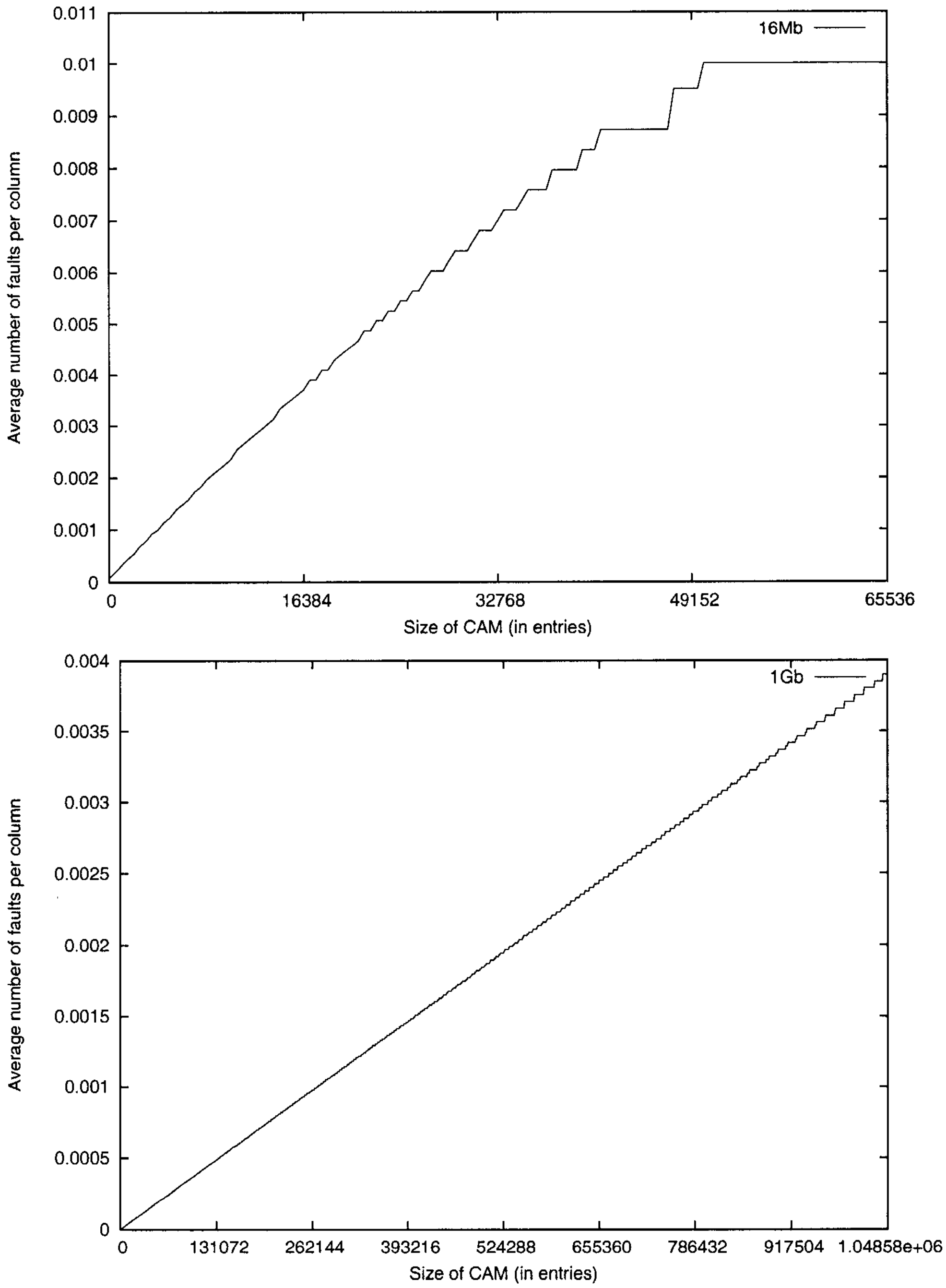


Figure C.15: 50% yield line of a 16-Mbit and a 1-Gbit DRAM with associative direct ternary CAM redundancy using a column fault model

### C.3.6 Associative Indirect Ternary CAM Redundancy

Once again, the column yield of the redundancy system has a great effect on the yield:

$$Y_{red_{col}} = Y_{CAM_{col}}^{2b_{addr}} \cdot Y_{flashcol}^{b_{addr}+b_{red}+b_{sec}} \tag{C.39}$$

Unlike associative direct, only a pointer is stored in the associative memory, giving greater yield.

The yield of a column of DRAM words is given by Equation C.36. Failing columns of words contribute to the DRAM yield as follows:

$$Y_{DRAM_{col}} = \sum_{i=0}^{c} \begin{pmatrix} \frac{N_{col}}{b_{word}} \\ i \end{pmatrix} \cdot Y_{word_{col}}^{\frac{N_{col}}{b_{word}}-i} \cdot (1 - Y_{word_{col}})^{i} \tag{C.40}$$

The yield of a DRAM using associative indirect redundancy is then:

$$Y_{DRAM} = Y_{DRAM_{col}} \cdot Y_{red_{col}} \cdot Y_{sec_{col}}^{b_{word}} \tag{C.41}$$

where $Y_{sec_{col}}$ is the column yield of the secondary memory array. The last term of the above equation will, in most cases, be equal to Equation C.36. The secondary memory array must also be large enough to contain the columns:

$$N_{sec} \geq c \cdot s \tag{C.42}$$

This equation does not include a yield. This yield is included in Equation C.41 because a column fault in the secondary memory will kill the chip, not only reduce the number of redundant elements.

The same CAM column fault densities are used as in the associative direct case. Figure C.16 shows that in order to meet the 50% yield mark of row and column, ECC and row and column redundancy with ECC, a 16-Mbit DRAM using associative indirect redundancy will require 30, 9 and 28 CAM entries with 16, 15 and 16 secondary memory address bits, respectively. The 1-Gbit DRAM will require 320, 32 and 304 ternary CAM entries with 21, 17 and 21 secondary memory address bits, respectively.

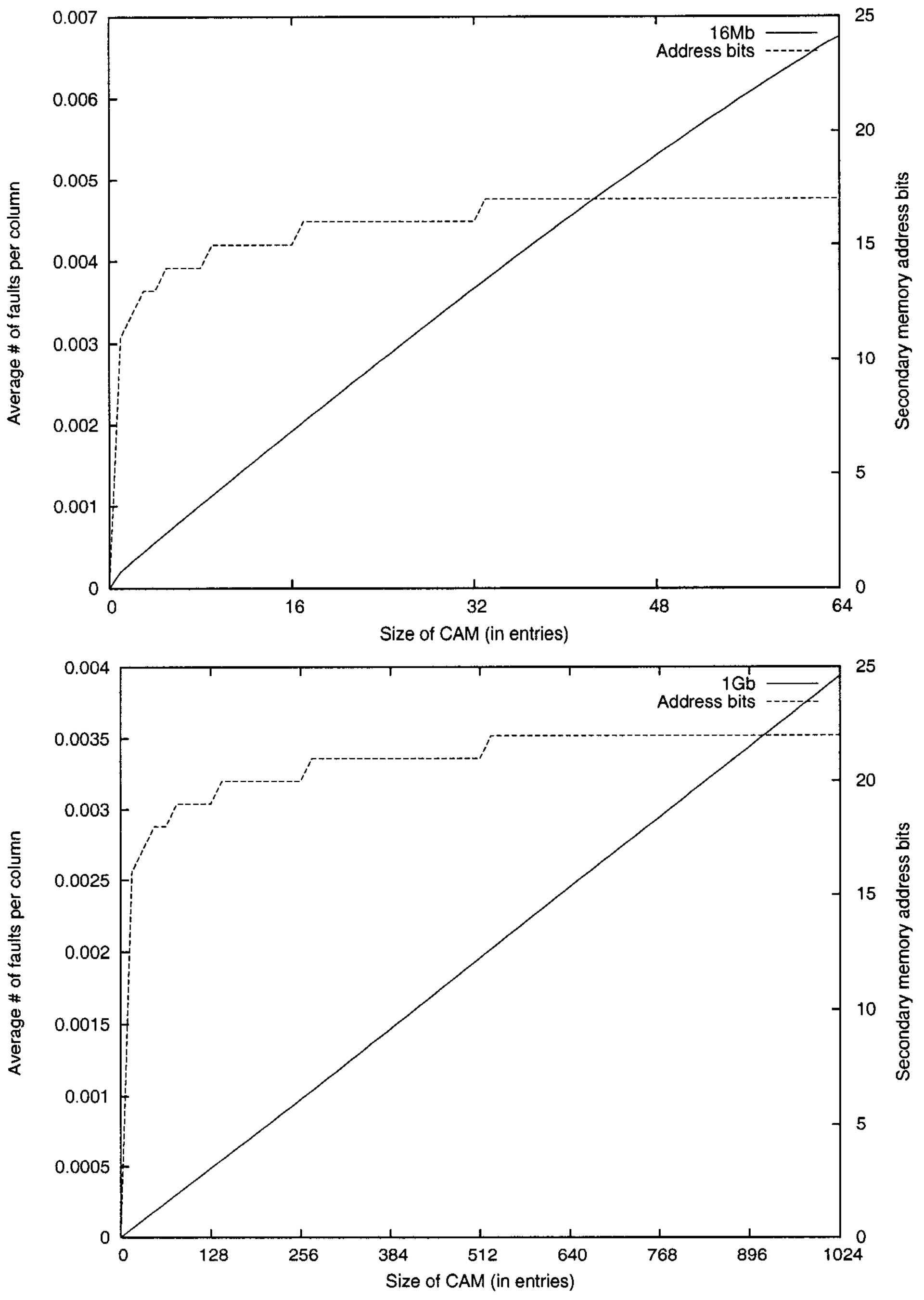


Figure C.16: 50% yield line of a 16-Mbit and a 1-Gbit DRAM with associative indirect ternary CAM redundancy using a column fault model

## C.4 Cluster Fault Model

The cluster fault model is based on the single cell fault model, however, with variable size faults. The faults are assumed to be rectangular clusters of cells with $x$ and $y$ sizes conforming to a Feris-Prabhu distribution [17]. Although this distribution is commonly used to describe the probability of a defect of a certain diameter occurring, a quantized version will be used to describe the probability of a cluster of cells of a certain dimension being covered by a fault:

$$S(\delta) = \begin{cases} c\frac{\delta^q}{\delta_o^{q+1}} & 0 \leq \delta \leq \delta_o \\ c\frac{\delta_o^{p-1}}{\delta^p} & \delta_o \leq \delta \leq \delta_M \end{cases} \tag{C.43}$$

$$c = \begin{cases} \frac{(q+1)(p-1)}{(q+p)-(q+1)\left(\frac{\delta_o}{\delta_M}\right)^{p-1}} & p \neq 1 \\ \frac{(q+1)}{1+(q+1)\ln\left(\frac{\delta_M}{\delta_o}\right)} & p = 1 \end{cases} \tag{C.44}$$

where $p$ and $q$ are parameters to fit the model, $\delta_o$ is the location of the peak in the probability density function and $\delta_M$ is the size of the largest fault.

### C.4.1 No Redundancy

The yield of a rectangular cluster of cells can be modeled as:

$$Y_{cluster} = e^{-\psi \cdot S(x) \cdot S(y)} \tag{C.45}$$

where $\psi$ is the average number of cluster faults per cell, and $S(x)$ and $S(y)$ are the probabilities of the cluster having those $x$ and $y$ dimensions.

This can be extended to all possible cluster sizes by taking the product of all possible cluster faults:

$$Y_{DRAM} = \prod_{x=1}^{x_{max}} \prod_{y=1}^{y_{max}} Y_{cluster}^{(N_x-x+1)(N_y-y+1)} \tag{C.46}$$

where $N_x$ and $N_y$ are the dimensions of the DRAM array in cells.

Figure C.17 shows the yield lines for the 16-Mbit and 1-Gbit DRAM with no redundancy. The Feris-Prabhu distributions coefficients for both the $x$ and $y$ coordinates have been set as follows:

$$\begin{aligned} p &= 2 \\ q &= 1 \\ \delta_o &= 2 \\ \delta_M &= 64 \end{aligned}$$

$p$ has been chosen to give a lower peak with a more gradual drop-off in the size. As usual, $q$ has been set for reasons of simplicity [8]. The placement of the peak is rather unusual. It should normally be smaller than the minimum feature size (in this unusual usage, less than one cell); however, clusters of that size are covered by the single cell fault model. The larger value has been chosen to emphasize the multi-cell nature of the cluster fault model. The maximum fault size has been arbitrarily set to 64 cells. These graphs show that approximately two faults for every three dies can be handled to achieve a 50% yield.

### C.4.2 Row and Column Redundancy

In order to simplify calculations, it is assumed that cluster faults do not cross page and section boundaries. As before, the yield of a cluster of cells is:

$$Y_{cluster} = e^{-\psi \cdot S(x) \cdot S(y)} \tag{C.47}$$

and the yield of the columns containing the cluster is:

$$Y_{column_{cluster}} = Y_{cluster}^{b_{column}+r_r-y+1} \tag{C.48}$$

The average yield of a column containing the cluster is given by:

$$Y_{column} = \left( Y_{column_{cluster}}^{x(p+r_c-2x+2)} \cdot \prod_{i=0}^{x-1} Y_{column_{cluster}}^{2i} \right)^{\frac{1}{p+r_c}} \tag{C.49}$$

This equation takes into account all clusters that can cross a column and requires some explanation. The first term accounts for all columns that can be traversed by a fault of a certain width. The second term accounts for the columns near the sides

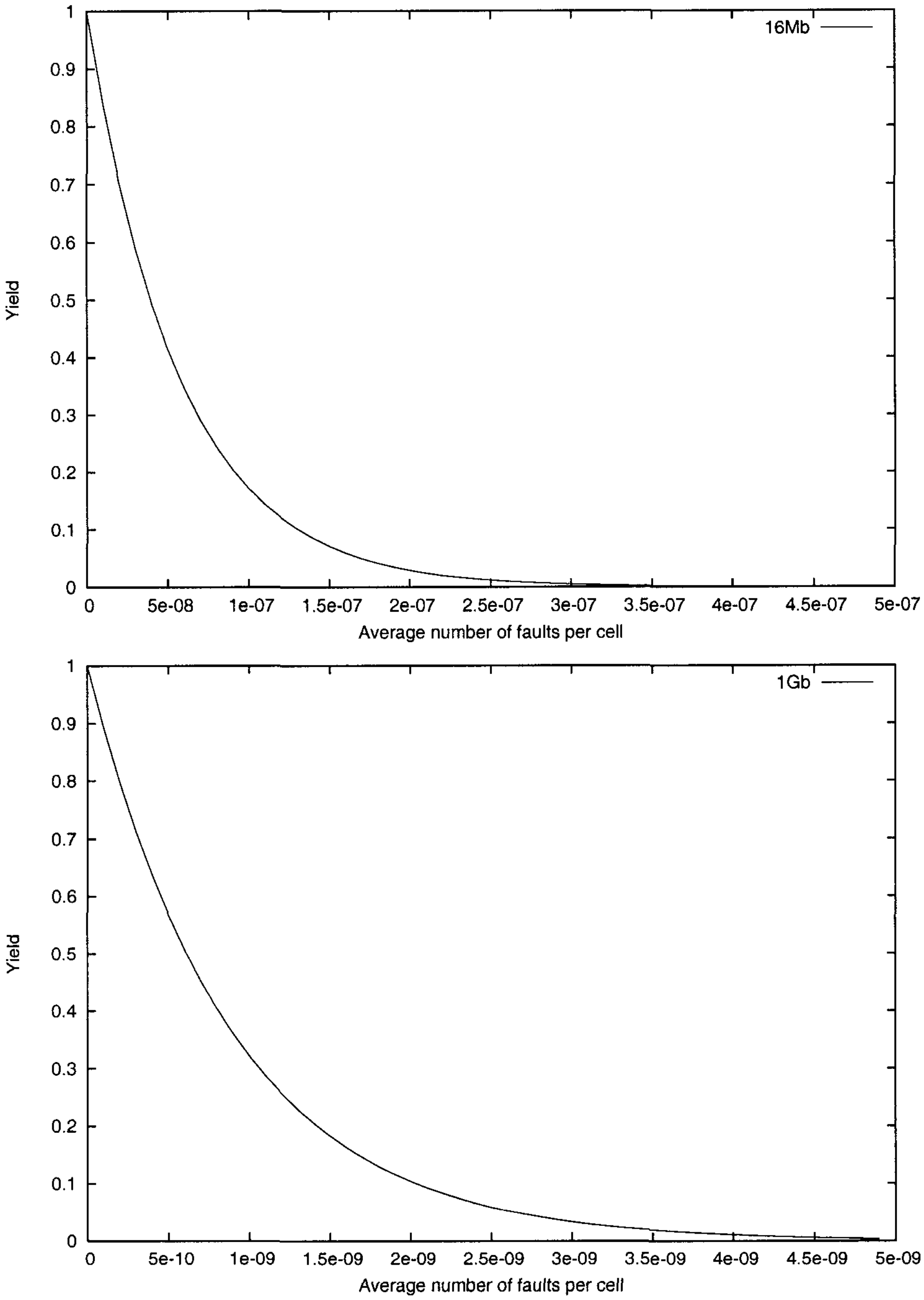


Figure C.17: Cluster fault model of a 16-Mbit and a 1-Gbit DRAM without redundancy

of the array that have reduced locations where the fault can be. The exponent $\frac{1}{p+r_c}$ averages the total for one effective column.

The column yield for one cluster fault size must then be extended to account for all possible fault sizes:

$$Y_{column_{eq}} = \prod_{x=1}^{x_{max}} \prod_{y=1}^{y_{max}} Y_{column} \tag{C.50}$$

The yield of a book is:

$$Y_{book_{cluster}} = \sum_{i=0}^{r_c} \binom{p+r_c}{i} \cdot Y_{column_{eq}}^{p+r_c-i} \cdot \left(1 - Y_{column_{eq}}\right)^i \tag{C.51}$$

This is used to determine the effective yield of a row in a DRAM with column redundancy:

$$Y_{row_{cluster}} = Y_{book_{cluster}}^{\frac{b_{row}}{b_{book}}} \tag{C.52}$$

The yield of a section is:

$$Y_{section_{cluster}} = \sum_{i=0}^{r_r} \binom{s+r_r}{i} \cdot Y_{row_{cluster}}^{s+r_r-i} \cdot (1 - Y_{row_{cluster}})^i \tag{C.53}$$

The yield of a DRAM for a cluster fault model is given by:

$$Y_{DRAM_{cluster}} = Y_{section_{cluster}}^{N_{sections}} \tag{C.54}$$

The yield versus cluster fault density is shown for the two DRAMs in Figure C.18. As expected, the yields are much lower than the single cell fault model case. They work out to about one fifth of the single cell model's values. This is because of the larger size of the faults, making it harder for the redundant rows and columns to cover and replace them.

### C.4.3 ECC Redundancy

Since ECC redundancy works along rows, the yield of the rows containing the cluster is calculated first.

$$Y_{row_{cluster}} = Y_{cluster}^{p-x+1} \tag{C.55}$$

The average yield of a row containing the cluster is given by:

$$Y_{row} = \left( Y_{row_{cluster}}^{y(b_{column}-2y+2)} \cdot \prod_{i=0}^{y-1} Y_{row_{cluster}}^{2i} \right)^{\frac{1}{b_{col}}} \tag{C.56}$$

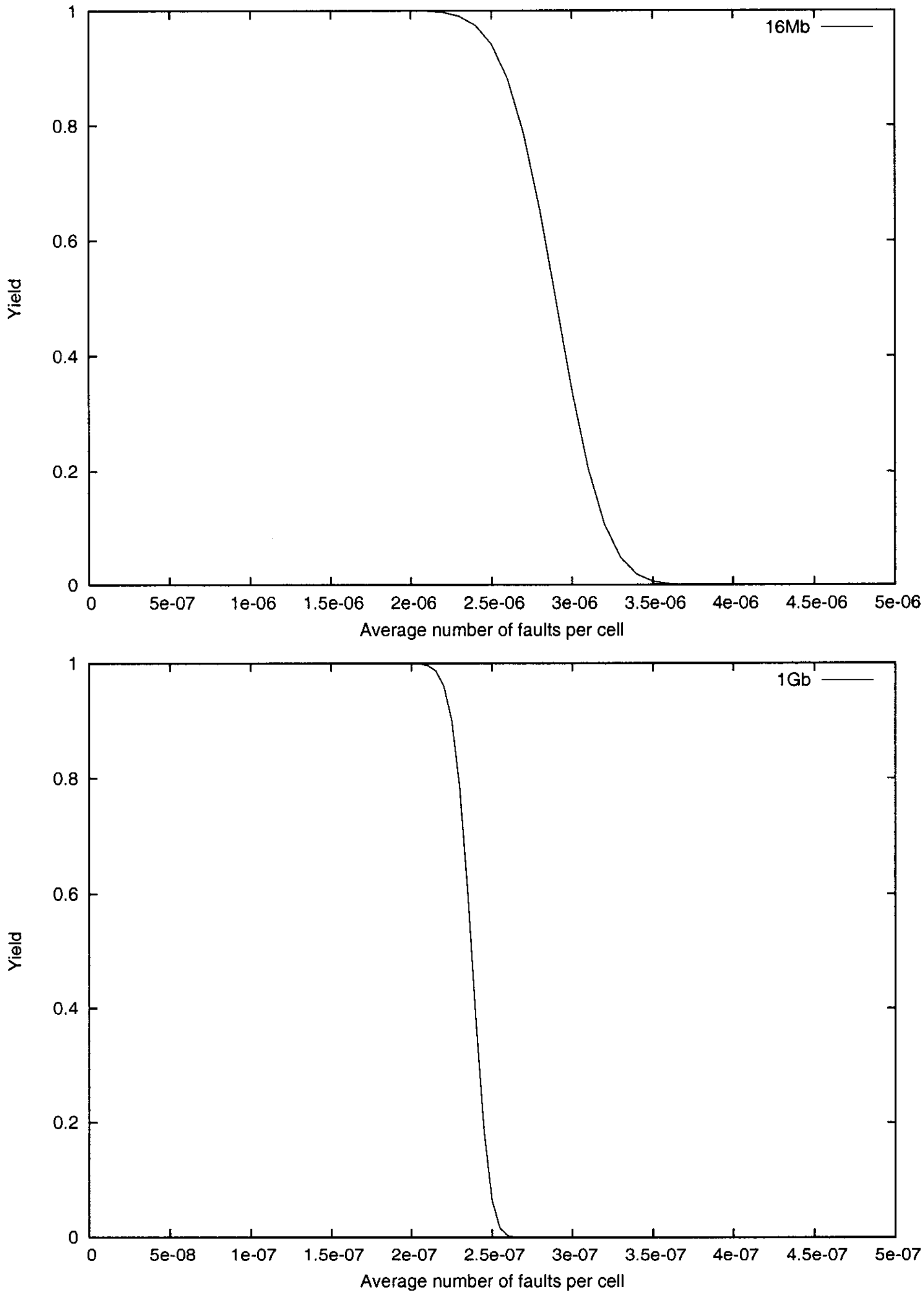


Figure C.18: Cluster fault model of a 16-Mbit and a 1-Gbit DRAM with row and column redundancy

The row yield for one cluster size is then extended for all possible cluster sizes:

$$Y_{row_{eq}} = \prod_{x=1}^{x_{max}} \prod_{y=1}^{y_{max}} Y_{row} \tag{C.57}$$

Clusters that only affect one bit-line can be repaired. The yield of such a cluster is:

$$Y_{cl_y} = e^{-\psi \cdot S(1) \cdot S(y)} \tag{C.58}$$

The average of the yields of these clusters in a row is:

$$Y_{1y_{row}} = \left( Y_{cl_y}^{y(b_{column}-2y+2)} \cdot \prod_{i=0}^{y-1} Y_{cl_y}^{2i} \right)^{\frac{1}{bcol}} \tag{C.59}$$

The product of all possible such clusters is:

$$Y_{1y} = \prod_{y=1}^{y_{max}} Y_{1y_{row}} \tag{C.60}$$

The effective yield of a codeword is then calculated:

$$Y_{cw} = Y_{row_{eq}}^{\frac{b_{cw}}{p}} + \binom{b_{cw}}{1} \cdot Y_{row_{eq}}^{\frac{b_{cw}-1}{p}} \cdot (1 - Y_{1y}) \tag{C.61}$$

The yield of a book is calculated as follows:

$$Y_{book} = Y_{cw}^{N_{cw}} \tag{C.62}$$

The yield of the DRAM is then simply the product of the yields of all of the books:

$$Y_{DRAM} = Y_{book}^{N_{books}} \tag{C.63}$$

The yields with ECC redundancy for the 16-Mbit and 1-Gbit DRAMs are shown in Figure C.19. Both give about one fifth of the yield with no redundancy. The extra elements required for ECC cannot cope with the arbitrary sizes of cluster-type faults. This should improve somewhat for DRAMs with interleaved codewords.

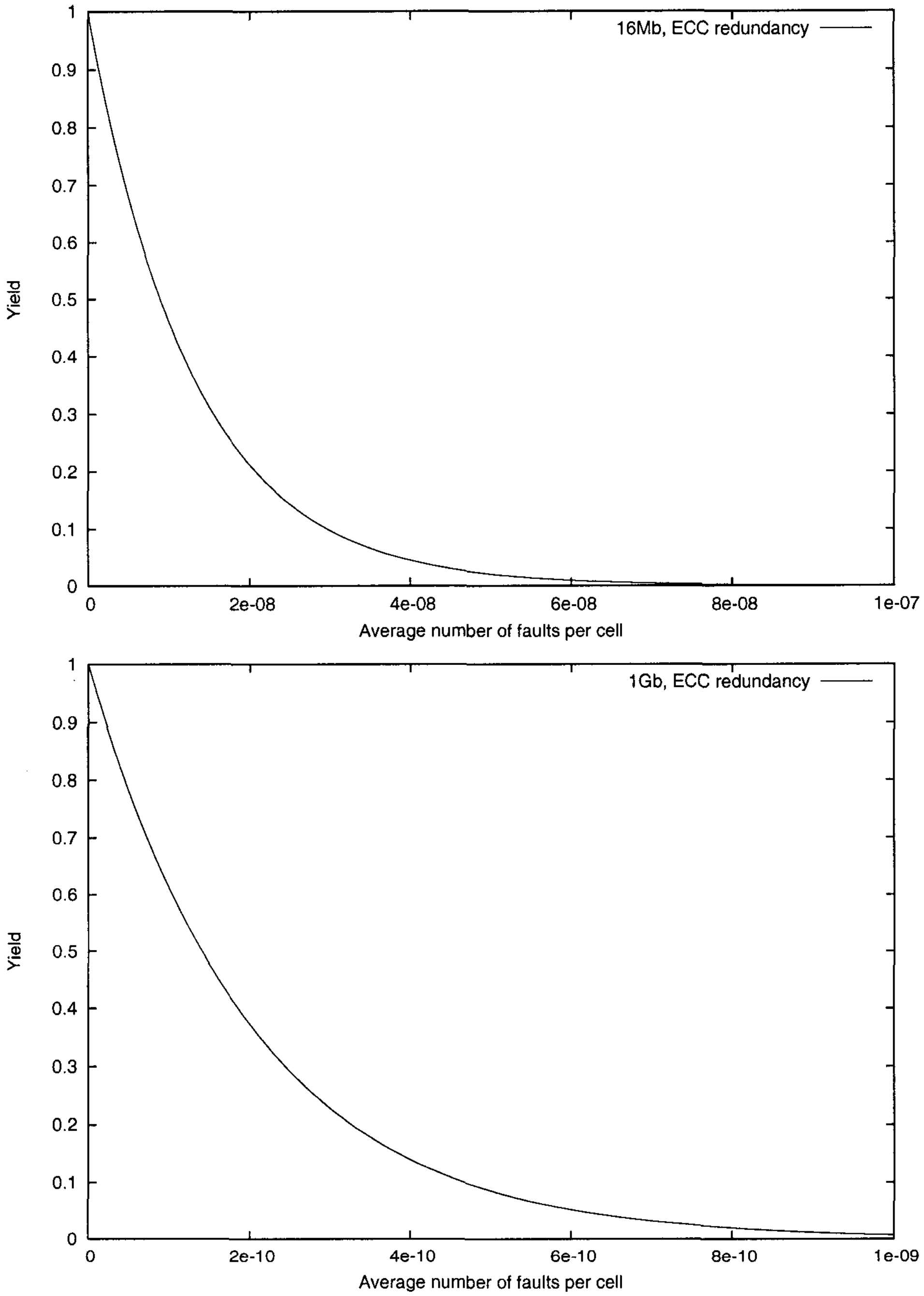


Figure C.19: Cluster fault model of a 16-Mbit and a 1-Gbit DRAM with ECC redundancy

### C.4.4 Row and Column Redundancy with ECC

The yield for row and column redundancy with ECC added is calculated the same as for row and column redundancy only, with Equation C.52 replaced by the following equations:

$$Y_{cw} = Y_{book_{cluster}}^{\frac{b_{cw}}{b_{book}}} + b_{cw} \cdot Y_{book_{cluster}}^{\frac{b_{cw}-1}{b_{book}}} \cdot (1 - Y_{1y}) \tag{C.64}$$

$$Y_{row_{cluster}} = Y_{cw}^{\frac{b_{row}}{b_{cw}}} \tag{C.65}$$

The term $(1 - Y_{1y})$ imparts a significant error because many failing cells may already have been repaired in the column redundancy equation.

This modification produces the plots shown in Figure C.20. The addition of ECC gives nearly a 30% increase in the number of cluster faults that can be handled over row and column redundancy alone.

### C.4.5 Associative Direct Ternary CAM Redundancy

Since ternary CAM redundancy replaces words, it gives different performance for DRAMs that use interleaved or contiguous word placement across a page. Although the cases are similar, both will be considered.

It is assumed that a cluster fault can occur anywhere in the DRAM, Although it is not very likely that cluster faults will cross sense amplifiers or word-line drivers, and even if this occurs, it will result in row or column failures, the difference in yields should be negligible. The entire DRAM is considered because this redundancy mechanism is pooled for the entire die and not specific to pages, sections or codewords, as with column, row or ECC redundancies.

As before, we begin with the yield of a cluster as in Equation C.45. A cluster of a certain size cannot overlap another. This means that there are:

$$N_{cl} = \frac{N_{bits}}{x \cdot y} \tag{C.66}$$

possible non-overlapping places where a cluster of faults can be located.

From this, we find the yield of DRAM due to the specific cluster:

$$Y_{DRAM_{cluster}} = \sum_{i=0}^{c_{xy}} \binom{N_{cl}}{i} \cdot Y_{cluster}^{N_{cl}-i} \cdot (1 - Y_{cluster})^{i} \tag{C.67}$$

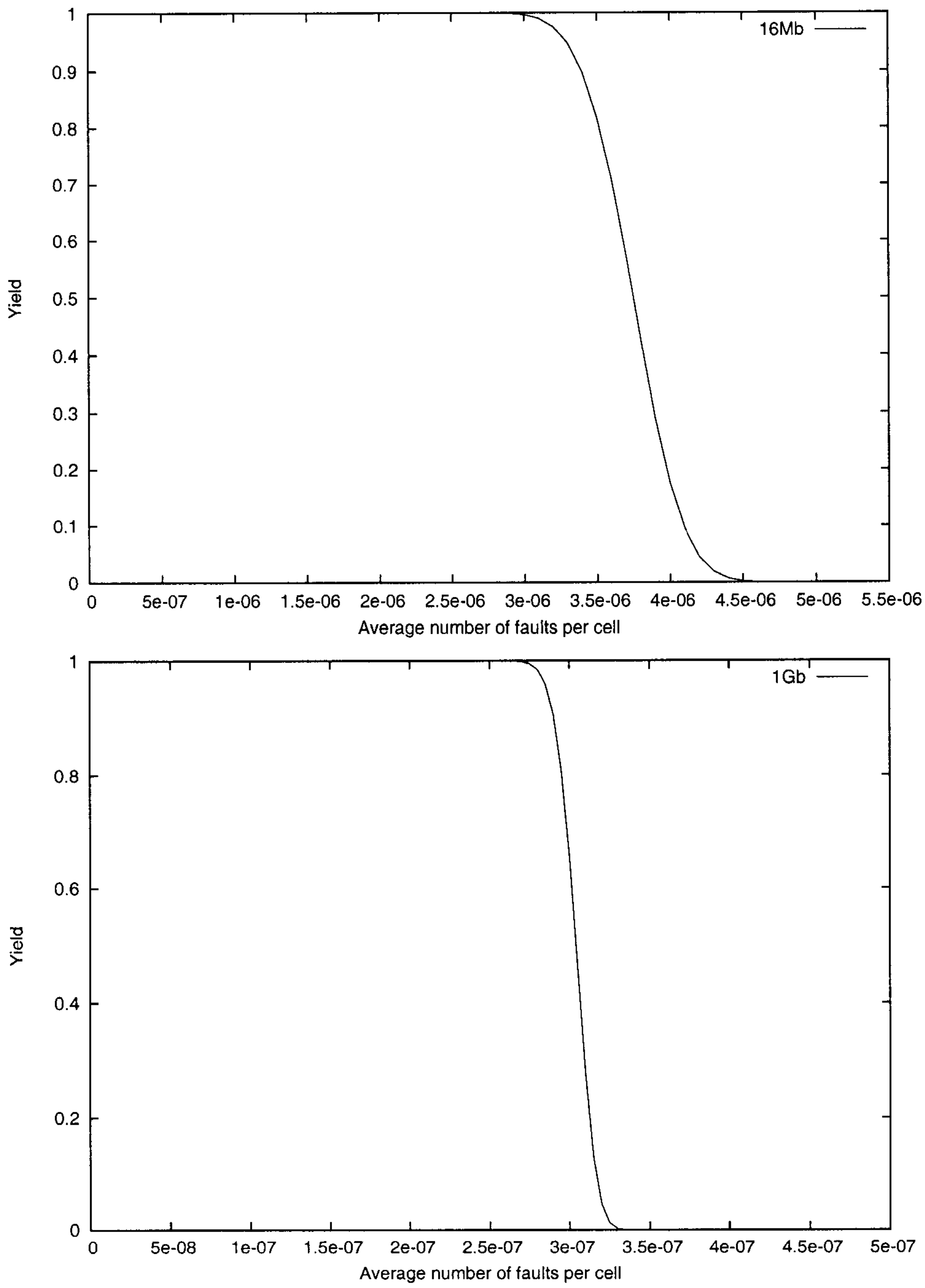


Figure C.20: Cluster fault model of a 16-Mbit and a 1-Gbit DRAM with row and column redundancy and ECC

The value $c_{xy}$ is the portion of redundant elements devoted to the replacement of a cluster of a certain size. For each faulty cluster size, the number of effective replacement clusters, $c_{xy}$ can be estimated by:

$$c_{xy} = \frac{N_{CAM_e} \cdot S(x) \cdot S(y)}{c_e} \tag{C.68}$$

where $N_{CAM_e}$ is the effective total number of CAM entries after yield and $c_e$ is the average number of CAM entries required to replace this specific cluster size. The number of CAM entries after yield can be roughly calculated as in Equation C.17. The number of required CAM entries depends on the fault size, the word size and whether the word bits are contiguous or interleaved.

