ERROR CONTROL CODING FOR SEMICONDUCTOR MEMORIES

A THESIS SUBMITTED TO
LAKEHEAD UNIVERSITY
IN PARTIAL FULFILMENT OF REQUIREMENTS
FOR THE DEGREE OF
MASTER OF SCIENCE

BY
HAO BAOMING ©

1993

ISBN 0-315-86154-1

ERROR CONTROL CODING FOR SEMICONDUCTOR MEMORIES

Abstract

All modern computers have memories built from VLSI RAM chips. Individually, these devices are highly reliable and any single chip may perform for decades before failing. However, when many of the chips are combined in a single memory, the time that at least one of them fails could decrease to mere few hours. The presence of the failed chips causes errors when binary data are stored in and read out from the memory. As a consequence the reliability of the computer memories degrade. These errors are classified into hard errors and soft errors. These can also be termed as permanent and temporary errors respectively.

In some situations errors may show up as random errors, in which both 1-to-0 errors and 0-to-1 errors occur randomly in a memory word. In other situations the most likely errors are unidirectional errors in which 1-to-0 errors or 0-to-1 errors may occur but not both of them in one particular memory word.

To achieve a high speed and highly reliable computer, we need large capacity memory. Unfortunately, with high density of semiconductor cells in memory, the error rate increases dramatically. Especially, the VLSI RAMs suffer from soft errors caused by alpha-particle radiation. Thus the reliability of computer could become unacceptable without error reducing schemes. In practice several schemes to reduce the effects of the memory errors were commonly used. But most of them are valid only for hard

errors. As an efficient and economical method, error control coding can be used to overcome both hard and soft errors. Therefore it is becoming a widely used scheme in computer industry today.

In this thesis, we discuss error control coding for semiconductor memories. The thesis consists of six chapters. Chapter one is an introduction to error detecting and correcting coding for computer memories. Firstly, semiconductor memories and their problems are discussed. Then some schemes for error reduction in computer memories are given and the advantages of using error control coding over other schemes are presented.

In chapter two, after a brief review of memory organizations, memory cells and their physical constructions and principle of storing data are described. Then we analyze mechanisms of various errors occurring in semiconductor memories so that for different errors different coding schemes could be selected.

Chapter three is devoted to the fundamental coding theory. In this chapter background on encoding and decoding algorithms are presented.

In chapter four, random error control codes are discussed. Among them error detecting codes, single error correcting/double error detecting codes and multiple error correcting codes are analyzed. By using examples, the decoding implementations for parity codes, Hamming codes, modified Hamming codes and majority logic codes are demonstrated. Also in this chapter it was shown that by combining error control coding and other schemes, the

reliability of the memory can be improved by many orders.

For unidirectional errors, we introduced unordered codes in chapter five. Two types of the unordered codes are discussed. They are systematic and nonsystematic unordered codes. Both of them are very powerful for unidirectional error detection. As an example of optimal nonsystematic unordered code, an efficient balanced code are analyzed. Then as an example of systematic unordered codes Berger codes are analyzed. Considering the fact that in practice random errors still may occur in unidirectional error memories, some recently developed t-random error correcting/all unidirectional error detecting codes are introduced. Illustrative examples are also included to facilitate the explanation.

Chapter six is the conclusions of the thesis.

The whole thesis is oriented to the applications of error control coding for semiconductor memories. Most of the codes discussed in the thesis are widely used in practice. Through the thesis we attempt to provide a review of coding in computer memories and emphasize the advantage of coding. It is obvious that with the requirement of higher speed and higher capacity semiconductor memories, error control coding will play even more important role in the future.

ACKNOWLEDGEMENT

I would like to take this opportunity to express my sincere thanks to my supervisor Dr. M. H. Kkan for his encouragement and support throughout the course of this program.

Thanks to Dr. Hasegawa for his invaluable advice.

Also I want to thank my family for their moral and patient support and understanding during the years of my study.

ERROR CONTROL CODING FOR SEMICONDUCTOR MEMORIES

Chapter One Introduction

1.1 Computer and Its Memory 1

1.2 Coding for Computer Memories 7

1.3 Summary 12

References 13

Chapter Two Semiconductor Memories

2.1 Memory Cells 17

2.2 Memory Organizations 31

2.3 Errors in Semiconductor Memories 41

2.4 Summary 48

References 49

Chapter Three Linear codes

3.1 Basic Concepts of Linear Codes 51

3.2 Hamming Codes 63

3.3 Cyclic Codes and BCH Codes 67

3.4 Summary 76

References 79

Chapter Four Codes Used in Computer Memory

4.1 Criteria for Code Selection 80

4.2 Error Detection Codes 83

4.3 SEC-DED Codes 96

4.4 Multiple Error Correction Codes 118

4.5 Erasures 133

4.6 Summary 134

References 137

Chapter Five Unidirectional Error Correction Codes

5.1 Unidirectional Error Detecting Codes 142

5.2 tEC-AUED Codes 150

5.3 Summary 158

References 161

Chapter Six Conclusions 163

CHAPTER ONE INTRODUCTIONS

Development in computer and its application has progressed rapidly during the last decades. Computer of today are much better in their performance and cheaper in cost. These have been achieved through dramatic improvement in hardware manufacturing as well as development of sophisticated software.

Reliability of the computers has also become a major issue as its applications become widespread. These criteria among other things require efficient and error free digital transmission and storage systems in computers. Computer memories, as one of the main subsystems of computer are playing a significant role towards computer's reliability and performance. As the memory system becomes larger, failures, including hardware and software failures, influence the behaviours of computer more seriously than ever.

1.1 Computer and Its Memory

A block diagram of a digital computer is shown in Fig. 1.1. In this diagram there are four basic units, the input unit, central processing unit (CPU), main memory and output unit.

The input unit enables operator to feed in the information data and instructions to the computer. The output unit allows the results of computing to be sent outside . Both input and output unit are equipments that interface with outside. CPU, which consists of control unit and arithmetic unit, is the heart of

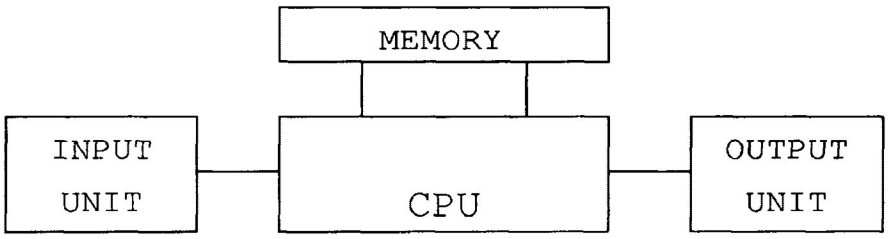


Fig.1.1 A bock diagram of a digital compute

computer. CPU controls flows of data and instructions between different parts of computer and processes these data such that computer can accomplish various functions. Because the rate of data transfer of the input unit is generally slow compared to the processing speed of CPU. It is necessary to hold the data and instructions in a place for immediate use which allows fast access to the stored data. This storage place is called main memory.

During operations, CPU directs the input information data to memory and reads them out as needed and applies arithmetic operations such as addition, subtraction, multiplication and division to the information data, and then, after being processed, the immediate results or final results are returned to the memory by CPU for preparation for output.

It can be seen that large amount of data communications between subsystems takes place in computer. Among these communications, the data traffic between CPU and main memory is of the highest rate. For example, for a high speed computer it may be of the order of 100 million bits every second[1.1]. A large computer memory stores more data and allows data communication between memory and CPU to be faster. Therefore a large main memory is essential for high speed computer systems.

High density memories and their problems

As a result of the greater need for storage capacity and speed, computer memories are becoming high density and high speed. This increased memory density has generally been achieved through

a reduction of storage cell size. Nowadays the memory chips containing 1M-bit are quite common. Table 1.1 [1.2] gives a view of the progress of densities of semiconductor DRAMs (Dynamic RAM, which is one type of the semiconductor memories. In chapter two various other semiconductor memories will be discussed). It shows that DRAM size is being quadrupled in about every two to four years in the past 20 years.

However, extremely small cell size and complexity of VLSI circuitry are more vulnerable to manufacturing failures and various interferences, especially alpha-particle radiation etc. This has largely increased the probability of failures in semiconductor memories. This in turn has increased error rates in computer systems and obstructed the progress of even higher density and higher speed memories.

The errors in semiconductor memories can be basically divided into two types: hard errors and soft errors. A hard error occurs when a memory location of hardware becomes permanently defective. It is an irreversible error caused by connection failures like internally shorted or open leads. Soft errors are temporary and random in time and locations. They may occur during one particular memory cycle time but disappear in the next cycle. The soft errors result from system noise, power surges, atmospheric interference and alpha-particle radiation[1.3].

Table 1.1 Storage geometry parameters in RAM

memory size (bits)	storage area (cm^2)	cell area (μ^2)	year
4K	0.07	1764	1973
16K	0.1	800	1976
64K	0.15	216	1978
256K	0.3	96	1982
1M	0.2	20	1984
4M	0.3	9	1986
16M	0.3	1.5	1987
64M	0.6	0.7	

Some schemes for increasing the reliability of memory system

It is important that the memory system enables to detect and correct errors as and when they occur. Otherwise the errors will lead to incorrect computation or even serious malfunction of the system operations.

There have been several approaches to reduce or overcome the consequences of errors in computer memories. Some of them are discussed as follows:

a) Memory Organizations Scheme[1.4, 1.5, 1.6]

For some hard errors, proper memory organizations can be used to limit the number of errors within a memory word or to disperse errors into single error per word so that simple error correcting scheme is used effectively. For example, one-bit-per-chip organization is one of such designs. In this organization every bit of a word is stored in a different memory chip. A word of n bits then is stored in n chips in the memory. As a result, when a whole chip in the memory fails, it can affect, at most, only one bit of the word.

b) Hardware Redundancy Scheme[1.7]

In this scheme, some amount of spare hardware components or memory cells are provided in memory chips during its fabrication. Whenever a memory cell is found permanently defective, the spare cell is automatically switched in to replace the defective ones. This scheme greatly increases the system reliability but causes a low efficiency in storage area usage.

c) Hardware Maintenance Strategy[1.8]

In a system environment, another option for memory reliability is the system maintenance strategy. This strategy allows memory to accumulate certain correctable failures in memory until they reach a threshold which is intolerable to the computer memory. Then the faulty memory chips are physically replaced. Such a substitution strategy is scheduled periodically during service time of computer.

d) Error Control Coding (ECC) Scheme[1.1, 1.9, 1.10, 1.11]

All methods mentioned above are valid only for hard errors. By adding some redundancy bits to information data, the scheme of ECC enables to combat both hard and soft errors occurring in memory words. By using the encoding and decoding logic, the ECC scheme enable to detect and even to correct errors as they occurred.

Because of its high efficiency, ECC along with some other techniques mentioned above produces a versatile and robust scheme to improve the reliability of semiconductor memories. Therefore they are becoming quite common features in modern high performance computer systems.

1.2 Coding for computer memories

As an example, a simple illustration is shown in Figure 1.2 to explain the concept of memory with ECC technique.

