A Model for Analysis of the Effects of Redundancy and Error Correction on DRAM Memory Yield and Reliability

by

Joseph Adam Croswell

B.S. Electrical Engineering and Computer Science (2000)

Massachusetts Institute of Technology

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

at the

Massachusetts Institute of Technology

September 2000

Signature of Author . . . .
Joseph A. Croswell
Department of Electrical Engineering and Computer Science
August 28, 2000

Certified by . . . .
Srinivas Devadas
Professor of Electrical Engineering and Computer Science
Thesis Supervisor

Accepted by . . . .
Arthur C. Smith
Chairman, Department Committee on Graduate Thesis

A Model for Analysis of the Effects of Redundancy and Error Correction
on DRAM Memory Yield and Reliability
by
Joseph Adam Croswell

Submitted to the
Department of Electrical Engineering and Computer Science

August 28, 2000

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

Abstract

Manufacturing a DRAM module that is error free is a very difficult process. This process is becoming more difficult when only utilizing the current methods for producing an error free DRAM. Error correction codes (ECCs) and cell replacement are two methods currently used in isolation of each other in order to solve two of the problems with this manufacturing process: increasing reliability and increasing yield, respectively. Possible solutions to this problem are proposed and evaluated qualitatively in discussion. Also, a simulation model is produced in order to simulate the impacts of various strategies in order to evaluate their effectiveness.

Thesis Supervisor: Srinivas Devadas
Title: Professor of Electrical Engineering and Computer Science

Table of Contents

Figure List

I. Introduction

Two of the biggest problems for memory manufacturers are memory yield and reliability. Memory yield is the percentage of memory chips that make it through extensive testing which is done before they are considered good enough to be sold to customers. Manufacturers want this number to be as high as possible. The higher the percentage of chips that make it through the testing, the more memory that is available to be sold. Costs can be spread out over more unit sales and the manufacturer can make more money per chip, or sell more chips by undercutting competitors.

Memory reliability is also of huge concern to memory manufacturers. The market for memory is, for the most part, a commodity market. A commodity market is one in which products from different manufacturers are indistinguishable from one another. The one way that manufacturers of computer memory can differentiate themselves is by offering more reliable memory. This differentiation allows manufacturer to increase margins or increase sales.

Not only are these two of the biggest problems that memory manufacturers face, but with the trends in computers, they are also becoming harder and harder to solve. As processor technology progresses, memory densities and speeds must also increase in order to realize an overall performance gain. According to Amdahl, as a rule of thumb, to keep a balanced system, memory capacity should grow linearly with CPU speed. Microprocessor performance has increased by 55% per year since 1987 and 35% per year before 1987 [1]. The vast majority of personal computers use Dynamic Random Access Memory (DRAM) for main memory. So far this type of memory, in terms of capacity, has been able to keep pace with processors, with DRAM capacity growing about 60% per year [1]. This rapid growth in memory capacity makes it harder and harder to make a DRAM perfect, as each chip now has more cells that could fail and cause the entire chip to be useless.

Also, in terms of speed, memory hasn't nearly kept pace with processor improvement. DRAM row access times have grown only 7% per year [1]. Test time, and consequently test cost, is increasing because memory speed isn't growing as rapidly as density. New methods must be developed to refine the manufacturing process and enable reliable DRAM to be produced in a cost efficient way. Two currently utilized methods to increase the yield and reliability of DRAMs, as the complexity of manufacturing and testing increases, are cell replacement and error correcting codes (ECC). These methods, however, are currently used completely independently to solve the memory yield and reliability issues separately. The cell replacement technique is used exclusively to solve the yield problem, while error correcting codes are used exclusively to solve the memory reliability problem. Still yet to be explored is whether a combination of these two techniques can be used to simultaneously solve both the memory yield and reliability problems. This is the topic of the research covered in the following pages.

II. Possible Solutions

The two techniques discussed above, cell replacement and error correction codes, are currently adequate to address these two problems facing DRAM manufacturers. However, also as mentioned above, current trends are making it harder and harder to produce reliable memory at a high enough yield to be financially successful. At some point, the techniques previously discussed may not be enough to overcome the difficulties in a cost-efficient way. Two ways of combining these techniques to solve both problems that have yet to be explored are:

1. Rather than using redundant rows and columns for cell replacement during manufacturing, use redundant cells as extra space within the DRAM that can be used to implement an ECC with higher error coverage, which would require more redundant bits. Thus making it appear as if there are no errors by correcting them while the memory is in use.

2. Implement a replacement method that could be used in connection with an error correcting code to perform replacement at any point in the life of the memory, not just during prepackaged testing.

2.1 Higher Coverage ECC

Many error correction codes have been discovered that can provide coverage for various levels of errors. Higher levels of error coverage require more redundant bits to implement the error correction code. However, if instead of using built in redundant cells for prepackaged replacement, they were used as redundant bits in an error correction code then the necessary extra bits might be a reasonably cost efficient way to provide the same level of yield and reliability as normal prepackaged replacement. This method would effectively mask failing cells by providing enough redundancy in the extra bits to be able to correct the error every time the failing cell is read. Currently error correction codes are used to provide a higher level of reliability, however, in this method, error correction codes would be used to provide an equivalent level of reliability. At the same time this method would allow a higher defect level in the manufacturing of the DRAM to be tolerated while producing the same quality products. This would enable a much lower level of testing to be performed, as most errors would be recoverable. Manufacturers would be able to effectively provide the same level of reliability to a consumer while cutting testing costs and enabling higher chip yields while accommodating higher defect levels in the new process.

2.2 Later Life Cell Replacement

One of the larger component costs of manufacturing DRAM is testing. During the testing process, cell replacement is performed in an attempt to make a malfunctioning DRAM module function correctly. If this replacement could be done at any point in the life of the DRAM then much of this cost could be cut. Complete testing would no longer be necessary, along with all of the expensive laser equipment needed to implement prepackaged cell replacement.

This method would provide the same benefits as the preceding one. Namely, higher defect levels could be accommodated in production and lower levels of testing would be needed, as errors will again be correctable. It would still involve an error correction code, as errors would need to be detected so that the failing cells could be replaced. It could be implemented in two ways. One would be to implement flash programmable data paths so that addresses that are reported to fail could be replaced while the memory isn't performing other operations. A second method would be for the memory controller to keep track of addresses that fail and discontinue using them, replacing them with redundant cells at addresses that are not being used. Either of these implementations would allow a low level of ECC coverage to be utilized to provide the same level of reliability, with lower accuracy and testing constraints. Errors would not be able to build up beyond the coverage of the ECC, because blocks containing errors would be replaced on the fly.

III Trends

Upon first inspection it might seem that these strategies might cost too much in added redundancy to make up for the savings in increased yield and lower testing and manufacturing costs. In fact this is almost certainly true currently, as the strategies already utilized are effective in producing adequate yields. However, this might not be the case in the future. The trends in test costs, silicon costs, and DRAM quality could lead to the current methods becoming inefficient in producing reliable memory.

3.1 Test Cost Trends

Already the cost of card testing is considered the biggest fraction of total cost for a DRAM memory card [2]. The trends show this fraction only increasing over time. The speed at which the DRAM has to operate is constantly increasing to allow system speed to increase with processor speed. This causes huge development and building costs for high-speed equipment capable of testing these memories. As DRAM speeds push toward physical limits, developing equipment to test them will only get more difficult. Already DRAM manufacturers buy test equipment from companies that specialize in developing testing solutions for the ever-advancing DRAM speeds and size.

The rapid increase in the complexity of the DRAM also leads to difficulty, and increased costs, in testing. Early memory cards consisted of just a few DRAM modules mounted on one side of the card with the only other components on the cards being a few capacitors. Today memory cards have been built with 18 or more DRAMs, mounted on both the front and back of the card and sometimes stacked on top of each other. They also contain many discrete components including buffering devices, capacitors, and PLL's. Consequently, isolating an error has become much more difficult, which again increases test costs. Increasingly complex solutions will also be needed to include more memory on fewer and smaller modules in order to keep pace with other system level advances, further frustrating the testing problem.

In addition, the size and density of memory being utilized has increased so much that even if you could test all the extra components at the same rate, it would still take orders of magnitude longer to test all of the cells. As an example, if the test time for a memory card with 256K addresses was 10 seconds, then to produce the same level of coverage on a 1GB card with 128 M addresses would require over 85 minutes. Even compensating for the speed increases between these two memories, it would still take 53 minutes to test the 1GB card [2]. Density increases, as cited in the overview, are greatly outpacing speed increases, causing this problem to get worse and worse. This can partially be compensated for in testing strategy. One example is testing multiple modules in parallel with the same piece of test equipment. This allows overall throughput of tested devices to increase. However, as mentioned above, testing strategies and test equipment are already becoming increasingly costly as they struggle to keep up with technology advances in the DRAMs being tested.