The yields due to each cluster size are then multiplied together to give the final yield of the DRAM:

$$Y_{DRAM} = \prod_{x=1}^{x_{max}} \prod_{y=1}^{y_{max}} Y_{DRAM_{cluster}} \tag{C.69}$$

Figure 4.5 shows the 50% yield line for DRAMs with associative direct ternary CAM redundancy. For simplicity, it is assumed that cluster faults do not occur in the CAM, flash and redundant memory. Instead, a single cell yield model is used in the redundant path with the same fault density as the cluster fault density. For both contiguous and interleaved 16-Mbit DRAMs, less than 1K CAM entries are required to more than best all conventional redundancy types. The contiguous 1-Gbit DRAM will require 3K CAM entries to match row and column redundancies, less than 1K to match ECC and slightly more and 3K CAM entries to match row and column redundancy with ECC. The interleaved 1-Gbit DRAM will require 6K CAM entries to best conventional row and column redundancy, less than 1K to best ECC and 12K to best row and column redundancy with ECC.

### C.4.6 Associative Indirect Ternary CAM Redundancy

The yield equation for associative indirect is the same as for associative direct; however, the number of CAM entries for most faults is much lower since there is a much larger limit on the number of words that can be replaced by a single CAM

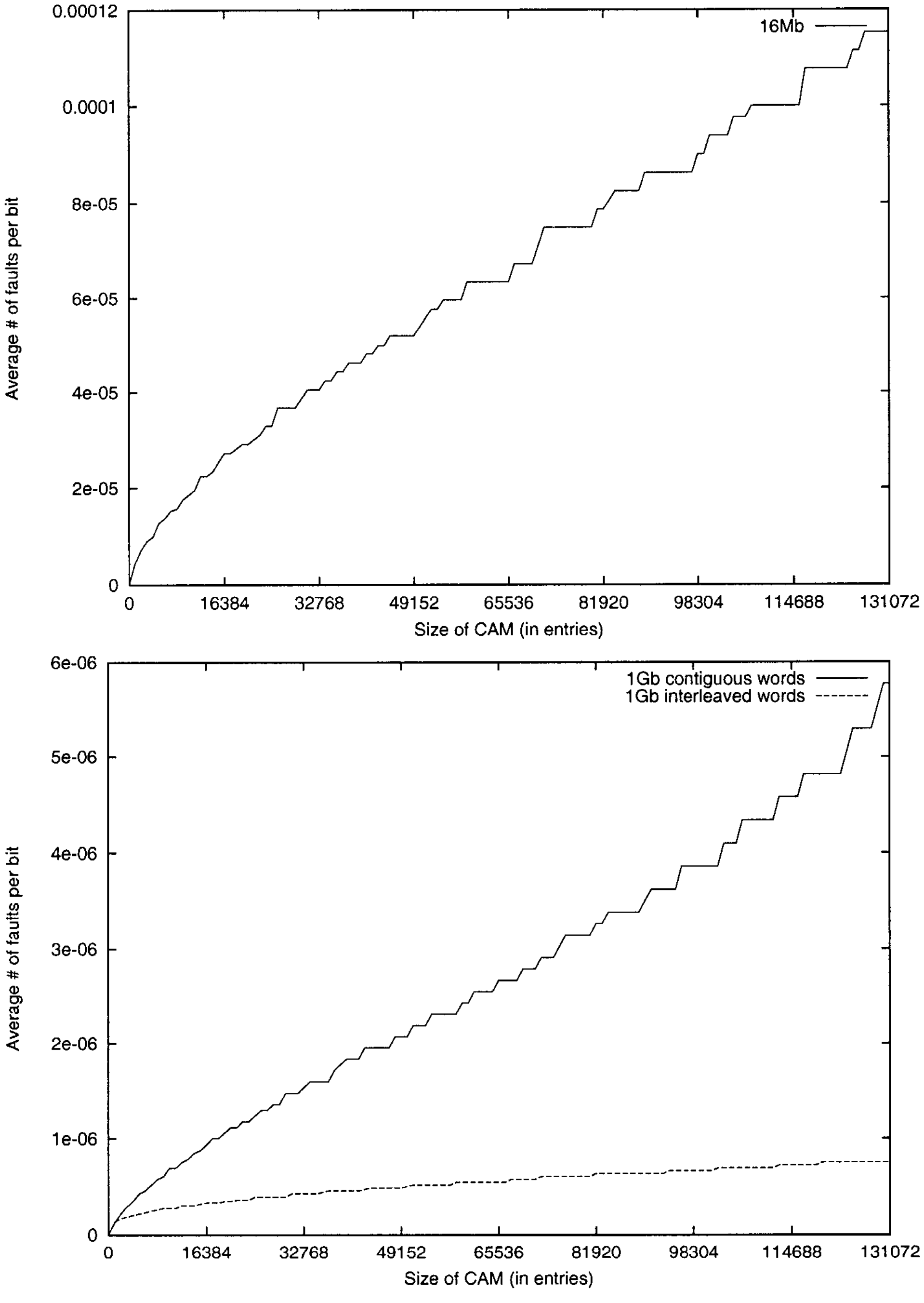


Figure C.21: 50% yield line of a 16-Mbit and a 1-Gbit DRAM with associative direct ternary CAM redundancy using a cluster fault model

entry. The secondary memory must be large enough to contain all of the replaced words. It should be able to hold at least:

$$N_{sec} \geq \sum_{x=1}^{x_{max}} \sum_{y=1}^{y_{max}} c_{xy} \cdot N_{xy} \tag{C.70}$$

where $N_{xy}$ is the average number of words required to replace this specific cluster size.

The 50% yield lines for the 16-Mbit and 1-Gbit DRAMs using associative indirect ternary CAM redundancy are shown in Figure C.22. In the case of the 16-Mbit DRAM, 256 CAM entries with 10 address bits and 384 CAM entries with 11 bits are required to match the performance of row and column redundancy and row and column redundancy with ECC, respectively. To match ECC, less than 10 CAM entries, with 2 address bits, are required. For the contiguous 1-Gbit DRAMs, 2K CAM entries with 16 address bits are required to beat the 50% yield points of row and column and row and column redundancy with ECC. The interleaved 1-Gbit DRAM will require 2K CAM entries with 14 address bits and 3K entries with 15 bits in for the same 50% yield points. In both cases, much less than 128 CAM entries will be required to match the performance of ECC.

## C.5 Combined Fault Model

### C.5.1 No Redundancy

For DRAMs with no redundancy, the yield equations already derived for the four models may be simply multiplied together:

$$Y_{DRAM} = e^{-\lambda \cdot b_{DRAM}} e^{-\chi \cdot N_{col}} e^{-\rho \cdot N_{row}} \prod_{x=1}^{x_{max}} \prod_{y=1}^{y_{max}} e^{-\psi \cdot S(x) \cdot S(y) \cdot (N_x - x+1)(N_y - y+1)} \tag{C.71}$$

The yields of the 16-Mbit and 1-Gbit DRAM are shown in Figure C.23.

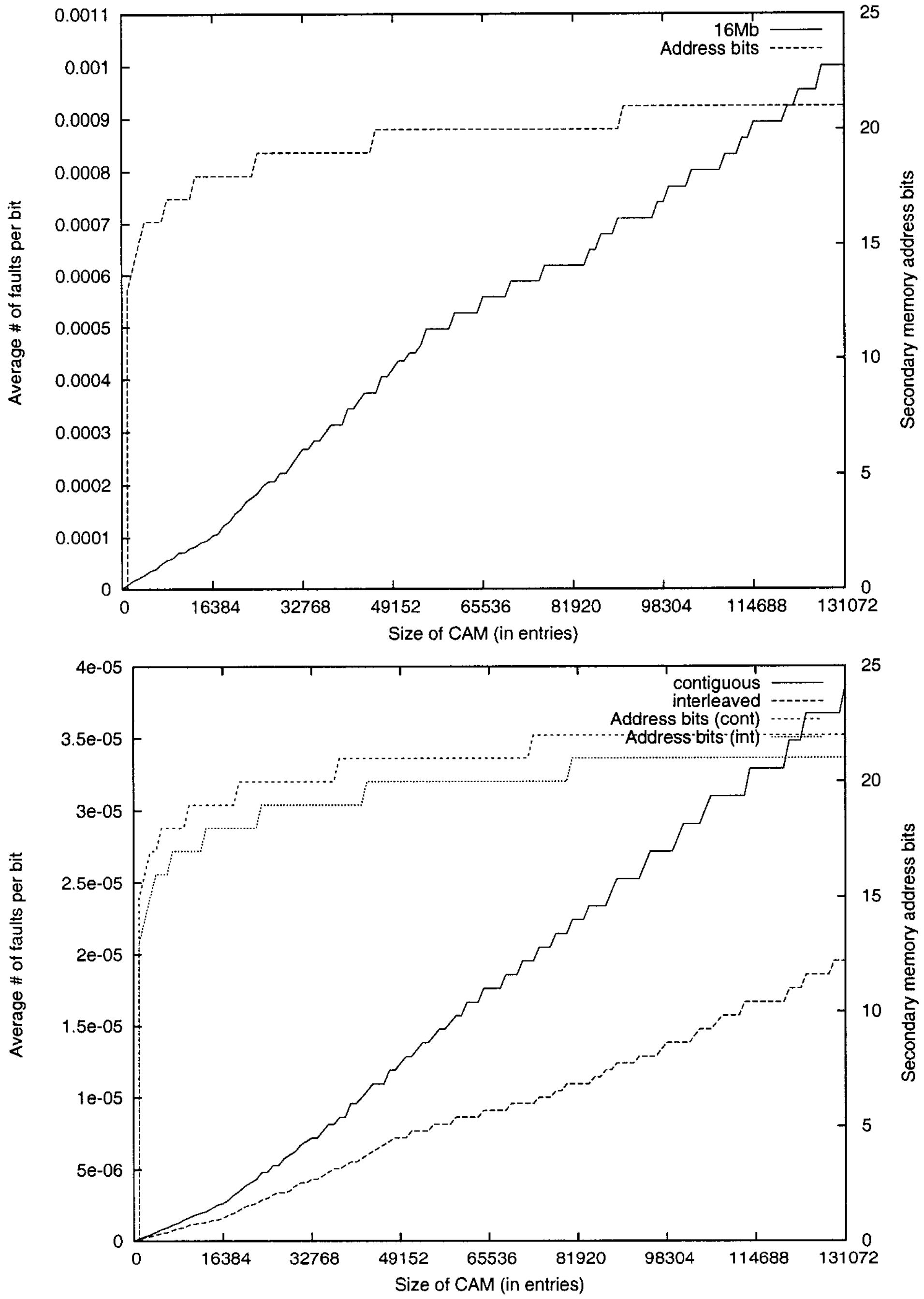


Figure C.22: 50% yield line of a 16-Mbit and a 1-Gbit DRAM with associative indirect ternary CAM redundancy using a cluster fault model

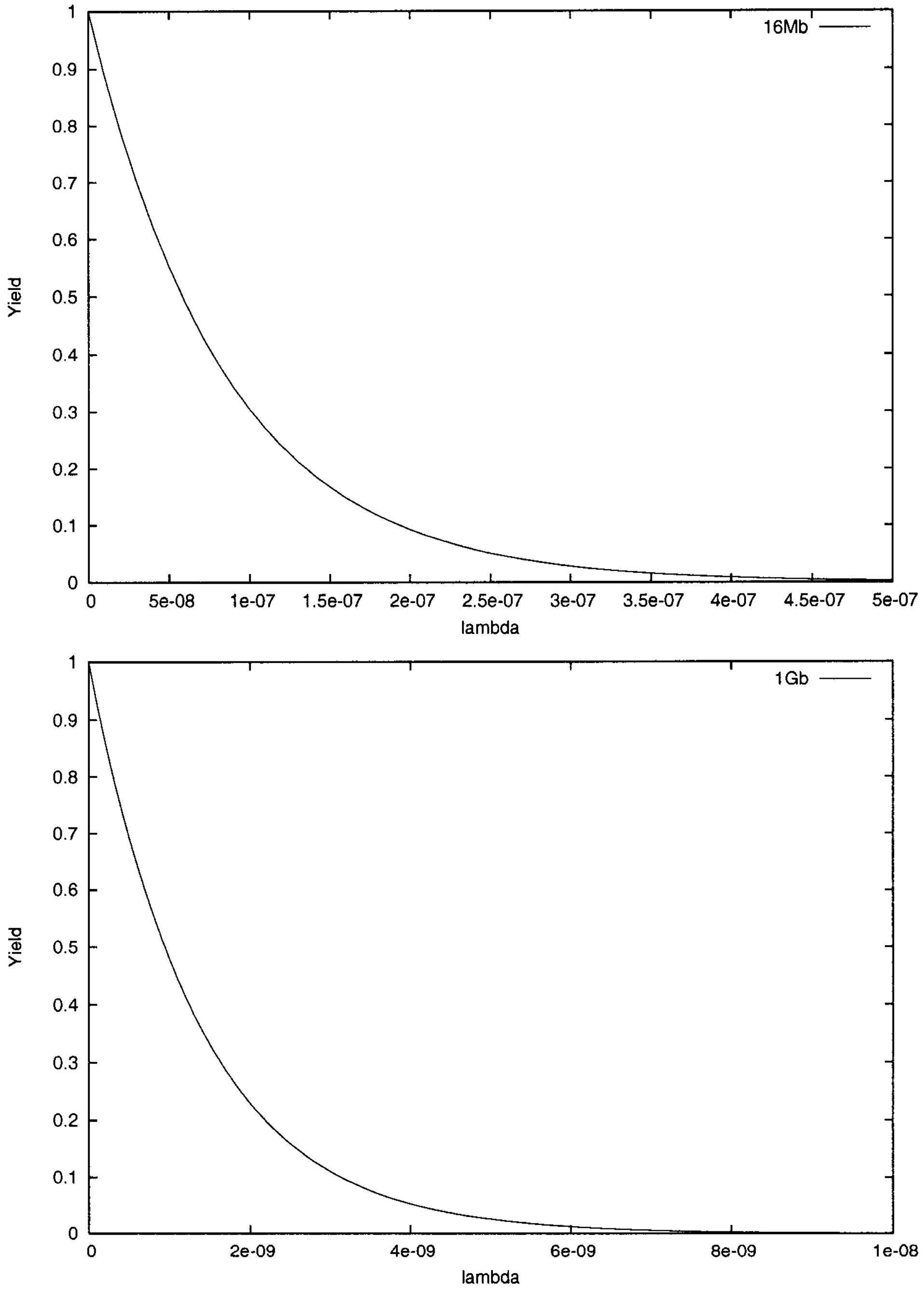


Figure C.23: Combined fault model of a 16-Mbit and a 1-Gbit DRAM without redundancy

### C.5.2 Row and Column Redundancy

The yield of a column, taking into account single cell, column and cluster faults is first determined:

$$Y_{col} = e^{-\lambda \cdot (b_{col}+r_r)} e^{-\chi} \prod_{x=1}^{x_{max}} \prod_{y=1}^{y_{max}} Y_{column_{eq}} \tag{C.72}$$

where $Y_{column_{eq}}$ is the column yield for all cluster sizes from Equations C.47 through C.50.

The yield of a book is found as usual for column redundancy:

$$Y_{book} = \sum_{i=0}^{r_c} \begin{pmatrix} p + r_c \\ i \end{pmatrix} \cdot Y_{col}^{p+r_c-i} \cdot (1 - Y_{col})^i \tag{C.73}$$

Then, the effective yield of a row, taking into account row faults, can be calculated:

$$Y_{row} = Y_{book}^{\frac{b_{row}}{b_{book}}} e^{-\rho} \tag{C.74}$$

From this point, the section and DRAM yield are found as usual:

$$Y_{section} = \sum_{i=0}^{r_r} \begin{pmatrix} s + r_r \\ i \end{pmatrix} \cdot Y_{row}^{s+r_r-i} \cdot (1 - Y_{row})^i \tag{C.75}$$

$$Y_{DRAM} = Y_{section}^{N_{sections}} \tag{C.76}$$

The yields of the 16-Mbit and 1-Gbit DRAM are shown in Figure C.24. Adding row and column redundancies allows the 16-Mbit DRAM to handle 135 times greater fault densities and still achieve 50% yield. The 1-Gbit DRAM can handle 2180 times greater fault densities than a DRAM without redundancy.

### C.5.3 ECC Redundancy

The yield of a codeword, taking into account single cell and cluster faults is calculated:

$$Y_{sc} = e^{-\lambda} \tag{C.77}$$

$$Y_{cw} = \left[Y_{sc}^{b_{cw}} + b_{cw} \cdot Y_{sc}^{b_{cw}-1}(1 - Y_{sc})\right] \left[Y_{row_{eq}}^{\frac{b_{cw}}{p}} + b_{cw} \cdot Y_{row_{eq}}^{\frac{b_{cw}}{p}-1} \cdot (1 - Y_{1y})\right] \tag{C.78}$$

where $Y_{row_{eq}}$ and $Y_{1y}$ are calculated from Equations C.55 through C.60.

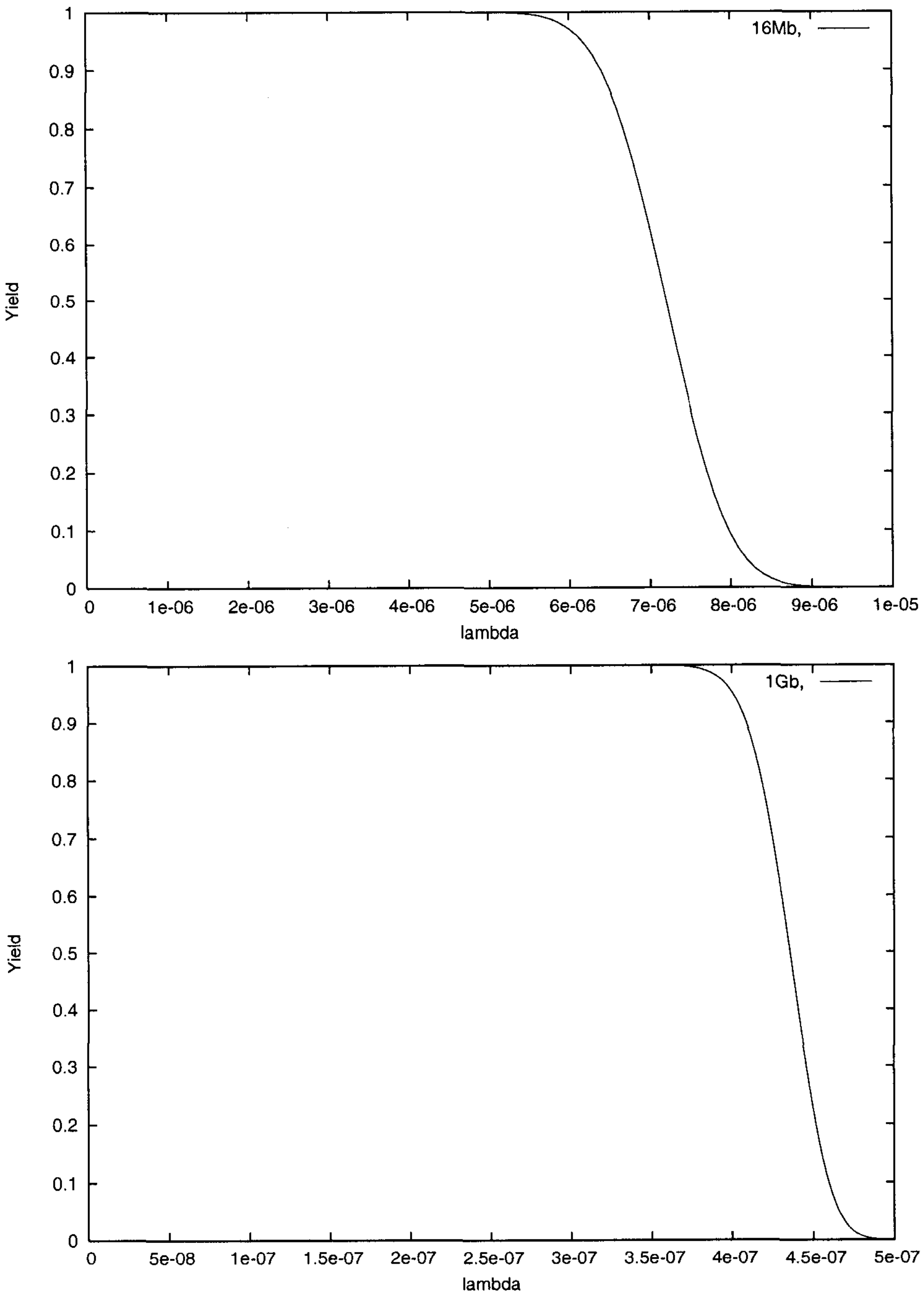


Figure C.24: Combined fault model of a 16-Mbit and a 1-Gbit DRAM with row and column redundancy

The yield of a column of codewords is then calculated:

$$Y_{col} = e^{-\chi} \tag{C.79}$$

$$Y_{col_{cw}} = Y_{cw}^{b_{col}} \left[ Y_{col}^{b_{cw}} + b_{cw} \cdot Y_{col}^{b_{cw}-1}(1 - Y_{col}) \right] \tag{C.80}$$

Finally, the yield of a book, and the yield of the DRAM, taking into account row yield, are found:

$$Y_{row} = e^{-\rho} \tag{C.81}$$

$$Y_{book} = Y_{col_{cw}}^{N_{cwp}} \tag{C.82}$$

$$Y_{DRAM} = Y_{book}^{N_{books}} \cdot Y_{row}^{N_{rows}} \tag{C.83}$$

where $N_{cwp}$ is the number of codewords across a page.

The yields of the 16-Mbit and 1-Gbit DRAM are shown in Figure C.25. Adding ECC redundancies allows the 16-Mbit DRAM to handle only 6.7 times greater fault densities and still achieve 50% yield. The 1-Gbit DRAM can handle 1.6 times greater fault densities. This low number, compared to row and column redundancy, is expected due to the previously shown ECC difficulty with row and cluster faults.

### C.5.4 Row and Column Redundancy with ECC

Equations C.77 through C.80 are first used to determine the yield of a column of codewords. Then, the equivalent yield of a column is found:

$$Y_{col_{eq}} = Y_{col_{cw}}^{\frac{1}{b_{cw}}} \tag{C.84}$$

From this point, the yield of the DRAM is found, as in the row and column redundancy case, with Equations C.73 through C.76.

The yields of the 16-Mbit and 1-Gbit DRAM are shown in Figure C.26. Adding row and column redundancies allows the 16-Mbit DRAM to handle 116 times greater fault densities and still achieve 50% yield. The 1-Gbit DRAM can handle 408 times greater fault densities. This is rather surprising. Intuitively, ECC should handle single cell faults, leaving more redundant rows and columns for other fault types, but instead, the extra cells required for ECC have lowered yield significantly compared to row and column only.

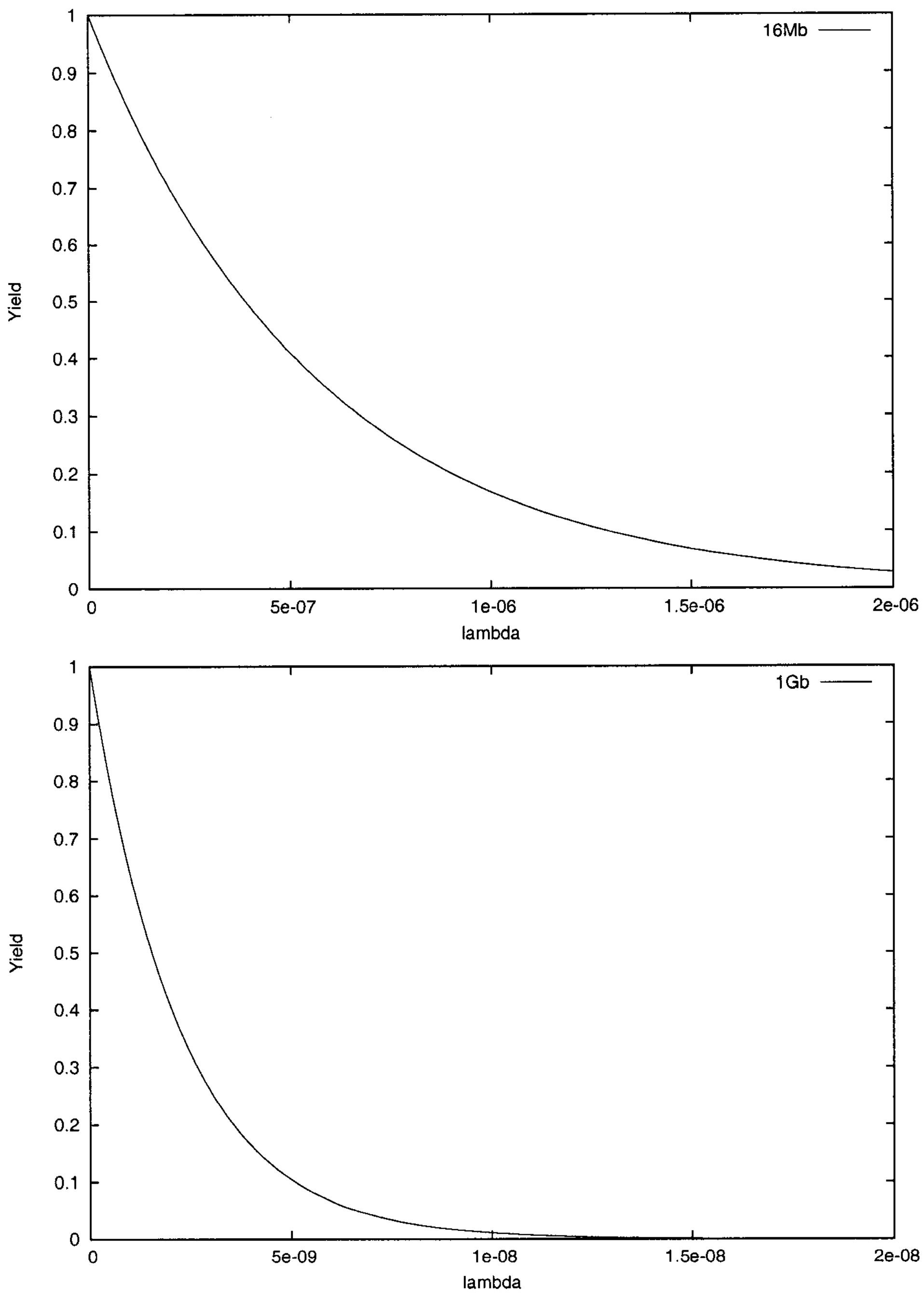


Figure C.25: Combined fault model of a 16-Mbit and a 1-Gbit DRAM with ECC redundancy

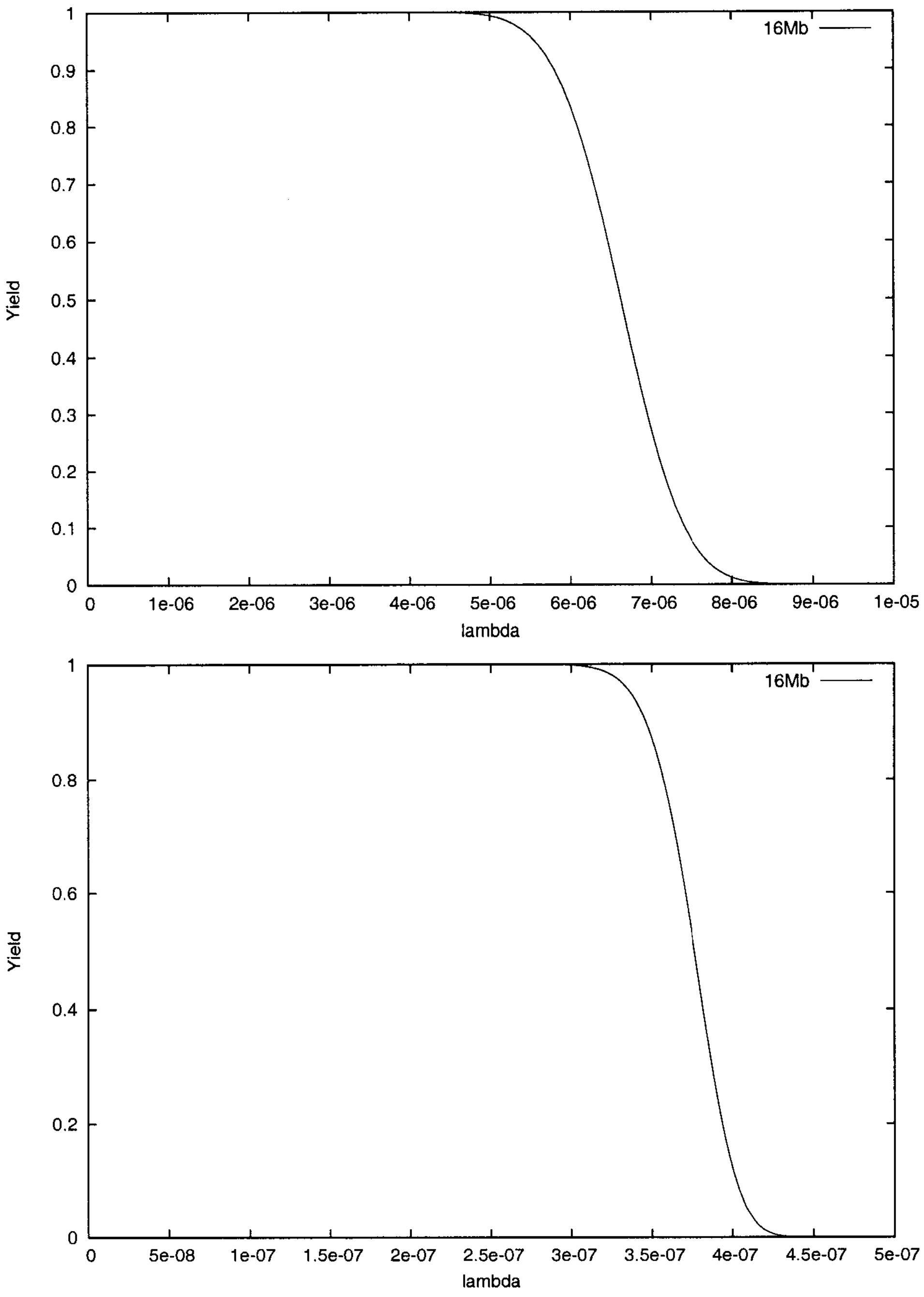


Figure C.26: Combined fault model of a 16-Mbit and a 1-Gbit DRAM with row and column redundancy and ECC

### C.5.5 Associative Direct Ternary CAM Redundancy

First, the number of CAM entries after yield must be found. This is split into four parts, one for each of the fault types:

$$Y_{re_{sc}} = Y_{CAM_{sc}}^{b_{addr}} \cdot Y_{flash_{sc}}^{b_{addr}+b_{sec}} \cdot Y_{am_{sc}}^{4b_{word}} \tag{C.85}$$

$$Y_{re_{row}} = Y_{row} \tag{C.86}$$

$$Y_{re_{col}} = Y_{col}^{3b_{addr}+b_{sec}+4b_{word}} \tag{C.87}$$

$$\begin{aligned} Y_{rc_{cl}} \quad = \quad & \prod_{x=1}^{2b_{addr}} \prod_{y=1}^{y_{max}} e^{-\psi\cdot(2b_{addr}-x+1)\cdot(N_c g-y+1)\cdot S(x)\cdot S(y)} \cdot \\ & \prod_{x=1}^{b_{addr}+b_{sec}} \prod_{y=1}^{y_{max}} e^{-\psi\cdot(b_{addr}+b_{sec}-x+1)\cdot(N_c g-y+1)\cdot S(x)\cdot S(y)} \cdot \\ & \prod_{x=1}^{w_{max}} \prod_{y=1}^{y_{max}} e^{-\psi\cdot(4b_{word}-x+1)\cdot(N_d g-y+1)\cdot S(x)\cdot S(y)} \end{aligned} \tag{C.88}$$

$$c = Y_{re_{sc}} \cdot Y_{re_{row}} \cdot Y_{re_{col}} \cdot Y_{re_{cl}} \cdot N_{CAM} \tag{C.89}$$

Next, the remaining working CAM entries, $N_e$ must be split up for each fault type. This is an optimization problem that is solved differently for each DRAM. The idea is to maximize the yield of all fault types for each fault density. See the source code in Appendix D.5.5 for how this is achieved in these two cases. The determined numbers are then entered into their respective individual fault type equation:

$$Y_{d_{sc}} = \sum_{i=0}^{N_{e_{sc}}} \begin{pmatrix} W \\ i \end{pmatrix} \cdot Y_{word}^{W-i} \cdot (1 - Y_{word})^{i} \tag{C.90}$$

$$Y_{d_{row}} = \sum_{i=0}^{\frac{4N_{e_{row}}\cdot b_{word}}{b_{row}}} \begin{pmatrix} N_{row} \\ i \end{pmatrix} \cdot Y_{row}^{N_{row}-i} \cdot (1 - Y_{row})^{i} \tag{C.91}$$

$$Y_{d_{col}} = \sum_{i=0}^{\frac{4N_{e_{col}}}{b_{col}}} \begin{pmatrix} \frac{N_{col}}{b_{word}} \\ i \end{pmatrix} \cdot Y_{word_{col}}^{\frac{N_{col}}{b_{word}}-i} \cdot (1 - Y_{word_{col}})^{i} \tag{C.92}$$

$$c_{xy} = \frac{N_{e_{cl}} \cdot S(x) \cdot S(y)}{c_e} \tag{C.93}$$

$$N_{cl} = \frac{N_{bits}}{x \cdot y} \tag{C.94}$$

$$Y_{d_{cl}} = \sum_{i=0}^{c_{xy}} \left( \begin{array}{c} N_{cl} \\ i \end{array} \right) \cdot Y_{cl}^{N_{cl}-i} \cdot (1 - Y_{cl})^{i} \tag{C.95}$$

$$Y_{d_{cl}} = \prod_{x=1}^{x_{max}} \prod_{y=1}^{y_{max}} Y_{DRAM_{cluster}} \tag{C.96}$$

$$Y_{DRAM} = Y_{d_{sc}} Y_{d_{row}} Y_{d_{col}} Y_{d_{cl}} \tag{C.97}$$

See Figure C.27 for plots of the DRAM yield with associative direct redundancy. The 2K entry associative direct method in a 16-Mbit DRAM does not supply enough entries to be match the 50% yield performance of row and column redundancy. It only handles 85% of the fault density. On the other hand, the 1-Gbit DRAM with 73K entries can handle 15 times and 7 times greater fault density in contiguous and interleaved word DRAM, respectively, than row and column redundancy. The discontinuities in these graphs are caused by entries being used for different fault types. The large jump in the 16-Mbit DRAM plot signifies that fault coverage can be better optimized for low fault densities.