Let us assume that the possible information data from input

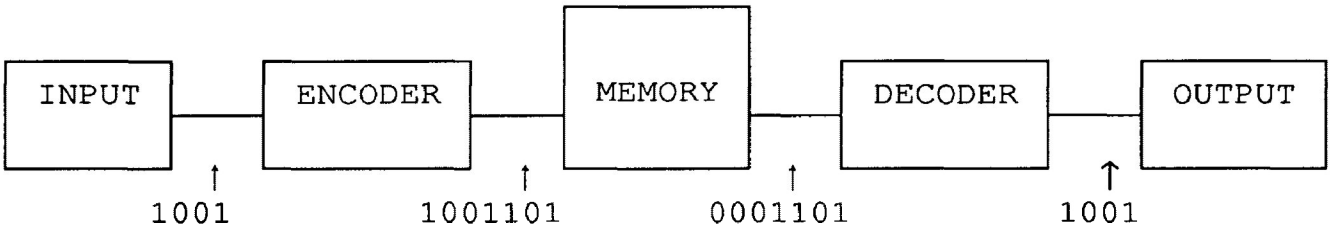


Fig.1.2 Simple illustration of a memory system with EDC scheme

(to be stored in the memory) are 1001. Instead of being stored in memory directly, the data is firstly sent to the encoder where, according to certain coding rules, some redundancy digits 101 are formed and added to the information data 1001. This process is called encoding and the new data which contain information digits and redundancy digits are called encoded word or codeword while the redundancy digits 101 are usually called parity check bits or check bits. Then the encoded data are sent to memory and stored there as 1001101.

Due to some faults in the memory, the codeword may be corrupted to an erroneous word. For example, suppose at the first position of the codeword, 1 is changed to 0 such that the actual codeword stored in the memory becomes 0001101 which differs from the original codeword.

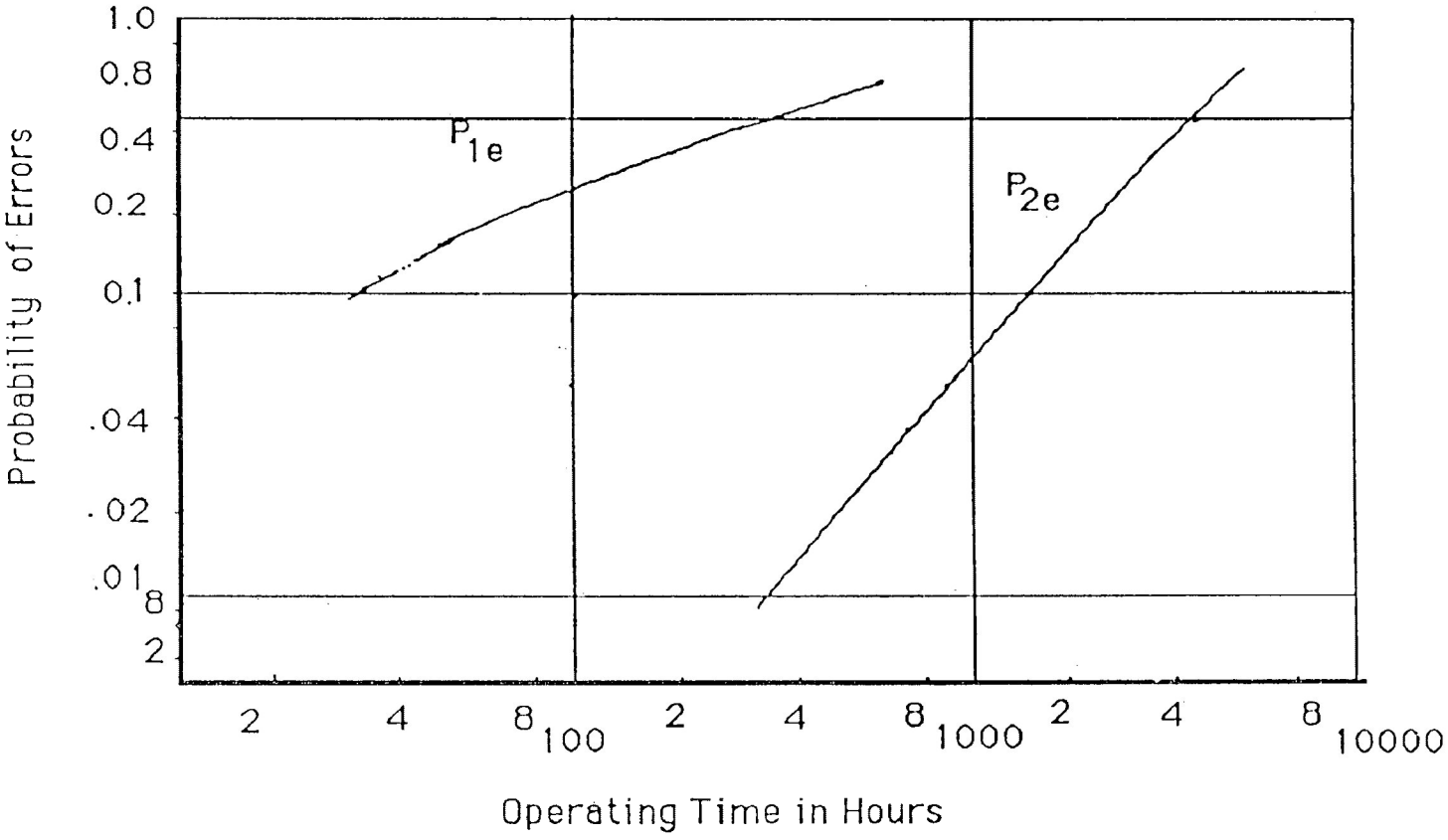
However, when the codeword is read out from the memory, the decoder will make it sure that the information portion of the codeword is 0001. Following the same rule as in the encoder, the decoder will regenerate new check bits, say 100, and compare them with the old check bits 101. If there is no differences between them, the codeword from the memory is accepted as correct. Otherwise error(s) is detected. By comparing the differences, the decoder will also be able to locate the position of the error and then correct it. As a result, after decoder, the original information data 1001 is recovered and sent to its destination.

By proper design, ECC coding enables to detect and correct either single error or multiple errors in a codeword [1.11].

Generally, most coding schemes do not require very complicated computations and implementations. Therefore it has found wide use in computer memories to improve the performance and reliability.

In early computers, for example, the IBM/650, UNIVAC and the Whirlwind computers, the simplest ECC schemes, single error detecting codes were used to enhance memory system reliability[1.11]. These codes added a single parity bit to the information data bits for error detection only. Today many different ECC codes are implemented in computer industry worldwide. The most common codes are Hamming codes[1.10], modified Hamming codes[1.12] and unidirectional error correcting codes[1.13, 1.14, 1.15]. Some VLSI chips which support modified Hamming codes are commercially available (e.g. SN54/74LS630)[1.16]. These chips are used externally to the memory while designing computer systems. In other ECC systems error correcting codes are implemented on memory chips [1.17].

Use of ECC enables modern semiconductor memory to maintain the advantage of low cost, low power, high density and high speed while still achieving acceptable level of memory reliability. Figure 1.3 [1.18] provides a comparison between memories with ECC and without ECC with respect to the operating hours under certain error probability. In the figure, P_{1e} represents the error probability for the memory without ECC and P_{2e} for the memory with ECC. As can be seen, a 32-bit, 64k word memory without ECC P_{1e} will reach 50% in 350 hours of operation. While for the memory with ECC, which can correct any single error within the 32-bit word, at the same error probability the operating time is extended up to 4500 hours.



P_{1e} : The probability of errors of the memory without ECC

P_{2e} : The probability of errors of the memory with ECC

Note: Only memory failures are considered here.

Fig. 1.3 Probability of errors for 32-bit, 64K word memory[18]

1.3 Summary

Main memory systems are of significance for computer's performance and reliability. Due to its fast development of VLSI and computer applications, semiconductor memories are becoming larger (in capacity) and faster. Meanwhile the extremely high density memory products suffer from various errors including hard errors and soft errors. To minimize the consequence of these errors, several schemes are practically implemented in main memory systems.

Among those schemes mentioned above, ECC is one of the most effective techniques. ECC is the art of adding redundancy effectively so that most messages, if corrupted, can be detected or recovered correctly. Combined with some other schemes, ECC coding has shown dramatic improvements in computer memory reliability.

Throughout the chapters of this thesis, based on considerations mentioned above, several ECC codes for computer memories are discussed. The main attention will be concentrated on semiconductor main memories. In the following, when memories are mentioned, they refer to semiconductor main memories.

References

1.1 T. R. N. Rao, E. Fujiwara, "Error-control Coding for Computer Systems" Prentice-Hall Inc., 1989

1.2 Pinaki Mazumder, Janak K. Patel, "Parallel Testing for Pattern-Sensitive Faults in Semiconductor Random-Access Memories" IEEE Transactions on Computers, No. 3, March 1989 pp 394

1.3 Suneel Rajpal, John. Mick, "Fast Error-correcting ICs Aid Large Memory System" Electronic Design, Feb. 1987, pp 123-126

1.4 Richard E. Matick "Computer Storage & Technology" A Wiley & Sons 1977

1.5 S. Middelhock, P. Gorge, P. Deker, "Physics of Computer Memory Devices" Academic Press, 1976

1.6 Chitoor V. Srinivsan, "Code for Error Correction in High-speed Memory Systems--Part I: Correction of Cell Defects in Integrated Memories" IEEE Transaction on Computers, vol. c-20, No. 8, Aug. 1971, pp 882-888

1.7 Ramachanda P. Kunda, Bharat Deep Rathi, "Improving Memory Subsystem Availability Using BIST" IEEE International

Conference on Computer Aided Design, ICCAD-87, Digest of Technical Papers, 1987, pp 340-343

1.8 D. C. Bossen, M. Y. Hsiao, "A System Solution to the Memory Soft Error Problem" IBM Journal of Research and Development, vol. 24, No. 3, May 1980, pp 390-397

1.9 Chen, C.L, Hsiao, M.Y "Error-correcting Code for Semiconductor Memories Application: a State-of-the-art Review" IBM J. Res. Develop., vol. 28, No. 2, Mar. 1984, pp 124-134

1.10 W. Wesley Peterson, E. J. Welson, JR. Error Correcting Codes MIT press 1986

1.11 Bary W. Johnson, "Design and Analysis of Fault Tolerant Digital System" Addison Wesley, 1989

1.12 M. Y. Hsiao, "A Class of Optimal Minimum Odd-weight-column SEC-DED Codes" IBM J. Res. and Develop. 14, (July 1970) pp 395-401

1.13 Serban D. Constantin, T. R. N. Rao, "On the Theory of Asymmetric Error Correcting Codes" Information and Control 40, 20-36 (1979) pp 20-35

1.14 Bella Bose, Thammavaram R. N. Rao, "Theory of Unidirectional Error correcting/detecting Codes" IEEE Transactions on

Computers, vol. c-31, No. 6, June 1982, pp 521-530

1.15 Dhiraj K. Pradhan, "A New Class of Error-correcting /detecting Codes for Fault-tolerant Computer Applications" IEEE Transactions on Computers, vol. c-29, No. 6, June 1980, pp 471-481

1.16 Dale Hunt, Thomas J. Tyson, "Error Detection and Correction Using SN54/74LS630 or SN54/74LS631" Microprocessors and Microsystems, vol. 13, No. 7, Sept. 1989, pp 473-480

1.17 Howard L. Kalter, "A 50-ns 16-Mb DRAM with a 10-ns Data Rate and On-Chip ECC" IEEE Journal of Solid-State Circuits, No. 5, Oct. 1990, pp 1118-1127

1.18 Len Levine, Ware Meyers, "Semiconductor Memory Reliability with Error Detecting and Correcting Codes", Computers, October 1976, pp 43-49

CHAPTER TWO SEMICONDUCTOR MEMORIES

In computer systems there are two types of memories. Main memory and secondary storage. By main memory it refers to the memory that is built-in along with the computer CPU. Main memory offers very fast access speed so that computer can take the advantage of the high processing speed of CPU.