3.2 Silicon Cost Trends

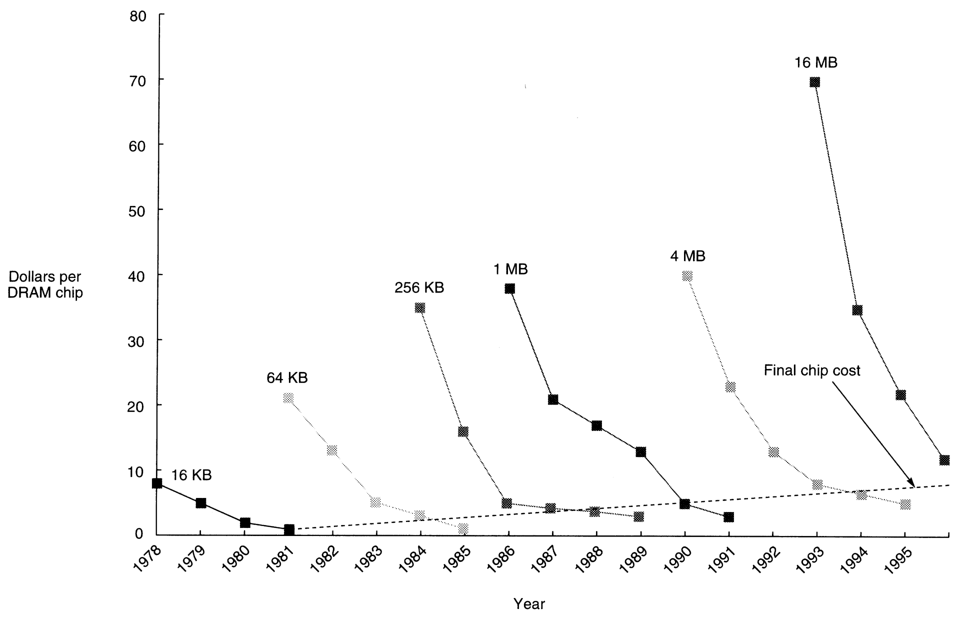

Figure 3.1:
Price of multiple generations of DRAM's over time in 1977 dollars [1]

As mentioned in the introduction, DRAM densities are increasing by just under 60% per year. This huge increase in memory size has lead to only a small increase in constant dollar prices per module. This has caused a rapid decrease in constant dollar cost per megabyte, as each new generation exhibits a higher capacity. Figure 3.1 shows prices of different generations of DRAMs over time in 1977 dollars [1]. By eliminating inflation, Figure 3.1 shows the real change in DRAM cost. The cost per megabyte of memory has dropped incredibly during this period, from over $5000 in 1977 to just over $6 in 1995 (1977 dollars) [1]. This implies that cost per megabyte of DRAM fell by over 40% per year during that time period. As cited in the previous section, testing costs during this same period have increased rapidly. This implies that the component of cost due to things besides testing, including silicon cost, is falling even more rapidly. The proportion of cost attributable exclusively to materials and manufacturing is consequently shrinking. A more efficient solution to the problem of manufacturing reliable DRAM might be possible if extra silicon, which is decreasing in cost, could be traded for lower levels of testing, which is increasing in cost. The solutions proposed in section II are both designed to take advantage of these trends.

3.3 Quality Trends

The quality demanded from DRAMs is also increasing rapidly. The market for DRAM memory is, for the most part, a commodity market where the products from different manufacturers are mostly identical and indistinguishable from one another. This leads to a situation where one cost prevails in the marketplace across all manufacturers. This allows buyers to be very selective in choosing between manufacturers. Quality and reliability are basically the only things that moderately differentiate manufacturers.

Large buyers, in particular, have the ability to research these factors before choosing a manufacturer. Since a lot of DRAM is purchased this way by companies manufacturing systems in which the DRAM will be included, these are very important factors for DRAM manufacturers. Thus the demands are getting even more constraining in these areas.

To be competitive, manufacturers must have very high Acceptable Quality Levels (AQLs). AQL is the percentage of the DRAM that function correctly. If a manufacturer ships a lot of malfunctioning chips to a PC builder, that PC builder is not likely to purchase DRAM from that manufacturer in the future. This again leads to increasing test costs, because of the desire to (as completely as possible) test all DRAMs before shipping in order to have the highest AQL possible. This presents another situation in which silicon costs, which are decreasing, could be substituted for testing costs, which are increasing, to create a more economically efficient process. This is again the aim of the solutions proposed in section II.

3.4 Trend Summary

The solutions proposed in section II are designed to address these exact trends and the problems that they are likely to create in producing reliable and quality DRAMs in the future. Both methods are aimed at allowing a limited error level within a DRAM that can still function completely correctly. This reduces the necessary testing, which, as pointed out above, is increasing in cost. Each method allows for this limited error level by using extra cells in a strategy to compensate for the failing cells. As pointed out above, these extra cells are decreasing in cost. Finally, both methods should result in higher quality memory because of the ability to survive failures within the DRAM that occur after all the manufacturer's testing is done. Failures that are missed during testing could also, potentially, be recoverable. This will give manufacturers utilizing these methods a competitive advantage in a market where they are price takers and can only differentiate themselves by producing more reliable DRAMs.

IV. Error Correction

One method currently utilized to produce reliable memory is the use of Error Correction Codes (ECC) to encode data before it is stored in the memory. Error correction codes take a set of information bits at the producer of the information and create a set of redundant bits based on the information bits. These redundant bits are sent or stored with the original set of information bits. The consumer of the information then uses the redundant bits to determine if any errors have occurred in transmission or storage. In the case of memory, the redundant bits are calculated and stored along with the original bits and then when they are read from the memory they are examined to determine if any errors have occurred between the time the information was stored and the time it was retrieved.

4.1 Basics

A simple example of an ECC is parity checking. Parity checking involves counting the number of ones in a sequence of bits. The parity bit is then created to make the total either even or odd, depending on which type of parity checking is being performed. For example, if you were checking for even parity and had the following sequence of bits: 10011010, then a 0 would be added as the parity bit to keep the total number of ones in the sequence an even number. This is useful because the consumer of the information, knowing that all information it receives should have even parity, can detect an error if it gets a sequence of bits with odd parity. Thus any sequence with only one incorrect bit would be detectable because the parity would necessarily be changed. This coding is not actually an error correction code since it can only detect errors, not correct them.

Other codes that are capable of not only detecting errors but also correcting them are actual error correction codes. A simple example is the (7,4) Hamming code, named in honor of R.W. Hamming. In the (7,4) Hamming code each block of bits contains seven total bits, four of which are data bits; with the other three being redundant check bits, thus the title of (7,4) Hamming code. In this particular example the data bits of the Hamming code are arranged so that when parity checking is performed a single failing bit can be located when the entire group is checked. The four data bits are located in bit positions three, five, six and seven, and the parity check bits are located in bit positions one, two, and four. To clarify how the code works think of representing each bit position of the seven total bits in each block with its binary representation:

Bit Position	Binary Representation
1	001
2	010
3	011
4	100
5	101
6	110
7	111

Now, a parity check is performed on each bit position from the table above that has a '1' in the least significant position (farthest right) of its binary representation in order to obtain the first parity check bit. Next, to obtain the second parity check bit, a parity check is performed on every bit position from the table above that has a ‘1’ in the middle position of its binary representation. And finally, to obtain the last parity check bit, a parity check is performed on all bit positions with a ‘1’ in the most significant position (farthest left) of its binary representation.

The parity bits of the block cannot be checked before they are generated, so the parity bits are generated just from the data bits. Thus, the first parity bit will be generated from bits three, five, and seven; the second from bits three, six, and seven; and the third from five, six, and seven. When the block is checked for errors, all bits, including parity bits, will be checked for parity and a syndrome will be generated. A syndrome is the binary number, in this case three bits long, that results from each parity check on each groupings of bits. If there are no errors then the syndrome will be zero. If there are errors the syndrome will contain the information necessary to decode where the error occurred. As an example to illustrate how this works for the (7,4) Hamming code being discussed, take 0101 as the data bits, producing 1001101 as the entire odd parity block (xx0x101 being the data and 10x1xxx being the bits generated by an odd parity check on the data bits). Performing an odd parity check to the same groupings of bits does in fact produce a syndrome of 000. Now, if we flip bit five the block becomes 1001001. Performing the parity checking again results in:

Parity check on bits 1,3,5,7 (**1**0**0**1**0**0**1**): 1
Parity check on bits 2,3,6,7 (1**00**10**01**): 0
Parity check on bits 4,5,6,7 (100**1001**): 1

This is a syndrome of 101. 101 is the binary representation of five, which is the bit that we flipped to induce an error.

The location of the data bits and parity bits within the block of this code allow the error bit to be identified directly from the syndrome by interpreting the syndrome as the binary representation of the failing bit position. The data bits and parity bits can be arranged in any way through the block, as long as the data bits are still checked in three groups of three, and the code will still have the same error correction properties. However, if the bits are located in different positions, a lookup table will be needed to determine which bit is incorrect from the syndrome.

This arrangement of bits allows decoding to be simple for an observer. Another arrangement that is used more often across a variety of codes is called systematic. A systematic arrangement is one in which all of the data bits are grouped together and all of the check bits are grouped together. A code arranged in this way is called a systematic code. In general, there can be many different arrangements of the bits in a code that can be used to create distinct codes. However, all codes generated this way are equivalent in terms of their error correction/detection properties, and are equivalent to some systematic code. Some of the reasons for arranging a code to be a systematic code will be discussed later.