### C.5.6 Associative Indirect Ternary CAM Redundancy

As has been the case with the separate fault models, the associative indirect case closely follows the associative direct case. The number of CAM entries is found by parts:

$$Y_{re_{sc}} = Y_{CAM_{sc}}^{b_{addr}} \cdot Y_{flash_{sc}}^{b_{addr}+b_{sec}+b_{red}} \tag{C.98}$$

$$Y_{re_{row}} = Y_{row} \tag{C.99}$$

$$Y_{re_{col}} = Y_{col}^{3b_{addr}+b_{sec}+b_{red}} \tag{C.100}$$

$$\begin{aligned} Y_{rc_{cl}} \quad = \quad & \prod_{x=1}^{2b_{addr}} \prod_{y=1}^{y_{max}} e^{-\psi\cdot(2b_{addr}-x+1)\cdot(N_c g-y+1)\cdot S(x)\cdot S(y)} \, . \\ & \prod_{x=1}^{(b_{addr}+b_{sec}+b_{red})} \prod_{y=1}^{y_{max}} e^{-\psi\cdot(b_{addr}+b_{sec}+b_{red}-x+1)\cdot(N_c g-y+1)\cdot S(x)\cdot S(y)} \end{aligned} \tag{C.101}$$

$$c = Y_{re_{sc}} \cdot Y_{re_{row}} \cdot Y_{re_{col}} \cdot Y_{re_{cl}} \cdot N_{CAM} \tag{C.102}$$

The number of CAM entries, $N_e$, devoted to each fault type is found through an iterative process that does not lend itself well to mathematical representation.

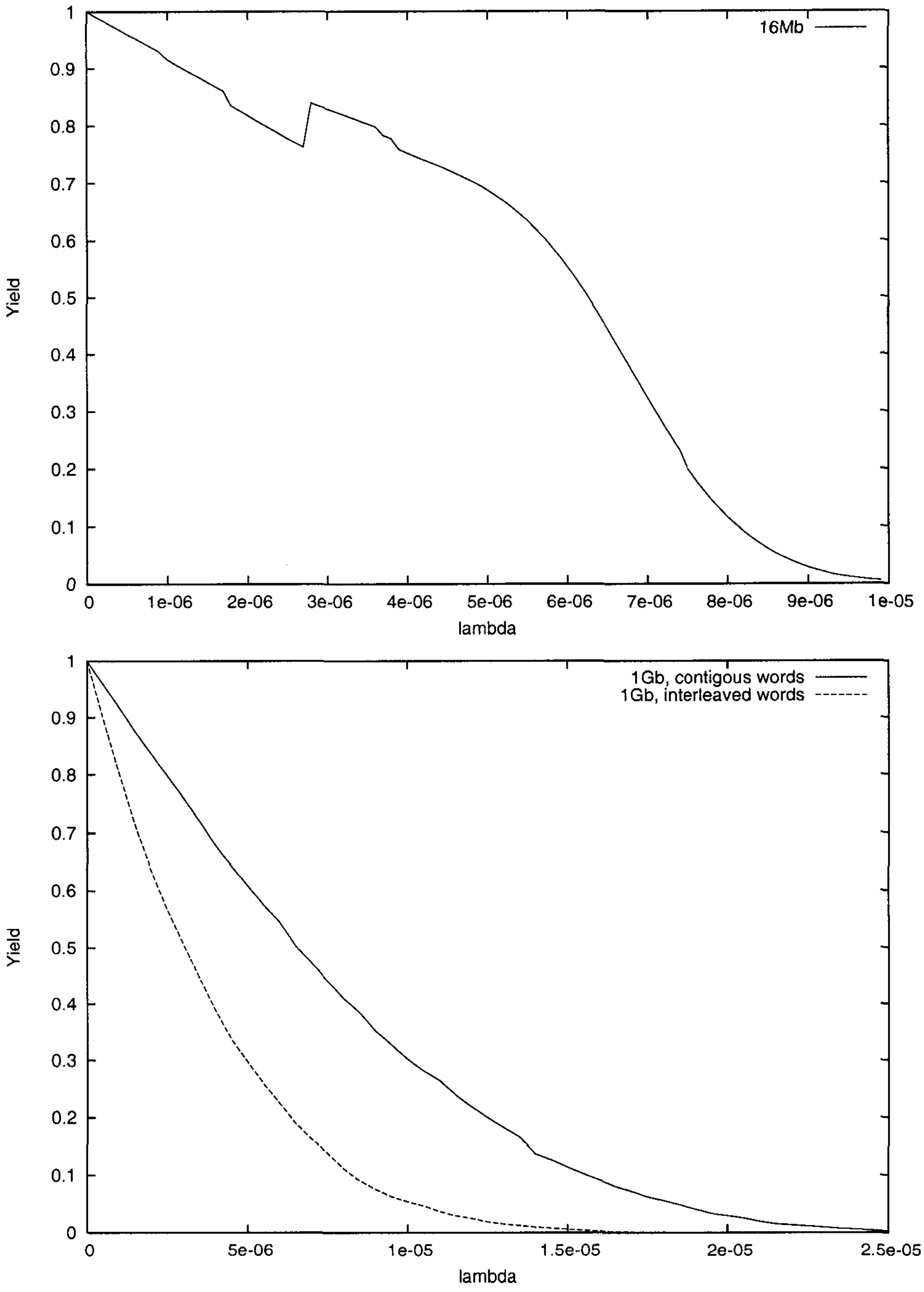


Figure C.27: Combined fault model of a 16-Mbit and a 1-Gbit DRAM with associative direct ternary CAM redundancy

See the C source code in Appendix D.5.6 for details. Once the number of CAM entries for each fault has been determined, the individual contributions to the yield are calculated:

$$Y_{d_{sc}} = \sum_{i=0}^{N_{e_{sc}}} \begin{pmatrix} W \\ i \end{pmatrix} \cdot Y_{word}^{W-i} \cdot (1 - Y_{word})^i \tag{C.103}$$

$$Y_{d_{row}} = \sum_{i=0}^{N_{e_{row}}} \begin{pmatrix} N_{row} \\ i \end{pmatrix} \cdot Y_{row}^{N_{row}-i} \cdot (1 - Y_{row})^i \tag{C.104}$$

$$Y_{d_{col}} = \sum_{i=0}^{N_{e_{col}}} \begin{pmatrix} \frac{N_{col}}{b_{word}} \\ i \end{pmatrix} \cdot Y_{word_{col}}^{\frac{N_{col}}{b_{word}}-i} \cdot (1 - Y_{word_{col}})^i \tag{C.105}$$

$$c_{xy} = \frac{N_{e_{cl}} \cdot S(x) \cdot S(y)}{c_e} \tag{C.106}$$

$$N_{cl} = \frac{N_{bits}}{x \cdot y} \tag{C.107}$$

$$Y_{d_{cl}} = \sum_{i=0}^{c_{xy}} \begin{pmatrix} N_{cl} \\ i \end{pmatrix} \cdot Y_{cl}^{N_{cl}-i} \cdot (1 - Y_{cl})^i \tag{C.108}$$

$$Y_{d_{cl}} = \prod_{x=1}^{x_{max}} \prod_{y=1}^{y_{max}} Y_{DRAM_{cluster}} \tag{C.109}$$

$$Y_{DRAM} = Y_{d_{sc}} Y_{d_{row}} Y_{d_{col}} Y_{d_{cl}} \tag{C.110}$$

See Figure C.28 for plots of the DRAM yield with associative indirect redundancy. As with the associative direct case, the plots are not smooth due to the adjusting of percentages of CAM entries used for each fault type. With an associative indirect ternary CAM redundancy of 1.1K entries and a 256-Kbit secondary memory, the 16-Mbit DRAM can handle twice the fault densities as the same DRAM with row and column redundancy. With 32K entries and a 16-Mbit redundant memory, a contiguous 1-Gbit DRAM can handle 24 times more faults than row and column redundancy, while an interleaved DRAM can handle 18 times more faults at 50% yield.

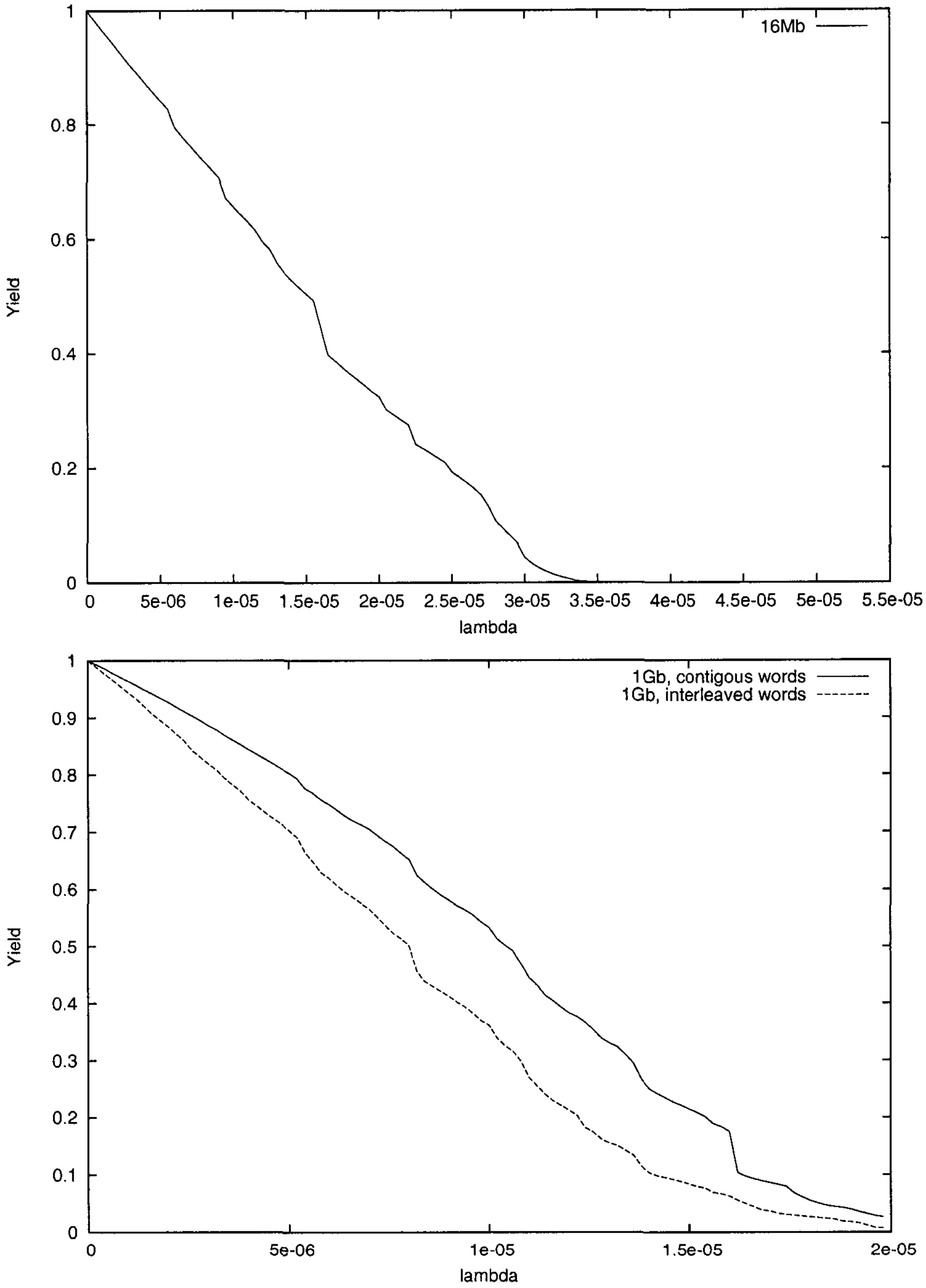


Figure C.28: Combined fault model of a 16-Mbit and a 1-Gbit DRAM with associative direct ternary CAM redundancy

# Appendix D

# Yield Model C Code

## D.1 Single Cell Fault Model

### D.1.1 No Redundancy

```
/* Program to calculate graphs for DRAM yield without redundancy using
 * the negative binomial yield model.
 * Craig Joly,  December 12, 2002
 */

#include <stdio.h>
#include <stdlib.h>
#include <getopt.h>
#include <math.h>

int main (int argc, char **argv)
{
    double alpha;
    double Ysc, Ydram;
    double lambda;
    double lstep, ltop;
    int b,c;

    opterr = 0;

    while ((c = getopt (argc, argv, "b:l:s:h")) != -1)
        switch (c)
        {
            case 'b' :
                b = atoi(optarg);
                break;
            case 'l' :
                ltop = atof(optarg);
                break;
            case 's' :
                lstep = atof(optarg);
                break;
            case 'h':
                fprintf(stderr, "Usage:%s␣-b␣bits␣-l␣end_lambda␣-s␣lambda_step\n", argv[0]);
                return 1;
            default:
                abort ();
        }

    for (lambda = 0; lambda < ltop; lambda += lstep) {
        Ysc = exp(-lambda);
        Ydram = pow(Ysc,b);
        printf ("%1.10f␣%1.10f\n", lambda, Ydram);
    }

    return 0;
}
```

### D.1.2 Row and Column Redundancy

```
/* Program to calculate graphs for DRAM yield with row and column
 * redundancy using the negative binomial yield model.
 * Craig Joly,  December 13, 2002
 */
```

## *D.1 Single Cell Fault Model*

```
#include <stdio.h>
#include <stdlib.h>
#include <getopt.h>
#include <gsl/gsl_sf_pow_int.h>
#include <gsl/gsl_sf_gamma.h>

#ifdef _16MDRAM
    #define BITS     ((unsigned int)(16*1024*1024))
    #define RED_COLS 2
    #define RED_ROWS 24
    #define NBOOKS 64
    #define NSECS 4
    #define BITS_P_ROW 1024
    #define BITS_P_COL 2048
    #define SEC_HEIGHT 4096
    #define PAGESIZE 128
#endif

#ifdef _1GDRAM
    #define BITS     ((unsigned int)(1024*1024*1024))
    #define RED_COLS 16
    #define RED_ROWS 64
    #define NBOOKS 32
    #define NSECS 8
    #define BITS_P_ROW 16384
    #define BITS_P_COL 4096
    #define SEC_HEIGHT 8192
    #define PAGESIZE 8192
#endif

int main (int argc, char **argv)
{
    double Ysc, Ycol, Yscbook, Yrow, Ydram;
    double Ybook, Ysec;
    double part1, part2, part3;
    double lambda;
    double lstep, ltop;
    int i, c;

    opterr = 0;

    while ((c = getopt (argc, argv, "l:s:h")) != -1)
    {
        switch (c)
        {
            case 'l' :
                ltop = atof(optarg);
                break;
            case 's' :
                lstep = atof(optarg);
                break;
            case 'h':
                fprintf(stderr, "Usage:%s␣-a␣alpha␣-l␣end_lambda␣-s␣lambda_step\n", argv[0]);
                return 1;
            default:
                abort ();
        }
    }

    Ydram = 1.0;

    for (lambda = 0 + lstep; lambda < ltop; lambda += lstep)
    {
        if (Ydram != 0)
        {
            Ysc = exp(-lambda);
            Ycol = gsl_sf_pow_int(Ysc, BITS_P_COL);

            Ybook = 0;
            for (i = 0; i <= RED_COLS; i++) {
                Ybook += gsl_sf_choose(PAGESIZE + RED_COLS, i) *
                         gsl_sf_pow_int(Ycol, PAGESIZE + RED_COLS - i) *
                         gsl_sf_pow_int(1 - Ycol, i);
            }

            Yscbook = exp(log(Ybook) / (BITS_P_COL * PAGESIZE));
            Yrow = gsl_sf_pow_int(Yscbook, BITS_P_ROW);

            Ysec = 0;
            for (i = 0; i <= RED_ROWS; i++) {
                Ysec += gsl_sf_choose(SEC_HEIGHT + RED_ROWS, i) *
                        gsl_sf_pow_int(Yrow, SEC_HEIGHT + RED_ROWS - i) *
                        gsl_sf_pow_int(1 - Yrow, i);
            }

            Ydram = gsl_sf_pow_int(Ysec, NSECS);
        }

        printf("%1.10f␣%1.10f\n", lambda, Ydram);
    }

    return 0;
}
```

### D.1.3 ECC Redundancy

```
/* Program to calculate graphs for DRAM yield with ECC
 * redundancy using the negative binomial yield model.
 * Craig Joly, December 16, 2002
 */

#include <stdio.h>
#include <stdlib.h>
#include <getopt.h>
#include <gsl/gsl_sf_pow_int.h>
#include <gsl/gsl_sf_exp.h>

#ifdef _16MDRAM
    #define NBOOKS 64
    #define NSECS 4
    #define CODEWORD 137
    #define NCW 2048
#endif

#ifdef _1GDRAM
    #define NBOOKS 32
    #define NSECS 8
    #define CODEWORD 523
    #define NCW 65536
#endif

int main (int argc, char **argv)
{
    double alpha;
    double Ysc, Ycw, Ydram;
    double Ybook;
    double lambda;
    double lstep, ltop;
    int i, c;

    opterr = 0;

    while ((c = getopt (argc, argv, "l:s:h")) != -1)
    {
        switch (c)
        {
            case 'l' :
                ltop = atof(optarg);
                break;
            case 's' :
                lstep = atof(optarg);
                break;
            case 'h':
                fprintf(stderr, "Usage:%s␣-l␣end_lambda␣-s␣lambda_step\n", argv[0]);
                return 1;
            default:
                abort ();
        }
    }

    Ydram = 1.0;

    for (lambda = 0 + lstep; lambda < ltop; lambda += lstep)
    {
        if (Ydram != 0)
        {
            Ysc = gsl_sf_exp(-lambda);

            Ycw = gsl_sf_pow_int(Ysc, CODEWORD) + CODEWORD *
                gsl_sf_pow_int(Ysc, CODEWORD-1) * (1 - Ysc);

            Ybook = gsl_sf_pow_int(Ycw, NCW);

            Ydram = gsl_sf_pow_int(Ybook, NBOOKS);
        }
//      printf("%1.5f %1.5f %1.5f\n", Ysc, Ycw, Ybook);
        printf("%1.10f␣%1.10f\n", lambda, Ydram);
    }

    return 0;
}
```

### D.1.4 Row and Column Redundancy with ECC

## *D.1 Single Cell Fault Model*

```
/* Program to calculate graphs for DRAM yield with row and column
 * redundancy using the negative binomial yield model.
 * Craig Joly,  December 13, 2002
 */

#include <stdio.h>
#include <stdlib.h>
#include <getopt.h>
#include <gsl/gsl_sf_pow_int.h>
#include <gsl/gsl_sf_log.h>
#include <gsl/gsl_sf_exp.h>
#include <gsl/gsl_sf_gamma.h>

#ifdef _16MDRAM
    #define BITS    ((unsigned int)(16*1024*1024*137/128))
    #define RED_COLS 2
    #define RED_ROWS 24
    #define NBOOKS 64
    #define NSECS 4
    #define BITS_P_ROW 1024
    #define BITS_P_COL 2048
    #define SEC_HEIGHT 4096
    #define CODEWORD 137
    #define PAGESIZE 137
#endif

#ifdef _1GDRAM
    #define BITS    ((unsigned int)(1024*1024*1024*523/512))
    #define RED_COLS 16
    #define RED_ROWS 64
    #define NBOOKS 32
    #define NSECS 8
    #define BITS_P_ROW 16384
    #define BITS_P_COL 4096
    #define SEC_HEIGHT 8192
    #define CODEWORD 523
    #define PAGESIZE 8368
#endif

int main (int argc, char **argv)
{
    double Ysc, Ycol, Yscbook, Yrow, Ydram;
    double Ybook, Yscecc, Ycw, Ysec;
    double lambda;
    double lstep, ltop;
    int i, c;

    opterr = 0;

    while ((c = getopt (argc, argv, "l:s:h")) != -1)
    {
        switch (c)
        {
            case 'l' :
                ltop = atof(optarg);
                break;
            case 's' :
                lstep = atof(optarg);
                break;
            case 'h':
                fprintf(stderr, "Usage:%s␣-l␣end_lambda␣-s␣lambda_step\n", argv[0]);
                return 1;
            default:
                abort ();
        }
    }

    Ydram = 1.0;

    for (lambda = 0 + lstep; lambda < ltop; lambda += lstep)
    {
        if (Ydram != 0)
        {
            Ysc = gsl_sf_exp(-lambda);

            Ycw = gsl_sf_pow_int(Ysc, CODEWORD) + CODEWORD *
                    gsl_sf_pow_int(Ysc, CODEWORD - 1) * (1 - Ysc);

            Yscecc = gsl_sf_exp(gsl_sf_log(Ycw) / (CODEWORD));

            Ycol = gsl_sf_pow_int(Yscecc, BITS_P_COL);
//          printf("Ycol=%f\n", Ycol);
            Ybook = 0;
            for (i = 0; i <= RED_COLS; i++)
            {
                Ybook += gsl_sf_choose(PAGESIZE + RED_COLS, i) *
                        gsl_sf_pow_int(Ycol, PAGESIZE + RED_COLS - i) *
                        gsl_sf_pow_int(1 - Ycol, i);
            }
//          printf("Ybook=%e\n", Ybook);
            Yscbook = gsl_sf_exp(gsl_sf_log(Ybook) / (BITS_P_COL * PAGESIZE));
//          printf("Yscbook=%f\n", Yscbook);
            Yrow = gsl_sf_pow_int(Yscbook, BITS_P_ROW);
//          printf("Yrow=%f\n", Yrow);
            Ysec = 0;
            for (i = 0; i <= RED_ROWS; i++)
            {
                Ysec += gsl_sf_choose(SEC_HEIGHT + RED_ROWS, i) *
                        gsl_sf_pow_int(Yrow, SEC_HEIGHT + RED_ROWS - i) *
                        gsl_sf_pow_int(1 - Yrow, i);
            }
//          printf("Ysec-%f\n", Ysec);
            Ydram = gsl_sf_pow_int(Ysec, NSECS);
        }

        printf("%1.10f_%1.10f\n", lambda, Ydram);
    }

    return 0;
}
```

### D.1.5 Associative Direct Redundancy

```
/* Program to calculate graphs for DRAM yield with associative direct
 * redundancy using the negative binomial yield model.
 * Craig Joly, December 13, 2002
 */

#include <stdio.h>
#include <stdlib.h>
#include <getopt.h>
#include <math.h>
#include <gsl/gsl_sf_gamma.h>

#define L_RATIO 0.2

#ifdef _16MDRAM
    #define BITS      ((unsigned int)(16*1024*1024))
    #define WORD_BITS   1
    #define ADDR_BITS   22
    #define SEC_BITS    2
    #define MAX_CAM_E   131072
#endif

#ifdef _1GDRAM
    #define BITS      ((unsigned int)(1024*1024*1024))
    #define WORD_BITS   16
    #define ADDR_BITS   25
    #define SEC_BITS    1
    #define MAX_CAM_E   131072
#endif

int main (int argc, char **argv)
{
    double Ysc, Yword, Ycam_sc, Ymask_sc, Ydram;
    double lambda, l_cam, l_mask;
    double l_h, l_l;
    double part1, part2, part3;
    int cam_e, n_cam_e;
    int i, c, reduce;

    opterr = 0;

    while ((c = getopt (argc, argv, "h")) != -1)
    {
        switch (c)
        {
            case 'h':
                fprintf(stderr, "Usage:%s\n", argv[0]);
                return 1;
            default:
                abort ();
        }
    }

    for (n_cam_e = 0 ; n_cam_e <= MAX_CAM_E ; n_cam_e += 1024)
    {
        lambda = 0.1;
        l_h = 0.1;
        l_l = 0;
        reduce = 1;
        while (1)
        {
            Ysc = exp(-lambda);
            Yword = pow(Ysc, WORD_BITS);
            Ydram = 0;
            l_cam = lambda / L_RATIO;
            l_mask = l_cam / 2;
            Ycam_sc = exp(-l_cam);
            Ymask_sc = exp(-l_mask);
            cam_e = floor(pow(Ycam_sc, ADDR_BITS) *
                    pow(Ymask_sc, ADDR_BITS + SEC_BITS) *
                    pow(Ysc, WORD_BITS) * n_cam_e);

            for (i = 0; i <= cam_e; i++)
            {
                part1 = gsl_sf_lnchoose(BITS / WORD_BITS, i);
                part2 = (BITS / WORD_BITS - i) * log(Yword);
                part3 = i * log(1.0 - Yword);
                Ydram += exp(part1 + part2 + part3);
            }

            if (Ydram > 0.53)
            {
                reduce = 0;
                l_l = lambda;
                lambda = (lambda + l_h) / 2;
            }
            else if (Ydram < 0.47)
            {
                if (reduce)
                    lambda /= 10;
                else
                {
                    l_h = lambda;
                    lambda = (lambda + l_l) / 2;
                }
            }
            else break;

            if (fabs(l_l - l_h) <= 1e-15) break;
        }

        printf("%1.10f␣%d␣%1.5f\n", lambda, n_cam_e, Ydram);
    }

    return 0;
}
```

### D.1.6 Associative Indirect Redundancy

```
#include <stdio.h>
#include <stdlib.h>
#include <getopt.h>
#include <math.h>
#include <gsl/gsl_sf_gamma.h>

#define L_RATIO 0.2

#ifdef _16MDRAM
    #define BITS         16*1024*1024
    #define WORD_BITS    1
    #define ADDR_BITS    22
    #define SEC_BITS     2
    #define MAX_CAM_E    128*1024
    #define CAM_STEP     1024
#endif

#ifdef _1GDRAM
    #define BITS         1024*1024*1024
    #define WORD_BITS    16
    #define ADDR_BITS    25
    #define SEC_BITS     1
    #define MAX_CAM_E    128*1024
    #define CAM_STEP     1024
#endif

int main (int argc, char **argv)
{
    long double Ysc, Yword, Ycam_sc, Yflash_sc, Ydram;
    long double part1, part2, part3;
    double lambda, l_cam, l_flash;
    double l_h, l_l;
    int cam_e, n_cam_e, red_bits, cam_e_p, cam_e_2p;
    int c, i, reduce, f;

    opterr = 0;

    while ((c = getopt (argc, argv, "h")) != -1)
    {
        switch (c)
        {
            case 'h':
                fprintf(stderr, "Usage:_%s\n", argv[0]);
                return 1;
            default:
                abort ();
        }
    }

    for (cam_e = 0; cam_e <= MAX_CAM_E; cam_e += CAM_STEP)
    {
        lambda = 0.1;
        l_h = 0.1;
        l_l = 0;
        reduce = 1;
        while (1)
        {
            Ysc = exp(-lambda);
            Yword = pow(Ysc, WORD_BITS);
            Ydram = 0;

            l_cam = lambda / L_RATIO;
            l_flash = l_cam / 2;

            Ycam_sc = exp(-l_cam);
            Yflash_sc = exp(-l_flash);

            red_bits = ceil(log(cam_e / Yword) / log(2));
            n_cam_e = ceil(cam_e / (pow(Ycam_sc, ADDR_BITS) *
                    pow(Yflash_sc, SEC_BITS + ADDR_BITS + red_bits)));

            for (i = 0; i <= cam_e; i++)
            {
                part1 = gsl_sf_lnchoose(BITS / WORD_BITS, i);
                part2 = (BITS / WORD_BITS - i) * log(Yword);
                part3 = i * log(1 - Yword);
                Ydram += exp(part1 + part2 + part3);
            }

            if (Ydram > 0.53)
            {
                reduce = 0;
                l_l = lambda;
                lambda = (lambda + l_h) / 2;
            }
            else if (Ydram < 0.47)
            {
                if (reduce)
                    lambda /= 10;
                else
                {
                    l_h = lambda;
                    lambda = (lambda + l_l) / 2;
                }
            }
            else break;

            if (fabs(l_l - l_h) < 1e-15) break;
        }

        printf("%1.10f_%d_%d_%d\n", lambda, n_cam_e, red_bits, cam_e);
    }

    return 0;
}
```

## D.2 Row Fault Model

### D.2.1 No Redundancy

```
/* Program to calculate graphs for DRAM yield without redundancy using
 * the negative binomial yield model.
 * Craig Joly, December 12, 2002
 */

#include <stdio.h>
#include <stdlib.h>
#include <getopt.h>
#include <math.h>

#ifdef _16MDRAM
    #define NROWS    16384
#endif

#ifdef _1GDRAM
    #define NROWS    65536
#endif
```

```
int main (int argc, char **argv)
{
    double Yrow, Ydram;
    double rho;
    double rstep, rtop;
    int b,c;

    opterr = 0;

    while ((c = getopt (argc, argv, "r:s:h")) != -1)
        switch (c)
        {
            case 'r' :
                rtop = atof(optarg);
                break;
            case 's' :
                rstep = atof(optarg);
                break;
            case 'h':
                fprintf(stderr, "Usage:%s␣-r␣end_rho␣-s␣rho_step\n", argv[0]);
                return 1;
            default:
                abort ();
        }

    for (rho = 0; rho < rtop; rho += rstep) {
        Yrow = exp(-rho);
        Ydram = pow(Yrow, NROWS);
        printf ("%1.10f␣%1.8f\n", rho, Ydram);
    }

    return 0;
}
```

### D.2.2 Row and Column Redundancy

```
/* Program to calculate graphs for DRAM yield with row and column
 * redundancy using the negative binomial yield model.
 * Craig Joly, Jan 18, 2003
 */

#include <stdio.h>
#include <stdlib.h>
#include <getopt.h>
#include <math.h>
#include <gsl/gsl_sf_pow_int.h>
#include <gsl/gsl_sf_gamma.h>

#ifdef _16MDRAM
    #define NSECS    4
    #define RED_ROWS 24
    #define SEC_HEIGHT 4096
#endif

#ifdef _1GDRAM
    #define NSECS    8
    #define RED_ROWS 64
    #define SEC_HEIGHT 8192
#endif

int main (int argc, char **argv)
{
    double alpha;
    double Yrow, Ysec, Ydram;
    double rho;
    double rstep, rtop;
    int i,b,c;

    opterr = 0;

    while ((c = getopt (argc, argv, "r:s:h")) != -1)
    {
        switch (c)
        {
            case 'r' :
                rtop = atof(optarg);
                break;
            case 's' :
                rstep = atof(optarg);
                break;
            case 'h':
                fprintf(stderr, "Usage:%s␣-r␣end_rho␣-s␣rho_step\n", argv[0]);
                return 1;
            default:
                abort ();
        }
    }
    for (rho = 0; rho < rtop; rho += rstep) {
        Yrow = exp(-rho);

        Ysec = 0;
        for (i = 0; i <= RED_ROWS; i++)
        {
            Ysec += gsl_sf_choose(SEC_HEIGHT + RED_ROWS, i) *
                    gsl_sf_pow_int(Yrow, SEC_HEIGHT + RED_ROWS - i) *
                    gsl_sf_pow_int(1 - Yrow, i);
        }

        Ydram = gsl_sf_pow_int(Ysec, NSECS);
        printf ("%1.10f_%1.8f\n", rho, Ydram);
    }

    return 0;
}
```

### D.2.3 Associative Direct Redundancy

```
/* Program to calculate graphs for DRAM yield with associative direct
 * redundancy using the negative binomial yield model.
 * Craig Joly, Jan 18, 2003
 */

#include <stdio.h>
#include <stdlib.h>
#include <getopt.h>
#include <math.h>
#include <gsl/gsl_sf_pow_int.h>
#include <gsl/gsl_sf_gamma.h>

#ifdef _16MDRAM
    #define NROWS        16384
    #define PAGESIZE     128
    #define MAX_CAM_E    128*1024
    #define CAM_STEP     1024
    #define ADDR_BITS    22
    #define BITS_P_ROW   1024
    #define WORD_BITS    1
#endif

#ifdef _1GDRAM
    #define NROWS        65536
    #define PAGESIZE     8192
    #define MAX_CAM_E    128*1024
    #define CAM_STEP     1024
    #define ADDR_BITS    25
    #define BITS_P_ROW   16384
    #define WORD_BITS    16
#endif

int main (int argc, char **argv)
{
    double Ydram, Yrow;
    double Ycam_row;
    double part1, part2, part3;
    double rho;
    double r_h, r_l, reduce;
    int cam_e, n_cam_e;
    int i,c;

    opterr = 0;


    while ((c = getopt (argc, argv, "h")) != -1)
    {
        switch (c)
        {
            case 'h':
                fprintf(stderr, "Usage:%s\n", argv[0]);
                return 1;
            default:
                abort ();
        }
    }

    for (n_cam_e = 0; n_cam_e <= MAX_CAM_E; n_cam_e += CAM_STEP)
    {
        rho = 0.1;
        r_h = 0.1;
        r_l = 0;
        reduce = 1;

        while (1)
        {
            Yrow = exp(-rho);
            Ycam_row = exp(-2*rho);
            cam_e = Ycam_row * n_cam_e;
            c = floor(4 * cam_e * WORD_BITS / BITS_P_ROW);

            Ydram = 0;
            for (i = 0; i <= c; i ++)
            {
                part1 = gsl_sf_lnchoose(NROWS, i);
                part2 = (NROWS - i) * log(Yrow);
                part3 = i * log(1 - Yrow);
                Ydram += exp(part1 + part2 + part3);
            }
//          printf("%d %f\n", c, Ydram);

            if (Ydram > 0.53)
            {
                reduce = 0;
                r_l = rho;
                rho = (rho + r_h) / 2;
            }
            else if (Ydram < 0.47)
            {
                if (reduce)
                    rho /= 10;
                else
                {
                    r_h = rho;
                    rho = (rho + r_l) / 2;
                }
            }
            else break;

            if (fabs(r_l - r_h) <= 0.0000000001) break;
        }

        printf("%1.10f_%d_%d\n", rho, n_cam_e, c);
    }

    return 0;
}
```