In modern computers, semiconductor devices are widely used as main memories. Implementation of VLSI (Very Large Scale Integration) technology has allowed many fold increase in the memory capacity within a compact size.

Main memory can be classified as Read-Only-Memory (ROM) and Random-Access-Memory (RAM). RAM memories allow the user (or CPU) to read data or instructions into it, read them out and also allow to change the data stored in the memory on demand. While ROM memory only allows CPU to read and use the information it has stored but not change them.

When enormous quantities of data are to be stored, a secondary storage is required. Secondary storage is external storage equipment, which provide very large storage capacity but needs more access time. Secondary storages store data relatively longer or even permanently. When needed, the data stored in the secondary storage have to be transferred to the main memory so that CPU can access them directly. Commonly used secondary storage devices are usually made of tapes, floppy disks and magnetic drums.

For both memories, the trends are towards extremely high

capacity and speed.

The basic unit of the memory which can store and retrieve a binary bit, 0′ or 1′, is called storage cell. Modern semiconductor techniques make it possible that hundred of thousands storage cells are integrated in a tiny silicon chip. Many such chips are organized together to form a whole computer memory. One of the main advantage of semiconductor memory is its high density and low cost.

It is beyond the scope of this thesis to describe either physics or fabrications of semiconductor memory and its cells. However, in order to have a better understanding of error control coding for semiconductor memories, in this chapter, we provide a brief review of semiconductor memories in principle. At First, in section 2.1, semiconductor storage cells are discussed. The discussion includes ROM memory cells, RAM memory cells and DRAM cells. Then in section 2.2, three types of organizations for computer memories are presented. All practical memories are organized in these forms with some small variations. In section 2.3, the mechanisms of semiconductor memory errors are analyzed. Different types of errors are also classified in this section so that for different type of errors the corresponding error control codes can be selected and employed.

2.1 Memory Cells

a) Read only memory cells

In a computer, many operations are carried out more than once

without changing the content. This fact makes fixed memory or so called read only memory (ROM) very useful. ROM are constantly used in computer for character generation, bootstrap programs and look-up tables etc[2.1].

A ROM consists of a matrix of addressable cells. Several types of ROM cells are implemented in practice.

i) ROM diode cell[2.2]

Fig.2.1 shows a diode memory array. The information bits 1' and 0' stored in cells are represented by the presence or absence of diode between x and y lines (the x and y lines are also called word line and bit line). The reading operation is very simple. When a 1' information stored in a cell (in which the diode is connected to x and y lines) is to be read out, a constant current is applied to the x line connected to the cell while the associated y line holds a low voltage so that the diode conducts and the corresponding output provides a low current. Otherwise if a 0' is stored in the cell, where the diode is absent, the constant current from x line will flow through the output directly. Therefore the information 1' and 0' is distinguished by a low current and a constant current in the output. In this example suppose the first x line W_0 and all four y lines are selected, then a word 1100 is read out.

It can be seen that to address a particular cell, the associated x and y selection lines have to be coincident.

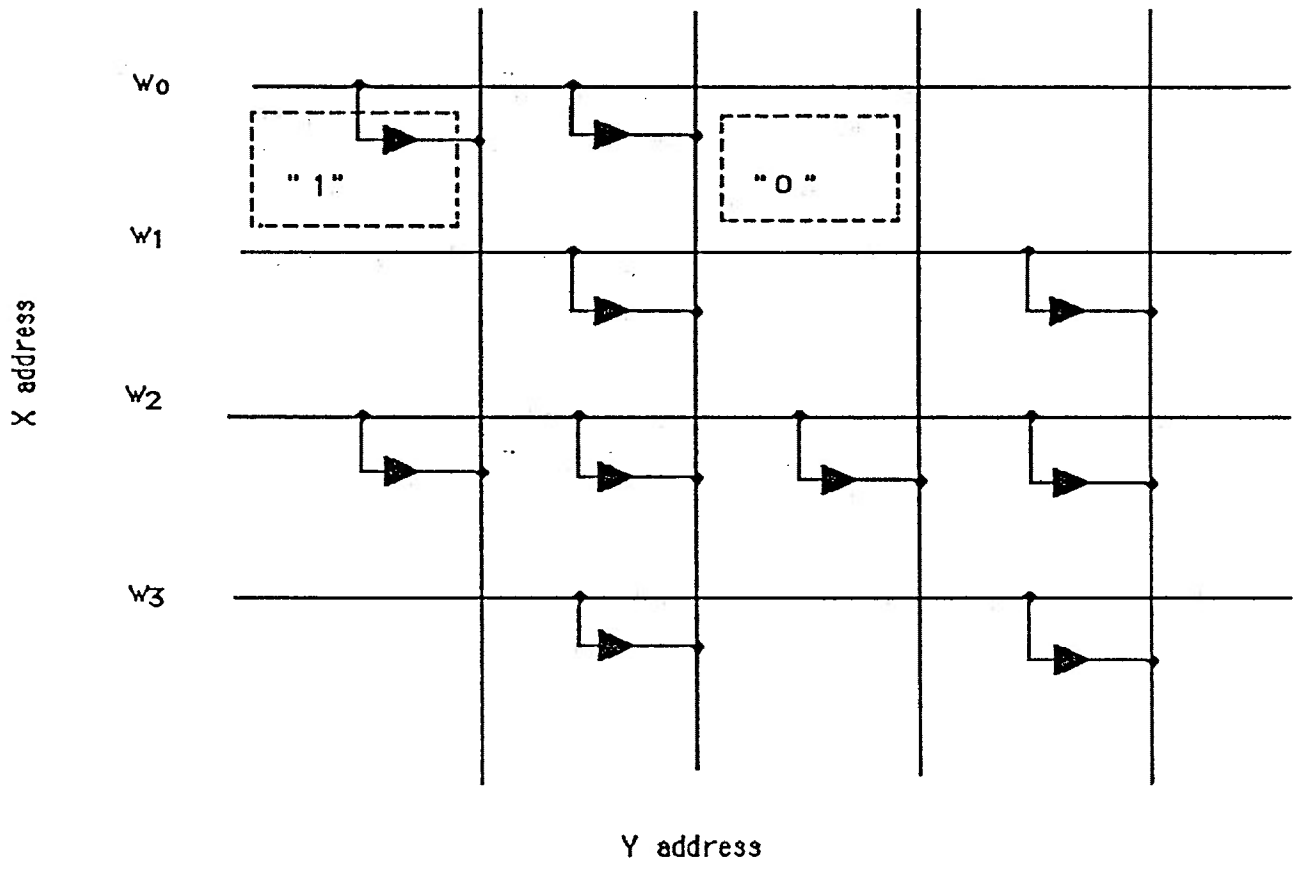


Fig. 2.1 ROM Diode Cells

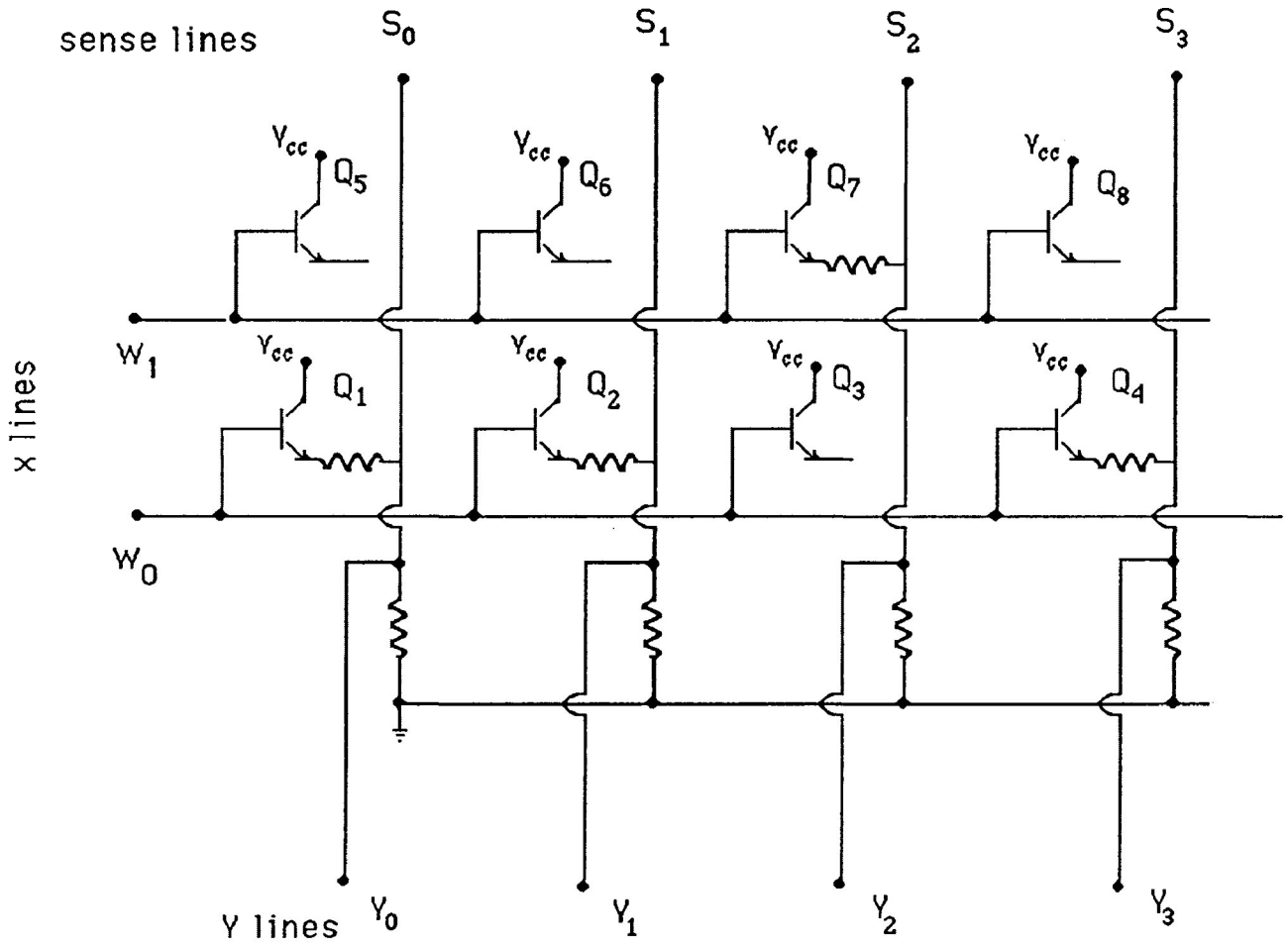


Fig. 2.2 Bipolar ROM Memory Cells

ii) ROM bipolar transistor cell[2.3]