A similar linear block code to the (7,4) Hamming code is used in Pentium Pro Processors. This code allows all single bit errors to be corrected, all double bit errors to be detected, and all errors confined to one nibble to be detected. This is abbreviated SEC-DED-S4ED for Single-bit Error Correction, Double-bit Error Detection, Single-nibble of four bits Error Detection. A nibble is a defined group of bits, in this case four. But not every four bits is a nibble, only bits grouped together in the ECC. In this case 0-3, 4-7,... and not 3-6, 7-10,... because the nibble boundaries for this code is every four bits. This code is 72 bits wide with 64 data bits and eight check bits. As before the check bits are generated by performing a parity check on groups of bits within the data. In this code a lookup table is necessary to determine where, within the block, an error has occurred from a non-zero syndrome. However, the syndrome does have the following properties that can be used to determine what kind of error has occurred simply by looking at the syndrome:

1. If one nibble of the syndrome contains three bits that are a '1', and the other nibble contains all zeros, then there is a data nibble that contains a three or four bit error
2. If the syndrome contains an odd number of ones, then the data contains a single bit error that is correctable using the lookup table.
3. All other values of the syndrome are double bit or greater, non-correctable errors.

Codes like this one used in the Pentium Pro are widely used in higher end applications that require higher reliability than standard implementations.

4.2 Definitions

This subsection will give definitions for some of the terminology that will be used in the rest of section IV. Every code has a certain number of data bits and a certain number of redundant check bits that are generated from the data bits. The number of data bits is generally referred to as k and the total number of bits in the code (data bits + check bits) is referred to as n. Codes are often identified by "type of code (n, k)". (Example: Hamming (7,4)) The code rate of a code is the number of data bits divided by the number of total bits (k/n). A higher code rate indicates that less redundancy has to be added. However, higher levels of error correction capacity (ability to survive more errors) generally lead to lower code rates because more redundancy is needed to survive more simultaneous errors. This tradeoff is illustrated visually in Figure 4.1 [3]. In the diagram, codes closer to the origin are better because they have both higher error correction capacity and code rate. Class B codes from the diagram are more efficient than class A codes from the diagram because all codes in class B are closer to the origin than corresponding codes in class A. However, class A codes could still be useful in certain applications. Codes, like class B codes, that are more efficient generally require more complex and slower hardware to decode. So in high speed or low cost applications, codes from class A might be more appropriate.

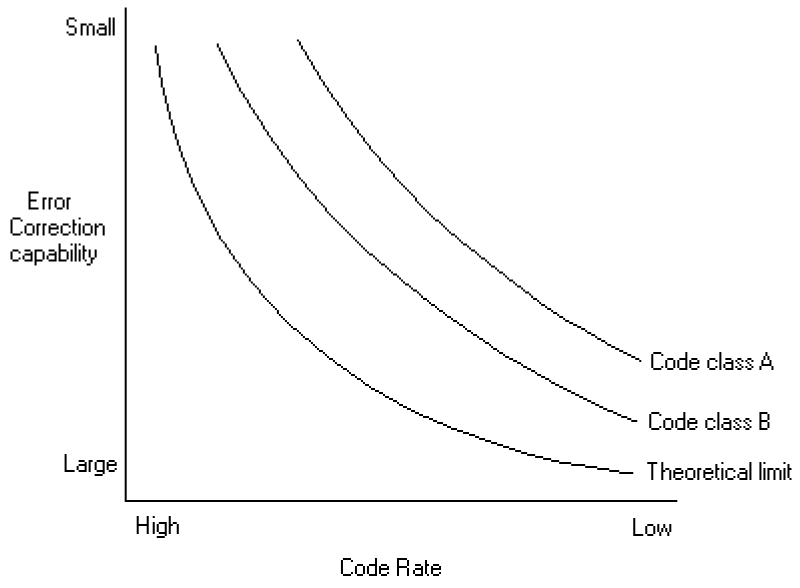

Figure 4.1
Characteristics of codes [3]

A single code type generally has a specific procedure for generating that particular code with different code lengths. However, the generation procedure usually doesn't allow all code lengths to be generated. One constraint that is often found in the generating procedure for binary codes is that the code length, n, must be a power of two. However, there are often accompanying procedures for modifying a code while retaining some or all of the properties of the original code. These procedures create a modified code, which isn't a code of the pure type that was generated, but can often still have all of the same error correction properties. This modified code can then be the length needed for a specific application.

One of the properties that these modifying procedures hope to retain is the minimum distance. This is the property of the code that determines the error correction coverage of the code. The distance between two code words is the number of bits by which the two codes differ. For example the distance between 10**0**01**01**0 and 10**1**01**10**0 is three, with the bits that differ in bold. The minimum distance of a code is the shortest distance between any two code words. For example if the following were all of the code words in a code:

001
010
111
100

then the distance of the code would be two since all of the code words differ from all of the other code words by at least two positions. If a code has a minimum distance d=2t+1, then the code can correct all t bit errors and detect all d-1 bit errors. Depending on other properties of the code, some higher bit errors might be detectible, but not all higher bit errors can be detected, so the distance gives a minimum reliability.

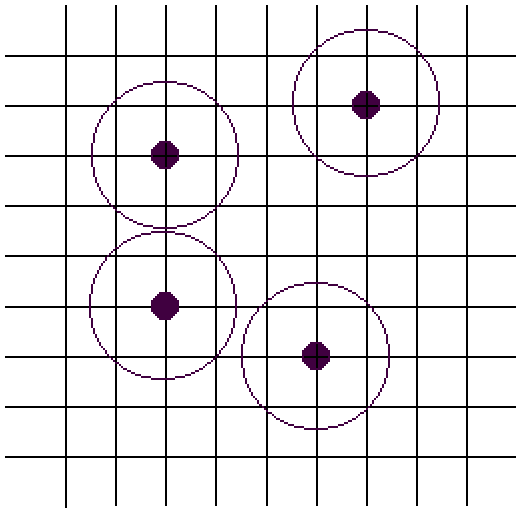

Figure 4.2
Code Words Graph

A code can detect all d-1 errors because for any of the code words to get changed into another code word, d bits would have to change. This is because d-1 is the number of bits that differ between the two code words with the most bits in common of any pair in the code. To see why the code can correct t errors, think of each code word as occupying a single spot in a grid. In Figure 4.2, a blackened circle located at a junction in the grid represents a code word. Moving between adjacent junctions represents changing one bit. So the minimum distance of the code pictured is three since the closest two code words are three junctions apart (three bits differ). The larger uncolored circles represent circles of distance t=1 away from the code word. Any received word that falls inside the larger circle surrounding a code word can be corrected to that code word. This is because errors are assumed to be equally likely and uncorrelated, and thus with a low probability of a single error, the chance of an additional error is much lower. Consequently, the probabilities lead to the conclusion that a received word that is located inside a larger circle on the graph in Figure 4.2 almost certainly originated as the code word at the center of the circle. This is called maximum-likelihood decoding.

4.3 Types of Codes

There are many different types of error correction codes. Some examples are Hamming codes, Bose-Chaudhuri-Hocquenghem (BCH) codes, Reed-Muller codes, Golay codes, and Reed-Solomon codes. These examples are all from a single class of codes called linear block codes. There are many other linear block codes that have been studied to varying levels, not to mention other classes of codes, including convolution codes and arithmetic codes.

However, the types of codes that are applicable for implementing error correction for DRAM used in personal computer applications is very limited. Arithmetic codes are designed to correct errors at the output of arithmetic operations. Convolution codes are designed to correct errors over multiple consecutive message blocks. Linear block codes are really the only class of codes applicable to DRAM memories.

Further, the number of linear block codes that are applicable to DRAM used in personal computers is even more limited. DRAM used in personal computers necessarily has to operate at very high speeds. Many codes have very complicated encoding and decoding schemes that are difficult and expensive to implement in hardware. Even codes that are easily implemented in hardware often don't have hardware implementations that perform fast enough to meet speed requirements for memory subsystems in personal computer systems.

Even with all of these limiting factors on the number of codes that are applicable to DRAM memory, there are still a wide variety of codes that can be used in DRAM applications. Many Hamming codes and modified Hamming codes have encoding and decoding hardware that is able to meet the speed requirements of DRAM. Also, because of the high code rate of BCH codes, a lot of study has been done to find fast hardware encoding and decoding schemes to make BCH codes applicable to high-speed memory.

	Check Bits
SEC	7
SEC-DED-SPD (4 bits/package)	8
SPC (2 bits/package)	8
SPC-DPD (2 bits/package)	10
SEC-TED	14
DEC-TED	15

Figure 4.3
Check bits needed for 64 data bits with different error correction capacities

The applicable codes still offer a wide variety of error correction properties. They range from simple parity checking with single error detection through triple bit error correcting and beyond. Some codes also offer special properties like the Pentium code cited before, which can detect burst errors of length four. Also besides being able to correct and detect all errors at some level, there are many codes that detect large percentages of higher bit errors, making them even better codes. All of these properties come at the cost of varying number of necessary redundant bits. Figure 4.3 shows the number of bits needed to implement varying levels of error correction capacity for 64 bit long data words.

Looking at the chart in Figure 4.3, the number of total bits needed to implement a DEC-TED code is nearly 10% more than the number of bits needed to implement a SEC-DED code. Even given that silicon only currently makes up about 50% of the total cost of a chip, this is a significant cost and efficiency penalty for adding just one more bit of error coverage. This calls into question the effectiveness that could be obtained currently by the first solution proposed in section II. A double error correction code would cost significantly more in terms of chip costs, while only barely satisfying the requirements of using a higher error correction code to mask all errors. If more than two hard errors lined up in a lowered accuracy manufacturing process, like the one proposed in solution one from section II, the chip would be entirely useless without also having later life cell replacement capability. Also, soft errors could line up with hard errors in an irrecoverable way with reasonable probability.