### D.2.4 Associative Indirect Redundancy

```
/* Program to calculate graphs for DRAM yield with associative direct
 * redundancy using the negative binomial yield model.
 * Craig Joly, Jan 18, 2003
 */

#include <stdio.h>
#include <stdlib.h>
#include <getopt.h>
#include <math.h>
#include <gsl/gsl_sf_pow_int.h>
#include <gsl/gsl_sf_gamma.h>

#ifdef _16MDRAM
    #define NROWS       16384
    #define PAGESIZE    128
    #define MAX_CAM_E   2*1024
    #define CAM_STEP    16
    #define ADDR_BITS   22
    #define BITS_P_ROW  1024
    #define WORD_BITS   1
#endif

#ifdef _1GDRAM
    #define NROWS       65536
    #define PAGESIZE    8192
    #define MAX_CAM_E   8*1024
    #define CAM_STEP    64
    #define ADDR_BITS   25
    #define BITS_P_ROW  16384
    #define WORD_BITS   16
#endif

int main (int argc, char **argv)
{
    double Ydram, Yrow;
    double Ycam_row;
    double part1, part2, part3;
    double rho;
    double r_h, r_l, reduce;
    int cam_e, n_cam_e, red_bits;
    int i,c;

    opterr = 0;
    while ((c = getopt (argc, argv, "h")) != -1)
    {
        switch (c)
        {
            case 'h':
                fprintf(stderr, "Usage:%s\n", argv[0]);
                return 1;
            default:
                abort ();
        }
    }

    for (n_cam_e = 0; n_cam_e <= MAX_CAM_E; n_cam_e += CAM_STEP)
    {
        rho = 0.1;
        r_h = 0.1;
        r_l = 0;
        reduce = 1;

        while (1)
        {
            Yrow = exp(-rho);
            Ycam_row = exp(-2*rho);

            cam_e = Ycam_row * n_cam_e;
            red_bits = ceil(log((cam_e * BITS_P_ROW)/(WORD_BITS * Yrow)) /
                    log(2));

            Ydram = 0;
            for (i = 0; i <= cam_e; i ++)
            {
                part1 = gsl_sf_lnchoose(NROWS, i);
                part2 = (NROWS - i) * log(Yrow);
                part3 = i * log(1 - Yrow);
                Ydram += exp(part1 + part2 + part3);
            }
//:w            printf("%d %f\n", cam_e, Ydram);

            if (Ydram > 0.53)
            {
                reduce = 0;
                r_l = rho;
                rho = (rho + r_h) / 2;
            }
            else if (Ydram < 0.47)
            {
                if (reduce)
                    rho /= 10;
                else
                {
                    r_h = rho;
                    rho = (rho + r_l) / 2;
                }
            }
            else break;

            if (fabs(r_l - r_h ) <= 0.0000000001) break;

        }
        printf("%1.10f_%d_%d_%d_%f\n", rho, n_cam_e, red_bits, cam_e, Ydram);
    }

    return 0;
}
```

# D.3 Column Fault Model

## D.3.1 No Redundancy

```
/* Program to calculate graphs for DRAM yield without redundancy using
 * the negative binomial yield model.
 * Craig Joly, December 12, 2002
 */

#include <stdio.h>
#include <stdlib.h>
#include <getopt.h>
#include <math.h>

#ifdef _16MDRAM
    #define NCOLS    8192
#endif
```

```
#ifdef _1GDRAM
    #define NCOLS    262144
#endif

int main (int argc, char **argv)
{
    double Ycol, Ydram;
    double chi;
    double cstep, ctop;
    int b,c;

    opterr = 0;

    while ((c = getopt (argc, argv, "c:s:h")) != -1)
        switch (c)
        {
            case 'c' :
                ctop = atof(optarg);
                break;
            case 's' :
                cstep = atof(optarg);
                break;
            case 'h':
                fprintf(stderr, "Usage:%s␣-l␣end_chi␣-s␣chi_step\n", argv[0]);
                return 1;
            default:
                abort ();
        }

    for (chi = 0; chi < ctop; chi += cstep) {
        Ycol = exp(-chi);
        Ydram = pow(Ycol, NCOLS);
        printf ("%1.10f␣%1.8f\n", chi, Ydram);
    }

    return 0;
}
```

### D.3.2 Row and Column Redundancy

```
/* Program to calculate graphs for DRAM yield with row and column
 * redundancy using the negative binomial yield model.
 * Craig Joly, Jan 18, 2003
 */

#include <stdio.h>
#include <stdlib.h>
#include <getopt.h>
#include <math.h>
#include <gsl/gsl_sf_pow_int.h>
#include <gsl/gsl_sf_gamma.h>

#ifdef _16MDRAM
    #define NCOLS    8192
    #define NBOOKS   64
    #define RED_COLS 2
    #define PAGESIZE 128
#endif

#ifdef _1GDRAM
    #define NCOLS    262144
    #define NBOOKS   32
    #define RED_COLS 16
    #define PAGESIZE 8192
#endif

int main (int argc, char **argv)
{
    double Ycol, Ybook, Ydram;
    double chi;
    double cstep, ctop;
    int i,b,c;

    opterr = 0;

    while ((c = getopt (argc, argv, "c:s:h")) != -1)
    {
        switch (c)
        {
            case 'c' :
                ctop = atof(optarg);
                break;
            case 's' :
                cstep = atof(optarg);
                break;
            case 'h':
                fprintf(stderr, "Usage:%s␣-c␣end_chi␣-s␣chi_step\n", argv[0]);
                return 1;
            default:
                abort ();
        }
    }

    for (chi = 0; chi < ctop; chi += cstep) {
        Ycol = exp(-chi);

        Ybook = 0;
        for (i = 0; i <= RED_COLS; i++)
        {
            Ybook += gsl_sf_choose(PAGESIZE + RED_COLS, i) *
                    gsl_sf_pow_int(Ycol, PAGESIZE + RED_COLS - i) *
                    gsl_sf_pow_int(1.0 - Ycol, i);
        }

        Ydram = gsl_sf_pow_int(Ybook, NBOOKS);
        printf ("%1.10f_%1.8f\n", chi, Ydram);
    }

    return 0;
}
```

### D.3.3 ECC Redundancy

```
/* Program to calculate graphs for DRAM yield with row and column
 * redundancy using the negative binomial yield model.
 * Craig Joly,  Jan 18, 2003
 */

#include <stdio.h>
#include <stdlib.h>
#include <getopt.h>
#include <math.h>
#include <gsl/gsl_sf_pow_int.h>
#include <gsl/gsl_sf_gamma.h>

#ifdef _16MDRAM
    #define NCOLS    8768
    #define NBOOKS   64
    #define CODEWORD 137
    #define NCWP 1
#endif

#ifdef _1GDRAM
    #define NCOLS    267776
    #define NBOOKS   32
    #define CODEWORD 523
    #define NCWP 16
#endif

int main (int argc, char **argv)
{
    double Ycol, Ybook, Ydram;
    double chi;
    double cstep, ctop;
    int i,b,c;

    opterr = 0;

    while ((c = getopt (argc, argv, "c:s:h")) != -1)
        switch (c)
        {
            case 'c' :
                ctop = atof(optarg);
                break;
            case 's' :
                cstep = atof(optarg);
                break;
            case 'h':
                fprintf(stderr, "Usage:%s_-c_end_chi_-s_chi_step\n", argv[0]);
                return 1;
            default:
                abort ();
        }

    for (chi = 0; chi < ctop; chi += cstep) {
        Ycol = exp(-chi);

        Ybook = gsl_sf_pow_int(gsl_sf_pow_int(Ycol, CODEWORD) +
                CODEWORD * gsl_sf_pow_int(Ycol, CODEWORD-1) *
                (1 - Ycol), NCWP);

        Ydram = gsl_sf_pow_int(Ybook, NBOOKS);
        printf ("%1.10f_%1.8f\n", chi, Ydram);
    }

    return 0;
}
```

### D.3.4 Row and Column Redundancy with ECC

```
/* Program to calculate graphs for DRAM yield with row and column
 * redundancy using the negative binomial yield model.
 * Craig Joly, Jan 18, 2003
 */

#include <stdio.h>
#include <stdlib.h>
#include <getopt.h>
#include <math.h>
#include <gsl/gsl_sf_pow_int.h>
#include <gsl/gsl_sf_exp.h>
#include <gsl/gsl_sf_log.h>
#include <gsl/gsl_sf_gamma.h>

#ifdef _16MDRAM
    #define NCOLS    8768
    #define NBOOKS   64
    #define RED_COLS 2
    #define PAGESIZE 137
    #define CODEWORD 137
    #define NCWP 1
#endif

#ifdef _1GDRAM
    #define NCOLS    267776
    #define NBOOKS   32
    #define RED_COLS 16
    #define PAGESIZE 8368
    #define CODEWORD 523
    #define NCWP 16
#endif

int main (int argc, char **argv)
{
    double Ycol, Ycw, Ycolecc, Ybook, Ydram;
    double chi;
    double part1, part2, part3;
    double cstep, ctop;
    int i,b,c;

    opterr = 0;

    while ((c = getopt (argc, argv, "c:s:h")) != -1)
        switch (c)
        {
            case 'c' :
                ctop = atof(optarg);
                break;
            case 's' :
                cstep = atof(optarg);
                break;
            case 'h':
                fprintf(stderr, "Usage:%s␣-c␣end_chi␣-s␣chi_step\n", argv[0]);
                return 1;
            default:
                abort ();
        }

    for (chi = 0; chi < ctop; chi += cstep) {
        Ycol = exp(-chi);

        Ycw = pow(Ycol, CODEWORD) + CODEWORD * pow(Ycol, CODEWORD-1) *
                (1 - Ycol);

        Ycolecc = exp(log(Ycw) / CODEWORD);

        Ybook = 0;
        for (i = 0; i <= RED_COLS; i++)
        {
            Ybook += gsl_sf_choose(PAGESIZE + RED_COLS, i) *
                    pow(Ycol, PAGESIZE + RED_COLS - i) *
                    pow(1 - Ycol, i);
        }

        Ydram = pow(Ybook, NBOOKS);


        printf ("%1.10f␣%1.8f\n", chi, Ydram);
    }

    return 0;
}
```

### D.3.5 Associative Direct Redundancy

```
/* Program to calculate graphs for DRAM yield with associative direct
 * redundancy using the negative binomial yield model.
 * Craig Joly,  Jan 18, 2003
 */

#include <stdio.h>
#include <stdlib.h>
#include <getopt.h>
#include <math.h>
#include <gsl/gsl_sf_pow_int.h>
#include <gsl/gsl_sf_gamma.h>

#ifdef _16MDRAM
    #define NCOLS        8192
    #define SEC_HEIGHT   4096
    #define MAX_CAM_E    64*1024
    #define CAM_STEP     512
    #define ADDR_BITS    22
    #define WORD_BITS    1
    #define BITS_P_COL   2048
    #define SEC_BITS     2
#endif

#ifdef _1GDRAM
    #define NCOLS        262144
    #define SEC_HEIGHT   8192
    #define MAX_CAM_E    1024*1024
    #define CAM_STEP     1024
    #define ADDR_BITS    25
    #define WORD_BITS    16
    #define BITS_P_COL   4096
    #define SEC_BITS     1
#endif

int main (int argc, char **argv)
{
    double Ydram, Ycol, Ywordcol, Yredcol;
    double Ydcol;
    double part1, part2, part3;
    double chi;
    double c_h, c_l, reduce;
    int cam_e, n_cam_e;
    int i,c;

    opterr = 0;

    while ((c = getopt (argc, argv, "h")) != -1)
    {
        switch (c)
        {
            case 'h':
                fprintf(stderr, "Usage:%s␣-a␣alpha\n", argv[0]);
                return 1;
            default:
                abort ();
        }
    }

    for (n_cam_e = 0; n_cam_e <= MAX_CAM_E; n_cam_e += CAM_STEP)
    {
        chi = 0.1;
        c_h = 0.1;
        c_l = 0;
        reduce = 1;

        while (1)
        {
            Ycol = exp(-chi);
            Ywordcol = pow(Ycol, WORD_BITS);
            Yredcol = pow(Ycol, 3 * ADDR_BITS + SEC_BITS + 4 * WORD_BITS);

            c = floor(4 * n_cam_e / BITS_P_COL);

            Ydcol = 0;
            for (i = 0; i <= c; i++)
            {
                part1 = gsl_sf_lnchoose(NCOLS / WORD_BITS, i);
                part2 = (NCOLS / WORD_BITS - i) * log(Ywordcol);
                part3 = i * log(1 - Ywordcol);
                Ydcol += exp(part1 + part2 + part3);
            }

            Ydram = Ydcol * Yredcol;

            if (Ydram > 0.53)
            {
                reduce = 0;
                c_l = chi;
                chi = (chi + c_h) / 2;
            }
            else if (Ydram < 0.47)
            {
                if (reduce)
                    chi /= 10;
                else
                {
                    c_h = chi;
                    chi = (chi + c_l) / 2;
                }
            }
            else break;
//          printf("%1.10f %1.10f %1.10f %1.10f\n", c_l, c_h, chi, Ydram);
            if (fabs(c_l - c_h) <= 0.0000000001) break;
        }
        printf("%1.10f_%d_%d_%1.10f\n", chi, n_cam_e, c, Ydram);
    }
    return 0;
}
```

### D.3.6 Associative Indirect Redundancy

```
/* Program to calculate graphs for DRAM yield with associative direct
 * redundancy using the negative binomial yield model.
 * Craig Joly, Jan 18, 2003
 */

#include <stdio.h>
#include <stdlib.h>
#include <getopt.h>
#include <math.h>
#include <gsl/gsl_sf_pow_int.h>
#include <gsl/gsl_sf_gamma.h>

#ifdef _16MDRAM
    #define NCOLS       8192
    #define SEC_HEIGHT  4096
    #define MAX_CAM_E   64
    #define CAM_STEP    1
    #define ADDR_BITS   22
    #define WORD_BITS   1
    #define BITS_P_COL  2048
    #define SEC_BITS    2
#endif

#ifdef _1GDRAM
    #define NCOLS       262144
    #define SEC_HEIGHT  8192
    #define MAX_CAM_E   1024
    #define CAM_STEP    16
    #define ADDR_BITS   25
    #define WORD_BITS   16
    #define BITS_P_COL  4096
    #define SEC_BITS    1
#endif

int main (int argc, char **argv)
{
    double Ydram, Ycol, Ywordcol, Yredcol;
    double Ydcol;
    double part1, part2, part3;
    double chi;
    double c_h, c_l, reduce;
    int cam_e, n_cam_e, red_bits;
    int i,c;

    opterr = 0;

    while ((c = getopt (argc, argv, "h")) != -1)
    {
        switch (c)
        {
            case 'h':
                fprintf(stderr, "Usage:%s\n", argv[0]);
                return 1;
            default:
                abort ();
        }
    }

    for (n_cam_e = 0; n_cam_e <= MAX_CAM_E; n_cam_e += CAM_STEP)
    {
        chi = 0.1;
        c_h = 0.1;
        c_l = 0;
        reduce = 1;

        while (1)
        {
            Ycol = exp(-chi);
            Ywordcol = pow(Ycol, WORD_BITS);

            red_bits = ceil(log(n_cam_e * BITS_P_COL) / log(2));

            Yredcol = pow(Ycol, 3 * ADDR_BITS + red_bits + SEC_BITS);

            Ydcol = 0;
            for (i = 0; i <= n_cam_e; i++)
            {
                part1 = gsl_sf_lnchoose(NCOLS / WORD_BITS, i);
                part2 = (NCOLS / WORD_BITS - i) * log(Ywordcol);
                part3 = i * log(1 - Ywordcol);
                Ydcol += exp(part1 + part2 + part3);
            }

            Ydram = Ydcol * Yredcol * Ywordcol;

            if (Ydram > 0.501)
            {
                reduce = 0;
                c_l = chi;
                chi = (chi + c_h) / 2;
            }
            else if (Ydram < 0.499)
            {
                if (reduce)
                    chi /= 10;
                else
                {
                    c_h = chi;
                    chi = (chi + c_l) / 2;
                }
            }
            else break;
//          printf("%1.10f %1.10f %1.10f %1.10f\n", r_l, r_h, Ytarget, Ydcol);

            if (fabs(c_l - c_h) <= 0.0000000001) break;

        }

        printf("%1.10f_%d_%d_%1.10f\n", chi, n_cam_e, red_bits, Ydram);
    }

    return 0;
}
```

# D.4 Cluster Fault Model

## D.4.1 No Redundancy

```
/* Program to calculate graphs for DRAM yield without redundancy using
 * a cluster fault model and a binomial (poisson) yiled model
 * Craig Joly, December 12, 2002
 */

#include <stdio.h>
#include <stdlib.h>
#include <getopt.h>
#include <math.h>

#ifdef _16MDRAM
    #define BITS     16*1024*1024
    #define N_X      2048
    #define N_Y      8192
#endif

#ifdef _1GDRAM
    #define BITS     1024*1024*1024
    #define N_X      32*1024
    #define N_Y      32*1024
#endif

#define Q        1.0
#define P        2.0
#define DNOT     2.0
#define DMAX     64.0
```

```
#define FPPARAM 0.68085

double sprob (int delta)
{
    return (delta < DNOT) ? (0.68085 * delta / 4.0):
                            (0.68085 * 2.0 / (delta * delta));
}

int main (int argc, char **argv)
{
    double Ycl, Ydcl, Ydram;
    double psi, fpparam;
    double pstep, ptop;
    int b,c, x, y;

    opterr = 0;

    while ((c = getopt (argc, argv, "b:l:s:h")) != -1)
        switch (c)
        {
            case 'l' :
                ptop = atof(optarg);
                break;
            case 's' :
                pstep = atof(optarg);
                break;
            case 'h':
                fprintf(stderr, "Usage:%s_-l_end_lambda_-s_lambda_step\n", argv[0]);
                return 1;
            default:
                abort ();
        }


    for (psi = 0; psi < ptop; psi += pstep) {
        Ydram = 1;
        for (x = 1; x <= DMAX; x++)
        {
            for (y = 1; y <= DMAX; y++)
            {
                Ycl = exp(-psi * sprob(x) * sprob(y));
                Ydcl = pow(Ycl, (N_X - x + 1) * (N_Y - y + 1));
                Ydram *= Ydcl;
            }
        }
        printf ("%1.10f_%1.10f\n", psi, Ydram);
    }

    return 0;
}
```

## D.4.2 Row and Column Redundancy

```
/* Program to calculate graphs for DRAM yield with row and column
 * redundancy using the negative binomial yield model.
 * Craig Joly, December 13, 2002
 */

#include <stdio.h>
#include <stdlib.h>
#include <getopt.h>
#include <math.h>
#include <gsl/gsl_sf_gamma.h>

#ifdef _16MDRAM
    #define BITS     ((unsigned int)(16*1024*1024))
    #define RED_COLS 2
    #define RED_ROWS 24
    #define NBOOKS 64
    #define NSECS 4
    #define BITS_P_ROW 1024
    #define BITS_P_COL 2048
    #define SEC_HEIGHT 4096
    #define PAGESIZE 128
#endif

#ifdef _1GDRAM
    #define BITS     ((unsigned int)(1024*1024*1024))
    #define RED_COLS 16
    #define RED_ROWS 64
    #define NBOOKS 32
    #define NSECS 8
    #define BITS_P_ROW 16384
    #define BITS_P_COL 4096
    #define SEC_HEIGHT 8192
    #define PAGESIZE 8192
#endif
```

```
#define Q       1.0
#define P       2.0
#define DNOT    2.0
#define DMAX    64.0
#define FPPARAM 0.68085

double sprob (int delta)
{
//  return (delta < DNOT) ? (FPPARAM * pow(delta , Q) / pow(DNOT, Q+1)) :
//                          (FPPARAM * pow(DNOT, P-1) / pow(delta , P));
    return (delta < DNOT) ? (FPPARAM * delta / 4.0) :
                            (FPPARAM * 2.0 / (delta*delta));
}

int main (int argc, char **argv)
{
    double Ycl, Ycol, Yscbook, Yrow, Ydram;
    double Ycolcl, Ybook, Ysec;
    double ycp1, ycp2, ycp3;
    double part1, part2, part3;
    double psi;
    double pstep, ptop;
    int x, y;
    int i, c;

    opterr = 0;

    while ((c = getopt (argc, argv, "l:s:h")) != -1)
    {
        switch (c)
        {
            case 'l' :
                ptop = atof(optarg);
                break;
            case 's' :
                pstep = atof(optarg);
                break;
            case 'h':
                fprintf(stderr, "Usage:%s -l end_psi -s psi_step\n", argv[0]);
                return 1;
            default:
                abort ();
        }
    }

    for (psi = 0 + pstep; psi < ptop; psi += pstep)
    {
        Ycol = 1.0;
        for (x = 1; x <= DMAX; x++)
        {
            for (y = 1; y <= DMAX; y++)
            {
                Ycl = exp(-psi * sprob(x) * sprob(y));
                Ycolcl = pow(Ycl, BITS_P_COL+RED_ROWS-y+1);
                ycp1 = pow(Ycolcl, x*(PAGESIZE + RED_COLS - 2*x + 2));
                ycp2 = 1;
                for (i = 1; i < x; i++)
                    ycp2 *= pow(Ycolcl, 2*i);
                Ycol *= exp(log(ycp1*ycp2)/(PAGESIZE+RED_COLS));
            }
        }
//      printf("%f\n", Ycol);
        Ybook = 0;
        for (i = 0; i <= RED_COLS; i++)
        {
            part1 = gsl_sf_lnchoose(PAGESIZE + RED_COLS, i);
            part2 = (PAGESIZE + RED_COLS-i) * log(Ycol);
            part3 = i * log(1 - Ycol);
//          printf("%1.10f %1.10f %1.10f\n", part1, part2, part3);
            Ybook += exp(part1 + part2 + part3);
        }


        Yscbook = exp(log(Ybook) / (PAGESIZE * BITS_P_COL));

        Yrow = pow(Yscbook, BITS_P_ROW);
//      printf("%1.10f %1.10f %1.10f\n", Ybook, Yscbook, Yrow);

//      printf("here3");

        Ysec = 0;
        for (i = 0; i <= RED_ROWS; i++)
        {
            part1 = gsl_sf_lnchoose(SEC_HEIGHT + RED_ROWS, i);
            part2 = (SEC_HEIGHT + RED_ROWS-i) * log(Yrow);
            part3 = i * log(1 - Yrow);
            Ysec += exp(part1 + part2 + part3);
        }
//          printf("%1.10f", Ysec);

            Ydram = pow(Ysec, NSECS);

            printf("%1.10f_%1.10f\n", psi, Ydram);
      }

      return 0;
}
```

### D.4.3 ECC Redundancy

```
/* Program to calculate graphs for DRAM yield with row and column
 * redundancy using the negative binomial yield model.
 * Craig Joly, December 13, 2002
 */

#include <stdio.h>
#include <stdlib.h>
#include <getopt.h>
#include <gsl/gsl_math.h>
#include <gsl/gsl_sf_log.h>
#include <gsl/gsl_sf_exp.h>
#include <gsl/gsl_sf_gamma.h>
#include <gsl/gsl_sf_pow_int.h>

#ifdef _16MDRAM
    #define BITS      ((unsigned int)(16*1024*1024))
    #define RED_COLS 2
    #define RED_ROWS 24
    #define NBOOKS 64
    #define NSECS 4
    #define BITS_P_COL 2048
    #define CODEWORD 137
    #define PAGESIZE 137
    #define NCW 2048
#endif

#ifdef _1GDRAM
    #define BITS      ((unsigned int)(1024*1024*1024))
    #define RED_COLS 16
    #define RED_ROWS 64
    #define NBOOKS 32
    #define NSECS 8
    #define BITS_P_COL 4096
    #define CODEWORD 523
    #define PAGESIZE 8192
    #define NCW 65536
#endif

#define Q       1.0
#define P       2.0
#define DNOT    2.0
#define DMAX    64.0
#define FPPARAM 0.68085

double sprob (int delta)
{
//  return (delta < DNOT) ? (FPPARAM * pow(delta, Q) / pow(DNOT, Q+1)) :
//                          (FPPARAM * pow(DNOT, P-1) / pow(delta, P));
    return (delta < DNOT) ? (FPPARAM * delta / 4.0) :
                            (FPPARAM * 2.0 / (delta*delta));
}

int main (int argc, char **argv)
{
    double Ycl, Ycw, Yrow, Ydram;
    double Yrowcl, Yly, Ybook;
    double yrp1, yrp2;
    double part1, part2, part3;
    double psi;
    double pstep, ptop;
    int x, y;
    int i, c;

    opterr = 0;

    while ((c = getopt (argc, argv, "l:s:h")) != -1)
    {
        switch (c)
        {
            case 'l' :
                ptop = atof(optarg);
                break;
            case 's' :
                pstep = atof(optarg);
                break;
            case 'h':
                fprintf(stderr, "Usage:%s␣-l␣end_psi␣-s␣psi_step\n", argv[0]);
                return 1;
            default:
                abort ();
        }
    }

    for (psi = 0; psi < ptop; psi += pstep)
    {
        Yrow = 1.0;
        for (x = 1; x <= DMAX; x++)
        {
            for (y = 1; y <= DMAX; y++)
            {
                Ycl = gsl_sf_exp(-psi * sprob(x) * sprob(y));
                Yrowcl = gsl_pow_int(Ycl, PAGESIZE-x+1);
                yrp1 = gsl_pow_int(Yrowcl, y*(BITS_P_COL - 2*y + 2));
                yrp2 = 1;
                for (i = 1; i < y; i++)
                    yrp2 *= gsl_pow_int(Yrowcl, 2*i);
                Yrow *= gsl_sf_exp(gsl_sf_log(yrp1*yrp2)/BITS_P_COL);
            }
        }
//      printf("%f\n", Ycol);

        Y1y = 1.0;
        for (y = 1; y <= DMAX; y++)
        {
            Ycl = gsl_sf_exp(-psi * sprob(1) * sprob(y));
            yrp1 = gsl_pow_int(Ycl, y*(BITS_P_COL -2*y + 2));
            yrp2 = 1;
            for (i = 1; i < y; i++)
                yrp2 *= gsl_pow_int(Ycl, 2*i);
            Y1y *= gsl_sf_exp(gsl_sf_log(yrp1 * yrp2)/BITS_P_COL);

        }

        part1 = gsl_sf_exp(gsl_sf_log(Yrow)*CODEWORD/PAGESIZE);
        part2 = CODEWORD * gsl_sf_exp(gsl_sf_log(Yrow)*((CODEWORD-1)/PAGESIZE));
        part3 = 1 - Y1y;
        Ycw = part1 + part2 * part3;

        Ybook = gsl_sf_pow_int(Ycw, NCW);
//      printf("here3");

//      printf("here4");

        Ydram = gsl_pow_int(Ybook, NBOOKS);

        printf("%1.13f␣%1.10f\n", psi, Ydram);
    }

    return 0;
}
```

### D.4.4 Row and Column Redundancy with ECC

```
/* Program to calculate graphs for DRAM yield with row and column
 * redundancy using the negative binomial yield model.
 * Craig Joly, December 13, 2002
 */

#include <stdio.h>
#include <stdlib.h>
#include <getopt.h>
#include <math.h>
#include <gsl/gsl_sf_gamma.h>

#ifdef _16MDRAM
    #define BITS    ((unsigned int)(16*1024*1024))
    #define RED_COLS 2
    #define RED_ROWS 24
    #define NBOOKS 64
    #define NSECS 4
    #define BITS_P_ROW 1096
    #define BITS_P_COL 2048
    #define SEC_HEIGHT 4096
    #define PAGESIZE 137
    #define CODEWORD 137
#endif

#ifdef _1GDRAM
    #define BITS    ((unsigned int)(1024*1024*1024))
    #define RED_COLS 16
    #define RED_ROWS 64
    #define NBOOKS 32
    #define NSECS 8
    #define BITS_P_ROW 16736
    #define BITS_P_COL 4096
    #define SEC_HEIGHT 8192
    #define PAGESIZE 8368
    #define CODEWORD 523
#endif

#define Q       1.0
#define P       2.0
#define DNOT    2.0
#define DMAX    64.0
#define FPPARAM 0.68085

double sprob (int delta)
{
//  return (delta < DNOT) ? (FPPARAM * pow(delta, Q) / pow(DNOT, Q+1)) :
//                          (FPPARAM * pow(DNOT, P-1) / pow(delta, P));
    return (delta < DNOT) ? (FPPARAM * delta / 4.0) :
                            (FPPARAM * 2.0 / (delta*delta));
}

int main (int argc, char **argv)
{
    double Ycl, Ycol, Yscbook, Yrow, Ydram;
    double Ycolcl, Ybook, Ysec;
    double Yly, yrp1, yrp2, Ycw;
    double ycp1, ycp2, ycp3;
    double part1, part2, part3;
    double psi;
    double pstep, ptop;
    int x, y;
    int i, c;

    opterr = 0;

    while ((c = getopt (argc, argv, "l:s:h")) != -1)
    {
        switch (c)
        {
            case 'l' :
                ptop = atof(optarg);
                break;
            case 's' :
                pstep = atof(optarg);
                break;
            case 'h':
                fprintf(stderr, "Usage:%s␣-l␣end_psi␣-s␣psi_step\n", argv[0]);
                return 1;
            default:
                abort ();
        }
    }

    for (psi = 0 + pstep; psi < ptop; psi += pstep)
    {
        Ycol = 1.0;
        for (x = 1; x <= DMAX; x++)
        {
            for (y = 1; y <= DMAX; y++)
            {
                Ycl = exp(-psi * sprob(x) * sprob(y));
                Ycolcl = pow(Ycl, BITS_P_COL+RED_ROWS-y+1);
                ycp1 = pow(Ycolcl, x*(PAGESIZE + RED_COLS - 2*x + 2));
                ycp2 = 1;
                for (i = 1; i < x; i++)
                    ycp2 *= pow(Ycolcl, 2*i);
                Ycol *= exp(log(ycp1*ycp2)/(PAGESIZE+RED_COLS));
            }
        }
//      printf("%f\n", Ycol);
        Ybook = 0;
        for (i = 0; i <= RED_COLS; i++)
        {
            part1 = gsl_sf_lnchoose(PAGESIZE + RED_COLS, i);
//          printf("die1");
            part2 = (PAGESIZE + RED_COLS-i) * log(Ycol);
//          printf("die2");
            part3 = i * log(1 - Ycol);
//          printf("die3");
            Ybook += exp(part1 + part2 + part3);
//          printf("die4");
        }

//      printf("here2");

        Yly = 1;
        for (y = 1; y <= DMAX; y++)
        {
            Ycl = exp(-psi * sprob(1) * sprob(y));
            yrp1 = pow(Ycl, y*(BITS_P_COL - 2*y + 2));
            yrp2 = 1;
            for (i = 1; i < y; i++)
                yrp2 *= pow(Ycl, 2*i);
            Yly *= exp(log(yrp1 * yrp2) / BITS_P_COL);
        }

        part1 = exp(log(Ybook)*CODEWORD/(PAGESIZE*BITS_P_COL));
        part2 = CODEWORD * exp(log(Ybook)*((CODEWORD-1)/(PAGESIZE*BITS_P_COL)));
        part3 = 1 - Yly;
        Ycw = part1 + part2 * part3;

        Yrow = pow(Ycw, BITS_P_ROW/CODEWORD);

//      printf("here3");

        Ysec = 0;
        for (i = 0; i <= RED_ROWS; i++)
        {
            part1 = gsl_sf_lnchoose(SEC_HEIGHT + RED_ROWS, i);
            part2 = (SEC_HEIGHT + RED_ROWS-i) * log(Yrow);
            part3 = i * log(1 - Yrow);
            Ysec += exp(part1 + part2 + part3);
        }

//      printf("here4");

        Ydram = pow(Ysec, NSECS);

        printf("%1.10f_%1.10f\n", psi, Ydram);
    }

    return 0;
}
```

### D.4.5 Associative Direct Redundancy

```
/* Program to calculate graphs for DRAM yield with associative direct
 * redundancy using the binomial yield model.
 * Craig Joly, December 13, 2002
 */

#include <stdio.h>
#include <stdlib.h>
#include <getopt.h>
#include <math.h>
#include <gsl/gsl_sf_gamma.h>

#define L_RATIO 0.2

#ifdef _16MDRAM
    #define BITS        16*1024*1024
    #define WORD_BITS   1
    #define NWORDS      4*4*1024*1024
    #define ADDR_BITS   22
    #define SEC_BITS    2
    #define MAX_CAM_E   128*1024
    #define CAM_STEP    1024
#endif

#ifdef _1GDRAM
    #define BITS        1024*1024*1024
    #define WORD_BITS   16
    #define NWORDS      2*32*1024*1024
    #define ADDR_BITS   25
    #define SEC_BITS    1
    #define MAX_CAM_E   128*1024
    #define CAM_STEP    1024
#endif

#define DNOT    2.0
#define DMAX    64.0

double sprob (int delta)
{
    return (delta < DNOT) ? (0.68085 * delta / 4.0):
                            (0.68085 * 2.0 / (delta * delta));
}

int main (int argc, char **argv)
{
    FILE *fc, *fw;
    float cams[64][64], words[64][64];
    double Ycl, Ydcl, Ydram;
    double Ycam_cl, Ymask_cl;
    double psi, psi_h, psi_l, psi_cam, psi_mask;
    double part1, part2, part3;
    int x, y;
    int cam_e, n_cam_c, cxy, nr;
    int c, i, j, reduce;
    opterr = 0;

#ifdef _16MDRAM
    fc = fopen("16M_ad.lut", "r");
    fw = fopen("16M_word.lut", "r");
#endif
#ifdef _cont
    fc = fopen("1G_ad_cont.lut", "r");
    fw = fopen("1G_word_cont.lut", "r");
#endif
#ifdef _int
    fc = fopen("1G_ad_int.lut", "r");
    fw = fopen("1G_word_int.lut", "r");
#endif

    for (j = 0; j < 64; j++)
        for (i = 0; i < 64; i++)
        {
            fscanf(fc, "%f", &(cams[i][j]));
            fscanf(fw, "%f", &(words[i][j]));
        }

    while ((c = getopt (argc, argv, "h")) != -1)
    {
        switch (c)
        {
            case 'h':
                fprintf(stderr, "Usage:%s\n", argv[0]);
                return 1;
            default:
                abort ();
        }
    }

    for (n_cam_e = 0; n_cam_e <= MAX_CAM_E; n_cam_e += CAM_STEP)
    {
        psi = psi_h = 1;
        psi_l = 0;
        reduce = 1;
        Ydram = 1.0;

        while (1)
        {
            Ydram = 1.0;
            for (x = 1; x <= DMAX; x++)
            {
                for (y = 1; y <=DMAX; y++)
                {
                    Ycl = exp(-psi * sprob(x) * sprob(y));

                    psi_cam = psi / L_RATIO;
                    psi_mask = psi_cam / 2;

                    Ycam_cl = exp(-psi_cam);
                    Ymask_cl = exp(-psi_mask);

                    cam_e = floor(pow(Ycam_cl, ADDR_BITS) *
                        pow(Ymask_cl, ADDR_BITS + SEC_BITS) *
                        pow(Ycl, 4 * WORD_BITS) * n_cam_e);

                    cxy = floor((double)cam_e * sprob(x) * sprob(y) / cams[x-1][y-1]);

                    nr = ceil(BITS / (x * y));
//                  printf("%d %f %d\n", NWORDS, words[x-1][y-1], nr);

                    Ydcl = 0;
                    for (i = 0; i <= cxy; i++)
                    {
                        part1 = gsl_sf_lnchoose(nr, i);
                        part2 = (nr - i) * log(Ycl);
                        part3 = i * log(1 - Ycl);
//                      printf("%f %f %f\n", part1, part2, part3);
                        Ydcl += exp(part1 + part2 + part3);
                    }
                    Ydram *=  Ydcl;
                }
            }

            if (Ydram > 0.51)
            {
                reduce = 0;
                psi_l = psi;
                psi = (psi + psi_h) / 2;
            }
            else if (Ydram < 0.49)
            {
                if (reduce)
                    psi /= 10;
                else
                    psi = (psi + psi_l) / 2;
            }
            else
            {
                break;
            }
        }

        printf("%1.10f_%d_%1.5f\n", psi, n_cam_c, Ydram);

    }

    return 0;
}
```