Bipolar transistor can be used in ROM in very nearly the same way as in diode memory. A typical circuit of a bipolar cell is shown in Fig.2.2. Two types of cells for 1′ and 0′ can be distinguished . In the 1′ cell, the emitter of the transistor is connected to y line while in 0′ cell the emitter of the transistor is disconnected from y line. Suppose the x-line W_1 in the figure is driven high. This would turn on all transistors connected to it. Current would then flow through the transistors, through their emitter resistors and through the resistors at the bottom of the sense lines. A voltage drops across the sense line resistor indicates a binary information 1′, such as Q_7 in the figure. While in the 0′ cells (i.e., Q_5, Q_6, and Q_8), since there is no such a emitter current at all, there would be no voltage changed at the corresponding sense line, hence the 0′ is sensed. A word 0010 is then read out.

iii) MOS ROM cell[2.1]

MOS (Metal-Oxide-Semiconductor) technology is ideal for ROM due to its high density. A typical MOS ROM cell is shown in Fig. 2.3. The 1′ and 0′ cells are distinguished by connecting or disconnecting the gate of FET to the word line. For example when a positive pulse is applied on line W_0, current will flow up through Q_1 (and Q_3) and FET load to $+V_{DD}$ line. The voltage across the load drops the y_0 and y_2. As a result, a 0′ is read out. Since Q_2 (and Q_4) is inactive, there is no voltage drop on y_1 (and y_3), it will stay high , indicating a 1′ being read out. Thus a word

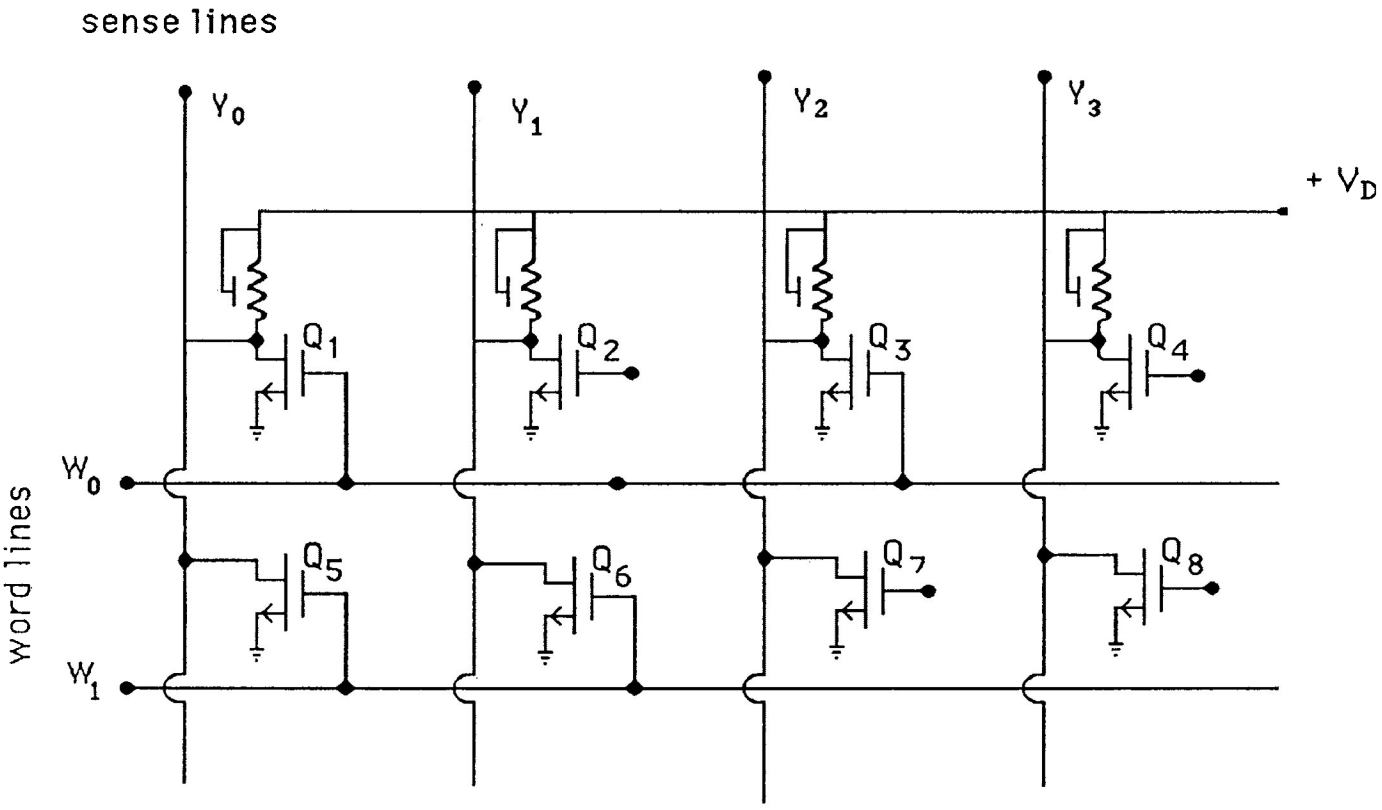


Fig. 2.3 MOS ROM Memory Cell

0101 is sensed.

b) Random access memory cells

Random Access Memory (RAM) allows not only to read out the content it stores but also to change (write) the content of the memory on demand.

i) Bipolar memory cell [2.1]

Fig.2.4 illustrates the circuitry of a TTL (Transistor-Transistor-logic) memory cell. Q_1 and Q_2 form a bistable flip-flop. The x and y lines are for selection or address of cells. The bit/sense line pair are for writing and reading, respectively. Write and read operations are described as follows.

Store a 0' To store a 0' in a cell, the corresponding x and y lines are driven high to address the cell. Then the write circuit place a low condition on E_1 of Q_1. This turns Q_1 on and Q_2 off regardless the previous state of the flip-flop. When the write pulse is gone, the flip-flop remains the state (Q_1 on and Q_2 off) unchanged. With the x and y lines back to normal (ground), the emitters E_2 and E_3 on Q_1 provide a path for emitter current. The current flowing in Q_1 is now from E_2 and E_3 to ground rather than from E_1. Once the digit 0' is stored, it is locked in the cell. The memory state can only be changed by grounding the write sense line on the off transistor. This change can only be made when the select lines are high.

Store a 1' If a 1' is to be stored, the x and y select lines

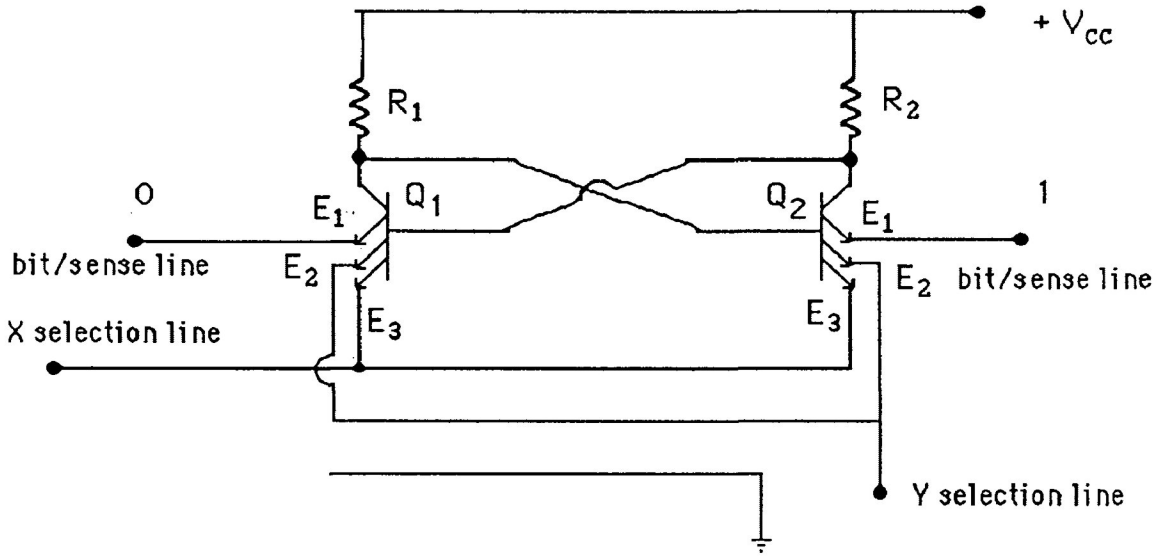


Fig. 2.4 Bipolar RAM Memory Cell

for the cell are driven high. The write circuit shorts E_1 on Q_2 to ground. The forward bias suddenly turns Q_2 on and the flip-flop action turns Q_1 off (regardless the previous state of the flip-flop). Then the state (Q_1 off and Q_2 on) is kept until a next write operation comes.

Sense To read the content of the memory we simply drive the x and y lines high. If Q_1 is on and Q_2 off which means a 0′ is stored in the cell, Q_1 will conduct through the resistor of sense amplifier. There will be no Q_2 current flowing in the sense amplifier connected to E_1 on Q_2, therefore there is no output from the 1′ sense amplifier. If the content of the cell is 1′, then only 1′ sense amplifier has output and 0′ sense amplifier keeps unchanged.

ii) MOSFET memory cell [2.1]

A basic MOS RAM memory cell is shown in Fig. 2.5. Q_1 and Q_2 constitute the bistable flip-flop. The drain load resistor is found by series MOSFETs. Many variations to the basic circuit are possible, but the principle of the operations are similar. The operations of writing and reading are described as follows.

Write the memory To enter a digit into the memory cell, x and y address lines have to be applied $+V_{dd}$. If a 0′ is to be stored, the 0′ bit line is placed to ground while the 1 bit line is held high. As a result, Q_1 will be turn on and Q_2 off. If a 1′ is to be stored, the 1′ bit line goes to ground and 0′ bit line holds high so that Q_1 is off and Q_2 on.