It is still possible that some time in the future a code with a low code rate could be paired with a compression code to produce an overall code rate that would be acceptable. However, compression codes have only very recently been applied to main memory in personal computers. In the near future, the second proposed solution has a much higher chance of producing a cost efficient result. Consequently, the remainder of this paper will concentrate on exploring the second proposed solution, which doesn't allow errors to build up along the same code word, with a modified Hamming (72,64) code.

4.4 Modified Hamming (72, 64) Code

This code is a distance four modified Hamming code. It can correct all single bit errors and detect all double bit errors. In addition it has some special properties that allow it to detect high percentages of higher bit errors. This code, proposed in [4] and described in [3], satisfies the conditions of the minimum-equal-weight code, the odd-weight-column code, and the rotational code.

Hardware encoding and decoding involve a number of modulo two additions that is proportional to the number of ones in the code's parity check matrix (parity check matrices will be explained in 4.5). This leads to two constraints on the parity check matrix that will lead to faster, lower cost, and more reliable hardware for encoding and decoding.

1. The total number of 1's in the parity matrix should be minimum
2. The number of 1's in each row of the parity check matrix should be made equal, or as close as possible, to the average number, i.e., the total number of 1's in the parity matrix divided by the number of rows. [3]

The parity check matrix of an odd-weight column code has an odd number of ones in all the column vectors of the matrix. Such a code gives "good discrimination of even number and odd number of errors by modulo 2 addition of the syndrome bits" [3]. Therefore the code is able to detect a larger percentage of errors which affect more bits than are covered by the code normally.

A rotational code is one in which the parity check matrix is made up of consecutive sub-matrices. The sub-matrices are identical except for being rotated once between each sub-matrix. Encoding/decoding hardware for such a code can be implemented by creating a circuit for one sub-matrix and then replicating that circuit for all other sub-matrices and simply changing the input output connections. This reduces the design time and cost for implementing the encoding/decoding hardware.

4.5 Decoding

```
    1 0 0 1 0 1 0 0 1 0 0 1 0 1 0 0 1 0 0 0 1 0 0 0 0 1 0 0 1 0 0 0 1 0 0 0 0 1 0 1 0 1 0 1 1 0 1 1 1 0 1 1 0 1 1 1 1 0 1 1 0 1 1 0 1 1 0 0 0 0 0 0
    0 1 0 0 1 0 0 1 0 1 0 0 1 0 0 1 0 1 0 0 0 1 0 0 1 0 0 1 0 1 0 0 0 1 0 0 1 0 1 1 1 0 1 1 0 1 0 1 0 1 0 1 0 1 1 1 1 0 1 1 1 0 0 0 1 1 0 0 0 0 0 0
    0 1 1 0 0 0 0 1 0 1 1 0 0 0 0 1 0 0 1 0 0 0 1 0 0 0 0 1 0 0 1 0 0 0 1 0 0 1 0 1 0 1 0 1 1 1 1 0 1 1 1 0 1 1 0 1 1 1 1 0 1 0 0 1 0 0 1 1 0 0 0 0
H = 0 0 0 1 0 1 1 0 0 0 0 1 0 1 1 0 0 0 0 1 0 0 0 1 0 1 1 0 0 0 0 1 0 0 0 1 1 1 1 0 1 1 1 0 0 1 0 1 0 1 0 1 1 1 0 1 1 1 1 0 0 0 1 0 0 0 1 1 0 0 0 0
    0 1 0 1 0 1 0 1 1 0 1 1 1 0 1 1 0 1 1 1 1 0 1 1 0 1 1 0 1 1 0 0 0 0 0 0 1 0 0 1 0 1 0 0 1 0 0 1 0 1 0 0 1 0 0 0 1 0 0 0 0 1 0 0 1 0 0 0 1 0 0 0
    1 0 1 1 1 0 1 1 0 1 0 1 0 1 0 1 0 1 1 1 1 0 1 1 1 0 0 0 1 1 0 0 0 0 0 0 0 1 0 0 1 0 0 1 0 1 0 0 1 0 0 1 0 1 0 0 0 1 0 0 1 0 0 1 0 1 0 0 0 1 0 0
    0 1 0 1 0 1 0 1 1 1 1 0 1 1 1 0 1 1 0 1 1 1 1 0 1 0 0 1 0 0 1 1 0 0 0 0 0 1 1 0 0 0 0 1 0 1 1 0 0 0 0 1 0 0 1 0 0 0 1 0 0 0 0 1 0 0 1 0 0 0 1 0
    1 1 1 0 1 1 1 0 0 1 0 1 0 1 0 1 1 1 0 1 1 1 1 0 0 0 1 0 0 0 1 1 0 0 0 0 0 0 0 1 0 1 1 0 0 0 0 1 0 1 1 0 0 0 0 1 0 0 0 1 0 1 1 0 0 0 0 1 0 0 0 1
```

Figure 4.4
Parity check matrix for Hamming (72, 64) code

A lot of study has been done in an effort to discover more efficient hardware implementations for encoding and decoding data in an error correction code. This is not the focus of this section and the next one. For the purpose of this research it is sufficient to know that such an efficient hardware encoder/decoder exists for the codes we are using. The focus of this section and the next one will be explanations of the basics of encoding and decoding linear block codes.

One way to describe an error correction code is by listing all of the vector code words. The parity check matrix is an alternative way to define a code. Figure 4.4 depicts the r by n, rank r=n-k parity check matrix for a Hamming (72,64) code. The code words of the code are the 2^k vectors that make up the null space of the parity check matrix. The null space of a matrix is all of the vectors, X, which satisfy the matrix equation $\mathbf{X}*\mathbf{H}^t=\mathbf{0}$ ($\mathbf{H}^t$ denotes the transpose of **H** and **0** represents the 1 by r zero matrix).

This leads to a very obvious and simple error detection scheme. Multiply the received vector by the transpose of the parity matrix and if the result, called the syndrome, is zero then the received word is a code word and either no errors occurred or an undetectable error occurred. The syndrome can also be used to find the error locations of errors that are correctable. There are, again, many ways to do this. One very simple approach for single error correcting codes is to note that the transpose of the syndrome for a single error matches one of the columns from the parity check matrix. The number of this matching column is the bit location in the code word that is in error. The bit in question can be flipped in order to correct the error.

4.6 Encoding

Another way of representing a code is by its k by n generator matrix. The generator matrix for the Hamming (72, 64) code is shown in Figure 4.5. The rows of the generator matrix are a set of basis vectors for the vector subspace that is the code. The 2^k code words are generated by all the linear combinations of the rows. This is why this matrix is called the generator matrix, because linear combinations of the rows produce code words. Also, a data vector is encoded in the code when a one by k data vector is multiplied by the generator matrix producing the one by n code word.

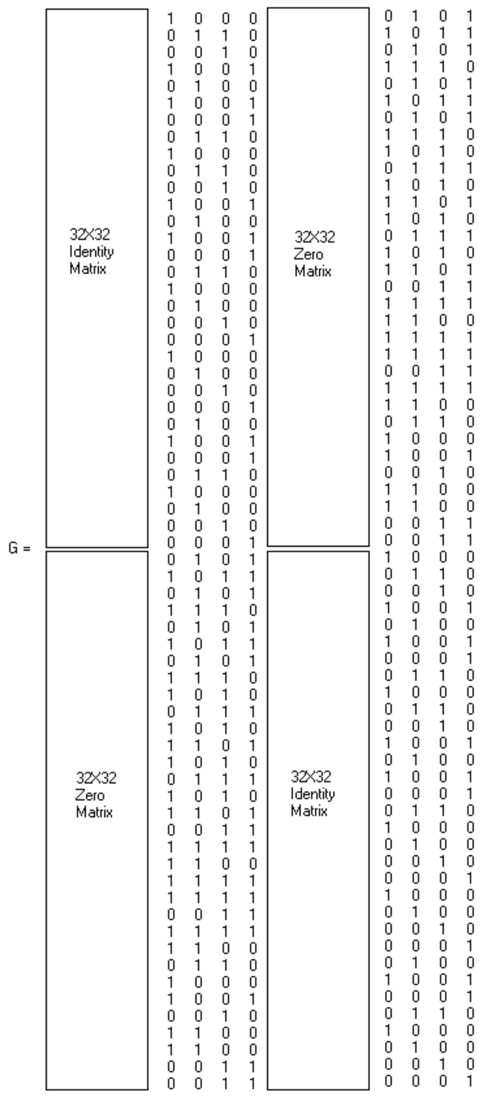

Figure 4.5
Generator matrix for Hamming (72,64) code

Every row, $\mathbf{X}_i$, of the generator matrix must satisfy the parity check equations given in the parity check matrix. Symbolically, $\mathbf{X}_i * \mathbf{H}^t = \mathbf{0}$. Since this must hold true for every row in the generator matrix, multiplying the generator matrix by the transpose of the parity check matrix must produce a k by r zero matrix. Symbolically, $\mathbf{G} * \mathbf{H}^t = \mathbf{0}$.

This leads to methods for deriving generator matrices from parity check matrices or parity check matrices from generator matrices. For systematic codes, this is trivial. Define P as the k by r matrix defined in the following equation for a systematic code $\mathbf{G}_1$: $\mathbf{G}_1 = [\mathbf{I}_k\ \mathbf{P}]$, where $\mathbf{I}_k$ is the k by k identity matrix. Then using the relationship between the generator matrix and the parity check matrix cited above, $\mathbf{G} * \mathbf{H}^t = \mathbf{0}$, it can be seen that the parity check matrix of the same systematic code defined in $\mathbf{G}_1$ can be defined by the following equation for $\mathbf{H}_1$: $\mathbf{H}_1 = [-\mathbf{P}^t\ \mathbf{I}_r]$, where $-\mathbf{P}^t$ is the additive inverse of $\mathbf{P}^t$ and $\mathbf{I}_r$ is the r by r identity matrix.