### D.4.6 Associative Indirect Redundancy

```
/* Program to calculate graphs for DRAM yield with associative direct
 * redundancy using the binomial yield model.
 * Craig Joly,  December 13, 2002
 */

#include <stdio.h>
#include <stdlib.h>
#include <getopt.h>
#include <math.h>
#include <gsl/gsl_sf_gamma.h>

#define L_RATIO 0.2

#ifdef _16MDRAM
    #define BITS         16*1024*1024
    #define WORD_BITS    1
    #define NWORDS       4*4*1024*1024
    #define ADDR_BITS    22
    #define SEC_BITS     2
    #define MAX_CAM_E    128*1024
    #define CAM_STEP     1024
#endif

#ifdef _1GDRAM
    #define BITS         1024*1024*1024
    #define WORD_BITS    16
    #define NWORDS       2*32*1024*1024
    #define ADDR_BITS    25
    #define SEC_BITS     1
    #define MAX_CAM_E    128*1024
    #define CAM_STEP     1024
#endif

#define DNOT     2.0
#define DMAX     64.0

double sprob (int delta)
{
    return (delta < DNOT) ? (0.68085 * delta / 4.0):
                            (0.68085 * 2.0 / (delta * delta));
}

int main (int argc, char **argv)
{
    FILE *fc, *fw;
    float cams[64][64], words[64][64];
    double Ycl, Ydcl, Ydram;
    double Ycam_cl, Yflash_cl;
    double psi, psi_h, psi_l, psi_cam, psi_flash;
    double part1, part2, part3;
    int x, y;
    int cam_e, n_cam_e, cxy, nr, dword, red_bits;
    int c, i, j, reduce;
    opterr = 0;

#ifdef _16MDRAM
    fc = fopen("16M_ai_cam.lut", "r");
    fw = fopen("16M_word.lut", "r");
#endif
#ifdef _cont
    fc = fopen("1G_ai_cam_cont.lut", "r");
    fw = fopen("1G_word_cont.lut", "r");
#endif
#ifdef _int
    fc = fopen("1G_ai_cam_int.lut", "r");
    fw = fopen("1G_word_int.lut", "r");
#endif

    for (j = 0; j < 64; j++)
        for (i = 0; i < 64; i++)
        {
            fscanf(fc, "%f", &(cams[i][j]));
            fscanf(fw, "%f", &(words[i][j]));
        }

    while ((c = getopt (argc, argv, "h")) != -1)
    {
        switch (c)
        {
            case 'h':
                fprintf(stderr, "Usage:%s\n", argv[0]);
                return 1;
            default:
                abort ();
        }
    }

    for (n_cam_e = 0; n_cam_e <= MAX_CAM_E; n_cam_e += CAM_STEP)
    {
        psi = psi_h = 1;
        psi_l = 0;
        reduce = 1;
        Ydram = 1.0;

        while (1)
        {
            Ydram = 1.0;
            dword = 0;
            for (x = 1; x <= DMAX; x++)
            {
                for (y = 1; y <=DMAX; y++)
                {
                    Ycl = exp(-psi * sprob(x) * sprob(y));

                    psi_cam = psi / L_RATIO;
                    psi_flash = psi_cam / 2;

                    Ycam_cl = exp(-psi_cam);
                    Yflash_cl = exp(-psi_flash);

                    cam_e = floor(pow(Ycam_cl, ADDR_BITS) *
                        pow(Yflash_cl, ADDR_BITS + SEC_BITS) *
                        n_cam_e);

                    cxy = floor((double)cam_e * sprob(x) * sprob(y) / cams[x-1][y-1]);

                    nr = ceil(BITS / (x * y));
//                  printf("%d %f %d\n", NWORDS, words[x-1][y-1], nr);

                    Ydcl = 0;
                    for (i = 0; i <= cxy; i++)
                    {
                        part1 = gsl_sf_lnchoose(nr, i);
                        part2 = (nr - i) * log(Ycl);
                        part3 = i * log(1 - Ycl);
//                      printf("%f %f %f\n", part1, part2, part3);
                        Ydcl += exp(part1 + part2 + part3);
                    }
                    Ydram *=  Ydcl;
                    dword += ceil((double)cxy * words[x-1][y-1]);
                }
            }

            if (Ydram > 0.51)
            {
                reduce = 0;
                psi_l = psi;
                psi = (psi + psi_h) / 2;
            }
            else if (Ydram < 0.49)
            {
                if (reduce)
                    psi /= 10;
                else
                    psi = (psi + psi_l) / 2;
            }
            else
            {
                break;
            }
        }

        red_bits = ceil(log((float)dword) / log(2.0));

        printf("%1.10f␣%d␣%1.5f␣%d\n", psi, n_cam_e, Ydram, red_bits);

    }

    return 0;
```

```
}
```

# D.5 Combined Fault Model

## D.5.1 No Redundancy

```
/* Program to calculate graphs for DRAM yield without redundancy using
 * a cluster fault model and a binomial (poisson) yiled model
 * Craig Joly, December 12, 2002
 */

#include <stdio.h>
#include <stdlib.h>
#include <getopt.h>
#include <math.h>

#ifdef _16MDRAM
    #define BITS    16*1024*1024
    #define NROWS   16*1024
    #define NCOLS   8768
    #define N_X     2192
    #define N_Y     8192
#endif

#ifdef _1GDRAM
    #define BITS    1024*1024*1024
    #define NROWS   65536
    #define NCOLS   262144
    #define N_X     32*1024
    #define N_Y     32*1024
#endif

#define Q       1.0
#define P       2.0
#define DNOT    2.0
#define DMAX    64.0
#define FPPARAM 0.68085

double sprob (int delta)
{
    return (delta < DNOT) ? (0.68085 * delta / 4.0):
                            (0.68085 * 2.0 / (delta * delta));
}

int main (int argc, char **argv)
{
    double Ysc, Yrow, Ycol, Ycla;
    double Ycl, Ydcl, Ydram;
    double lambda;
    double lstep, ltop;
    int b,c, x, y;

    opterr = 0;

    while ((c = getopt (argc, argv, "l:s:h")) != -1)
        switch (c)
        {
            case 'l' :
                ltop = atof(optarg);
                break;
            case 's' :
                lstep = atof(optarg);
                break;
            case 'h':
                fprintf(stderr, "Usage:%s␣-l␣end_lambda␣-s␣lambda_step\n", argv[0]);
                return 1;
            default:
                abort ();
        }

    for (lambda = 0; lambda < ltop; lambda += lstep)
    {
        Ysc = exp(-lambda * BITS / 3);
        Ycol = exp(-lambda / 6 * NCOLS);
        Yrow = exp(-lambda / 6 * NROWS);
        Ycla = 1;
        for (x = 1; x <= DMAX; x++)
        {
            for (y = 1; y <= DMAX; y++)
            {
                Ycl = exp(-lambda/3 * sprob(x) * sprob(y));
                Ydcl = pow(Ycl, (N_X - x + 1) * (N_Y - y + 1));
                Ycla *= Ydcl;
            }
        }
        Ydram = Ysc * Ycol * Yrow * Ycla;
        printf ("%1.10f␣%1.10f\n", lambda, Ydram);
    }

    return 0;
}
```

### D.5.2 Row and Column Redundancy

```
/* Program to calculate graphs for DRAM yield with row and column
 * redundancy using the negative binomial yield model.
 * Craig Joly,  December 13, 2002
 */

#include <stdio.h>
#include <stdlib.h>
#include <getopt.h>
#include <gsl/gsl_sf_gamma.h>
#include <gsl/gsl_pow_int.h>
#include <gsl/gsl_sf_exp.h>
#include <gsl/gsl_sf_log.h>

#ifdef _16MDRAM
    #define BITS     ((unsigned int)(16*1024*1024))
    #define RED_COLS 2
    #define RED_ROWS 24
    #define NBOOKS 64
    #define NSECS 4
    #define BITS_P_ROW 1024
    #define BITS_P_COL 2048
    #define SEC_HEIGHT 4096
    #define PAGESIZE 128
#endif

#ifdef _1GDRAM
    #define BITS     ((unsigned int)(1024*1024*1024))
    #define RED_COLS 16
    #define RED_ROWS 64
    #define NBOOKS 32
    #define NSECS 8
    #define BITS_P_ROW 16384
    #define BITS_P_COL 8192
    #define SEC_HEIGHT 8192
    #define PAGESIZE 8192
#endif

#define DMAX    64.0
#define DNOT    2.0
#define FPPARAM 0.68085

double sprob (int delta)
{
    return (delta < DNOT) ? (FPPARAM * delta / 4.0) :
                            (FPPARAM * 2.0 / (delta*delta));
}

int main (int argc, char **argv)
{
    double Ysc, Ycol, Yscbook, Yrow, Ydram;
    double Ycl, Ycoleq, Ycolcl, Ycolcluster;
    double ycp1, ycp2, Yc1, Yc2, Yr1;
    double Yroweq, Ybook, Ysec;
    double part1, part2, part3;
    double lambda;
    double lstep, ltop;
    int x, y, i, c;

    opterr = 0;

    while ((c = getopt (argc, argv, "l:s:h")) != -1)
    {
        switch (c)
        {
            case 'l' :
                ltop = atof(optarg);
                break;
            case 's' :
                lstep = atof(optarg);
                break;
            case 'h':
                fprintf(stderr, "Usage:%s␣-l␣end_lambda␣-s␣lambda_step\n", argv[0]);
                return 1;
            default:
                abort ();
        }
    }

    for (lambda = 0 + lstep; lambda < ltop; lambda += lstep)
    {
        Ysc = exp(-lambda/3);
        Ycol = exp(-lambda/6);
        Yrow = exp(-lambda/6);
        Ycl = gsl_pow_int(Ysc, BITS_P_COL);

        Yc2 = 1.0;
        for (x = 1; x <= DMAX; x++)
        {
            for (y = 1; y <= DMAX; y++)
            {
                Ycl = exp(-lambda / 3 * sprob(x) * sprob(y));
                Ycolcl = gsl_pow_int(Ycl, BITS_P_COL+RED_ROWS-y+1);
                ycp1 = gsl_pow_int(Ycolcl, x*(PAGESIZE + RED_COLS - 2*x + 2));
                ycp2 = 1;
                for (i = 1; i < x; i++)
                    ycp2 *= gsl_pow_int(Ycolcl, 2*i);
                Yc2 *= gsl_sf_exp(gsl_sf_log(ycp1*ycp2)/(PAGESIZE+RED_COLS));
            }
        }

        Ycoleq =  Ycl * Ycol * Yc2;
//      printf("Ycol=%f\n", Ycol);
        Ybook = 0;
        for (i = 0; i <= RED_COLS; i++)
        {
            part1 = gsl_sf_lnchoose(PAGESIZE + RED_COLS, i);
            part2 = (PAGESIZE + RED_COLS - i) * log(Ycoleq);
            part3 = i * log(1.0 - Ycoleq);
            Ybook += exp(part1 + part2 + part3);
        }
        Yscbook = exp(log(Ybook) / (BITS_P_COL * PAGESIZE));
        Yrl = gsl_pow_int(Yscbook, BITS_P_ROW);
        Yroweq = Yrl * Yrow;
        Ysec = 0;
        for (i = 0; i <= RED_ROWS; i++)
        {
            part1 = gsl_sf_lnchoose(SEC_HEIGHT + RED_ROWS, i);
            part2 = (SEC_HEIGHT + RED_ROWS - i) * log(Yroweq);
            part3 = i * log(1 - Yroweq);
            Ysec += exp(part1 + part2 + part3);
        }
        Ydram = gsl_pow_int(Ysec, NSECS);

        printf("%1.10f_%1.10f\n", lambda, Ydram);
    }

    return 0;
}
```

### D.5.3 ECC Redundancy

```
/* Program to calculate graphs for DRAM yield with row and column
 * redundancy using the negative binomial yield model.
 * Craig Joly,  Jan 18, 2003
 */

#include <stdio.h>
#include <stdlib.h>
#include <getopt.h>
#include <math.h>

#ifdef _16MDRAM
    #define NCOLS    8768
    #define NBOOKS   64
    #define NROWS    16*1024
    #define NSECS    4
    #define CODEWORD 137
    #define PAGESIZE 137
    #define BITS_P_COL  2048
    #define SEC_HEIGHT  4096
    #define NCWP 1
#endif

#ifdef _1GDRAM
    #define NCOLS    267776
    #define NBOOKS   32
    #define NROWS    65536
    #define NSECS    8
    #define CODEWORD 523
    #define PAGESIZE 8368
    #define BITS_P_COL  4096
    #define SEC_HEIGHT  8192
    #define NCWP 16
#endif

#define DMAX     64.0
#define DNOT     2.0
```

```
#define FPPARAM 0.68085

double sprob (int delta)
{
    return (delta < DNOT) ? (FPPARAM * delta / 4.0) :
                            (FPPARAM * 2.0 / ( delta*delta));
}

int main (int argc, char **argv)
{
    double Ysc, Ycw1, Ycw2, Ycw, Ycl;
    double yrp1, yrp2, part1, part2, part3, Y1y, Ycolcw1, Ycolcw2;
    double Yrowcl, Ycolcw;
    double Yrow, Ydrow, Ycol, Ybook, Ysec, Ydram;
    double lambda;
    double lstep, ltop;
    int x, y;
    int i,b,c;

    opterr = 0;

    while ((c = getopt (argc, argv, "l:s:h")) != -1)
        switch (c)
        {
            case 'l' :
                ltop = atof(optarg);
                break;
            case 's' :
                lstep = atof(optarg);
                break;
            case 'h':
                fprintf(stderr, "Usage:%s␣-l␣end_lambda␣-s␣lambda_step\n", argv[0]);
                return 1;
            default:
                abort ();
        }

    for (lambda = 0; lambda < ltop; lambda += lstep) {

        Ysc = exp(-lambda/3);
        Ycol = exp(-lambda/6);
        Ydrow = exp(-lambda/6 * NCOLS);

        Ycw1 = pow(Ysc, CODEWORD) + CODEWORD * pow(Ysc, CODEWORD-1) * (1 - Ysc);

        Yrow = 1.0;
        for (x = 1; x <= DMAX; x++)
        {
            for (y = 1; y <= DMAX; y++)
            {
                Ycl = exp(-lambda/3 * sprob(x) * sprob(y));
                Yrowcl = pow(Ycl, PAGESIZE-x+1);
                yrp1 = pow(Yrowcl, y*(BITS_P_COL - 2*y + 2));
                yrp2 = 1;
                for (i = 1; i < y; i++)
                    yrp2 *= pow(Yrowcl, 2*i);
                Yrow *= exp(log(yrp1 * yrp2) / BITS_P_COL);
            }
        }

        Y1y = 1.0;
        for (y = 1; y <= DMAX; y++)
        {
            Ycl = exp(-lambda/6 * sprob(1) * sprob(y));
            yrp1 = pow(Ycl, y*(BITS_P_COL - 2*y + 2));
            yrp2 = 1;
            for (i = 1; i < y; i++)
                yrp2 *= pow(Ycl, 2*i);
            Y1y *= exp(log(yrp1 * yrp2)/BITS_P_COL);
        }

        part1 = exp(log(Yrow) * CODEWORD / PAGESIZE);
        part2 = CODEWORD * exp(log(Yrow) * ((CODEWORD - 1) / PAGESIZE));
        part3 = 1 - Y1y;
        Ycw2 = part1 + part2 * part3;

        Ycw = Ycw1 * Ycw2;
        Ycolcw1 = pow(Ycw, BITS_P_COL);
        Ycolcw2 = (pow(Ycol, BITS_P_COL) + BITS_P_COL *
                      pow (Ycol, BITS_P_COL - 1) * (1 - Ycol));
        Ycolcw = Ycolcw1 * Ycolcw2;
        Ybook = pow(Ycolcw, NCWP);
        Ydram = pow(Ybook, NSECS) * Ydrow;

        printf("%1.10f␣%1.10f\n", lambda, Ydram);

    }

    return 0;
}
```

### D.5.4 Row and Column Redundancy with ECC

```
/* Program to calculate graphs for DRAM yield with row and column
 * redundancy using the negative binomial yield model.
 * Craig Joly,  Jan 18, 2003
 */

#include <stdio.h>
#include <stdlib.h>
#include <getopt.h>
#include <math.h>
#include <gsl/gsl_sf_gamma.h>
#include <gsl/gsl_pow_int.h>

#ifdef _16MDRAM
    #define RED_COLS 2
    #define RED_ROWS 24
    #define NCOLS     8768
    #define NBOOKS    64
    #define NROWS     16*1024
    #define NSECS     4
    #define CODEWORD 137
    #define PAGESIZE 137
    #define BITS_P_COL   2048
    #define BITS_P_ROW   1096
    #define SEC_HEIGHT   4096
    #define NCWP 1
#endif

#ifdef _1GDRAM
    #define RED_COLS 16
    #define RED_ROWS 64
    #define NCOLS     267776
    #define NBOOKS    32
    #define NROWS     65536
    #define NSECS     8
    #define CODEWORD 523
    #define PAGESIZE 8368
    #define BITS_P_COL   4096
    #define BITS_P_ROW   16736
    #define SEC_HEIGHT   8192
    #define NCWP 16
#endif

#define DMAX     64.0
#define DNOT     2.0
#define FPPARAM 0.68085

double sprob (int delta)
{
    return (delta < DNOT) ? (FPPARAM * delta / 4.0) :
                            (FPPARAM * 2.0 / ( delta*delta));
}

int main (int argc, char **argv)
{
    double Ysc, Ycw1, Ycw2, Ycw, Ycl;
    double yrp1, yrp2, Yr1, part1, part2, part3, Y1y, Ycolcw1, Ycolcw2;
    double Yrowcl, Ycolcw;
    double Ycoleq, Yroweq, Yscbook;
    double Yrow, Ydrow, Ycol, Ybook, Ysec, Ydram;
    double lambda;
    double lstep, ltop;
    int x, y;
    int i,b,c;

    opterr = 0;

    while ((c = getopt (argc, argv, "l:s:h")) != -1)
        switch (c)
        {
            case 'l' :
                ltop = atof(optarg);
                break;
            case 's' :
                lstep = atof(optarg);
                break;
            case 'h':
                fprintf(stderr, "Usage:%s␣-l␣end_lambda␣-s␣lambda_step\n", argv[0]);
                return 1;
            default:
                abort ();
        }

    for (lambda = 0 + lstep; lambda < ltop; lambda += lstep) {
        Ysc = exp(-lambda/3);
        Ycol = exp(-lambda/6);
        Yrow = exp(-lambda/6);

        Ycw1 = pow(Ysc, CODEWORD) + CODEWORD * pow(Ysc, CODEWORD-1) * (1 - Ysc);

        Yrow = 1.0;
        for (x = 1; x <= DMAX; x++)
        {
            for (y = 1; y <= DMAX; y++)
            {
                Ycl = exp(-lambda/3 * sprob(x) * sprob(y));
                Yrowcl = pow(Ycl, PAGESIZE-x+1);
                yrp1 = pow(Yrowcl, y*(BITS_P_COL - 2*y + 2));
                yrp2 = 1;
                for (i = 1; i < y; i++ )
                    yrp2 *= pow(Yrowcl, 2*i);
                Yrow *= exp(log(yrp1 * yrp2) / BITS_P_COL);
            }
        }

        Y1y = 1.0;
        for (y = 1; y <= DMAX; y++)
        {
            Ycl = exp(-lambda/6 * sprob(1) * sprob(y));
            yrp1 = pow(Ycl, y*(BITS_P_COL - 2*y + 2));
            yrp2 = 1;
            for (i = 1; i < y; i++)
                yrp2 *= pow(Ycl, 2*i);
            Y1y *= exp(log(yrp1 * yrp2)/BITS_P_COL);
        }

        part1 = exp(log(Yrow) * CODEWORD / PAGESIZE);
        part2 = CODEWORD * exp(log(Yrow) * ((CODEWORD - 1) / PAGESIZE));
        part3 = 1 - Y1y;
        Ycw2 = part1 + part2 * part3;

        Ycw = Ycw1 * Ycw2;
        Ycolcw1 = pow(Ycw, BITS_P_COL);
        Ycolcw2 = (pow(Ycol, BITS_P_COL) + BITS_P_COL *
                    pow (Ycol, BITS_P_COL - 1) * (1 - Ycol));
        Ycolcw = Ycolcw1 * Ycolcw2;

        Ycoleq = exp(log(Ycolcw) / CODEWORD);

        Ybook = 0;
        for (i = 0; i <= RED_COLS; i++)
        {
            part1 = gsl_sf_lnchoose(PAGESIZE + RED_COLS, i);
            part2 = (PAGESIZE + RED_COLS - i) * log(Ycoleq);
            part3 = i * log(1.0 - Ycoleq);
            Ybook += exp(part1 + part2 + part3);
        }

        Yscbook = exp(log(Ybook) / (BITS_P_COL * PAGESIZE));
        Yr1 = gsl_pow_int(Yscbook, BITS_P_ROW);
        Yroweq = Yr1 * Yrow;
        Ysec = 0;
        for (i = 0; i <= RED_ROWS; i++)
        {
            part1 = gsl_sf_lnchoose(SEC_HEIGHT + RED_ROWS, i);
            part2 = (SEC_HEIGHT + RED_ROWS - i) * log(Yroweq);
            part3 = i * log(1 - Yroweq);
            Ysec += exp(part1 + part2 + part3);
        }
        Ydram = gsl_pow_int(Ysec, NSECS);

        printf("%1.10f_%1.10f\n", lambda, Ydram);
    }

    return 0;
}
```

### D.5.5 Associative Direct Redundancy

```
#include <stdio.h>
#include <stdlib.h>
#include <getopt.h>
#include <math.h>
#include <gsl/gsl_sf_gamma.h>
#include <gsl/gsl_pow_int.h>

#define L_RATIO 0.2

#ifdef _16MDRAM
    #define BITS        16*1024*1024
    #define WORD_BITS   1
    #define N_WORDS     16*1024*1024
    #define N_ROWS       16384
    #define N_COLS       8192
    #define ADDR_BITS    22
    #define SEC_BITS     2
    #define BITS_P_ROW   1024
    #define BITS_P_COL   2048
    #define NCAM         2048
    #define NSUP         8
    #define NCRDN        2
#endif

#ifdef _1GDRAM
    #define BITS         1024*1024*1024
    #define WORD_BITS    16
    #define N_WORDS      64*1024*1024
    #define N_ROWS       65536
    #define N_COLS       262144
    #define ADDR_BITS    25
    #define SEC_BITS     1
    #define BITS_P_ROW   16384
    #define BITS_P_COL   4096
    #define NCAM         74752
#endif

#define DNOT     2.0
#define DMAX     64.0
#define FPPARAM 0.68085

double sprob (int delta)
{
    return (delta < DNOT) ? (FPPARAM * delta / 4.0) :
                            (FPPARAM * 2.0 / (delta * delta));
}


int main (int argc, char **argv)
{
    FILE *fc, *fw;
    float cams[64][64], words[64][64];
    double Ysc, Yword, Ycol, Ywordcol, Yrow, Ycl, Ydram;
    double lambda, l_cam, l_mask;
    double ltop, lstep;
    double Ycam, Ymask, Ycamsrc, Ycamcl;
    double Ycamcl1, Ycamcl2, Ycamcl3, Ycamsrc1, Ycamsrc2, Ycamsrc3;
    double part1, part2, part3;
    double Yd1, Yd2, Yd3, Yd4, Ydcl;
    int Nesc, Nerow, Necol, Necl, nr, cxy;
    int cam_e;
    int c, i,j,k,l,x,y, xtop;

    opterr = 0;

#ifdef _16MDRAM
    fc = fopen("16M_ad.lut", "r");
    fw = fopen("16M_word.lut", "r");
#endif
#ifdef _cont
    fc = fopen("1G_ad_cont.lut", "r");
    fw = fopen("1G_word_cont.lut", "r");
#endif
#ifdef _int
    fc = fopen("1G_ad_int.lut", "r");
    fw = fopen("1G_word_int.lut", "r");
#endif

    for (j = 0; j < 64; j++)
        for (i = 0; i < 64; i++)
        {
            fscanf(fc, "%f", &(cams[i][j]));
            fscanf(fw, "%f", &(words[i][j]));
        }

    while ((c = getopt (argc, argv, "l:s:h")) != -1)
    {
        switch (c)
        {
            case 'l' :
                ltop = atof(optarg);
                break;
            case 's' :
                lstep = atof(optarg);
                break;
            case 'h' :
                fprintf(stderr, "Usage:␣%s␣-l␣max_lambda␣-s␣lambda_step\n", argv[0]);
                return 1;
            default :
                abort ();
        }
    }
    Nesc = -50;
    for (lambda = 0; lambda <ltop; lambda += lstep)
    {
        l_cam = -lambda / (3 * L_RATIO);
        l_mask = l_cam / 2;

        Ysc = exp(-lambda / 3);
        Ycol = exp(-lambda / 6);
        Yrow = exp(-lambda / 6);

        Ycam = exp(-l_cam);
        Ymask = exp(-l_mask);

        Ycamsrc1 = gsl_pow_int(Ycam, ADDR_BITS) *
                gsl_pow_int(Ymask, ADDR_BITS + SEC_BITS) *
                gsl_pow_int(Ysc, WORD_BITS);
        Ycamsrc2 =  Yrow;
        Ycamsrc3 =  gsl_pow_int(Ycol, 3 * ADDR_BITS + SEC_BITS + 4* WORD_BITS);

        Ycamsrc = Ycamsrc1 * Ycamsrc2 * Ycamsrc3;

        Ycamcl1 = 1.0;
        for (x = 1; x <= 2 * ADDR_BITS; x++)
            for (y = 1; y <= DMAX; y++)
                Ycamcl1 *= exp(-lambda / 3 * (2 * ADDR_BITS - x + 1) *
                    (1024 - y + 1) * sprob(x) * sprob(y));

        Ycamcl2 = 1.0;
        for (x = 1; x <= ADDR_BITS + SEC_BITS; x++)
            for (y = 1; y <= DMAX; y++)
                Ycamcl2 *= exp(-lambda / 3 * (ADDR_BITS + SEC_BITS - x + 1) *
                    (1024 - y + 1) * sprob(x) * sprob(y));

        Ycamcl3 = 1.0;
        xtop = (4 * WORD_BITS > DMAX) ? 4 * WORD_BITS : DMAX;
        for (x = 1; x <= xtop; x++)
            for (y = 1; y <= DMAX; y++)
                Ycamcl3 *= exp(-lambda / 3 * (4 * WORD_BITS - x + 1) *
                    (256 -y + 1) * sprob(x) * sprob(y));

        Ycamcl = Ycamcl1 * Ycamcl2 * Ycamcl3;

        cam_e = NCAM * Ycamsrc * Ycamcl;

#ifdef _16MDRAM
        Nesc = NSUP * ceil((float)cam_e /
            (2.0 + (float)(BITS_P_COL + BITS_P_ROW) / 8.0));
        Nerow = (Nesc / NSUP - NCRDN) * BITS_P_ROW / (8 * WORD_BITS);
        Necol = (Nesc / NSUP - NCRDN) * BITS_P_COL / 8;
#endif
#ifdef _1GDRAM
        Nesc += 200;
        Nerow = 1024;
        Necol = 5120;
#endif

        Necl = cam_e - Necol - Nerow - Nesc;

        Yword = gsl_pow_int(Ysc, WORD_BITS);
        Ywordcol = gsl_pow_int(Ycol, WORD_BITS);

        Yd1 = 0;
        for (i = 0; i < Nesc; i++)
        {
            part1 = gsl_sf_lnchoose(N_WORDS, i);
            part2 = (N_WORDS - i) * log(Yword);
            part3 = i * log(1 - Yword);
            Yd1 += exp(part1 + part2 + part3);
        }
        if (Yd1 > 1.0) Yd1 = 1.0;

        Yd2 = 0;
        for (j = 0; j < 4 * Nerow * WORD_BITS / BITS_P_ROW; j++)
        {
            part1 = gsl_sf_lnchoose(N_ROWS, j);
            part2 = (N_ROWS - j) * log(Yrow);
            part3 = j * log(1 - Yrow);
            Yd2 += exp(part1 + part2 + part3);
        }

        Yd3 = 0;
        for (k = 0; k < 4 * Necol / BITS_P_COL; k++)
        {
            part1 = gsl_sf_lnchoose(N_COLS / WORD_BITS, k);
            part2 = (N_COLS / WORD_BITS - k) * log(Ywordcol);
            part3 = k * log(1 - Ywordcol);
            Yd3 += exp(part1 + part2 + part3);
        }
        Yd4 = 1.0;
        for (x = 1; x <= DMAX; x++)
        {
            for (y = 1; y <= DMAX; y++)
            {
                Ycl = exp(-lambda / 3 * sprob(x) * sprob(y));

                cxy = floor((double)Necl * sprob(x) * sprob(y) / cams[x-1][y-1]);
                nr = ceil((float)BITS / (float)(x * y));

                Ydcl = 0;
                for (l = 0; l <= cxy; l++)
                {
                    part1 = gsl_sf_lnchoose(nr, l);
                    part2 = (nr - l) * log(Ycl);
                    part3 = l * log(1 - Ycl);
                    Ydcl += exp(part1 + part2 + part3);
                }
                Yd4 *= Ydcl;
            }
        }

        Ydram = Yd1 * Yd2 * Yd3 * Yd4;

//      printf("%d %d %d %d %d\n", cam_e, Nesc, Nerow, Necol, Necl);
        printf("%1.10f_%1.10f_%1.10f_%1.10f_%1.10f_%1.10f\n", lambda, Yd1, Yd2, Yd3, Yd4, Ydram);
    }

    return 0;
}
```