Read the memory To read the condition of the cell, the x and y

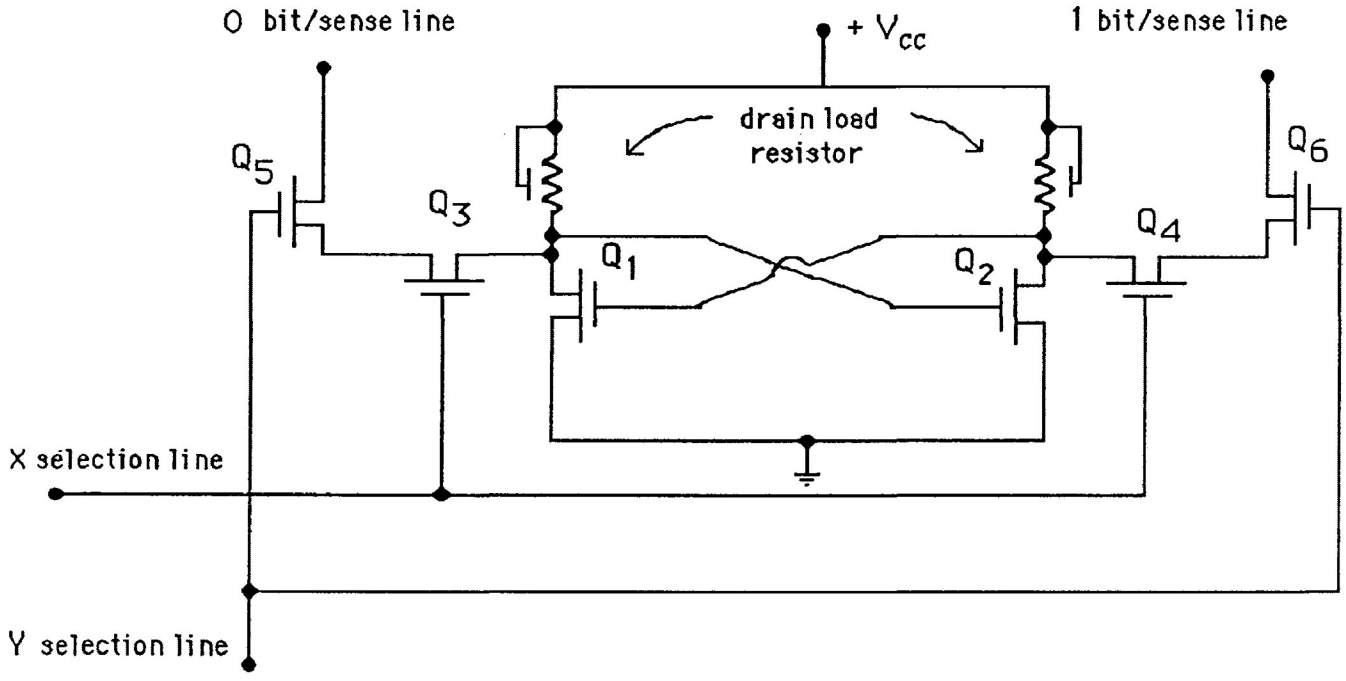


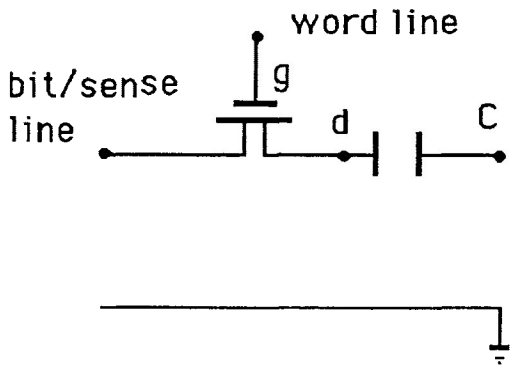
Fig. 2.5 MOS ROM Memory Cell

address lines are switched from ground to $+V_{DD}$. This high voltage will turn on Q_3, Q_4, Q_5 and Q_6. These transistors act now as closed switches such that the bit lines are both at $+V_{DD}$. However only the line connected to the 'on' flip-flop will conduct. Suppose Q_1 is on and Q_2 off. In this case, current can flow from ground up through Q_1, Q_3, Q_5 to 0' bit line. The current in this line is then sensed and amplified. The output is recognized as a 0' in the memory. Meanwhile for the 1' bit line, there is no current at all as Q_2 is off.

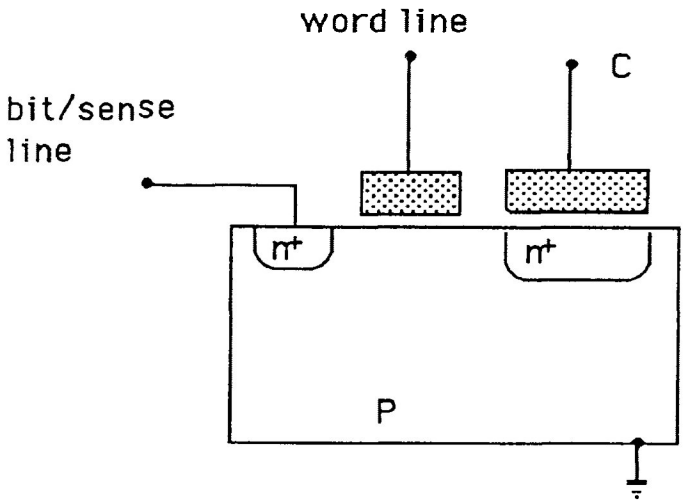
c) Dynamic RAM memory cell[2.4]

The RAM memory cells discussed above are all static memory cells. That is, the state or condition of the cell would remain unchanged until another write operation applies on the cell. To hold two stable states for storing information '1 and 0', the static RAM cell needs at least 2 transistors or MOSFETs. In the following we introduce a kind of dynamic memory cells. It contains only one transistor hence a higher density fabrication can be achieved.

A simple scheme of the single transistor cell is shown in Fig. 2.6 (a). The capacitor C_s is the storage capacitance and the MOSFET transistor acts as a switch. The capacitor C_s can be made in the same technology with that of the transistor. The cell structure in integrated technology is shown in Fig. 2.6 (b). In both circuits bit line and sense line share a same line called

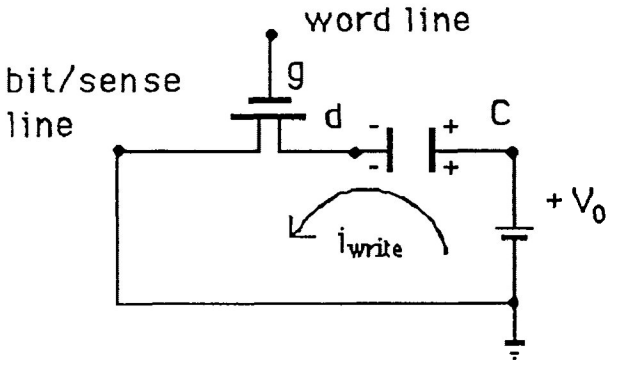


(a) Single Transistor DRAM Cell
in Separated Devices

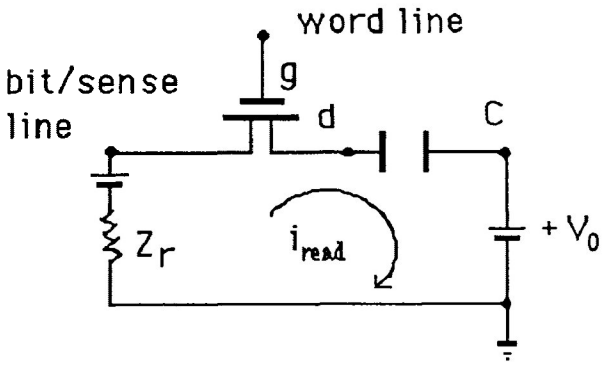


(b) Integrated Structure

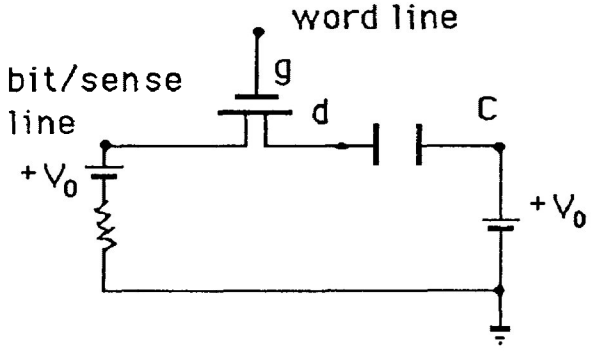
Fig. 2.6 Single-Transistor DRAM Cell



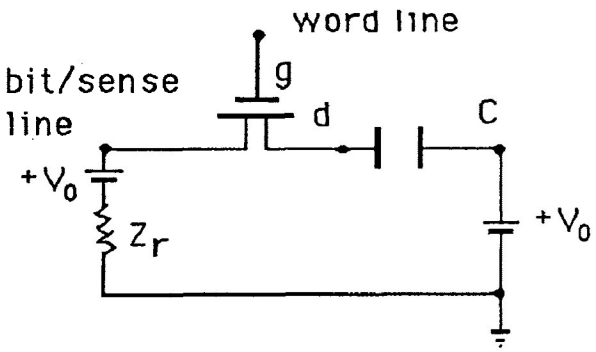
(a) Write a '0' in the cell



(b) read the cell



(c) Write a '1' in the cell



(d) read the cell

Fig. 2.7 Read and Write Operations

bit/sense line. The electrode C is always applied at a voltage $+V_0$. The information will be stored in the capacitor by presence and absence of negative charges in the capacitor. The read and write operations are described as follows.

Write and read a 0' When a 0' is to be written the cell, the word line is pulsed high and the bit/sense line connected to ground as shown in Fig. 2.7 (a). The dc voltage V_0 charges the capacitor C_s with current i_{write}. As a result, the capacitor is charged with negative charges which represents a 0' being stored in the cell. When read the cell, the word line is pulsed high and the sense/bit line is connected to a voltage of V_0 through the sense amplifier impedance Z_r. The equivalent circuit is shown in Fig. 2.7 (b). The current i_{read} will create a small signal across Z_r, thus the 0' is sensed. We can see that the current i_{read} will discharge the 0' information stored in C_s during the reading operation. Therefore the system has to be able to rewrite the information repeatedly or refresh the cell periodically.

Write and read a 1' When write a 1', the word line is pulsed high and bit/sense line is connected to $+V_0$ as shown in Fig. 2.7 (c). Since there is no current flowing in the capacitor circuit, there would be no charges being stored in the capacitor. This indicates that a 1' has been stored in the cell. When read the cell, the word line is driven high and the sense/bit line is connected to V_0 through the sense amplifier impedance Z_r. The equivalent circuit is shown in Fig. 2.7 (d).

Since there is no current flowing in the circuit, there would be no voltage across the Z_r. Thus the information 1' is sensed.

2.2 Memory Organizations [2.4]

There are three basic classes of memory organizations. They are the two, three, and two and half dimensional organizations (represented by 2D, 3D and 2-1/2D). The 2D and 3D types require a basic storage cell with two and three functional terminals, respectively. The 2-1/2D can use either types of cells. In practice, there would be some variations to these basic organization forms but the principle is the same. In some applications several schemes are combined to meet different needs. In the following we only discuss the three basic organizations which are useful for our purpose.

1 2D-memory organization

The simplest practical memory is organized as 2D memory. Fig.2.8 shows a block diagram of the scheme. All the memory cells are arranged in the form of a matrix. The cells in the same row are connected to a so-called word line. The cells in the same column are connected to a bit line. Any cell can be accessed randomly by the coincidence of a word line and a bit line. In the read and write operations, when a word line is selected, all cells associated with the word line will be available to be accessed. Actually, all these bit lines are accessed simultaneously.