Since every code is equivalent to a systematic code, this method can be used to find the parity check matrix from the generator matrix or the generator matrix from the parity check matrix for any code. A non-systematic code would need to be multiplied by a permutation matrix to produce the equivalent systematic code. Then the unknown matrix could be generated from the first by the above procedure. Finally this matrix would need to be permuted back to the order from the original matrix.

V. Cell Replacement

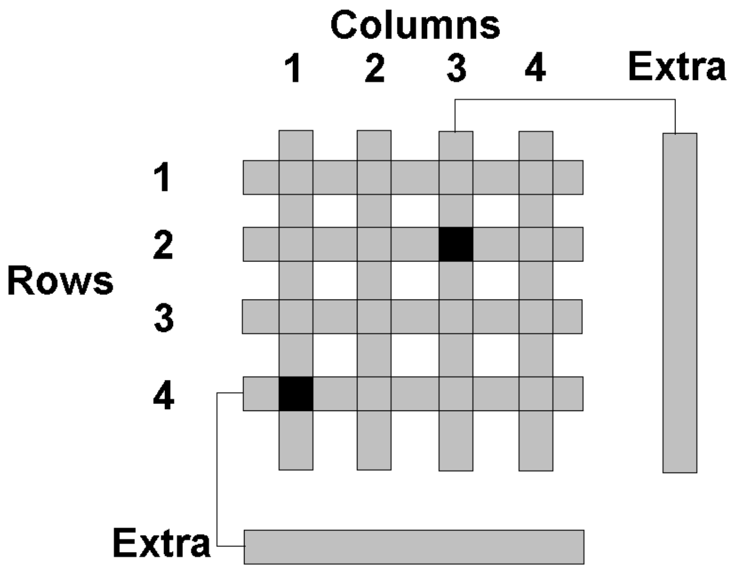

Figure 5.1
Cell replacement

5.1 Basics

As mentioned above, DRAM densities are increasing at an incredible rate. As each cell gets smaller, it becomes more difficult to make every cell perfect. Thus the probability of any particular cell being a failure increases with these density increases. In addition, the total number of cells on a chip also increases with the density. With both the probability of a failing cell and the number of cells per chip increasing, the probability of having a chip with a failing cell is increasing very rapidly. In order to combat this problem, memory manufacturers have begun building in extra, redundant rows and columns of cells that can be used to replace rows and columns that have cells that are known to fail. Figure 5.1 shows an example of how replacement would work for a very small array. Column three and Row four are being replaced using lasers to connect in the extra row and column.

A somewhat valid analogy for this is a spare tire in a car. It is completely unused unless another tire blows out, in which case the spare tire is used to replace the blown out tire. In this way the car can still serve it's original purpose using only parts that were originally made with it, even when one part fails. In the same way, there are "spare" rows and columns made within a chip that can be used to replace "blow outs".

There are, however, two things that make the analogy incomplete. First, another new tire can always be bought to fit a car. However, only rows and columns originally built within a memory chip can be used to replace broken cells. Thus if the number of rows or columns that need to be replaced exceeds the number of "spares" originally manufactured with the memory then the chip cannot be made to function properly. Second, any time in the life of the car, the spare can replace a blown out tire. However, with the way redundancy is currently implemented, this cannot be done with memory. Extra rows and columns are used to replace failing ones by using a laser beam on links to remove the defective rows or columns and connect in the extras. Because of the way this is done, the process must take place before the memory chip is packaged and sent to customers. So, only rows and columns that fail in prepackaged testing can be replaced using this method.

Thus this method is not a full proof way to manufacture functional memory. However it can increase the percentage (also known as yield) of the memory manufactured which can be made to function properly. This increased yield leads to a greater profit for the particular process because more of the product can be sold.

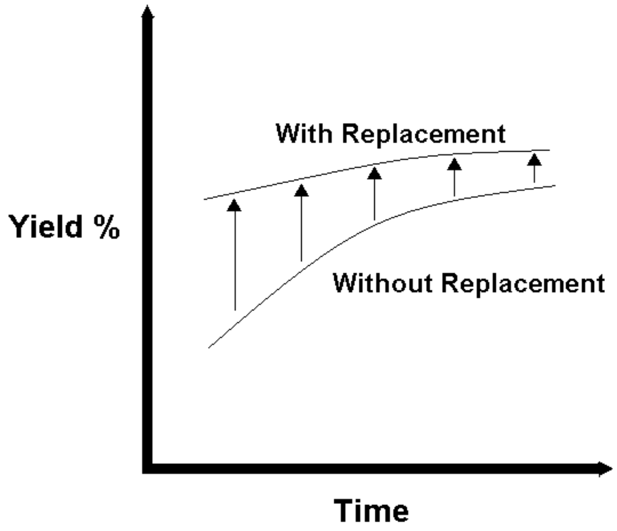

Figure 5.2
The yield advantage of using cell replacement throughout the life of a process.

The impact of this method is even greater when it is taken into account that process technology changes rapidly. Every time a new process is utilized, yields are low until the process can be refined. This replacement method can be used to decrease this initial poor yield problem, increase profits over the life of a process, and enable much greater profits in the early life of the process. Figure 5.2 shows an example of how the yield curve might be flattened out by using cell replacement. This can happen if a significant percentage of the failures in early life can be corrected with cell replacement. Also, more cell redundancy can be built into the DRAM in the early life of a new process in order to increase yield in the early life. As the process gets refined, the amount of redundancy can be reduced. This would enable more chips to fit on a wafer, because of the reduced size, and lead to a higher number of saleable modules at only marginal extra cost.

This initial poor yield problems can be seen in the prices of DRAM over the life of a process. Figure 3.1 from section 3.2 shows a drop in the constant dollar price by a factor of 8 to 10 over the lifetime of a generation of DRAM. This consistent drop in the price over the lifetime of a generation of DRAM can be attributed to a number of factors including upfront costs of manufacturing equipment for new generations, but low yields is a large factor. If this problem could be reduced as discussed in the previous paragraph, both consumers and manufacturers could benefit. Consumers could see lower initial prices on new technology as well as lower average prices. While manufacturers could see their profit margins increase, even with reduced prices, because of the greater yields distributing fixed cost over more modules.

5.2 Implementation

As stated previously, current implementations of cell replacement use lasers to blow fuses before the DRAM is packaged. This implementation is a sufficient, cost effective method for implementing cell replacement for failures that are discovered in pre-packaged testing. However, all failures are not found in pre-packaged testing. Some error patterns just aren't found during this testing. Also, failures can occur in later life or get induced before the module makes it through packaging. In addition soft errors can be induced by alpha particles hitting the DRAM at any time. These errors are transient but if they line up with existing hard errors that were not found in pre-packaged testing, then they can go undetected by an error code being used to compensate for these soft errors. For these reasons, and many others stated previously, an implementation of cell replacement that could work in the later life of the DRAM is desirable.

In section II, two implementations were proposed that would allow latter life cell replacement. One of the solutions proposed was to have the memory controller keep up with the addresses that had failed and discontinue using them, replacing them with redundant cells at addresses that were not being used. Upon deeper consideration, this method is not at all applicable to DRAM being used for personal computer main memory.

The first problem with this implementation is the amount of storage space it would require to implement. Memory controllers generally don't keep much if any state. The amount of space required to implement this version of cell replacement would be ludicrous in comparison. Each failing address would need to be stored along with a replacement address to associate with the failing address. This could add up to a significant portion of the total memory.

The second problem is that this storage space would need to be fault free also. Making such a large amount of memory error free would require using identical methods and lead to the need to implement the same strategy recursively over and over again.

The third problem is speed. The speed of the storage space implementing the memory address mapping would need to be much faster than the memory in order to maintain current access speeds. Again this would be a recursive problem leading to higher and higher costs. Also, even give a much faster memory mapping table, the search for failing address mappings would require a large amount of time, reducing the speed of the system further.

The fourth problem is a lack of resolution on replacements. Currently utilized cell replacement strategies offer the ability to replace parts of code words without replacing entire addresses. This would not be possible with a scheme implemented in the memory controller because the memory controller isn't able to resolve locations any further than allowed by it's addressing scheme. It is clear that the only reasonable implementation solution for later life cell replacement is flash programmable data paths.

VI. Memory Models

There are many different and varied DRAM memory models to describe what types of errors can occur in a DRAM. These models give varying levels of abstraction away from the actual physical implementation of the DRAM memory. Memory models found in academic work often involve many types of errors and many rules for how to test for them. These various error types are attempts to model specific failure modes within memory modules while abstracting away from them and trying to cover as many possible architectures as possible. Within industry, memory models are more likely to drift to the two extremes. Either the model nearly completely describes every aspect of a particular DRAM product, or the model includes only a few types of errors that cover the basic failure modes for any DRAM no matter what the architecture. One example of typical memory error models from each of academia and industry are described in the following sections.