### D.5.6 Associative Indirect Redundancy

```
#include <stdio.h>
#include <stdlib.h>
#include <getopt.h>
#include <math.h>
#include <gsl/gsl_sf_gamma.h>
#include <gsl/gsl_pow_int.h>

#define L_RATIO 0.2

#ifdef _16MDRAM
    #define BITS          16*1024*1024
    #define WORD_BITS     1
    #define N_WORDS       16*1024*1024
    #define N_ROWS        16384
    #define N_COLS        8192
    #define ADDR_BITS     22
    #define SEC_BITS      2
    #define RED_BITS      18
    #define BITS_P_ROW    1024
    #define BITS_P_COL    2048
    #define NCAM          1126
    #define N_X           256         // dimensions of redundant mem
    #define N_Y           1024
    #define MIN_SC        100
    #define MAX_C         1
    #define MAX_R         1
#endif

#ifdef _1GDRAM
    #define BITS          1024*1024*1024
    #define WORD_BITS     16
    #define N_WORDS       64*1024*1024
    #define N_ROWS        65536
    #define N_COLS        262144
    #define ADDR_BITS     25
    #define SEC_BITS      1
    #define RED_BITS      20
    #define BITS_P_ROW    16384
    #define BITS_P_COL    4096
    #define NCAM          32768
    #define N_X           1024        // dimensions of redundant mem
    #define N_Y           512
//  #define MIN_SC        1000
    #define MAX_C         3
    #define MAX_R         2
#endif

#define DNOT     2.0
#define DMAX     64.0
#define FPPARAM  0.68085

double sprob (int delta)
{
    return (delta < DNOT) ? (FPPARAM * delta / 4.0) :
```

```
                                    (FPPARAM * 2.0 / (delta * delta));
}

int main (int argc, char **argv)
{
    FILE *fc, *fw;
    float cams[64][64], words[64][64];
    double Ysc, Yword, Ycol, Ywordcol, Yrow, Ycl, Ydram;
    double lambda, l_cam, l_mask;
    double ltop, lstep;
    double Ycam, Ymask, Ycamsrc, Ycamcl;
    double Ycamcl1, Ycamcl2, Ycamcl3, Ycamsrc1, Ycamsrc2, Ycamsrc3;
    double Ycla, Yclb, Yclc;
    double part1, part2, part3;
    double Yd1, Yd2, Yd3, Yd4, Ydcl;
    float Rl1, Rl2, Rl3, Rl4;
    int Nrw, Nesc, Nerow, Necol, Necl, nr, cxy, N1, N2;
    int Wesc, Werow, Wecol, Wecl, dword;
    int cam_e, min_sc;
    int c, i,j,k,l,x,y, xtop;

    opterr = 0;

#ifdef _16MDRAM
    fc = fopen("16M_ai_cam.lut", "r");
    fw = fopen("16M_word.lut", "r");
#endif
#ifdef _cont
    fc = fopen("1G_ai_cam_cont.lut", "r");
    fw = fopen("1G_word_cont.lut", "r");
#endif
#ifdef _int
    fc = fopen("1G_ai_cam_int.lut", "r");
    fw = fopen("1G_word_int.lut", "r");
#endif

    for (j = 0; j < 64; j++)
        for (i = 0; i < 64; i++)
        {
            fscanf(fc, "%f", &(cams[i][j]));
            fscanf(fw, "%f", &(words[i][j]));
        }

    while ((c = getopt (argc, argv, "l:s:m:h")) != -1)
    {
        switch (c)
        {
            case 'l' :
                ltop = atof(optarg);
                break;
            case 's' :
                lstep = atof(optarg);
                break;
            case 'm' :
                min_sc = atof(optarg);
                break;
            case 'h' :
                fprintf(stderr, "Usage:_%s_-l_max_lambda_-s_lambda_step\n", argv[0]);
                return 1;
            default :
                abort ();
        }
    }

    for (lambda = 0; lambda <ltop; lambda += lstep)
    {
        l_cam = -lambda / (3 * L_RATIO);
        l_mask = l_cam / 2;

        Ysc = exp(-lambda / 3);
        Ycol = exp(-lambda / 6);
        Yrow = exp(-lambda / 6);

        Ycam = exp(-l_cam);
        Ymask = exp(-l_mask);

        Ycamsrc1 = gsl_pow_int(Ycam, ADDR_BITS) *
                gsl_pow_int(Ymask, ADDR_BITS + SEC_BITS + RED_BITS);
        Ycamsrc2 = Yrow;
        Ycamsrc3 =  gsl_pow_int(Ycol, 3 * ADDR_BITS + SEC_BITS + RED_BITS);

        Ycamsrc = Ycamsrc1 * Ycamsrc2 * Ycamsrc3;

        Ycamcl1 = 1.0;
        for (x = 1; x <= 2 * ADDR_BITS; x++)
            for (y = 1; y <=DMAX; y++)
                Ycamcl1 *= exp(-lambda / 3 * (2 * ADDR_BITS - x + 1) *
                    (1024 - y + 1) * sprob(x) * sprob(y));
        Ycamcl2 = 1.0;
        for (x = 1; x <= ADDR_BITS + SEC_BITS + RED_BITS; x++)
            for (y = 1; y <= DMAX; y++)
                Ycamcl2 *= exp(-lambda / 3 * (ADDR_BITS + SEC_BITS + RED_BITS
                    - x + 1) * (1024 - y + 1) * sprob(x) * sprob(y));

        Ycamcl = Ycamcl1 * Ycamcl2;

        cam_e = NCAM * Ycamsrc * Ycamcl;

        Ycla = 1;
        for (x = 1; x <= DMAX; x++)
            for (y = 1; y <= DMAX; y++)
            {
                Yclb = exp(-lambda / 3 * sprob(x) * sprob(y));
                Yclc = gsl_pow_int(Yclb, (N_X - x + 1) * (N_Y - y + 1));
                Ycla *= Yclc;
            }

        Rl1 = lambda / 3 * gsl_pow_int(2, RED_BITS) * WORD_BITS;
        Rl2 = lambda / 6 * N_Y * gsl_pow_int(2, RED_BITS) /
            (N_X * N_Y);
        Rl3 = lambda / 6 * N_X * gsl_pow_int(2, RED_BITS) /
            (N_X * N_Y);
        Rl4 = lambda / 3 * gsl_pow_int(2, RED_BITS) * WORD_BITS * 1024;

        Nrw = gsl_pow_int(2, RED_BITS) - Rl1 - Rl2 - Rl3 - Rl4;;

        Wesc = Nrw / (5 +  (BITS_P_COL + BITS_P_ROW / WORD_BITS) / 2);

        if (Wesc < min_sc) Wesc = min_sc;
#ifdef _cont
        if (lambda > 0.000016) Wesc += 2000;
#endif

        Nerow = Wesc / 2;
        Necol = Wesc / 2;

        if (Nerow > MAX_R) Nerow = MAX_R;
        if (Necol > MAX_C) Necol = MAX_C;

        Nesc = Wesc;
        Necl = cam_e - Nesc - Nerow - Necol;
        Werow = Nerow * BITS_P_ROW / WORD_BITS;
        Wecol = Necol * BITS_P_COL;
        Wecl = Nrw - Wesc - Werow - Wecol;

        Yword = gsl_pow_int(Ysc, WORD_BITS);
        Ywordcol = gsl_pow_int(Ycol, WORD_BITS);

        dword = Wecl + 10;
        Necl += 500;
        while (dword > Wecl)
        {
            Necl -= 500 ;
            Yd4 = 1.0;
            dword = 0;
            for (x = 1; x <= DMAX; x++)
            {
                for (y = 1; y <= DMAX; y++)
                {
                    Ycl = exp(-lambda / 3 * sprob(x) * sprob(y));

                    cxy = floor((double)Necl * sprob(x) * sprob(y) /
                                cams[x-1][y-1]);
                    nr = ceil((float)BITS / (float)(x * y));

                    Ydcl = 0;
                    for (l = 0; l <= cxy; l++)
                    {
                        part1 = gsl_sf_lnchoose(nr, l);
                        part2 = (nr - l) * log(Ycl);
                        part3 = l * log(1 - Ycl);
                        Ydcl += exp(part1 + part2 + part3);
                    }
                    Yd4 *= Ydcl;
                    dword += ceil((double)cxy * words[x-1][y-1]);
                }
            }
        }

        N1 = Nrw - (Werow + Wecol + dword);
        N2 = cam_e - Necl - Necol - Nerow;
        Nesc = (N1 < N2) ? N1 : N2;
        Wesc = Nesc;

        Yd1 = 0;
        for (i = 0; i < Nesc; i++)
        {
            part1 = gsl_sf_lnchoose(N_WORDS, i);
            part2 = (N_WORDS - i) * log(Yword);
            part3 = i * log(1 - Yword);
            Yd1 += exp(part1 + part2 + part3);
        }
        if (Yd1 > 1.0) Yd1 = 1.0;

        Yd2 = 0;
        for (j = 0; j < Nerow; j++)
        {
            part1 = gsl_sf_lnchoose(N_ROWS, j);
            part2 = (N_ROWS - j) * log(Yrow);
            part3 = j * log(1 - Yrow);
            Yd2 += exp(part1 + part2 + part3);
        }

        Yd3 = 0;
        for (k = 0; k < Necol; k++)
        {
            part1 = gsl_sf_lnchoose(N_COLS / WORD_BITS, k);
            part2 = (N_COLS / WORD_BITS - k) * log(Ywordcol);
            part3 = k * log(1 - Ywordcol);
            Yd3 += exp(part1 + part2 + part3);
        }


        Ydram = Yd1 * Yd2 * Yd3 * Yd4;

        printf("%1.10f␣%1.10f␣%1.10f␣%1.10f␣%1.10f␣%1.10f\n", lambda, Yd1, Yd2, Yd3, Yd4, Ydram);

    }

    return 0;
}
```

# Appendix E

# Model VHDL and Ruby Code

## E.1 Top Level

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
library work;
use work.redun_pkg.all;

entity assoc_i_16 is
    port (  gclk         : in std_logic;
            a            : in std_logic_vector (21 downto 0);
            d            : in std_logic_vector (3 downto 0);
            qp           : in std_logic_vector (3 downto 0);
            q            : out std_logic_vector (3 downto 0);
            cas_n        : in std_logic;
            ras_n        : in std_logic;
            cs_n         : in std_logic;
            rw_n         : in std_logic;
            mem_en_n     : inout std_logic
    );
end entity assoc_i_16;

architecture struct of assoc_i_16 is

constant cam_depth : positive := 1152;
constant a_bits : positive := 22;
constant row_bits : positive := 11;
constant c_bits : positive := 11;
constant red_width : positive := 18;
constant sec_width : positive := 2;
constant off_width : positive := 12;
constant word_width : positive := 4;
constant s_word_width : positive := 1;

signal match : std_logic_vector(cam_depth - 1 downto 0);
signal red_en : std_logic;
signal section : std_logic_vector(sec_width - 1 downto 0);
signal base, pointer : std_logic_vector(red_width - 1 downto 0);
signal mask : std_logic_vector(a_bits - 1 downto 0);
signal offset : std_logic_vector(off_width - 1 downto 0);
signal wl_sel, bl_sel : std_logic_vector(511 downto 0);
signal dr, qr : std_logic_vector(s_word_width - 1 downto 0);
signal qp0, qp1, qp2, qp3 : std_logic_vector(s_word_width - 1 downto 0);
signal q0, q1, q2, q3 : std_logic_vector(s_word_width - 1 downto 0);
signal d0, d1, d2, d3 : std_logic_vector(s_word_width - 1 downto 0);

signal sel0, sel1, sel2, sel3 : std_logic;
signal selin0, selin1 : std_logic;

begin

    compare : component match_array
        generic map (
            addr_width    => a_bits,
            entries       => cam_depth,
            row_width     => row_bits,
            col_width     => c_bits
        )
        port map (
            addr          => a,
            cas_n         => cas_n,
            ras_n       => ras_n,
            match       => match,
            red_en      => red_en,
            mem_en_n    => mem_en_n
        );

    data : component flash_array
        generic map (
            addr_width  => a_bits,
            red_bits    => red_width,
            sec_bits    => sec_width,
            entries     => cam_depth
        )
        port map (
            match   => match,
            section => section,
            base    => base,
            mask    => mask
        );

    shift : component switch_22_12
        port map (
            addr    => a,
            dc_mask => mask,
            offset  => offset
        );

    compute : component nc_adder
        generic map (
            b_red   => red_width,
            b_off   => off_width
        )
        port map (
            base    => base,
            offset  => offset,
            pointer => pointer
        );

    x_decode : component decode_9_512
        port map (
            input   => pointer(17 downto 9),
            enable  => red_en,
            output  => wl_sel
        );

    y_decode : component decode_9_512
        port map (
            input   => pointer(8 downto 0),
            enable  => red_en,
            output  => bl_sel
        );

    -- redundant memory array
    r_array : component red_array
        generic map (
            row_bits    => 9,
            col_bits    => 9,
            word_width  => 1
        )
        port map (
            rows    => wl_sel,
            cols    => bl_sel,
            d       => dr,
            q       => qr,
            rw_n    => rw_n
        );

    -- combinational logic for four output muxes
    sel3 <= mem_en_n and section(1) and section(0);
    sel2 <= mem_en_n and section(1) and not section(0);
    sel1 <= mem_en_n and not section(1) and section(0);
    sel0 <= mem_en_n and not section(1) and not section(0);

    -- splice vectors
        qp3(0) <= qp(3);
        qp2(0) <= qp(2);
        qp1(0) <= qp(1);
        qp0(0) <= qp(0);

    -- output muxes to determine if data should come from primary or
    -- redundant memory
    outmux3 : component mux_2_1
        generic map (
            width   => 1
        )
        port map (
            in_0    => qp3,
            in_1    => qr,
            sel     => sel3,
            output   => q3
        );

    outmux2 : component mux_2_1
        generic map (
            width    => 1
        )
        port map (
            in_0     => qp2,
            in_1     => qr,
            sel      => sel2,
            output   => q2
        );

    outmux1 : component mux_2_1
        generic map (
            width    => 1
        )
        port map (
            in_0     => qp1,
            in_1     => qr,
            sel      => sel1,
            output   => q1
        );

    outmux0 : component mux_2_1
        generic map (
            width    => 1
        )
        port map (
            in_0     => qp0,
            in_1     => qr,
            sel      => sel0,
            output   => q0
        );

    q <= q3 & q2 & q1 & q0;

    -- input mux to place data in the redundant memory array if
    -- necessary
    selin0 <= mem_en_n and section(0);
    selin1 <= mem_en_n and section(1);

    d3(0) <= d(3);
    d2(0) <= d(2);
    d1(0) <= d(1);
    d0(0) <= d(0);

    inmux : component mux_4_1
        generic map (
            width    => 1
        )
        port map (
            in_0     => d0,
            in_1     => d1,
            in_2     => d2,
            in_3     => d3,
            sel0     => selin0,
            sel1     => selin1,
            output   => dr
        );

end architecture struct;
```

## E.2 Package

```
-- package file for associatiev indirect ternary CAM redundancy
-- components

library ieee;
use ieee.std_logic_1164.all;

package redun_pkg is

    -- DRAM specific contsats
    constant data_width      : positive := 32;
    constant addr_width      : positive := 25;
    constant row_bits        : positive := 12;
    constant col_bits        : positive := 11;
    constant s_word_width    : positive := 16;
    constant sec_bits        : positive := 1;

    -- redundancy specific constants
    constant red_bits        : positive := 21;
    constant off_width       : positive := 12;
    constant cam_depth       : positive := 24268;
```

```
-- redundancy system components

component match_array is
    generic (
        addr_width   : positive := addr_width;
        entries      : positive := cam_depth;
        row_width    : positive := row_bits;
        col_width    : positive := col_bits
    );
    port (
        addr         : in std_logic_vector(addr_width -1 downto 0);
        cas_n        : in std_logic;
        ras_n        : in std_logic;
        match        : out std_logic_vector(entries - 1 downto 0);
        red_en       : out std_logic;
        mem_en_n     : out std_logic
    );
end component match_array;

component flash_array is
    generic(
        addr_width   : positive := addr_width;
        red_bits     : positive := red_bits;
        sec_bits     : positive := sec_bits;
        entries      : positive := cam_depth
    );
    port (
        match   : in std_logic_vector(entries - 1 downto 0);
        section : out std_logic_vector(sec_bits - 1 downto 0);
        base    : out std_logic_vector(red_bits - 1 downto 0);
        mask    : out std_logic_vector(addr_width - 1 downto 0)
    );
end component flash_array;

component switch_22_12 is
    port (
        addr    : in  std_logic_vector(21 downto 0);
        dc_mask : in  std_logic_vector(21 downto 0);
        offset  : out std_logic_vector(11 downto 0)
    );
end component switch_22_12;

component nc_adder is
    generic (
        b_red   : positive := red_bits;
        b_off   : positive := off_width
    );
    port (
        base    : in  std_logic_vector(b_red - 1 downto 0);
        offset  : in  std_logic_vector(off_width - 1 downto 0);
        pointer : out  std_logic_vector(b_red - 1 downto 0)
    );
end component nc_adder;

component decode_9_512 is
    port (
        input   : in std_logic_vector (8 downto 0);
        enable  : in std_logic;
        output  : out std_logic_vector (511 downto 0)
    );
end component decode_9_512;

component mux_2_1 is
    generic (
        width   : positive := s_word_width
    );
    port (
        in_0    : in  std_logic_vector(width - 1 downto 0);
        in_1    : in  std_logic_vector(width - 1 downto 0);
        sel     : in  std_logic;
        output  : out std_logic_vector(width - 1 downto 0)
    );
end component mux_2_1;

component mux_4_1 is
    generic (
        width   : positive  := s_word_width
    );
    port (
        in_0    : in std_logic_vector(width - 1 downto 0);
        in_1    : in std_logic_vector(width - 1 downto 0);
        in_2    : in std_logic_vector(width - 1 downto 0);
        in_3    : in std_logic_vector(width - 1 downto 0);
        sel0    : in std_logic;
        sel1    : in std_logic;
        output  : out std_logic_vector(width - 1 downto 0)
    );
end component mux_4_1;
```

```
    component red_array is
        generic (
            row_bits : positive := 9;
            col_bits : positive := 9;
            word_width   : positive := s_word_width
        );
        port (
            rows     : in std_logic_vector(row_bits - 1 downto 0);
            cols     : in std_logic_vector(col_bits - 1 downto 0);
            d        : in std_logic_vector(word_width - 1 downto 0);
            q        : out std_logic_vector(word_width - 1 downto 0);
            rw_n     : in std_logic
        );
    end component red_array;

end package redun_pkg;
```

## E.3 Match Array

```
library ieee;
use ieee.std_logic_1164.all;
use std.textio.all;
use work.txt_util.all;

entity match_array is
    generic (
        addr_width   : positive := 25;
        entries      : positive := 1024;
        row_width    : positive := 12;
        col_width    : positive := 13
    );
    port (
        addr         : in std_logic_vector(addr_width - 1 downto 0);
        cas_n        : in std_logic;
        ras_n        : in std_logic;
        match        : out std_logic_vector(entries - 1 downto 0);
        red_en       : out std_logic;         -- enable red. mem output
        mem_en_n     : out std_logic      -- disable primary memory
    );
end entity match_array;

architecture struct of match_array is

signal match_row     : std_logic_vector(entries - 1 downto 0);
signal match_col     : std_logic_vector(entries - 1 downto 0);
signal row_addr      : std_logic_vector(row_width - 1 downto 0);
signal col_addr      : std_logic_vector(col_width - 1 downto 0);
signal row_addr_n    : std_logic_vector(row_width - 1 downto 0);
signal col_addr_n    : std_logic_vector(col_width - 1 downto 0);

subtype word_rs is std_logic_vector(0 to row_width - 1);
subtype word_cs is std_logic_vector(0 to col_width - 1);
type row_rom is array (0 to entries - 1) of word_rs;
type col_rom is array (0 to entries - 1) of word_cs;

signal row_carom, row_carom_n    : row_rom;
signal col_carom, col_carom_n    : col_rom;

begin

    load_caroms : process is
        variable l: line;
        variable sr: string(word_rs'range);
        variable sc: string(word_cs'range);

        variable index : natural;

        file load_file_rs  : TEXT open read_mode is "cam_row_true.dat";
        file load_file_rs_n  : TEXT open read_mode is "cam_row_not.dat";
        file load_file_cs  : TEXT open read_mode is "cam_col_true.dat";
        file load_file_cs_n  : TEXT open read_mode is "cam_col_not.dat";
    begin
        index := 0;
        while not endfile(load_file_rs) loop
            readline(load_file_rs, l);
            read(l,sr);
            row_carom(index) <= to_std_logic_vector(sr);
            readline(load_file_rs_n, l);
            read(l,sr);
            row_carom_n(index) <= to_std_logic_vector(sr);
            readline(load_file_cs, l);
            read(l,sc);
            col_carom(index) <= to_std_logic_vector(sr);
            readline(load_file_cs_n, l);
            read(l,sc);
            col_carom_n(index) <= to_std_logic_vector(sr);
            index := index + 1;
        end loop;
    end process load_caroms;

    row_addr <= addr(addr_width - 1 downto addr_width - col_width);
    col_addr <= addr(col_width - 1 downto 0);
    row_addr_n <= not row_addr;
    col_addr_n <= not col_addr;

    matchlines : for depth in 0 to entries - 1 generate
    begin

        row_precharge : process is
        begin
            if falling_edge(ras_n) then
                match_row(depth) <= '1';
            end if;
        end process row_precharge;

        row_match : for i in row_width - 1 downto 0 generate
        begin
            match_row(depth) <= '0' when (row_carom(depth)(i) = '0'
                                            and col_addr(i) = '1') or
                                        (row_carom_n(depth)(i) = '0'
                                            and col_addr_n(i) = '1') else
                                'Z';
        end generate row_match;

        col_precharge : process (cas_n) is
        begin
            if (falling_edge(cas_n) and match_row(depth) = '1') then
                match_col(depth) <= '1';
            end if;
        end process col_precharge;

        col_match : for j in col_width - 1 downto 0 generate
        begin
            match_col(depth) <= '0' when (col_carom(depth)(j) = '0'
                                            and col_addr(j) = '1') or
                                        (col_carom_n(depth)(j) = '0'
                                            and col_addr_n(j) = '1') else
                                'Z';
        end generate col_match;
    end generate matchlines;

end architecture struct;
```

## E.4 Flash (Data) Array

```
library ieee;
use ieee.std_logic_1164.all;
use std.textio.all;
library work;
use work.txt_util.all;

entity flash_array is
    generic (
        addr_width  : positive := 25;
        red_bits    : positive := 21;
        sec_bits    : positive := 1;
        entries     : positive := 1024
    );
    port (
        match   : in std_logic_vector(entries - 1 downto 0);
        section : out std_logic_vector(sec_bits - 1 downto 0);
        base    : out std_logic_vector(red_bits - 1 downto 0);
        mask    : out std_logic_vector(addr_width - 1 downto 0)
    );
end entity flash_array;

architecture struct of flash_array is
    subtype word_sec is std_logic_vector(sec_bits - 1 downto 0);
    subtype word_bse is std_logic_vector(red_bits - 1 downto 0);
    subtype word_msk is std_logic_vector(addr_width - 1 downto 0);
    type sec_rom is array (0 to entries - 1) of word_sec;
    type bse_rom is array (0 to entries - 1) of word_bse;
    type msk_rom is array (0 to entries - 1) of word_msk;

    signal sec : sec_rom;
    signal bse : bse_rom;
    signal msk : msk_rom;

begin

    load_roms : process is
        variable l: line;
        variable ss: string(word_sec'range);
        variable sb: string(word_bse'range);
        variable sm: string(word_msk'range);

        variable index : natural;
        file sec_dat : TEXT open read_mode is "sec.dat";
        file bse_dat : TEXT open read_mode is "bse.dat";
        file msk_dat : TEXT open read_mode is "msk.dat";
    begin
        index := 0;
        while not endfile(sec_dat) loop
            readline(sec_dat, l);
            read(l,ss);
            sec(index) <= to_std_logic_vector(ss);
            readline(bse_dat, l);
            read(l,sb);
            bse(index) <= to_std_logic_vector(sb);
            readline(msk_dat, l);
            read(l,sm);
            msk(index) <= to_std_logic_vector(sm);
            index := index + 1;
        end loop;
    end process load_roms;

    wordline : for depth in 0 to entries - 1 generate
    begin
        section <= sec(depth) when match(depth) = '1';
        base <= bse(depth) when match(depth) = '1';
        mask <= msk(depth) when match(depth) = '1';
    end generate wordline;

end architecture struct;
```

## E.5 Shifter (Ruby)

```
#!/usr/bin/env ruby

require "optparse"

a_bits = 25
o_bits = 12

ARGV.options { |opt|
    opt.banner = "Usage: #{$0} -a addr_bits -o offset_bits\n"

    opt.on("Options:")
    opt.on("-a", "--addr_bits NUM", Integer, "Number of address bits") {|a_bits|}
    opt.on("-o", "--offset_bits NUM", Integer, "Number of offset bits") {|o_bits|}
    opt.on_tail("-h", "--help", "Show this message") { puts opt; exit 0}

    opt.parse!
}

# print a header, entity and start of architecture

puts "-- Crossbar shifter automagically generated by gen_shift"
puts "-- input address and mask widths are #{a_bits}"
puts "-- output width is #{o_bits}"
puts ""
puts "library ieee;"
puts "use ieee.std_logic_1164.all;"
puts ""
puts "entity shifter is"
puts "\tgeneric ("
puts "\t\tb_addr\t: positive := #{a_bits};"
puts "\t\tb_off\t: positive := #{o_bits}"
puts "\t);"
puts "\t port ("
puts "\t\taddr\t: in\tstd_logic_vector(b_addr - 1 downto 0);"
puts "\t\tdc_mask\t: in\tstd_logic_vector(b_addr - 1 downto 0);"
puts "\t\toffset\t: out\tstd_logic_vector(b_off - 1 downto 0)"
puts "\t);"
puts "end entity shifter;"
puts ""
puts "architecture rtl of shifter is"
puts "begin"
puts ""

# create the crossbar

0.upto(o_bits - 1) {|vline|
    print "\toffset(#{vline}) <= "
    vline.upto(a_bits - 1) {|hline|
        print "addr(#{hline}) when ("
        m = Array.new(hline, -1)
        m[hline] = 1
        0.upto(vline-1) {|t| m[t] = 1}
        m.each_index {|h|
            print "dc_mask(#{h}) = "
            if m[h] == 1
                print "'1'"
            else
                print "'0'"
            end
            if (m.size - 1) == h
                if hline == (a_bits - 1)
                    puts ");"
                else
                    print ") else\n\t\t\t\t"
                end
            else
                print " and "
            end
        }
    }
    puts ""
}

puts "end architecture rtl;"
```

## E.6 OR (No-carry Adder)

```
library ieee;
use ieee.std_logic_1164.all;

entity nc_adder is
    generic (
        b_red    : in positive := 21;
        b_off    : in positive := 12
    );
    port (
        base     : in std_logic_vector(b_red - 1 downto 0);
        offset   : in std_logic_vector(b_off - 1 downto 0);
        pointer  : out std_logic_vector(b_red - 1 downto 0)
    );
end entity nc_adder;

architecture rtl of nc_adder is
begin

    adder : for bit_index in 0 to b_off - 1 generate
    begin
        pointer(bit_index) <= base(bit_index) or offset(bit_index);
    end generate adder;

    pointer(b_red - 1 downto b_off) <= base(b_red - 1 downto b_off);

end architecture rtl;
```

## E.7 Redundant DRAM decoder (Ruby)

```
#!/usr/bin/env ruby

require "optparse"

a_bits = 25

ARGV.options {|opt|
    opt.banner = "Usage: #{$0} -a addr_bits\n"

    opt.on("Options:")
    opt.on("-a", "--addr_bits NUM", Integer, "Number of address bits") {|a_bits|}
    opt.on_tail("-h", "--help", "Show this message") { puts opt; exit 0 }

    opt.parse!
}

# print a header, entity and start of architecture

o_lines = 2 ** a_bits

puts "-- Address decoder automagically generated by gendecode"
puts "-- input address width is #{a_bits} bits"
puts "-- there are #{o_lines} output lines"
puts ""
puts "library ieee;"
puts "use ieee.std_logic_1164.all;"
puts ""
puts "entity decoder is"
puts "\t port ("
puts "\t\t input\t: in\tstd_logic_vector(#{a_bits - 1} downto 0);"
puts "\t\t enable\t: in\tstd_logic;"
puts "\t\t output\t: out\tstd_logic_vector(#{o_lines - 1} downto 0)"
puts "\t );"
puts "end entity decoder;"
puts ""
puts "architecture rtl of decoder is"
puts ""
puts "\t signal in_t : std_logic_vector(#{a_bits} downto 0);"
puts ""
puts "begin"
puts ""
puts "\tin_t <= input & enable;"
puts ""
puts "\twith in_t select output <="

out_a = Array.new(o_lines);
in_a = Array.new(a_bits + 1);

0.upto(o_lines - 1) {|int|
    in_a.clear
    a_bits.times {in_a << 0}
    out_a.clear
    o_lines.times {out_a << 0}

    out_a[int] = 1
    out_a.reverse!

    i = 0
    while 0
        if int == 0
            break
        elsif int == 1
            in_a[i] = 1
            break
        end

        if int.modulo(2) == 0
            int /= 2
        else
            in_a[i] = 1
            int = (int - 1) / 2
        end

        i += 1
    end

    in_a.reverse!
    in_a << 0

    print "\t\t \""
    print out_a
    print "\" when \""
    print in_a
    puts "\","
}

out_a.collect{|bit| bit = 0}
print "\t\t \""
print out_a
puts "\" when others;"
puts ""
puts "end architecture rtl;"
```

## E.8 Multiplexer (2-1)

```
library ieee;
use ieee.std_logic_1164.all;

entity mux_2_1 is
    generic (
        width    : positive := 32
    );
    port (
        in_0     : in   std_logic_vector(width - 1 downto 0);
        in_1     : in   std_logic_vector(width - 1 downto 0);
        sel      : in   std_logic;
        output   : out  std_logic_vector(width - 1 downto 0)
    );
end entity mux_2_1;

architecture rtl of mux_2_1 is
begin
    with sel select
        output <= in_0 when '0',
                  in_1 when others;
```

```
end architecture rtl;
```

## E.9 Multiplexer (4-1)

```
library ieee;
use ieee.std_logic_1164.all;

entity mux_4_1 is
    generic (
        width    : positive := 32
    );
    port (
        in_0     : in   std_logic_vector(width - 1 downto 0);
        in_1     : in   std_logic_vector(width - 1 downto 0);
        in_2     : in   std_logic_vector(width - 1 downto 0);
        in_3     : in   std_logic_vector(width - 1 downto 0);
        sel0     : in   std_logic;
        sel1     : in   std_logic;
        output   : out  std_logic_vector(width - 1 downto 0)
    );
end entity mux_4_1;

architecture rtl of mux_4_1 is
    signal lower, upper : std_logic_vector(width - 1 downto 0);
begin
    with sel0 select
        lower <= in_0 when '0',
                 in_1 when others;

    with sel0 select
        lower <= in_2 when '0',
                 in_3 when others;

    with sel1 select
        output <= lower when '0',
                  upper when others;

end architecture rtl;
```

## E.10 Redundant Data Array

```
library ieee;
use ieee.std_logic_1164.all;

entity red_array is
    generic (
        row_bits     : positive := 11;
        col_bits     : positive := 10;
        word_width   : positive := 16
    );
    port (
        rows     : in std_logic_vector(2**row_bits - 1 downto 0);
        cols     : in std_logic_vector(2**col_bits - 1 downto 0);
        d        : in std_logic_vector(word_width -1 downto 0);
        q        : out std_logic_vector(word_width -1 downto 0);
        rw_n     : in std_logic
    );
end entity red_array;

architecture behaviour of red_array is
    subtype word is std_logic_vector(0 to word_width - 1);
    type d_array is array (0 to 2**row_bits - 1, 0 to 2** col_bits - 1) of word;

    signal dram : d_array;

begin

    wordline : for depth in 0 to 2**row_bits - 1 generate
    begin
        bitline : for column in 0 to 2**col_bits - 1 generate
        begin
            q <= dram(depth,column) when rows(depth) = '1'
                and cols(column) = '1' and rw_n = '0';

            dram(depth,column) <= d when rows(depth) = '1'
                and cols(column) = '1' and rw_n = '1';

        end generate bitline;
    end generate wordline;

end architecture behaviour;
```

# Appendix F

# Testbench VHDL Listings

## F.1 Testbench

```
library ieee;
use ieee.std_logic_1164.all;

package test_pkg is
    component lfsr_generic is
        generic (Width: positive := 4);      -- length of sequence
        port (
            clock    : in std_logic;
            reset    : in std_logic;         -- active low
            load     : in std_logic;         -- active high
            enable   : in std_logic;         -- active high
            parallel_in  : in std_logic_vector(Width - 1 downto 0);
            parallel_out    : out std_logic_vector(Width - 1 downto 0);
            serial_out   : out std_logic
        );
        end component lfsr_generic;

end package test_pkg;

library ieee;
use ieee.std_logic_1164.all;

package assoc_16_pkg is
    component assoc_i_16 is
       port (  gclk        : in std_logic;
            a           : in std_logic_vector (21 downto 0);
            d           : in std_logic_vector (3 downto 0);
            qp          : in std_logic_vector (3 downto 0);
            q           : out std_logic_vector (3 downto 0);
            cas_n       : in std_logic;
            ras_n       : in std_logic;
            cs_n        : in std_logic;
            rw_n        : in std_logic;
            mem_en_n    : inout std_logic
        );
    end component assoc_i_16;

end package assoc_16_pkg;


library ieee;
use ieee.std_logic_1164.all;
use std.textio.all;
library work;
use work.test_pkg.all;
use work.assoc_16_pkg.all;
use work.txt_util.all;

entity assoc_16_test is
end assoc_16_test;

architecture mixed of assoc_16_test is

    constant T_halfclock : time := 3.75 ns; --(corresponds to 266 MHz DDR)
    constant T_prop : time := 0.5 ns;        -- avoid hold time violations
    constant T_latch : time := 1 ns;

    signal clock, reset : std_logic;
    signal seed_addr : std_logic_vector(21 downto 0);
    signal seed_data : std_logic_vector(3 downto 0);
signal serial_addr, serial_data : std_logic;
signal addr_internal, addr_delay : std_logic_vector(21 downto 0);
signal data_internal, data_delay, qdata : std_logic_vector(3 downto 0);
signal main_mem_q : std_logic_vector(3 downto 0);
signal load_ga, load_gd, req_addr, req_data : std_logic;
signal cas_n, ras_n, cs_n, rw_n : std_logic;
signal match : std_logic;
signal start : std_logic;

type state_type is (s_reset, s_0, s_1, s_2);
signal state, next_state : state_type;


begin

seed_addr <= "0101010111110101000100";
seed_data <= "1001";

main_mem_q <= "UUUU";

start <= '0';
start <= '1' after 10 ns;
start <= '0' after 11 ns;

gen_addr : component lfsr_generic
    generic map (Width => 22)
    port map (
        clock   => clock,
        reset   => reset,
        load    => load_ga,
        enable  => req_addr,
        parallel_in     => seed_addr,
        parallel_out    => addr_internal,
        serial_out      => serial_addr
    );

gen_data : component lfsr_generic
    generic map (Width => 4)
    port map (
        clock   => clock,
        reset   => reset,
        load    => load_gd,
        enable  => req_data,
        parallel_in     => seed_data,
        parallel_out    => data_internal,
        serial_out      => serial_data
    );

redun : component assoc_i_16
    port map (
        gclk    => clock,
        a       => addr_delay,
        d       => data_delay,
        qp      => main_mem_q,
        q       => qdata,
        cas_n   => cas_n,
        ras_n   => ras_n,
        cs_n    => cs_n,
        rw_n    => rw_n,
        mem_en_n => match
    );

clock_gen   : process
begin
    clock <= '1';
    wait for T_halfclock;
    clock <= '0';
    wait for T_halfclock;
end process clock_gen;

state_logic : process (state)
    variable count : integer;
    file write_out : TEXT open write_mode is "write.dat";
    file read_out : TEXT open write_mode is "read.dat";
    variable read_line, write_line : line;

begin
    case state is
        when s_reset =>
            ras_n <= '1';
            cas_n <= '1';
            req_data <= '0';
            load_gd <= '1';
            load_ga <= '1';
            rw_n <= '1';
            count := 0;
            next_state <= s_reset;
        when s_0 =>
            load_gd <= '0';
```

```
                load_ga <= '0';
                req_data <= '1';
                cas_n <= '1';
                if count < 262144 then
                    addr_delay <= addr_internal after T_prop;
                    data_delay <= data_internal after T_prop;
                else
                    rw_n <= '0';
                end if;
                ras_n <= '0' after T_latch;
                next_state <= s_1;
            when s_1 =>
                req_data <= '0';
                ras_n <= '1';
                cas_n <= '0' after T_latch;
                next_state <= s_2;
            when s_2 =>
                if count = 262143 then
                    load_gd <= '1';
                    load_ga <= '1';
                end if;
                if count < 262144 then
                    write ( write_line , str(addr_delay));
                    write ( write_line , string'("_"));
                    write ( write_line , str(data_delay));
                    writeline ( write_out , write_line);
                else
                    write ( read_line , str(addr_delay));
                    write ( read_line , string'("_"));
                    write ( read_line , str(match));
                    write ( read_line , string'("_"));
                    write ( read_line , str(qdata));
                end if;

                if count = 524287 then
                    next_state <= s_reset;
                else
                    count := count + 1;
                    next_state <= s_0;
                end if;
        end case;
    end process state_logic;

    state_register  : process (clock, reset)
    begin
        if reset = '1' then
            state <= s_reset;
        elsif rising_edge(clock) then
            if start = '1' then
                state <= s_0;
            else
                state <= next_state;
            end if;
        end if;
    end process state_register;

end architecture mixed;
```

## F.2 lfsr_generic

```
-----------------------------------------------------------
-- lfsr_generic.vhd
--
-- Revised 2001/02/09
--
-- Authors:   Raymond Sung
-- Date:      Oct 10, 2000
-- Course:    EE552
-- Modified:  John Koob & Ray Sung
-- Source:    EE 552 Student Application Note
-- Desc:
--
-- Autononmous Linear Feedback Shift Register
--
-- Description: Implementation of a Linear Feedback Shift Register
--              that generates a sequence of 2^N-1 non-repeating pseudo-random numbers.
--
--              The Bit-Width or Length of the Pseudo Random Sequence
--              can be instantiated with the generic "Width."
--
--              The length of the pseudorandom sequence can be anywhere from 2 to 16.
--              (2^2 = 4 numbers in sequence and 2^16 numbers sequence !!
--
--              The seed input variable allows one to start the pseudo-random sequence at
--              a certain position in the pseudo-random sequence.
--
```

## F.2 lfsr_generic