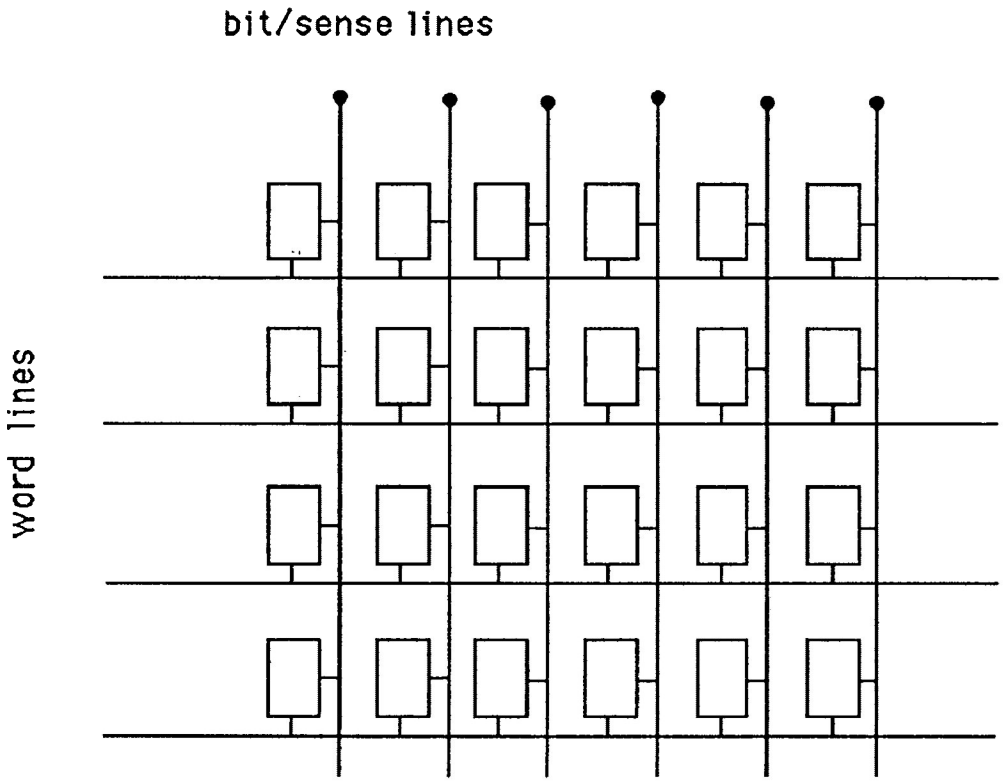


Fig. 2.8 2D Organized Memory

To write a word which consists of several bits in the memory, a word address is firstly required from the CPU. According to the address, a appropriate word line is energized. At the same time, the content of the word which comes from the data register arrives at bit lines. This causes the memory cell to be set to 1′ state. If a bit line signal is absent, the cell then remains in the 0′ state. To read a word from the memory, the desired word line is addressed and the content of each cell connected to the word line will appear on the corresponding bit line. Then through the sense amplifiers, all the bits of the word are sensed simultaneously.

2 3D-memory organization

There are two basic types of 3D storage. The series and parallel connections for the selection line. In semiconductor memories parallel version is more favourable. We only consider parallel 3D storage. Fig.2.9 is a illustration of 3D parallel organization. The x and y selection lines are connected in parallel between planes, the bit lines are connected in series on a plane. Every plane is of the same structure and contains the same number of cells.

The basic operations require the use of an x line, a y line and bit line for writing, and an x line, a y line, and a sense line for reading. For example, in a flip-flop memory, a coincidence of the x and y lines, and 1′ signal on the 1′ bit lines cause a 1′ to be written in a corresponding cells and a coincidence of x and y

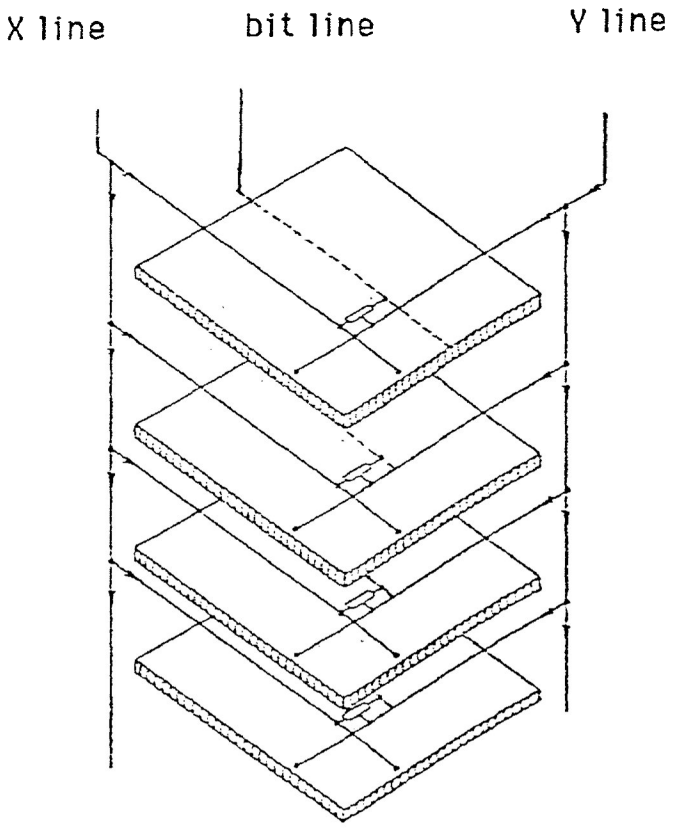


Fig. 2.9 3D Organized Memory

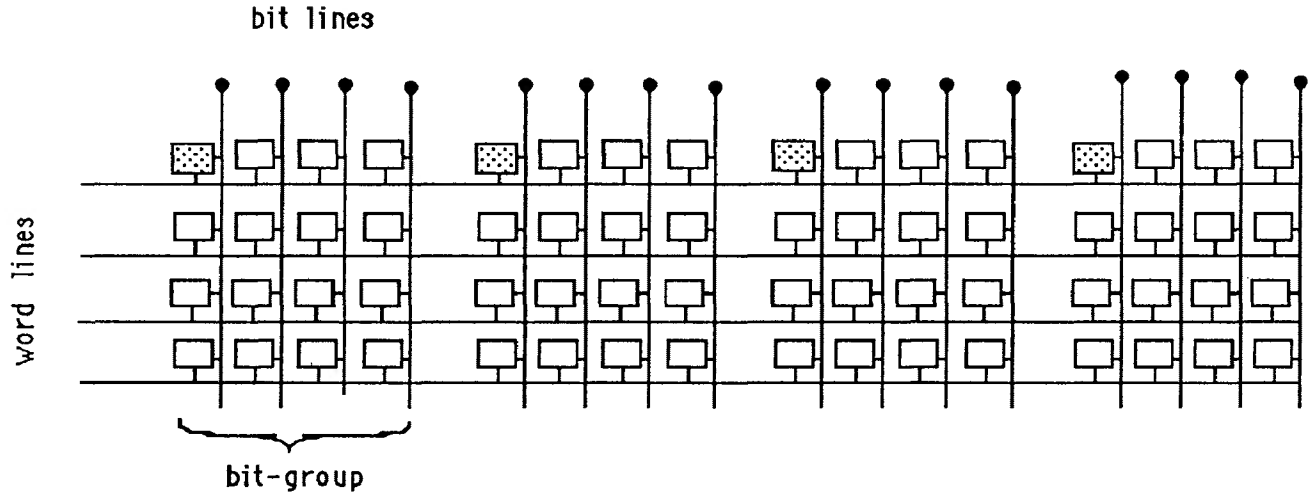
lines, and 0′ signal on the 0′ bit lines will store 0′s in the corresponding memory cell. For reading, energizing the x and y lines makes all the intersection cells ready to be sensed on the sense lines. In this scheme it can be seen that when a pair of x and y line are selected all the cells in the vertical direction are activated. All these cells each from different plane form a word. Thus the number of cells in each plane is equal to the number of the word that the memory contains. The number of the planes is equal to the number of the bits per word.

3 2-1/2D memory organization

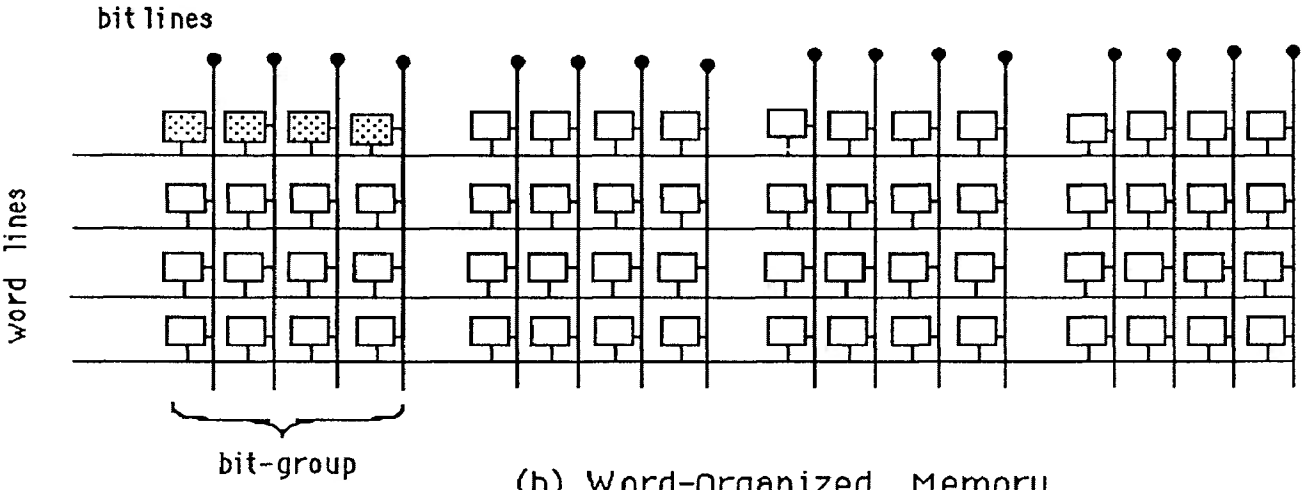
Almost all the main memories today are of 2-1/2D organization. Both two and three terminal storage cells can be used in 2-1/2D memory, therefore there are two fundamental forms of 2-1/2D for each of the cells respectively.

In all main memories the number of word is greatly exceeds the number of bits per word. This fact causes a difficulty that the word address circuitry would be very large and therefore need a long word addressing delay. To overcome the problem, we need to reduce the number of word lines without reducing the number of words which the memory contains. That is, one word line would energize more than one words. Fig. 2.10 (a) and (b) illustrate the principle of such organized memory matrices using two terminal storage cells.

Each row of the memory matrix is now divided into four bit_groups (can be any number of bit-groups in a row in practice). When a word line is selected, four bit-groups are half activated.