6.1 Academic Memory Model

The memory testing model described in this section is described in more detail in [5]. This model starts by listing the functional faults that can occur in a DRAM. These include:

1. Opens and shorts in connecting lines
2. Open decoders: The total memory cannot be truly addressed

3. Wrong access: Wrong cells are accessed
4. No access
5. Multiple access: More cells than the addressed ones are accessed
6. Crosstalk between cells
7. Stuck cell
8. Pattern sensitive interaction between cells
9. Chip select line stuck
10. Read/write line stuck
11. Data register stuck
12. Crosstalk between data lines or cells of the data register
13. Drivers stuck
14. Cell can be set to 0 but not to 1 (or vice versa)

These fault types are based on a memory model that includes an address latch, row and column decoders, refresh latch, write driver, data register, sense amplifiers, and memory cell array. While, in some situations, it might be useful to locate which of these modules an error has occurred in, most of the time errors outside of the memory array cannot be corrected. So, next, this model maps these functional errors based on the above memory module breakdown into a simpler memory model that includes an address decoder, memory cell array, and read/write logic. These new fault types are:

1. Stuck-At Faults	(SAF)
2. Transition Faults	(TF)
3. Coupling Fault	(CF)
4. Neighborhood Pattern Sensitive Faults	(NPSF)

Stuck at faults are cells where the memory always takes on 0 or all ways takes on 1 as a value. No matter what is written, the value read from the cell never changes. Transition faults are faults where a cell either cannot make a $0 \rightarrow 1$ transition ($TF\uparrow$) or cannot make a $1 \rightarrow 0$ transition ($TF\downarrow$). Coupling faults are faults where changing the value of one cell affects a single other cell value. There are two types of coupling faults: idempotent coupling faults and inversion coupling faults. Idempotent coupling faults occur when a transition in one cell forces the contents of another cell to a specific value.

Inversion coupling faults occur when a transition in one cell causes an inversion in another cell. Neighborhood pattern sensitive faults are faults where a pattern in cells in close physical proximity to a cell under test causes the cell under test to be affected. There are three types of neighborhood pattern sensitive faults: active NPSFs, passive NPSFs, and static NPSFs. Active NPSFs involve the base cell's (cell under test) contents changing because of changes in the neighborhood pattern. Passive NPSFs involve the base cell's contents not being changeable because of the neighborhood pattern. Static NPSFs involve the base cell's contents being forced to a certain state due to the neighborhood pattern.

The next element of the model is a set of tests that cover various combinations of error types from the reduced function error types. All of these tests are march tests. A march test is a sequence of march elements. A march element is a set of reads and writes to perform at every address in order from one end of the memory to the other. An example march element is: $\uparrow$(r0, w1). This means go up through the address starting at zero. At every address, first try to read 0 from the memory and then try to write 1. Here is some example pseudo code for this particular march element to more clearly illustrate exactly what the element means:

```
for i=0 to memory_size-1 do
     temp=Memory[i]
     if temp ~= 0 then
          report error
     end
     Memory[i]=1
end
```

The march tests defined in this model and their fault coverages are listed in Figure 6.1.

MATS	↑(w0); ↑(r, w1); ↑(r)	4n or 4*2N	SAF, some AF
MATS+	↑(w0); ↑(r, w1); ↓(r, w0)	5n or 5*2N	SAF, AF
MATS++	↑(w0); ↑(r, w1, r);	7n or 7*2N	SAF, TF, AF
	↓(r, w0, r)		
March C	↑(r, w1); ↑(r, w0); ↑(r);	11n	SAF, TF, AF, Unlinked CF
	↓(r, w1); ↓ (r, w0); ↓ (r)		
March A	↑(r, w1, w0, w1); ↑(r, w0, w1);	15n	SAF, AF, linked CF
	↓(r, w0, w1, w0); ↓ (r, w1, w0)		
March B	↑(r, w1, r, w0, r, w1); ↑(r, w0, w1);	17n	SAF, AF, linked CF, linked TF
	↓(r, w0, w1, w0); ↓ (r, w1, w0)		

Figure 6.1
March Tests

6.2 Industry Memory Model

As stated in the introduction to this section, industry memory models are most often of one extreme or the other. They are either completely detailed and specific to a particular DRAM product or very general with almost no linking to any type of memory architecture. A very general model will be discussed here. One very general model involves four types of failures: single cell failure, bit line failure, word line failure, and decoder failure. These are very simple error types that stress the results of the failure rather than the cause as in the model from the previous section. A single cell failure is, as its name implies, a failure isolated to a single cell. Bit line and word line failures are failures that affect entire rows or columns within the memory's internal layout. A Decoder failure, also as its name implies, involves a failure of the decoding hardware for the memory array.

While this model is more generalized with less complex error types than the one from the previous section, it is tied more to the memory architecture than the previous model. Word line and bit line errors affect pieces of memory, whose size depends on the internal organization of the memory array. The details of how the memory array is laid out are very important to memory manufacturers. This layout determines how redundancy can be placed throughout the memory array. So, the compromise of making the model less general is one worth making for memory manufacturers.

VII. Simulation Model

A simulation tool was created in order to test the reliability achieved for different proposed error rates. Matlab was chosen as the implementation language. Conceptually DRAM memory is a large storage matrix of ones and zeros. Also, error correction is conceptually based on generator and parity check matrices. So, Matlab was chosen because of the ease of implementing matrices and the extensive collection of matrix operations.

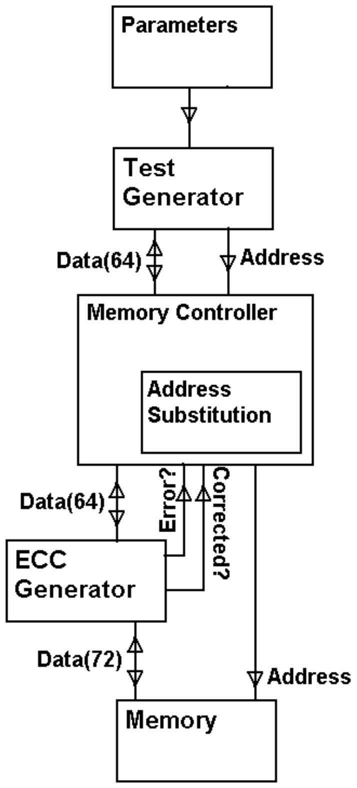

Figure 7.1:
Module Dependency Diagram

This simulation tool consists of four modules: Test Generator, Memory Controller, ECC Generator/Decoder, and Memory. The Test Generator sends test patterns into the other modules of the system. Each of the other modules performs the basic operations that their logical counterparts perform in an actual system. Figure 7.1 shows a block diagram of the interactions between the modules. Each module is a .m file, a file of Matlab scripting commands. There are also two supporting .m files: hmatrix.m and parameters.m.

7.1 Setup

Hmatrix.m defines the specific error correction code to be used. In this module the parity check matrix, or H-matrix, is specified as a matrix of zeros and ones. Matrix operations are then performed on the parity check matrix in order to derive the generator matrix. First, if the code is not a systematic code a permutation matrix is defined which will transform the code into an equivalent systematic code. The check bit portion of the equivalent systematic code, generated by multiplying the original parity matrix by the permutation matrix, is then separated from the rest of the code. Next the transpose of this part of the systematic parity matrix is taken. An identity matrix, which is the number of information bits by the number of information bits in size, is then joined with the resulting matrix from the previous transpose operation. This combined matrix is the generator matrix for the equivalent systematic code that was derived from the original code. A second permutation matrix is generated to transform this generator matrix into an equivalent generator matrix for the original non-systematic code. Finally the generator matrix is multiplied by the permutation matrix to produce the generator matrix for the original code. The error correction code to be used is now fully defined by the generator matrix, G, and the parity matrix, H.

The definition of the error correction code in a separate m file was done in order to be able to substitute different codes into the simulation tool without any re-coding. Unfortunately the decoding method for different error correction coverages isn't exactly the same. So, some amount of re-coding does have to be done in the ECC decoder when using different codes. However, this attempt at modularity was at least partially successful in that the generation process for different error correction codes is exactly the same.

Parameter.m is another attempt at modularizing the program to allow different configurations to be run without having to change any of the other modules. First parameter.m clears the memory so that all of the memory is available to model the largest size memory possible. Next, the error correction code (which is a version of hmatrix.m) to be used is defined. A variable is set to determine whether replacement is used or not. Variables that define the architecture of the memory module are set. These variables include the number of chips and the number of sections per chip. Modeling the memory as a number of chips with a number of sections allows the redundancy to be modeled as extra rows within each section. DRAM devices are generally laid out as roughly square sections of cells grouped together into sections where bit lines or word lines might fail. However, the exact architecture varies greatly between DRAM modules. So this model can accurately describe the features that are common between most DRAM modules. Next, which test algorithm(s) to use is defined. The size of the memory and the amount of redundant cells available for replacement is defined. The error rates are defined. Finally the test generator is started to run a simulation with the desired parameters.

Defining these things in a parameters file allows for many test cases to be run without changing any of the codes from the actual modules of the memory. You can set the error coverage to be used. Determine whether replacement will be used or not. Determine exactly what areas of the memory have what amounts of replacement. Define the total size of the memory, set which error rates will be used, and pick from a set of standard test algorithms to run on the memory configuration.

7.2 Test Generator

The test generator module does exactly what its title implies: generates test sequences to uncover errors in the memory. First it sets up some structures to allow the functioning of the other modules. The number of address lines set in the parameters file is used to set a variable that describes how many addressable memory locations there are. If replacement is being used, a memory status array is created to keep track of when and where errors occur. The memory matrix is created. The number of replacements available per block of memory is calculated by multiplying the percentage of total memory available for replacement, defined in the parameters.m, by the number of addressable memory locations. Each section in each chip has this many replacements available because each replacement only replaces the number of bits in that section and chip. An error occurred indicator and an error count indicator are both initialized to zero.