```
--              Uses:     Digital Signal Processing
--                        Wireless Communications
--                        Encryption/Decryption
--                        Direct Sequence Spread Spectrum
--                        Pseudo-Random Number Generation
--                        Scrambler/Descrambler
--                        Built-In Self Test
--
-- References:  "HDL Chip Design" - Douglas J. Smith
--              "Linear Feedback Shift Register MegaFunction" - Nova Engineering


library IEEE;
use IEEE.STD_Logic_1164.all;

entity LFSR_GENERIC is

    generic(Width: positive := 4);      -- length of pseudo-random sequence
    port    (   clock: in std_logic;
                reset: in std_logic;      -- active high reset
                load: in std_logic;       -- active high load (assert this to use as regular reg)
                enable: in std_logic;     -- active high enable
                parallel_in: in std_logic_vector(Width-1 downto 0); -- parallel seed input
                parallel_out: out std_logic_vector(Width-1 downto 0); -- parallel data out
                serial_out: out std_logic     -- serial data out (From last shift register)
            );

end entity LFSR_GENERIC;

architecture RTL of LFSR_GENERIC is

type TapsArrayType is array(2 to 16) of std_logic_vector(15 downto 0);
signal Taps: std_logic_vector(Width-1 downto 0);

begin

    LFSR: process (clock)

    -- internal registers and signals
    variable LFSR_Reg: std_logic_vector(Width-1 downto 0);
    variable XOR_Out: std_logic_vector(Width-2 downto 0);
    variable OR_Chain: std_logic_vector(Width-2 downto 0);
    variable NOR_All: std_logic;
    variable Feedback: std_logic;
    variable TapsArray: TapsArrayType;

    begin

        Taps <= TapsArray(Width)(Width-1 downto 0); -- get tap points from lookup table

        if reset='1' then

            LFSR_Reg := (others=>'1');
            OR_Chain := (others=>'0');
            Feedback := '0';
            NOR_All := '0';
            -- Look-Up Table for Tap points to insert XOR gates as feedback into D-FF
            -- outputs.  Taps are designed so that 2^N-1 (N=Width of Register) numbers
            -- are cycled through before the sequence is repeated

            TapsArray(2) :=         "0000000000000011";
            TapsArray(3) :=         "0000000000000101";
            TapsArray(4) :=         "0000000000001001";
            TapsArray(5) :=         "0000000000010010";
            TapsArray(6) :=         "0000000000100001";
            TapsArray(7) :=         "0000000001000001";
            TapsArray(8) :=         "0000000010001110";
            TapsArray(9)      :=    "0000000100001000";
            TapsArray(10)     :=    "0000001000000100";
            TapsArray(11)     :=    "0000010000000010";
            TapsArray(12)     :=    "0000100000101001";
            TapsArray(13)     :=    "0001000000001101";
            TapsArray(14)     :=    "0010000000010101";
            TapsArray(15)     :=    "0100000000000001";
            TapsArray(16)     :=    "1000000000010110";


        elsif rising_edge(clock) then
            -- load signal asserted, use seed value on port to determine where
            -- to start in the pseudo-random sequence

            if enable = '1' then

                if load = '1' then

                    LFSR_Reg := parallel_in;

                else
```

```
          Feedback := LFSR_Reg(Width-1) xor NOR_All;
          -- look at old value;

          for N in Width-1 downto 1 loop

            if (Taps(N-1)='1') then

              XOR_Out(N-1) := LFSR_Reg(N-1) xor parallel_in(N) xor Feedback;

            else

              XOR_Out(N-1) := LFSR_Reg(N-1) xor parallel_in(N);

            end if;

            LFSR_Reg(N) := XOR_Out(N-1);

          end loop;


          OR_Chain(0) := XOR_Out(0) or XOR_Out(1);

          for N in 2 to Width-2 loop
            OR_Chain(N-1):= OR_Chain(N-2) or XOR_Out(N);
          end loop;

          NOR_All := not OR_Chain(Width-2);

          LFSR_Reg(0) := parallel_in(0) xor Feedback;

        end if;

      end if;

    end if;

      parallel_out <= LFSR_Reg;     -- parallel data out
      serial_out <= LFSR_Reg(Width-1);     -- serial data out

    end process;

end RTL;
```

### *F.2 lfsr_generic*

# Appendix G

# SKILL Code

All functions beginning with tf_ are part of the tf functions written by Tyler Brandon at the University of Alberta VLSI Design Lab.

## G.1 Row match array

```
procedure( ai_match_row_array(filename)
    let( (cv inst_pfet inst_cN inst_cX inst_c0 inst_c1 inst_tnet
        inst_noConn inPort word bits x y i)

    cv = tf_getWinCV()
    inst_pfet = dbOpenCellViewByType("cmosp18" "pfet" "symbol")
    inst_inv = dbOpenCellViewByType("vst_n18_sc_tsm_c4" "INVD1" "symbol")
    inst_cN=dbOpenCellViewByType("assoc_i" "carom_off" "symbol")
    inst_cX=dbOpenCellViewByType("assoc_i" "carom_dc" "symbol")
    inst_c0=dbOpenCellViewByType("assoc_i" "carom_0" "symbol")
    inst_c1=dbOpenCellViewByType("assoc_i" "carom_1" "symbol")
    inst_tnet = dbOpenCellViewByType("assoc_i" "t_model" "symbol")
    inst_noConn = dbOpenCellViewByType("basic" "noConn" "symbol")

    inPort = infile(filename)
    when( inPort
        y = 0
        while( fscanf( inPort "%s" word )
            bits = strlen( word )
            x = 0
            dbCreateParamInst(cv inst_pfet nil -0.5:3*y+0.375 "R0"
                        1 list(list("w" "float" 800n)))
            tf_schCreateWire(cv list(-5:48*y+6 -4:48*y+6 -4:48*y+12)
                            "vdd!" -5:48*y+12 "R270")
            tf_schCreateWire(cv list(-8:48*y+6 -10:48*y+6))
            for(i 1 bits
                case( substring(word i 1)
                    ("X" dbCreateInst(cv inst_cX nil x*3:y*3 "R0" 1))
                    ("0" dbCreateInst(cv inst_c0 nil x*3:y*3 "R0" 1))
                    ("1" dbCreateInst(cv inst_c1 nil x*3:y*3 "R0" 1))
                    (t dbCreateParamInst(cv inst_cN nil x*3:y*3 "R0" 1
                        list(list("mult" "float" 1))))
                )
                dbCreateParamInst(cv inst_tnet nil x*3+1.125:y*3+0.25 "R0" 1
                        list(list("RES" "float" 1.41)
                            list("CAP" "float" 1.31f)))
                tf_schCreateWire(cv list(48*x-4:48*y+3 48*x+23:48*y+3))

                dbCreateParamInst(cv inst_tnet nil x*3+0.4375:y*3-0.25 "R270" 1
                        list(list("RES" "float" 0.366)
                            list("CAP" "float" 0.265f)))
                dbCreateParamInst(cv inst_tnet nil x*3+1.125:y*3-0.25 "R270" 1
                        list(list("RES" "float" 0.366)
                            list("CAP" "float" 0.265f)))

                if( y == 0 then
                    tf_schCreateWire(cv list(48*x+6:48*y+18
                        48*x+6:48*y-9) strcat("cl_n<" concat( x )
                        ">") 48*x+5:48*y+18 "R90")
                    tf_schCreateWire(cv list(48*x+17:48*y+18
                        48*x+17:48*y-9) strcat("cl<" concat( x )
                        ">") 48*x+16:48*y+18 "R90")
                else
                    tf_schCreateWire(cv list(48*x+6:48*y+18
                        48*x+6:48*y-9))
                    tf_schCreateWire(cv list(48*x+17:48*y+18
                        48*x+17:48*y-9))
                )

                x = x + 1
            )

            dbCreateParamInst(cv inst_pfet nil x*3+0.5:3*y+0.375 "MY"
                            1 list(list("w" "float" 250n)
                                    list("l" "float" 360n)))
            dbCreateInst(cv inst_inv nil x*3+0.75:3*y+0.1875 "MX" 1)
            tf_schCreateWire(cv list(48*x-4:48*y+3 48*x+12:48*y+3)
                            strcat("ml<" concat( -y ) ">")
                            48*x:48*y+2 "R0")
            tf_schCreateWire(cv list(48*x+35:48*y+3 48*x+40:48*y+3
                            48*x+40:48*y+8 48*x+8:48*y+8
                            48*x+8:48*y+6)
                            strcat("ml_n<" concat( -y ) ">")
                            48*x+40:48*y+3 "R0")

            tf_schCreateWire(cv list(48*x+5:48*y+6 48*x+4:48*y+6
                            48*x+4:48*y+9 48*x+4:48*y+12)
                            "vdd!" 48*x+4:48*y+12 "R0")
            tf_schCreateWire(cv list(48*x+23:48*y-5 48*x+23:48*y-7)
                            "vss!" 48*x+22:48*y-5 "R270")
            tf_schCreateWire(cv list(48*x+25:48*y-5 48*x+25:48*y-7)
                            "vdd!" 48*x+24:48*y-5 "R270")

            dbCreateParamInst(cv inst_tnet nil -0.5625:y*3-0.25 "R270" 1
                    list(list("RES" "float" 0.366) list("CAP" "float" 0.265f)))
            tf_schCreateWire(cv list(-10:48*y+18 -10:48*y-9))


            y = y - 1
        )
        close( inPort )
    )



    for(i 0 x-1                    ; the loads for placing up to 1024
        dbCreateParamInst(cv inst_tnet nil i*3+0.4375:y*3 "R270" 1
                list(list("RES" "float" 372) list("CAP" "float" 0.269p)))
        dbCreateParamInst(cv inst_tnet nil i*3+1.125:y*3 "R270" 1
                list(list("RES" "float" 372) list("CAP" "float" 0.269p)))
        dbCreateParamInst(cv inst_cN nil i*3:y*3-2.1875 "R0" 1
                list(list("mult" "float" 1016)))
        dbCreateInst(cv inst_noConn nil i*3:y*3-2 "R0" 1)
        tf_schCreateWire(cv list(48*i+6:48*y+18 48*i+6:48*y-5))
        tf_schCreateWire(cv list(48*i+17:48*y+18 48*i+17:48*y-5))
    )
    dbCreateParamInst(cv inst_pfet nil -0.5:3*y-2 "R0" 1
                list(list("w" "float" 800n) list("multiplier" "float" 1016)))
    tf_schCreateWire(cv list(-5:48*y-32 -4:48*y-32 -4:48*y-26)
                "vdd!" -5:48*y-26 "R270")
    dbCreateParamInst(cv inst_tnet nil -0.5625:y*3 "R270" 1
                list(list("RES" "float" 372) list("CAP" "float" 0.269p)))
    tf_schCreateWire(cv list(-10:48*y+18 -10:48*y-5))
    tf_schCreateWirc(cv list(-10:48*y-26 -10:48*y-32 -8:48*y-32))

    dbCreateInst(cv inst_noConn nil -0.25:y*3-2.1875 "R0" 1)


    tf_schCreatePin(cv strcat("ml_n<" concat( -y-1 ) ":0>") "output"
                    (x+1)*48:y*10)
    tf_schCreatePin(cv strcat("ml<" concat( -y-1 ) ":0>") "output"
                    (x+1)*48:(y-1)*10)
    tf_schCreatePin(cv strcat("cl_n<" concat( x-1 ) ":0>") "input"
                    x*12-8:25, "R270")
    tf_schCreatePin(cv strcat("cl<" concat( x-1 ) ":0>") "input"
                    x*12+8:25, "R270")
    tf_schCreatePin(cv "precharge_n" "input" -10:18 "R270")

;   schHiCheckAndSave()
))
```

## G.2 Column match array

```
procedure( ai_match_col_array(filename)
    let( (cv inst_pfet inst_nfet inst_inv inst_or inst_cN inst_cX
        inst_c0 inst_c1 inst_noConn inst_tnet inPort word bits x y i)
```

```
;   cv = dbOpenCellViewByType("assoc_i" "match_col_array" "schematic" nil "w")
    cv = tf_getWinCV()
    inst_pfet = dbOpenCellViewByType("cmosp18" "pfet" "symbol")
    inst_nfet = dbOpenCellViewByType("cmosp18" "nfet" "symbol")
    inst_inv = dbOpenCellViewByType("vst_n18_sc_tsm_c4" "INVD1" "symbol")
    inst_or = dbOpenCellViewByType("vst_n18_sc_tsm_c4" "OR2D4" "symbol")
    inst_cN=dbOpenCellViewByType("assoc_i" "carom_off" "symbol")
    inst_cX=dbOpenCellViewByType("assoc_i" "carom_dc" "symbol")
    inst_c0=dbOpenCellViewByType("assoc_i" "carom_0" "symbol")
    inst_c1=dbOpenCellViewByType("assoc_i" "carom_1" "symbol")
    inst_tnet = dbOpenCellViewByType("assoc_i" "t_model" "symbol")
    inst_noConn = dbOpenCellViewByType("basic" "noConn" "symbol")

    inPort = infile(filename)
    when( inPort
        y = 0
        while( fscanf( inPort "%s" word )
            bits = strlen( word )
            x = 0
            dbCreateInst(cv inst_or nil -2.75:3*y+0.4375 "R0")
            dbCreateParamInst(cv inst_pfet nil -0.4375:3*y+0.4375 "MYR90"
                        1 list(list("w" "float" 800n))) ; pre_charge
            dbCreateInst(cv inst_nfet nil -0.3125:3*y "R0") ; pull down
            tf_schCreateWire(cv list(-20:48*y+7 -7:48*y+7)) ; OR - pull-up
            tf_schCreateWire(cv list(-49:48*y+7 -44:48*y+7)
                strcat("match_row_n<" concat(-y) ">") -51:48*y+8 "R0")
            tf_schCreateWire(cv list(-46:48*y+5 -44:48*y+5))     ; cp - OR
            tf_schCreateWire(cv list(-7:48*y+4 -7:48*y+3 -15:48*y+3)
                    "vdd!" -15:48*y+4)  ; PC pull-up vdd!
            tf_schCreateWire(cv list(-36:48*y+15 -33:48*y+15) "vss!"
                    -37:48*y+15 "R0")    ; vss! for OR
            tf_schCreateWire(cv list(-31:48*y+15 -28:48*y+15) "vdd!"
                    -30:48*y+15 "R0")    ; vdd! for OR
            tf_schCreateWire(cv list(-2:48*y -1:48*y -1:48*y-3 -1:48*y-6)
                    "vss!" -2:48*y-4 "R270")     ; vss! for pull-down
            tf_schCreateWire(cv list(-5:48*y -12:48*y)  ; pull-down ctrl
                    strcat("match_row_n<" concat(-y) ">") -18:48*y+1 "R0")

            for(i 1 bits
                case( substring(word i 1)
                    ("X" dbCreateInst(cv inst_cX nil x*3:y*3 "R0" 1))
                    ("0" dbCreateInst(cv inst_c0 nil x*3:y*3 "R0" 1))
                    ("1" dbCreateInst(cv inst_c1 nil x*3:y*3 "R0" 1))
                    (t dbCreateParamInst(cv inst_cN nil x*3:y*3 "R0" 1
                        list(list("mult" "float" 1))))
                )

                dbCreateParamInst(cv inst_tnet nil x*3+1.125:y*3+0.25 "R0" 1
                        list(list("RES" "float" 1.41)
                            list("CAP" "float" 1.31f)))
                tf_schCreateWire(cv list(48*x-4:48*y+3 48*x+23:48*y+3))

                dbCreateParamInst(cv inst_tnet nil x*3+0.4375:y*3-0.25 "R270" 1
                        list(list("RES" "float" 0.366)
                            list("CAP" "float" 0.265f)))
                dbCreateParamInst(cv inst_tnet nil x*3+1.125:y*3-0.25 "R270" 1
                        list(list("RES" "float" 0.366)
                            list("CAP" "float" 0.265f)))

                if( y == 0 then
                    tf_schCreateWire(cv list(48*x+6:48*y+18
                        48*x+6:48*y-9) strcat("cl_n<" concat( x ) ">")
                        48*x+5:48*y+18 "R90")
                    tf_schCreateWire(cv list(48*x+17:48*y+18
                        48*x+17:48*y-9) strcat("cl<" concat( x ) ">")
                        48*x+16:48*y+18 "R90")
                else
                    tf_schCreateWire(cv list(48*x+6:48*y+18
                        48*x+6:48*y-9))
                    tf_schCreateWire(cv list(48*x+17:48*y+18
                        48*x+17:48*y-9))
                )
                x = x + 1
            )


            dbCreateParamInst(cv inst_pfet nil x*3+0.5:3*y+0.375 "MY"
                            1 list(list("w" "float" 250n)
                                    list("l" "float" 360n)))
            dbCreateInst(cv inst_inv nil x*3+0.75:3*y+0.1875 "MX" 1)
            tf_schCreateWire(cv list(48*x-4:48*y+3 48*x+12:48*y+3)
                strcat("ml<" concat( -y ) ">") 48*x:48*y+2 "R0")
            tf_schCreateWire(cv list(48*x+35:48*y+3 48*x+40:48*y+3
                    48*x+40:48*y+8 48*x+8:48*y+8 48*x+8:48*y+6)
                    strcat("ml_n<" concat( -y ) ">") 48*x+40:48*y+3 "R0")

            tf_schCreateWire(cv list(48*x+5:48*y+6 48*x+4:48*y+6
                            48*x+4:48*y+9 48*x+4:48*y+12)
                            "vdd!" 48*x+4:48*y+12 "R0")
            tf_schCreateWire(cv list(48*x+23:48*y-5 48*x+23:48*y-7)
                            "vss!" 48*x+22:48*y-5 "R270")
            tf_schCreateWire(cv list(48*x+25:48*y-5 48*x+25:48*y-7)
                            "vdd!" 48*x+24:48*y-5 "R270")

            dbCreateParamInst(cv inst_tnet nil -2.8125:y*3-0.25 "R270" 1
                list(list("RES" "float" 0.366) list("CAP" "float" 0.265f)))
            tf_schCreateWire(cv list(-46:48*y+18 -46:48*y-9))

            y = y - 1
        )
        close( inPort )
    )

    for(i 0 x-1                    ; the loads for placing up to 1024
        dbCreateParamInst(cv inst_tnet nil i*3+0.4375:y*3 "R270" 1
                list(list("RES" "float" 372) list("CAP" "float" 0.269p)))
        dbCreateParamInst(cv inst_tnet nil i*3+1.125:y*3 "R270" 1
                list(list("RES" "float" 372) list("CAP" "float" 0.269p)))
        dbCreateParamInst(cv inst_cN nil i*3:y*3-2.1875 "R0" 1
                list(list("mult" "float" 1016)))
        dbCreateInst(cv inst_noConn nil i*3:y*3-2 "R0" 1)
        tf_schCreateWire(cv list(48*i+6:48*y+18 48*i+6:48*y-5))
        tf_schCreateWire(cv list(48*i+17:48*y+18 48*i+17:48*y-5))
    )
    dbCreateParamInst(cv inst_tnet nil -2.8125:y*3 "R270" 1
                list(list("RES" "float" 372) list("CAP" "float" 5.43p)))
    tf_schCreateWire(cv list(-46:48*y+18 -46:48*y-5))
    dbCreateInst(cv inst_noConn nil -2.875:3*y-1.625 "R0" 1)


    tf_schCreatePin(cv "precharge_n" "input" -46:18 "R270")
    tf_schCreatePin(cv strcat("ml<" concat( -y-1 ) ":0>") "output"
                    (x+1)*48:y*10)
    tf_schCreatePin(cv strcat("match_row_n<" concat( -y-1 ) ":0>")
                    "input" -60:y*7)
    tf_schCreatePin(cv strcat("cl_n<" concat( x-1 ) ":0>") "input"
                    x*12-8:25, "R270")
    tf_schCreatePin(cv strcat("cl<" concat( x-1 ) ":0>") "input"
                    x*12+8:25, "R270")

;   schHiCheckAndSave()
))
```

## G.3 Associated data array

```
procedure( ai_data_array(filename )
    let( (cv inst_r0 inst_r1 inst_pfet inst_tnet inst_noConn inPort
        word bits x y c i j)

    cv = tf_getWinCV()
    inst_r0=dbOpenCellViewByType("assoc_i" "rom_0" "symbol")
    inst_r1=dbOpenCellViewByType("assoc_i" "rom_1" "symbol")
    inst_pfet=dbOpenCellViewByType("cmosp18" "pfet" "symbol")
    inst_tnet = dbOpenCellViewByType("assoc_i" "t_model" "symbol")
    inst_noConn = dbOpenCellViewByType("basic" "noConn" "symbol")
    inst_nand = dbOpenCellViewByType("vst_n18_sc_tsm_c4" "NAN2D2" "symbol")
    inst_mux = dbOpenCellViewByType("vst_n18_sc_tsm_c4" "MUX4D2" "symbol")

    inPort = infile(filename)
    when( inPort
        y = 0
        while( fscanf( inPort "%s" word )
            bits = strlen( word )
            x = 0
            for(i 1 bits
                c = substring(word i 1)
                if( strcmp(c "1") == 0 then
                    dbCreateInst(cv inst_r1 nil 3*x:3*y "R0" 1)
                else
                    dbCreateInst(cv inst_r0 nil 3*x:3*y "R0" 1)
                )

                dbCreateParamInst(cv inst_tnet nil x*3+1.125:y*3+0.0625 "R0" 1
                    list(list("RES" "float" 0.705) list("CAP" "float" 0.565f)))
                if( x == 0 then
                    tf_schCreateWire(cv list(48*x-4:48*y 48*x+23:48*y)
                        strcat("wl<" concat( -y ) ">") 48*x-4:48*y "R0")
                else
                    tf_schCreateWire(cv list(48*x-4:48*y 48*x+23:48*y))
                )
                dbCreateParamInst(cv inst_tnet nil x*3+0.75:y*3-0.25 "R270" 1
                    list(list("RES" "float" 0.366) list("CAP" "float" 0.265f)))
            if( y == 0 then                       ; precharge circuity
                dbCreateParamInst(cv inst_pfet nil 3*x+0.4375:1.32 "R0" 1
                    list(list("w" "float" 1u))) ; pull-up
                tf_schCreateWire(cv list(48*x+10:21 48*x+11:21 48*x+11:27)
                    "vdd!" 48*x+10:27 "R270") ; pull down vdd
                tf_schCreateWire(cv list(48*x+7:21 48*x+5:21 48*x+5:29))
                dbCreateParamInst(cv inst_tnet nil x*3+1.125:1.875
                    "R0" 1 list(list("RES" "float" 0.705)
                                list("CAP" "float" 0.565f)))
                if( x == 0 then
                    tf_schCreateWire(cv list(48*x-4:29 48*x+23:29)
                        "precharge_data_n<0>" 48*x-4:29 "R0")
                else
                    tf_schCreateWire(cv list(48*x-4:29 48*x+23:29))
                )
            )

            tf_schCreateWire(cv list(48*x+11:48*y+18 48*x+11:48*y-9))

            x = x + 1;
        )
        dbCreateInst(cv inst_noConn nil 3*(x-1)+2.75:y*3 "R0")
        y = y - 1
    )
    close( inPort )
)

dbCreateInst(cv inst_noConn nil 3*(x-1)+2.75:1.8125 "R0")

for(i 0 x-1          ; the loads for placing up to 256
    dbCreateParamInst(cv inst_tnet nil i*3+0.75:y*3 "R270" 1
        list(list("RES" "float" 90.8) list("CAP" "float" 1.32p)))
    tf_schCreateWire(cv list(48*i+11:48*y+18 48*i+11:48*y-5))
    tf_schCreateWire(cv list(48*i+11:48*y-26 48*i+11:48*y-32)
        strcat("bl0<" concat( i ) ">") 48*i+10:48*y-30 "R90")
)

y = y - 6

for(j y y+2          ; up to 1024
    for(i 0 x-1
        dbCreateParamInst(cv inst_pfet nil 3*i+0.4375:3*j "R0" 1
            list(list("w" "float" 1u))) ; pull-up
        tf_schCreateWire(cv list(48*i+10:48*j 48*i+11:48*j 48*i+11:48*j+6)
            "vdd!" 48*i+10:48*j+6 "R270") ; pull down vdd
        tf_schCreateWire(cv list(48*i+7:48*j 48*i+5:48*j 48*i+5:48*j+8))
        dbCreateParamInst(cv inst_tnet nil i*3+1.125:j*3+0.5625
            "R0" 1 list(list("RES" "float" 0.705)
                        list("CAP" "float" 0.565f)))
        if( i == 0 then
            tf_schCreateWire(cv list(48*i-4:48*j+8 48*i+23:48*j+8)
                strcat("precharge_data_n<" concat(-j-11) ">") 48*i-4:48*j+8 "R0")
        else
            tf_schCreateWire(cv list(48*i-4:48*j+8 48*i+23:48*j+8))
        )
        dbCreateParamInst(cv inst_tnet nil i*3+0.75:j*3 "R270" 1
            list(list("RES" "float" 93.7) list("CAP" "float" 1.36p)))
        tf_schCreateWire(cv list(48*i+11:48*j-3 48*i+11:48*j-5))
        tf_schCreateWire(cv list(48*i+11:48*j-26 48*i+11:48*j-32)
            strcat("bl" concat( -j-11 ) "<" concat( i ) ">")
            48*i+10:48*j-30 "R90")
    )
    dbCreateInst(cv inst_noConn nil 137.75:j*3+0.1875 "R0")
)

dbCreateInst(cv inst_nand nil 20:-48 "R0")
dbCreateInst(cv inst_nand nil 20:-52 "R0")
dbCreateInst(cv inst_mux strcat("IMUX<" concat( x-1 ) ":0>") 25:-50 "R0")
tf_schCreateWire(cv list(316:-768 320:-768) "row_match_n<0>" 308:-768 "R0")
tf_schCreateWire(cv list(316:-770 320:-770) "row_match_n<2>" 308:-770 "R0")
tf_schCreateWire(cv list(316:-832 320:-832) "row_match_n<0>" 308:-832 "R0")
tf_schCreateWire(cv list(316:-834 320:-834) "row_match_n<1>" 308:-834 "R0")
tf_schCreateWire(cv list(344:-768 360:-768 360:-808 400:-808))
tf_schCreateWire(cv list(344:-832 360:-832 360:-810 400:-810))
tf_schCreateWire(cv list(425:-800 432:-800))
tf_schCreateWire(cv list(400:-800 390:-800)
    strcat("bl0<" concat( x-1 ) ":0>") 390:-800)
tf_schCreateWire(cv list(400:-802 390:-802)
    strcat("bl1<" concat( x-1 ) ":0>") 390:-802)
tf_schCreateWire(cv list(400:-804 390:-804)
    strcat("bl2<" concat( x-1 ) ":0>") 390:-804)
tf_schCreateWire(cv list(400:-806 390:-806)
    strcat("bl3<" concat( x-1 ) ":0>") 390:-806)

tf_schCreateWire(cv list(331:-760 331:-750) "vss!" 330:-755 "R90")
tf_schCreateWire(cv list(333:-760 333:-750) "vdd!" 332:-755 "R90")
tf_schCreateWire(cv list(331:-824 331:-814) "vss!" 330:-819 "R90")
tf_schCreateWire(cv list(333:-824 333:-814) "vdd!" 332:-819 "R90")
tf_schCreateWire(cv list(412:-792 412:-782) "vss!" 411:-787 "R90")
```

```
    tf_schCreateWire(cv list(414:-792 414:-782) "vdd!" 413:-787 "R90")

    tf_schCreatePin(cv "precharge_data_n<3:0>" "input" -10:24*y "R0"); bit_line
    tf_schCreatePin(cv strcat("wl<" concat( -y-7 ) ":0>") "input" -4:y*7)
    tf_schCreatePin(cv strcat("bl<" concat( x-1 ) ":0>") "output"
                    432:-800 "R0")
    tf_schCreatePin(cv "row_match_n<3:0>" "input" 288:-800 "R0")

;   schHiCheckAndSave()