(a) Bit-Organized Memory



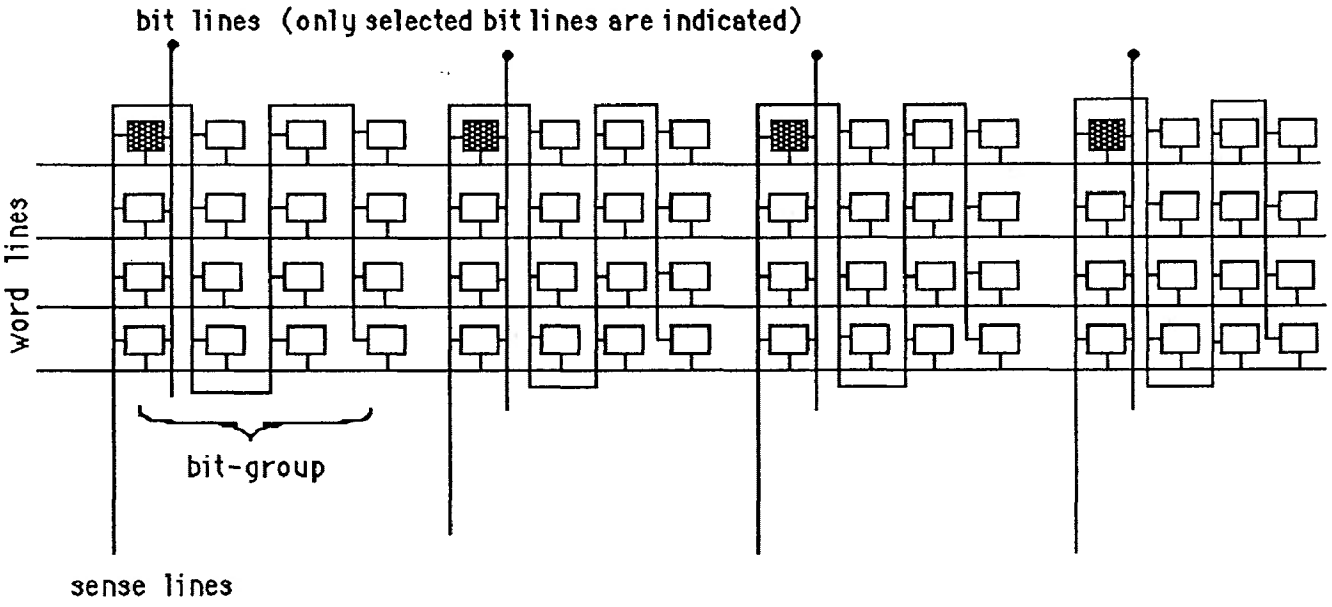
(b) Word-Organized Memory

Fig. 2.10 2-1/2D Organization Using Two Terminal Storage Cells

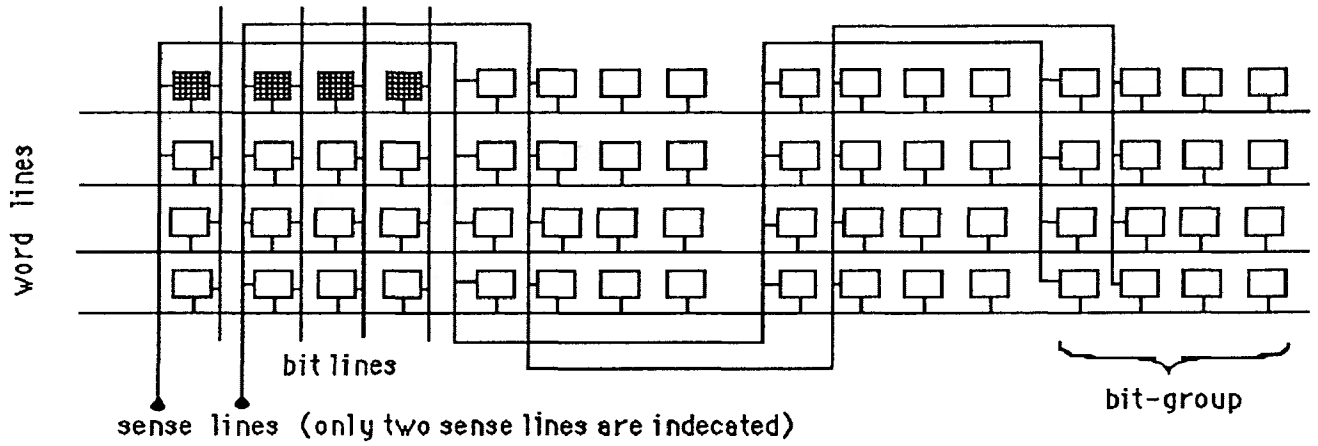
In Fig. 2.10 (a), a word consists of four bits, each bit comes from different bit-group. This is a Bit-organized 2-1/2D memory organization. In Fig. 2.10 (b), a word consists of the bits that come from a single bit-group. Hence it is a word-organized 2-1/2D memory organization. Since two terminal cells are used in this organization, the bit line and sense line are the same line called bit/sense line.

To write a 1' into the memory, a word line and appropriate bit line have to be coincident. If there is no signal on the bit line, the cell remains in the 0' state. To read the memory, there is no coincidence required. When a word line is selected, the contents of all cells connected to the word line appear on the corresponding sense/bit lines (four words are available to be read in Fig. 2.10). But only the desired sense lines are switched to the sense amplifiers.

For three terminal cells, there are also two configurations for the 2-1/2D organization. They are bit-organized and word-organized organizations. The simplified illustrations are shown in Fig. 2.11 (a) and (b), respectively. Since three terminal cells are used, the bit lines and sense lines are separated. In the figures, the connection of sense line are shown (for simplicity, only two sense lines are illustrated, the other two sense lines can be connected in a similar way). The word lines and bit line remain identical to those for two terminal cells as shown in Fig. 2.10. The difference between Fig. 2.10 and Fig 2.11 is that in the latter, the reading operation also requires a coincidence of a word



(a) Bit-Organized Memory



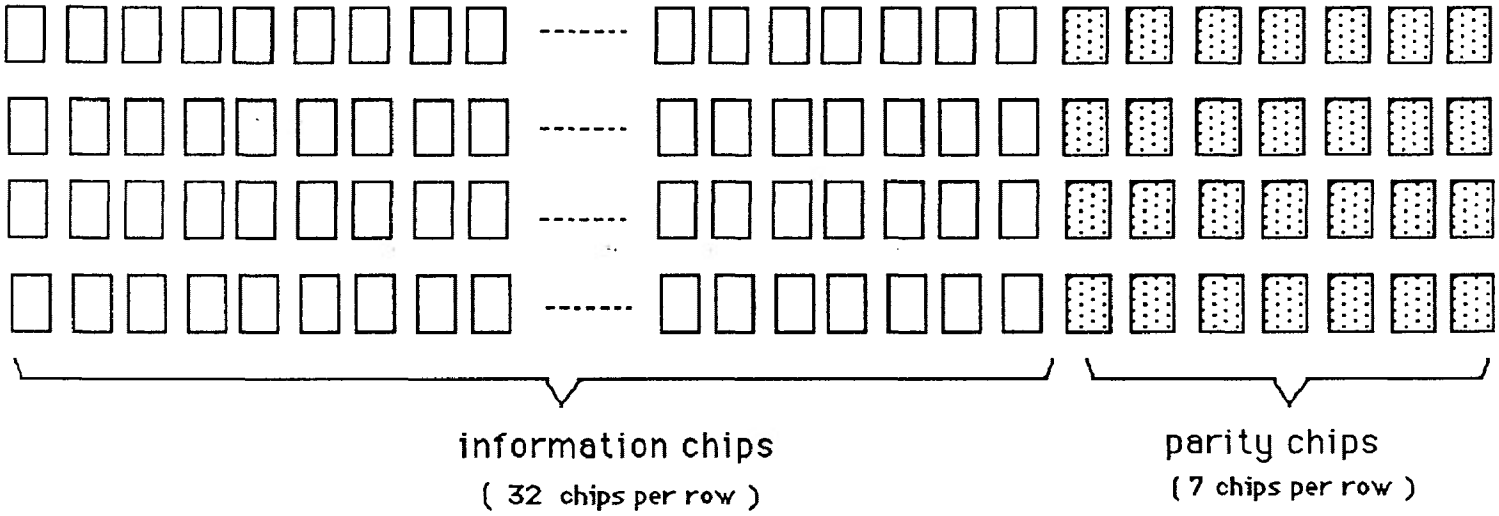
(b) Word-Organized Memory

Fig. 2.11 2-1/2D Organization Using Three Terminal Storage Cells

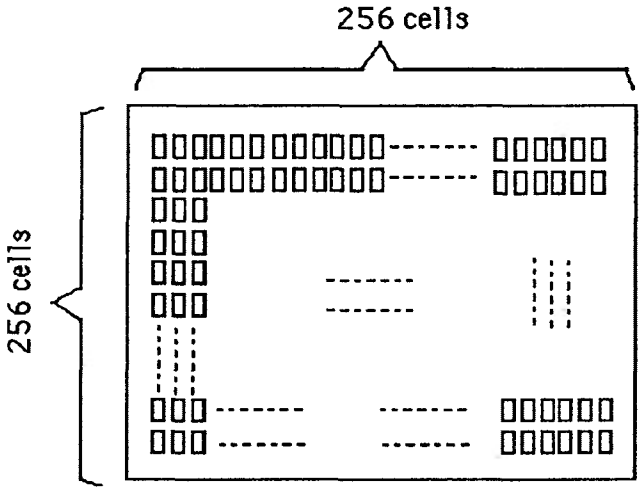
line and appropriate bit lines. This is because, for example, in Fig 2.11 (a), on the first sense line there are as many as 16 cells available to be read, but only the shadowed one is desired. The coincidence can be used to specify the desired cell from many cells associated with the same sense line. LSI (Large Scale Integrated) semiconductor technology makes it possible to integrate many cells on a tiny chip so that the memory capacity can be increased dramatically. In our organizations shown in Fig.2.10 and Fig.2.11, all bit-groups in vertical direction can be fabricated on a single chip respectively. Thus the memory showed in Fig.2.11 is made up four chips with each chip of 16 cells. Modern VLSI can make 64k cells or more on a single chip. The high density chips not only significantly increase the capacity of the main memory but also greatly reduce the joints and connections between cells, therefore reduce the potential circuitry faulty. On the other hand, the maintenance of the memory on the level of chips becomes much easier than that of on the level of individual cells. When chips are used in the memory discussed above in Fig.2.11 (a) and (b), they are usually called bit-per-chip and word-per-chip organization, respectively. There are also byte-per-chip organized memory in practice.

An example of chip organized 2-1/2D memory organization

A typical chip organized memory board with 1M-byte sold for VAX computers is shown in Fig. 2.12 (a) [2.5, 2.6]. Each chip is organized internally as 256 X 256 square matrix of bits. This 64K chip has 65536 single-bit locations as shown in Fig. 2.12 (b). The



(a) 1M-byte memory organization



(b) Structure of chip

Fig. 2.12 1M-Byte Memory

memory board is organized as 4 x 39 matrix of chips. The first 32 chips in each row are used for storing information bits, and the remaining 7 chips in the row are for parity check bits (we will discuss the function of these parity check bits in the next chapters latter on). Every word consists of 39 bits, each bit is stored in a different chip in one row. Therefore this is a bit-per-chip organized memory.

According to the organization, we can simply calculate the parameters of the memory as follows:

Number of words in the memory (N_w) = 256 x 256 x 4 = 256K words

Number of bits in the memory (N_b) = N_w x 39 = 10M bits

Number of bytes in the memory (N_B) = N_b / 8 = 1M bytes

Number of bits per word (word size W_s) = 39 bits

It can be seen that in practice the number of words and word lines are very large. The selection of a word or a cell is therefore somehow complicated. In memory system the appropriate selection is done by a so-called address decoder. The concept of the address decoder was eliminated in the above discussions.

2.3 Errors in Semiconductor Memories

Due to the defective cells, failed connections between selection lines, bit/sense lines and various interferences, there exist faults in semiconductor memories. Particularly when memory cells trend to be extremely small and the chips extremely dense, the defects increase significantly. As a result, some bit(s), when

they are read out from the storage cells, would be different from those which has been stored. In this case, we say error(s) occurring in the word. The presence of these errors greatly decreases the reliability of the computer memory.