After all of this setup, the test algorithms; designated in parameters.m; run. The available test algorithms are the same ones discussed section 6.1. However, as cited in section 6.2 the industry memory module is much simpler than a complete academic memory model. Also because of the variation in the exact memory architecture, it would not be possible to reach a general conclusion by basing simulations on any exact memory architecture, even if the more detailed academic memory models were used. For these reasons, the memory test that runs in the tests for this research is the MATS algorithm. Because of the model used to generate errors, this algorithm offers sufficient coverage to discover all errors introduced by the memory error generation model used.

7.3 Memory Controller

Memorycontroller.m is the logical equivalent of a memory controller in a real memory subsystem. It translates logical addresses into physical memory accesses. In addition to this function it implements the actual replacement of cells. In the physical world this would be equivalent to the memory controller directing the blowing of fuses within the memory to replace specific groups of cells within DRAMs. An actual system could be built this way or the logic to implement the fuse blowing functionality could be placed within the DRAM itself. Regardless of the way this is implemented, in reality this simulation model delivers the same functionality and fully tests the logical soundness of either implementation. The reason the logic was placed within the memory controller logic in the simulation model was to separate the error inducing logic, which needed to be included in the memory module, from the replacement and correction logic.

In this simulation model the memory controller module takes in an address, a word of data, and a read or write command from the test generator module. It then tests whether replacement is being used or not. If replacement is not being used it merely forwards the read or write command, with the correct physical address, to the error correction module so that the data can be coded for a write operation or decoded for a read operation.

If replacement is being used, the memory controller module first tests whether a write or a read is being performed. If the operation is a write, it again merely forwards the operation on to the error correction module with the correct physical address. If the operation is a read, it still forwards the command in the same way to the error correction module. The error correction module requests the data from the memory and decodes it. After the data is returned, some extra functionality is implemented in the memory controller module. First it tests to determine if a detectable error occurred. A detectable error is one that the error correction code currently being implemented has coverage to detect. The error correction module sends back a flag to the memory controller module that alerts the memory controller module that a detectable error has or has not occurred. If no detectable error occurred then the data is sent back to the test generator module and control is returned to that module to determine if any undetectable error has occurred. If a detectaible error did occur, the memory controller module first determines which chip and then which section within that chip the error occurred. The error correction module passes back the bit number that failed and this is then used to determine within which chip and section number the bit is located. Once this is done the memory controller module tests whether there are any replacements left within the particular chip and section where the error occurred. If there is a replacement available, a replacement operation is sent to the memory with the correct chip and section, and the number of replacements used for that section and chip number is incremented by one. Finally control and the data is returned to the test generator.

The replacement logic is implemented in the memory controller be means of an array that holds the number of replacements that have already been used. The dimensions of this array are the number of chips by the number of sections so that the number of replacements can be tracked by where actual replacements exist within the DRAM. This array is tested and updated within the memory controller module. For the same reasons discussed in section 5.3 in relation to keeping up with address mapping replacements in the memory controller, this would not be a realistic way to actually implement this functionality. A fairly large amount of storage would be needed within the memory controller, which also would need to be error free for the scheme to work. The process of making that memory error free would add an extra level of complexity and would be an inefficient way to implement the functionality. A memory system implemented this way would not be able to operate at the speeds demanded of memory subsystems. In a real memory a single replacement request could be sent to the memory, which would test for available replacements and return a success or failure status based on whether there were replacements available. However, the way it is actually implemented in this simulation model is logically equivalent and will still provide complete testing coverage equivalent to the way an actual physical implementation would be tested. The reason this was separated out in the simulation module rather than exactly matching the way a physical implementation would be implemented, was, again, to separate the error introducing logic in the memory module from the memory correction logic in the error correction module, and the replacement logic in the memory controller.

7.4 ECC generator/decoder

The ECC generator/decoder module, eccgenerator.m, encodes and decodes the data into the error correction code being used. As discussed earlier, this module has to be modified to function differently depending on the error correction code being used. The simulations discussed in this paper are limited to one error correction code. This implementation is sufficient while only one code is being used for simulation. However, if more, or other, codes are used this solution isn't very scalable or modular. For each new code, modification has to occur. A better original design might have been to define the code for decoding in the hmatrix.m module, which defines the error correction code, and call it from the ECC module. However, this was not an anticipated complication, so the current implementation is a reasonable solution to resolve the problem within the framework in which it is currently being used.

Within the ECC generator/decoder module, the first thing that happens is a test for whether the operation is a write or a read. If the operation is a write, the data vector is multiplied by the generator matrix modulo two. The modulo two is necessary because the Matlab matrix multiplication function is a base 10 operation, but what is needed is a binary multiply. Simply taking modulo two of the result is sufficient because all of the multiples in the matrix multiply operation are one times zero, zero times zero, or zero times one. The results of all of the multiplications are still binary numbers. The problem is introduced when the numbers are added together. However, the sum of ones and zeros in base 10 taken modulo two is the same as the binary sum of the same numbers. So taking the result of multiplying the data vector by the generator matrix modulo two is sufficient to encode the data in the error correction code. Next the coded data is passed along to the memory module to be stored in the memory.

If the memory operation is a read, the memory module is accessed first to retrieve the data for the address being read. Next a syndrome is produced. Multiplying the retrieved coded vector from the memory by the parity check matrix modulo two produces this syndrome. The matrix multiplication modulo two works in this situation for the same reasons described above. Next the syndrome is tested to determine if any detectable errors occurred. If the syndrome is composed of all zeros then no detectable errors occurred. In this case, the data bits are picked out of the coded word to produce the decoded data. Control is then returned to the memory controller module along with the data produced. If the syndrome is not composed of all zeros then a detectable error has occurred. If this happens then the syndrome is compared to each column of the parity matrix. If it matches any of the columns then a single bit error has occurred and the bit location of the error is the same as the column number that was matched. If a match is found the digit at the bit position that was matched is flipped to make the data word correct. A variable is then set to indicate that a correctable error occurred. A variable is set indicating which bit was corrected so that the information can be passed back to the memory controller for possible substitution in the memory. If no match is found on the syndrome then a detectable, non-correctable error has occurred. This information is passed back to the memory controller by the combination of the erroroccurred indicator and the errorcorrected indicator. If erroroccurred is one and errorcorrected is zero then a detectable, non-correctable error has occurred. Regardless of whether the error was correctable or not, the bits which should represent the data word are now picked out of the coded data word to be returned to the memory controller.

7.5 Memory

The memory module is where the actual matrix that represents the storage array of the memory is maintained. Assignments and lookups are performed within this module. In addition to serving as the logical equivalent of a physical memory, the intentional introduction of errors occurs within this module. Errors are introduced into the memory with the probability specified in the parameters module. Stuck at bits are also accounted for in the memory so that the bit remains stuck at the same value between accesses unless the memory location is replaced. Stuck bits are simply represented by a two or a three in the memory matrix. The Matlab matrix is not restricted to binary numbers so an error can be represented by any value in the array that is not a one or a zero. Twos represent bits stuck at one and threes represent bits stuck at zero. The write operation in the memory is coded to preserve these twos and threes when new data is written to a location with stuck at bits. The read operation is coded to translate the twos and threes into ones and zeros when the data is sent back to the ECC decoder. The replacement operation is coded to eliminate the twos and threes from the location being replaced. The number of replacements left for a particular section is maintained in the memory controller module, so any replacement request sent to the memory by the memory controller is assumed to have replacements available. Once the maximum number of replacements have been reached for a section, the memory controller will stop requesting replacements for that section.

First the memory module tests whether the access is a read or a write. If the operation is a write, then errors are randomly introduced into the data before it is stored. This is done by creating a vector of random numbers that is the length of a code word. A call to the Matlab rand function generates the random number vector. This produces a vector of numbers between zero and one. Next, ½ plus the error rate percentage is subtracted from all of the elements of the random number vector. This shifts the center of the random function. Now only numbers that are less than the error rate percentage will be more negative than –½. If all of the elements are then rounded, everything will round to zero except for numbers which were originally smaller than the error probability, which will round to -1. Since the numbers are uniformly distributed these numbers should occur with the same probability as specified by the error probability. So the error vector is now a vector of zeros and negative ones, which occur with the error rate probability. Next this vector is multiplied by negative two. This produces a vector of zeros and twos, which are distributed in the same way as the negative ones. This vector is then added to the data vector to induce errors. If no errors (twos) were present in the error vector then the data vector will be unchanged. However, if errors were present, they cause the bit that they match up with to flip. This is because zero plus two is two, which represents a bit stuck at one in the memory, and one plus two equals three, which represents a bit stuck at zero in the memory. After the error injection is done, the address in the memory matrix is checked to determine if there are any existing stuck at bits in that location. If there are any pre-existing stuck at bits in the memory they are copied into the data word about to be written. The new data word is then copied into the correct location in the memory regardless of the pre-existing or induced errors.

If the memory operation is a read then the contents of the memory at the current physical address is put into a data word. Next, the data word is checked for stuck at bits so that they can be translated into their binary equivalents before they are passed back to the ECC generator. Zeros are substituted for threes and ones are substituted for twos.

If the memory operation is a replacement, the memory status is reset to indicate that no errors are present in that memory location. Next the stuck at bits are taken out of the memory array for that physical address, chip number, and section number. Twos are replaced in the memory by ones and threes are replaced in the memory by zeros.