))
```

# Appendix H

# Schematics

These are the schematics for an ternary CAM associative indirect design to be placed in a 1-Gbit DRAM. I assumes four-way set associativity, but all 25 address bits and 20 redundant memory bits are still in the design.

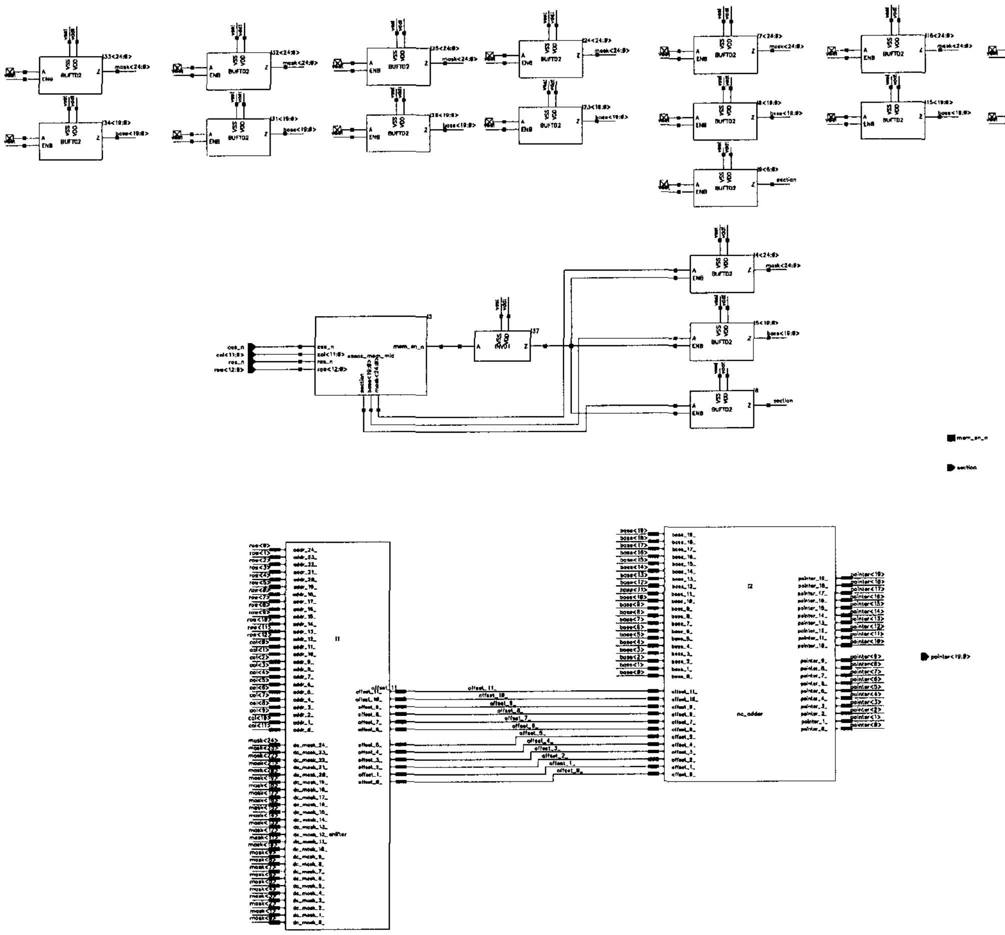


Figure H.1: Top level schematic of associative indirect redundancy. It does not include the redundant memory (connected via pointer).

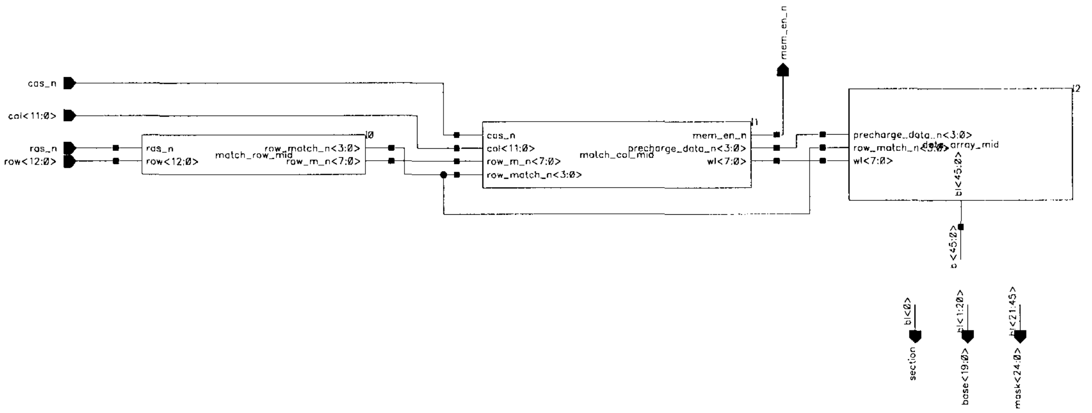


Figure H.2: Associative memory (match and data arrays)

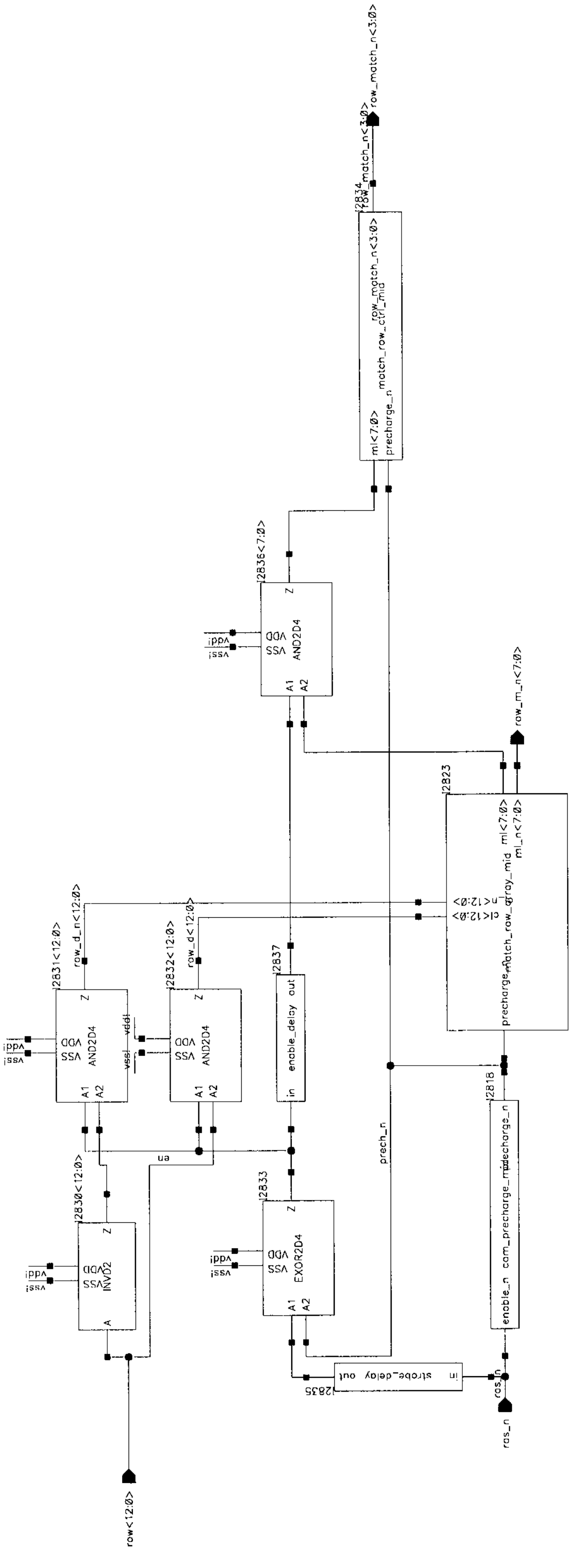


Figure H.3: Row match component

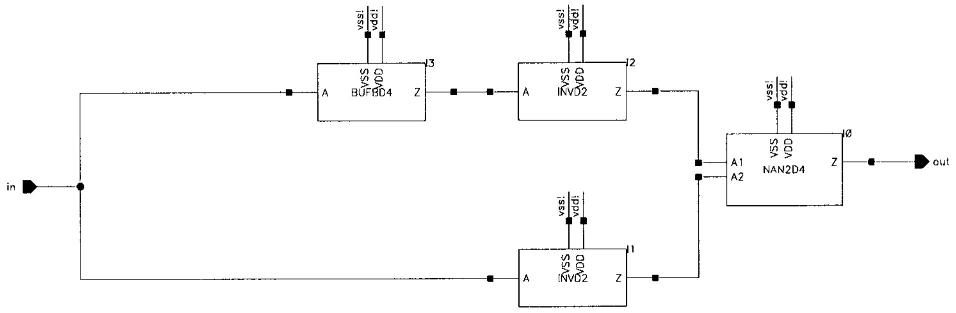


Figure H.4: Row access strobe delay

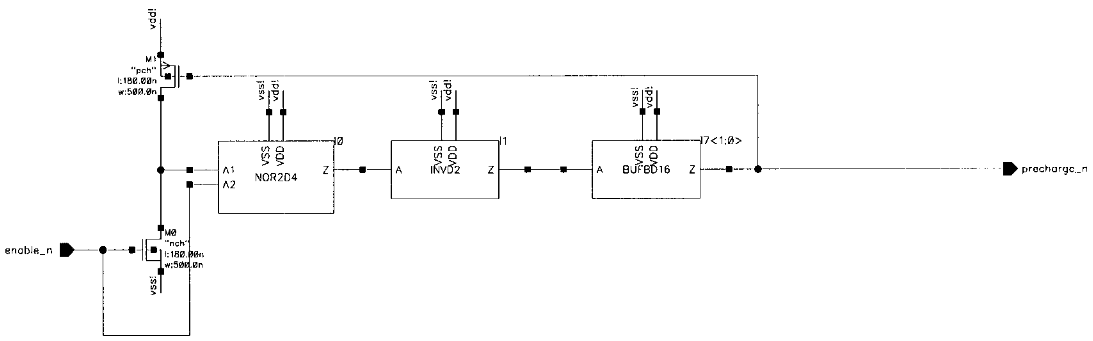


Figure H.5: Row match array match-line pre-charge

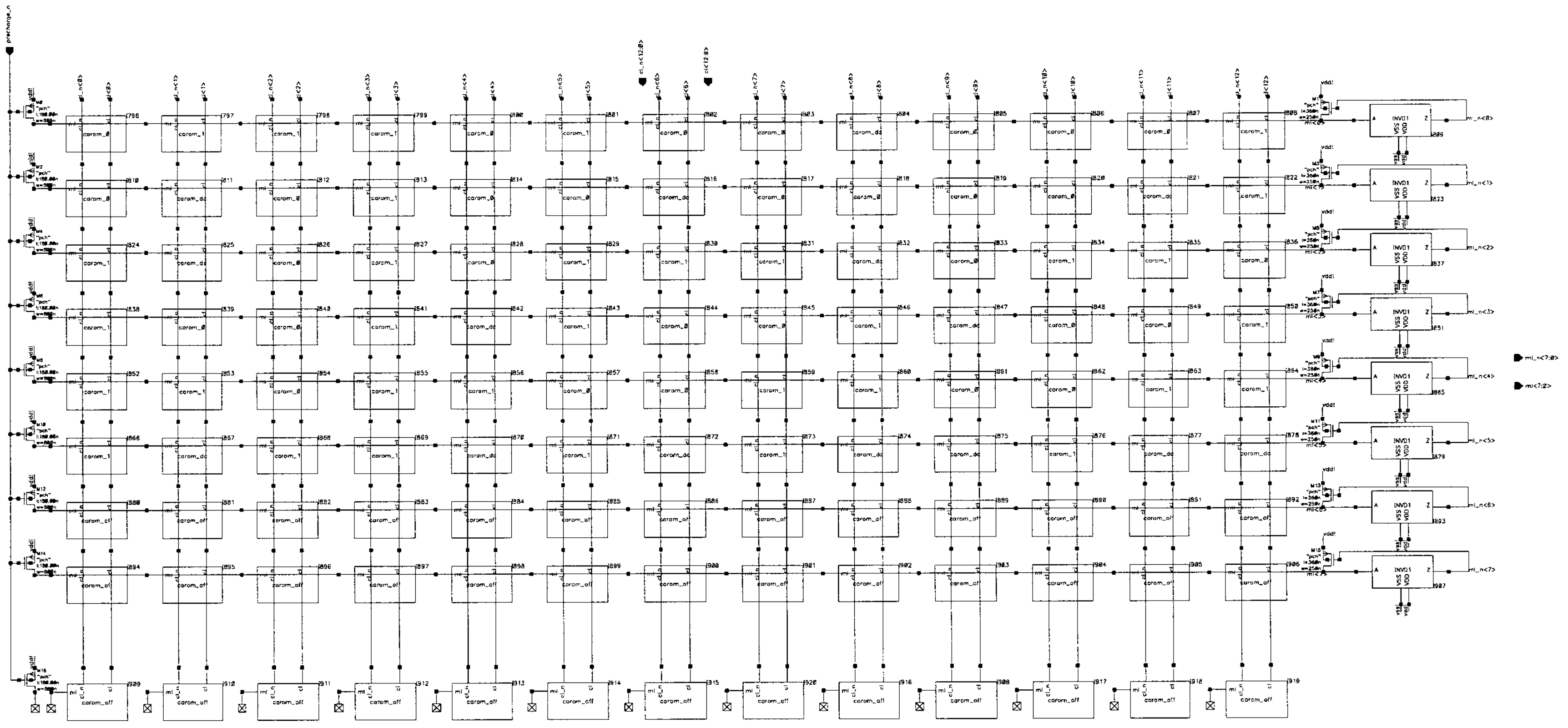


Figure H.6: Row match array

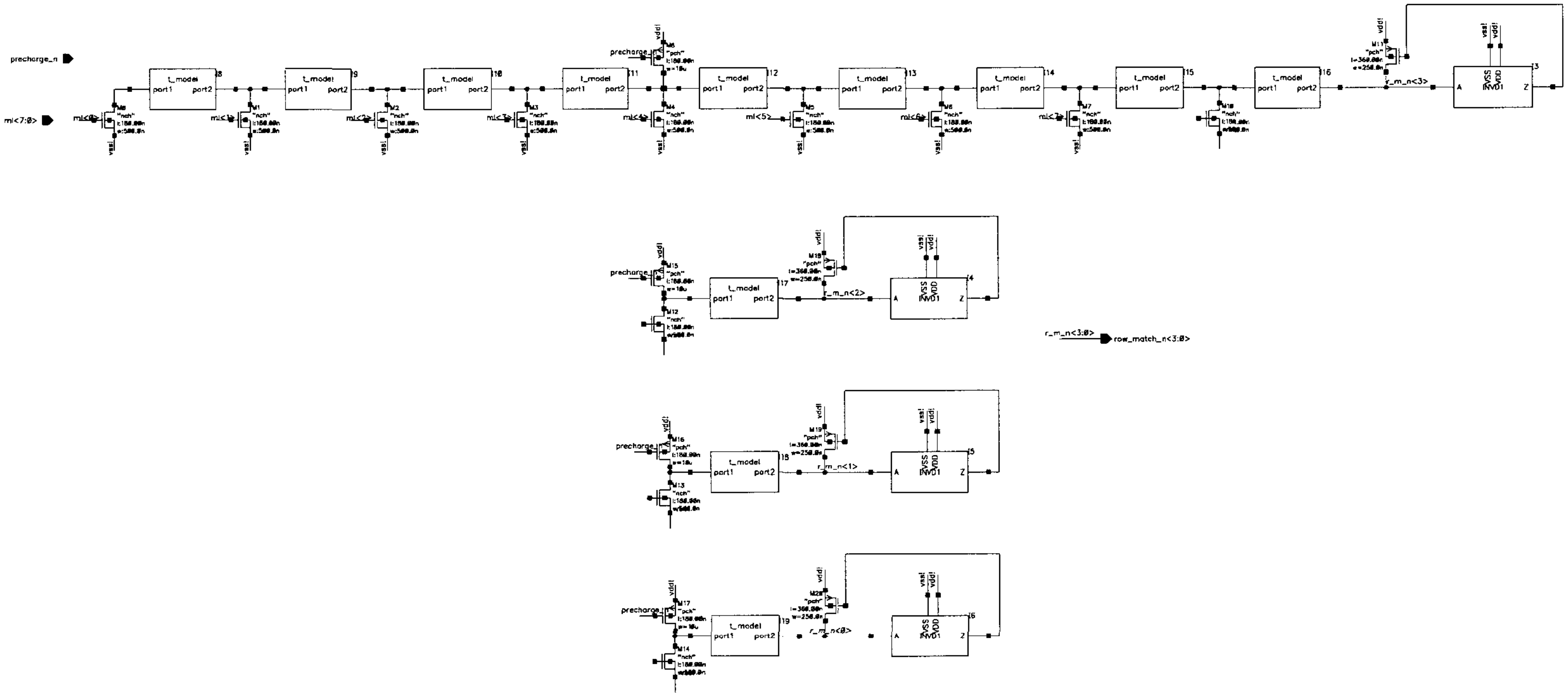


Figure H.7: Row match control (did a match occur anywhere in the array?)

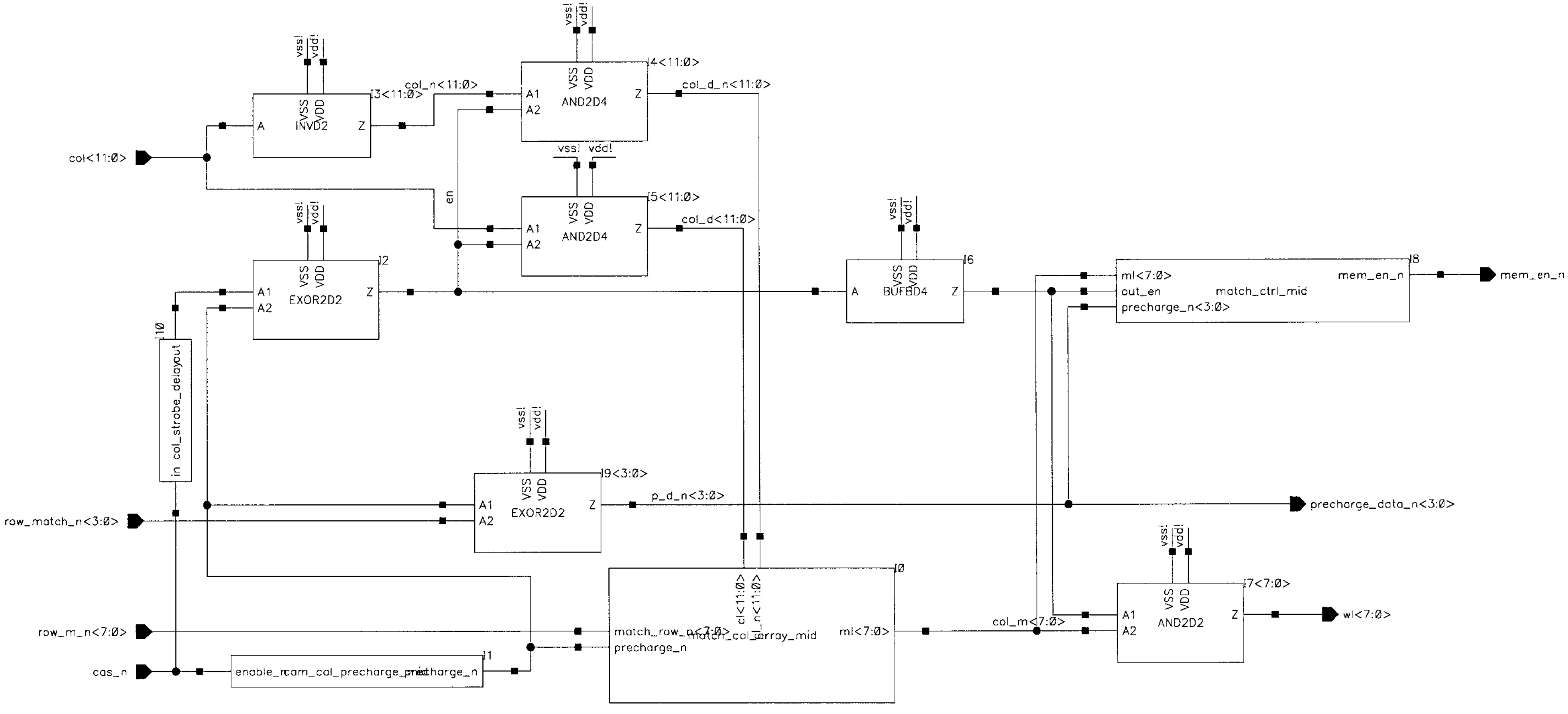


Figure H.8: Column match component

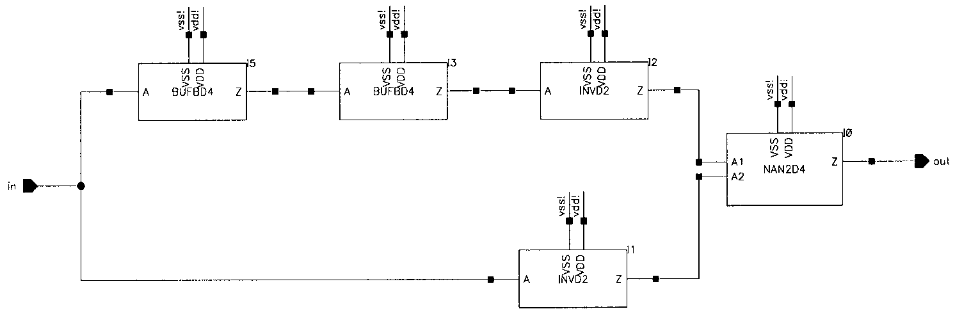


Figure H.9: Column access strobe delay

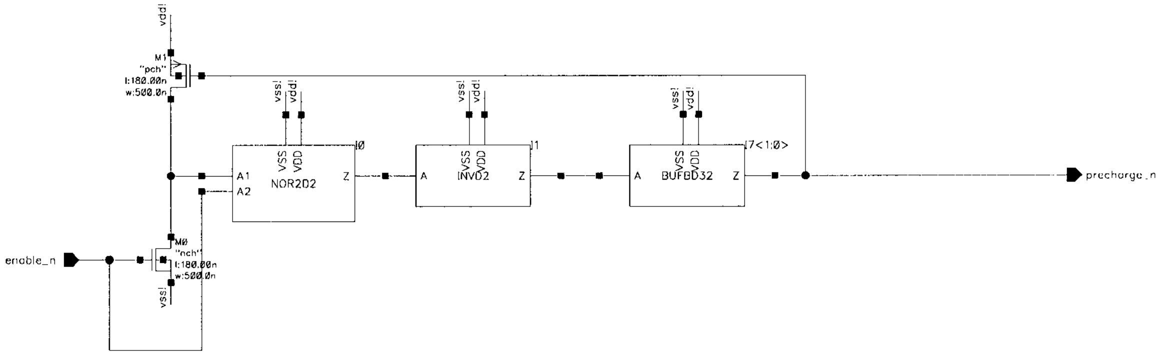


Figure H.10: Column match array match-line pre-charge

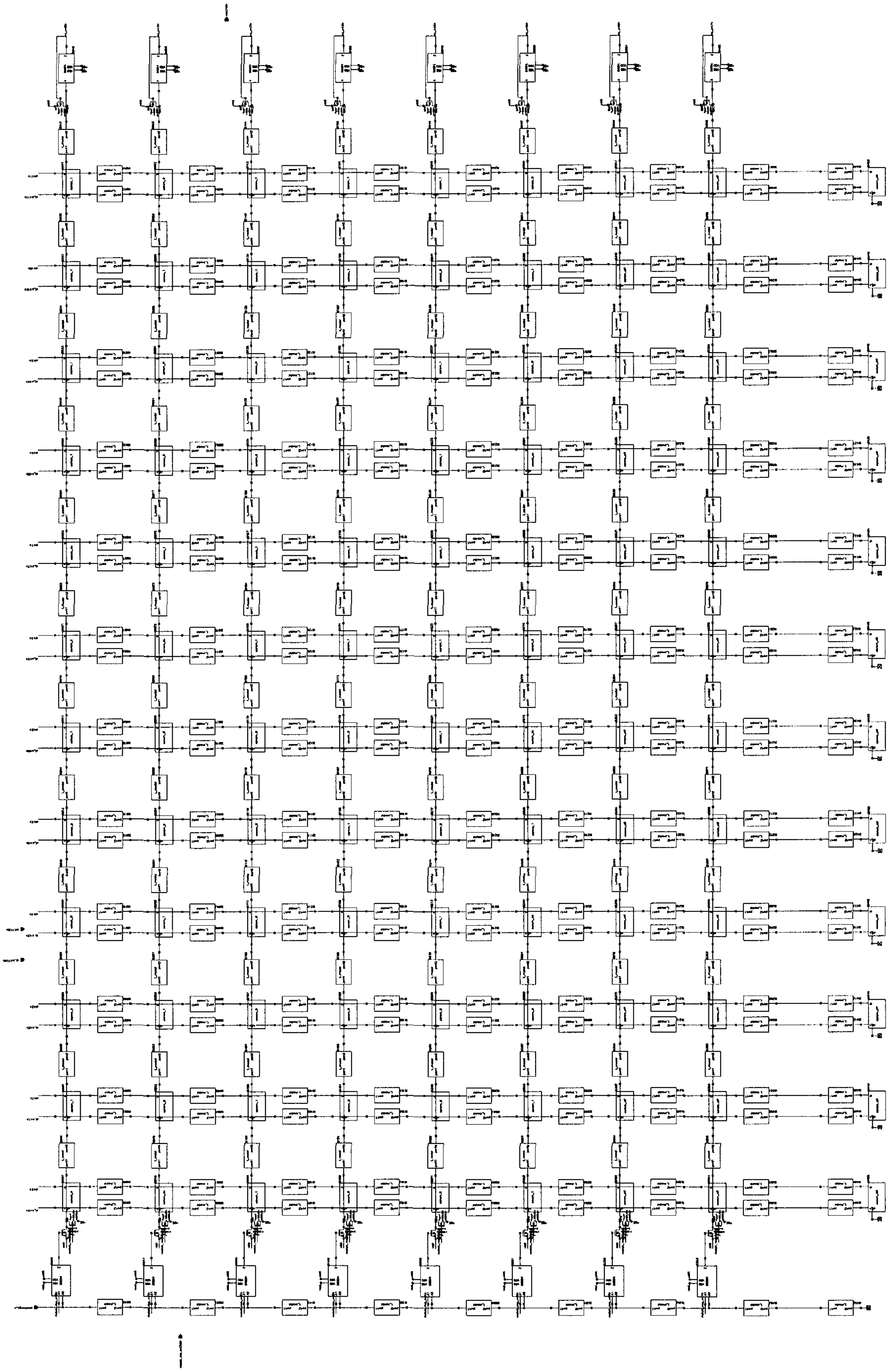


Figure H.11: Column match array

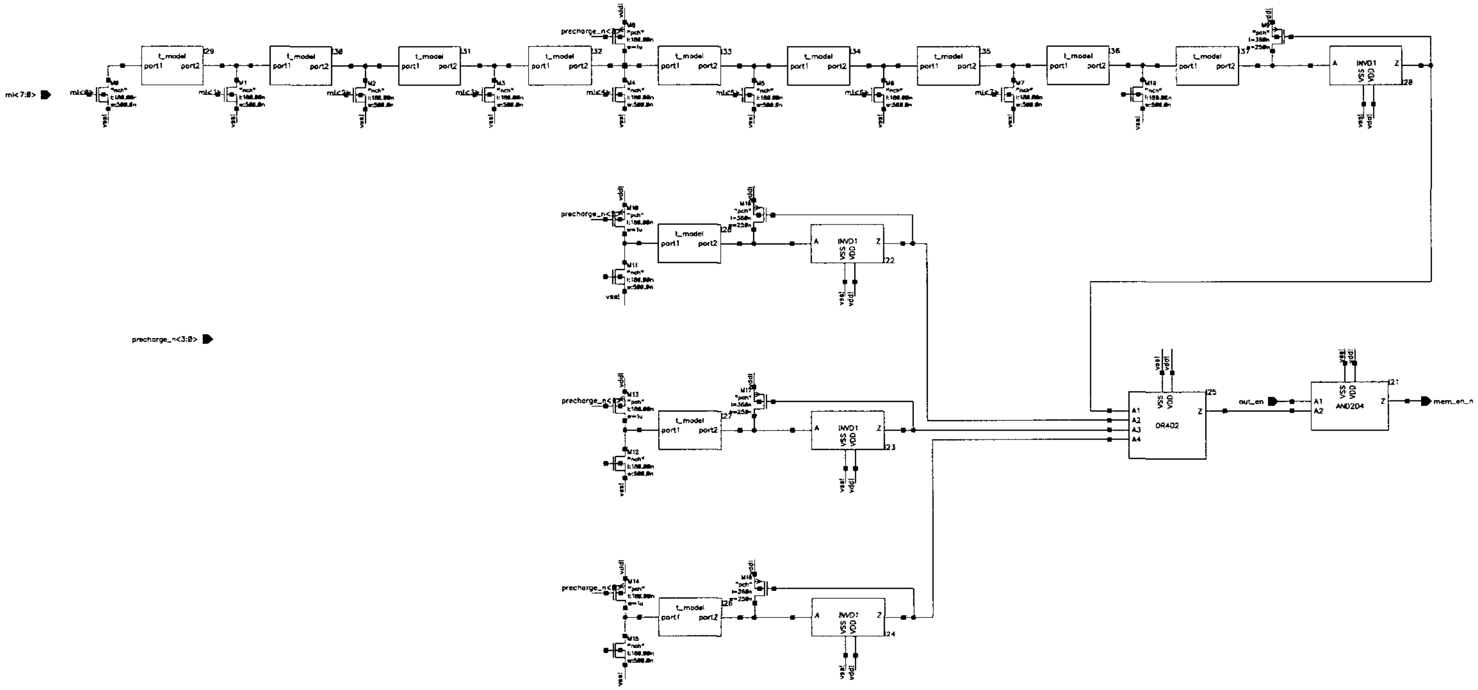


Figure H.12: Match control (did an address match?)

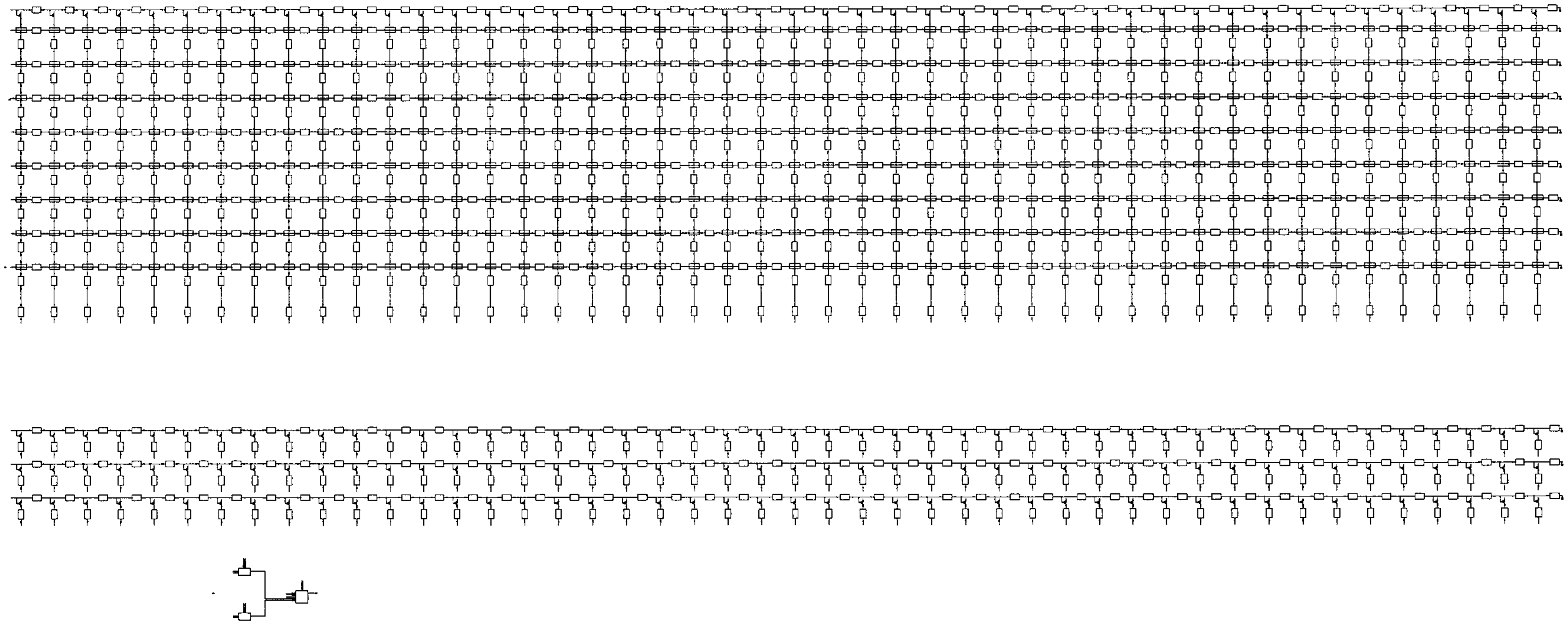


Figure H.13: Data array (containing section, base and mask)

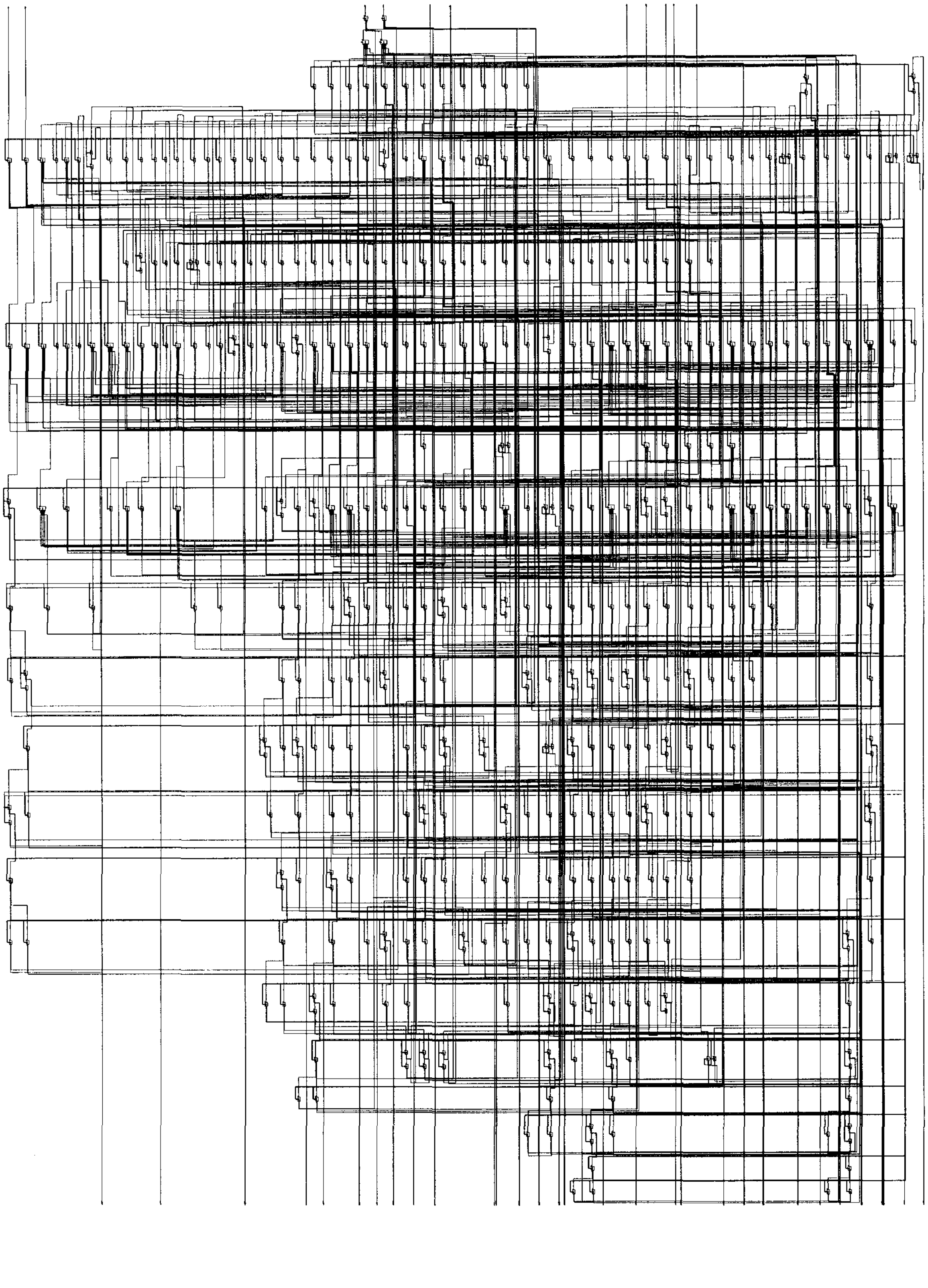


Figure H.14: Move *don't care* bits in address to lowest order bits

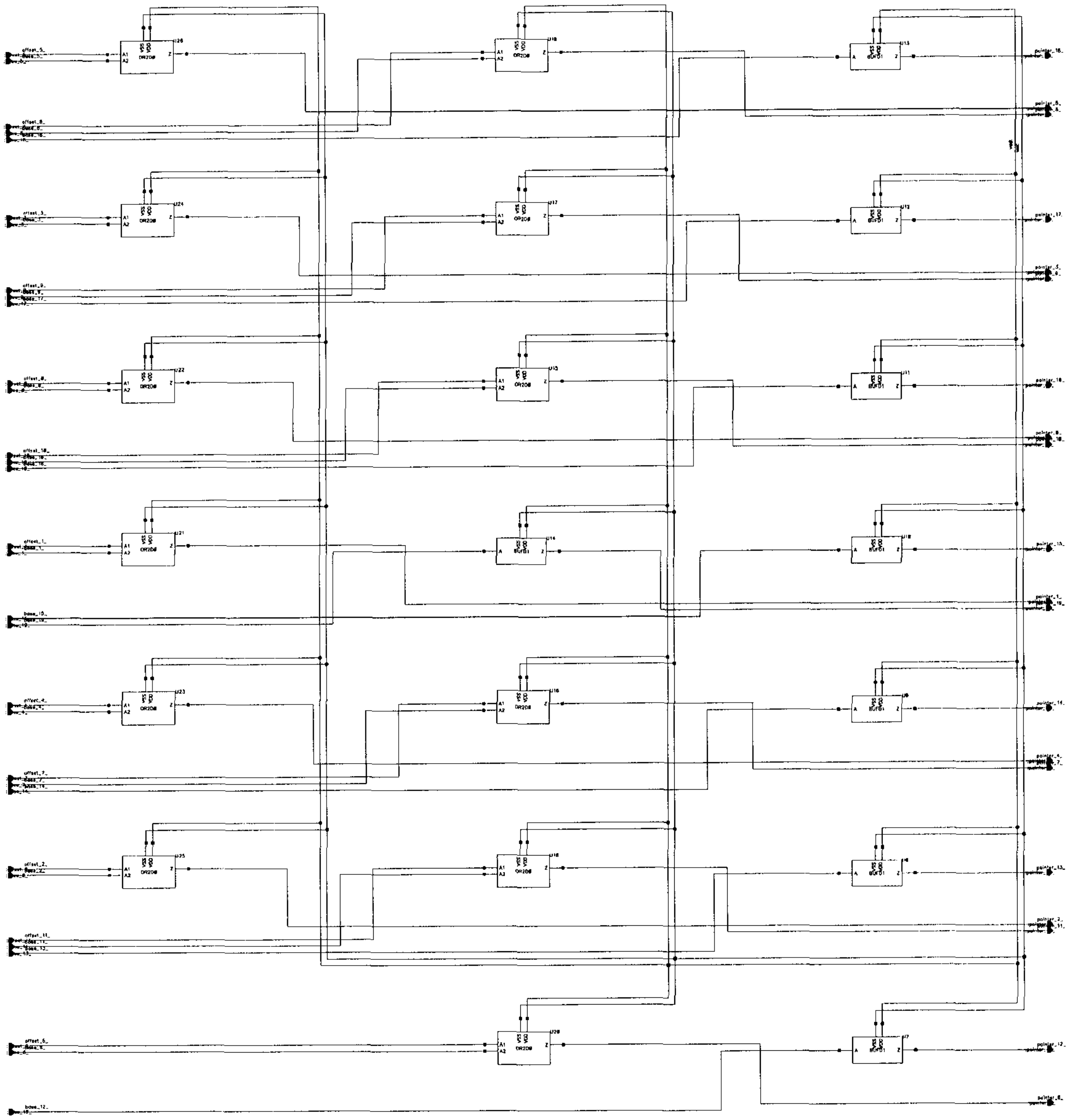


Figure H.15: OR (No-carry adder)