VIII. Conclusions

The simulation module described above was run over a range of error rates. For the purpose of the simulations the error rate was defined as the probability that any single address location has one or more failing bits. Error rates ranging from .1% to 50% were tested. Figure 8.1 shows the total number of errors that occurred versus the error probability. This is roughly a direct proportionality relationship as expected. The total number of errors should vary directly with the error rate.

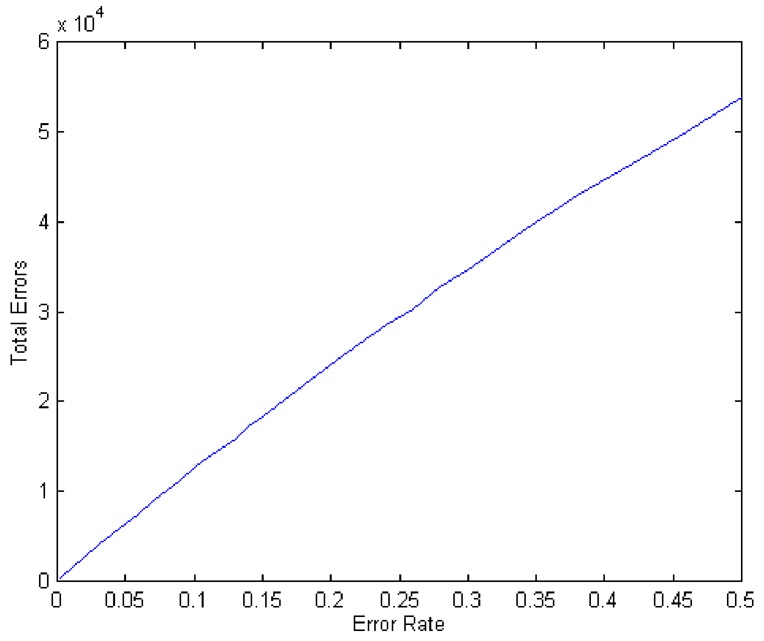

Figure 8.1
Total Errors

Figure 8.2 shows the number of errors corrected versus the error rate. This graph also shows roughly a direct proportionality relationship. However, the curve is a little more concave than the total errors graph, indicating that a higher percentage of errors were corrected at a lower error rate. This can be seen even more clearly in Figure 8.3, which graphs the number of errors that show up outside the memory. Since this number is the only other component of the total number of errors and the graph of it is convex, the correctable graph must be more concave than the total errors graph. This also makes sense, because at a lower error rate, there is a lower probability of more than one bit failing at a time. Also, it should be noted that the corrected errors make up the largest portion of the total errors. Even at the highest error rate, the total number of errors is about $5.4*10^4$, while the number of corrected errors is about $4.6*10^4$, or greater than 80% of the total.

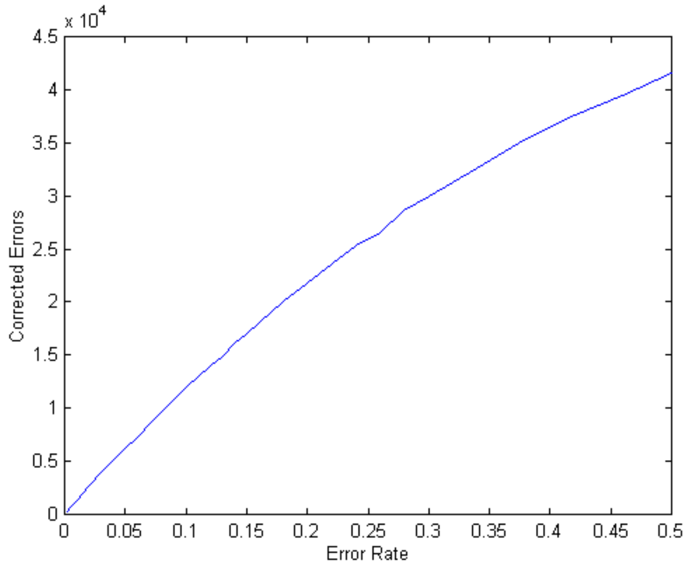

Figure 8.2
Errors Corrected

Finally, figure 8.4 shows the number of errors that were neither correctable nor detectible. This graph shows an obvious sharp descent as the error rate goes down. Even at a 15% error rate, virtually no errors are able to make it through all of the error correction methods. This is a promising result, which upon further, more detailed study might produce a better solution to memory manufacturing problems this research has discussed.

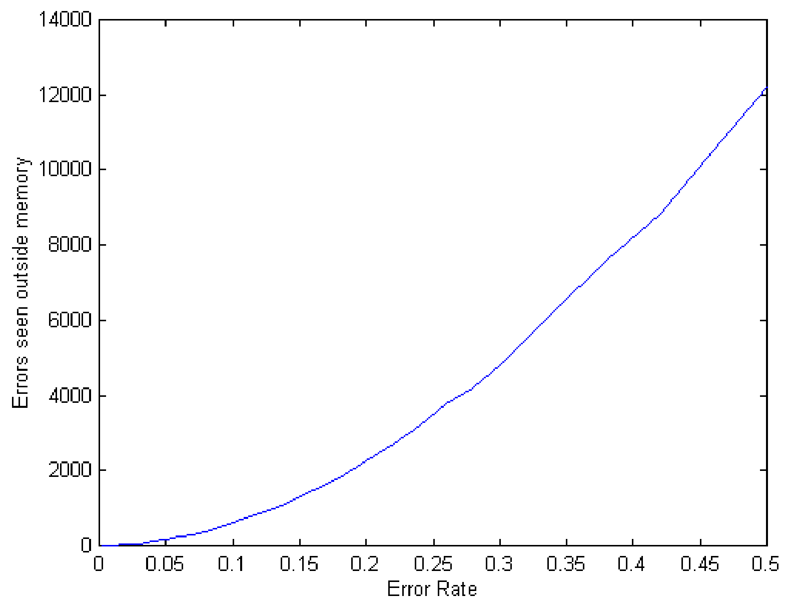

Figure 8.3

Errors found outside the memory

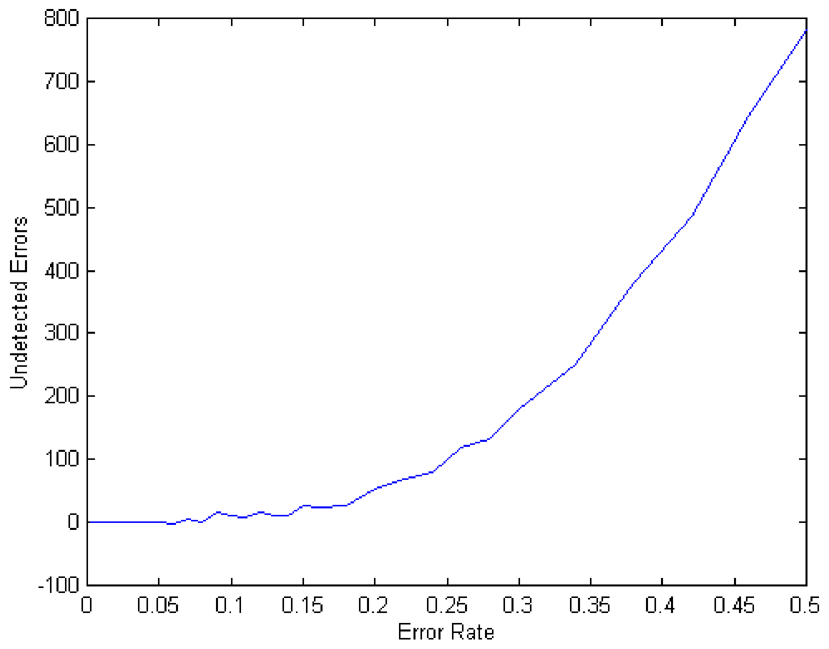

Figure 8.4
Undetectable Errors

References

[1] Hennessy, John L. and Patterson, David A., Computer Architecture a Quantitative Approach, Second Edition. San Francisco, CA: Morgan Kaufmann Publishers, Inc. 1996.

[2] Gagne, Cyrille L., “Memory Cards: To Test of Not to Test?,” MicroNews, 1998 Vol.4, No. 4, p.9-11.

[3] Imai, Hideki, Essentials of Error-Control Coding Techniques. San Diego, CA: Academic Press, Inc. 1990.

[4] Hsiao, M. Y., Bossen, D. C., and Chien, R. T., “Orthogonal Latin square codes” IBM J. Res. Dev. 14, 1970, p.390-394.

[5] Van De Goor, A. J., and Verruijt, C. A., “An Overview of Deterministic Functional RAM Chip Testing” ACM Computing Surveys, Vol. 22, No.1, March 1990.

[6] Poli, Alain and Huguet, Llorenc, Error Correcting Codes Theory and Applications. Hertfordshire: Prentice Hall International Ltd. 1992.

[7] Peterson, W. Wesley, Error Correcting Codes. Cambridge, MA: MIT Press. 1961.

[8] Pless, Vera, Introduction to the Theory of Error-Correcting Codes. New York. 1989.

[9] Wakerly, John, Error Detecting Codes, Self-Checking Circuits and Applications. New York: North-Holland. 1978.

[10] Vanstone, Scott A. and van Oorschot, Paul C., An Introduction to Error Correcting Codes with Applications. Boston, MA: Kluwer Academic Publishers. 1989.