There are several types of errors. Some of them are permanent in position and nature, the others are temporary and random. These errors are classified as hard and soft errors. On the other hand, Some errors are symmetric in error directions (the errors can be either 1'-0' and 0'-1'), and others are asymmetric errors (only one 1'-0' or 0'-1' is possible, but not both). In this section we will briefly discuss the errors and their mechanisms, including hard errors, soft errors and unidirectional errors.

Hard errors

If the error in any position of a word is permanent in nature, then it is called a hard error [2.7]. One of the properties of hard errors is that once it has happened at some locations the output of this location is permanently stuck at 1' or 0' state regardless of what was written in. Defective cells, internally failed connections between selection lines, such as short and open leads, are most likely sources of the hard errors. Several kinds of hard failures have been reported[2.5]. A single cell failure , for example, can occur as a hard error. There are also multiple errors caused by row failure, column failure, row-column failure and even a whole chip failure. However by proper memory organizations, the error patterns can be limited to single errors in every memory

word.

Hard errors are random, permanent, and irreversible. They can occur during the fabrication or latter on in the service time. In practice some schemes are used to deal with the hard errors in semiconductor memories. Such as on-line testing and switching to spares, replacing the defective chips etc. With the improvement of VLSI technology, the rate of hard errors has been decreased dramatically.

By the improvement of VLSI techniques and proper memory organizations, for example, one-bit-per-chip organization, the hard errors in computer memory appear usually as single errors.

Erasures

A hard error has a fixed position, therefore once it has been detected, the position is then known in next memory operation cycles. Thus in the next read operation, we know if the bit is correct or not. This type of errors is called erasures. With erasures we define a symbol in a word with known location but with unknown value[2.6]. Clearly a hard error is random in position while an erasure's position is known.

Soft errors

If the error is of a transient type, then it is called soft error. Soft errors are caused by system noise, power surges and alpha-particles radiation. Compared with hard errors, soft errors are random, temporary and reversible. In the next memory cycle the

faulty bit does not show any greater chance of being in error than that of other bits in the memory. Therefore the schemes mentioned above for hard errors are not effective to soft errors.

Experimental evidence indicates that most (more than 90 percent) soft errors seen in devices from several manufactures are due to alpha-particle radiation from packaging materials. In the following , we will give a brief discussion about the mechanism of alpha-particle inducing soft errors in DRAM memories [2.8, 2.9].

In a integrated DRAM storage cell, the presence and absence of the negative charges in the capacitor represents the 0' and 1' state respectively (in N-channel the minority carriers are electrons). If the negative charges stored in the capacitor are discharged or the no-charged capacitor acquires enough electrons by rather than an ordinary memory operation, the 0' state might change to 1' state or vice versa, then the stored bit would be erroneous. We will see that this is the way that alpha-particles cause errors.

An alpha-particle is emitted by radioactive decay of Uranium and Thorium which are present in packaging materials. When an alpha-particle penetrates semiconductor memory devices, it can create enough electron-hole pairs near a storage node. These electrons and holes diffuse through the bulk silicon. For those which reach the edge of storage region (depletion region), the electrons are swept into depletion region while the holes ohmically move through the substrate. If the normal state of the capacitor is in the no charged state (1' state), the depletion region is said 'empty'. In this situation, when an alpha-particle generates

enough electrons and holes near the depletion region, the swept electrons will be collected in the region such that the empty region is filled with the electrons which represent a 0′ state. Thus an alpha-particle causes an 1′-to-0′ error. On the other hand, if the normal state of the capacitor is in the charged state (0′ state) originally, the depletion region is initially full of electrons. In this situation when some generated electrons are swept into the depletion region, the 'full' state will not be changed, hence the cell remains the 0′ state without being affected by the alpha-particle (holes can not been swept into the depletion region because holes do not move without electron's moving). Therefore alpha-particles only disturb the 'empty' state of the capacitor. Figure 2-13 shows graphically the mechanism of the alpha-particle inducing soft errors.

Further research also shows that alpha-particles have other properties which is critical to the soft errors as following

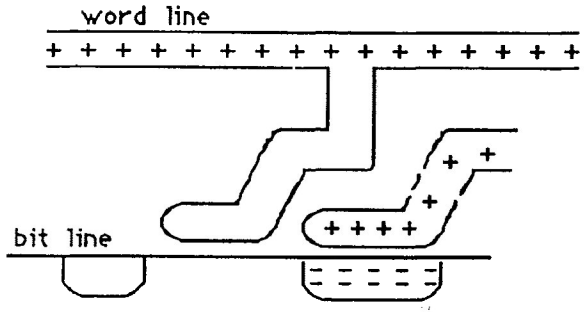
a) alpha-particle travels in nearly straight line

b) the scatter in range is small

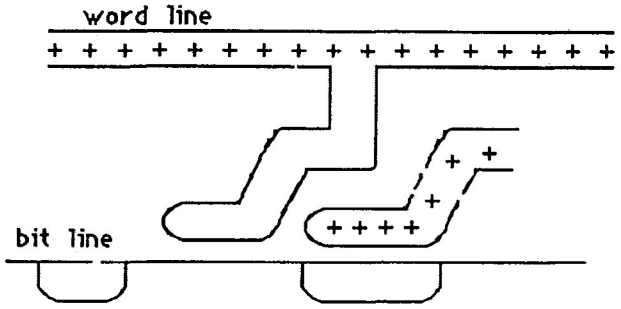
c) alpha-particle emits at discrete energies

d) alpha-particle emission is nuclear event and is unaffected by temperature, pressure, etc.

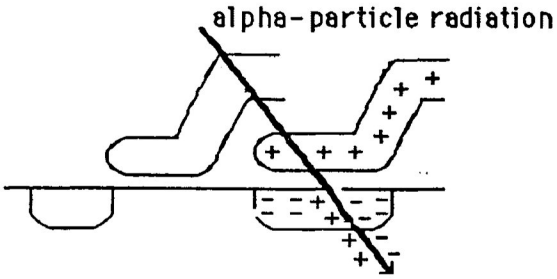
Alpha-particle affects not only DRAM memories but also some high speed bipolar memories[2.10]. When an alpha-particle hit the cell, it induces short transient current between the junctions of devices and circuit elements. These current flows through the collector of an N-P-N transistor to the substrate cause a potential



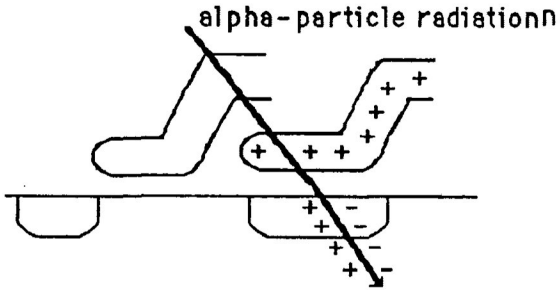
(a) '0' state is represented by 'full well'



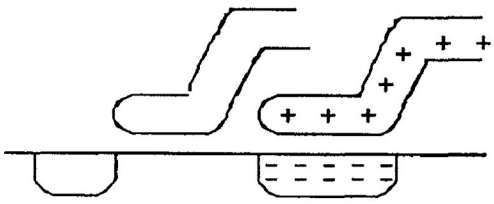
(a') '1' state is represented by 'empty well'



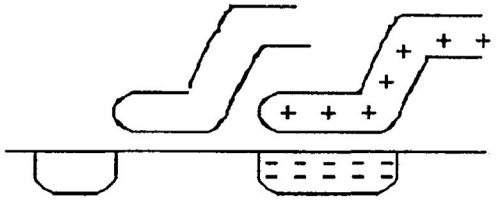
(b) alpha-particle penertrating and disturbing the '0' state



(b') alpha-particle penertrating and disturbing the '1' state



(c) 'full well' is still in '0' state without being affected by alpha-particle



(c') 'empty well' is changed to full by alpha-particle and '1' state is changed to '0' state

Fig. 2-13 Mechanisms of soft error induced by alpha-particle

dip at the collector. If that happens to be the collector of OFF transistor, it may cause a state changing. As a result, the state of the flip-flop is changed. Thus an error occurs because of the alpha-particle.

According to the discussion it can be seen that the distribution of the soft errors in semiconductor memories show up as single, random, and temporary errors.

Unidirectional errors

If all errors in a memory are of either 1′-to-0′, or 0′-to-1′ type, but not both, then the errors are called asymmetric errors. If all errors in a particular word are of either 1′-to-0′, or 0′-to-1′ type, but not both, then the errors are called unidirectional errors. The most likely faults in some of the recently developed LSI/VLSI ROM and RAM memories cause unidirectional errors[2.11, 2.12]. Such as the faults that affect address decoder, word lines, power supply,and stuck-fault in a sense bus, etc[2.13].

i) Address decoder: Single and multiple faults in address decoder may result in either no access or multiple access. No access yields an all-0-word read out from the memory and multiple access cause the OR of the several word to be read out. In both cases the resulting errors are unidirectional errors.

ii) Word line: An open word line may cause all bits in the word beyond the point of failure to be stuck at 0′. On the other hand, two word lines shorted together will form an OR function beyond the point where they shorted.

iii) Power supply: A failure in the power supply usually results in a unidirectional error.

2.4 Summary

We have discussed semiconductor storage cells, Memory organizations and mechanisms of various errors. An important concept is that in modern computers almost all main memories are made of LSI/VLSI chips. Thousands of cells fabricated on a single chips makes it possible to form very large capacity memories. Any cell failure on a chip will cause the chip to be failed. Any type of chip failure will cause errors. The common type of chip failure is single-cell failure caused by alpha-particle radiation. However there are also several other types of chip failures caused by hardware defects. These failures may cause multiple errors in the memory as well.

By proper memory organization design, for example, a bit-per-chip organized memory, it is able to disperse the multiple errors into different words such that when a chip is in failure, no matter what kind of failure, it only affects a single bit in a word. Thus in most cases the single bit error in a word is the dominating situation (comparing with memories of magnetic tapes and disks, in which, the usual errors are of random and burst distribution caused by defects, dust particles and magnetic head noises etc). Therefore for a semiconductor computer memory system the single error detecting and correcting ability is of particular significance.

References

2.1 O.R. Lawrence "Computer Technology" McGraw-Hill Ryerson Limited, 1984

2.2 S. Middelhock, P.K. George, P.D. Dekker "Physics of Memory Devices" Academic Press Inc. 1976

2.3 Walter A. Triebel, Alfred E. Chu "Handbook of Semiconductor and Bubble Memories" Prentice-Hall, Inc. , Englewood Cliffs, NJ 07623, 1982

2.4 Richard E. Matick "Computer Storage & Technology" Wiley-Inter science Publication John Wiley & sons, 1977

2.5 Mario Blaum, Rodney Goodman, and Robert Mceliece "The Reliability of Single-Error Protected Computer Memories" IEEE Transactions on Computer No. 1, Jan. 1988 pp 114-119

2.6 Carl-Erik W. Sundberg "Erasure and Error Decoding for Semiconductor Memories" IEEE Transactions on Computers vol.c-27 No.8, August 1978, pp 696-705

2.7 T. R. N. Rao E. Fujiwara "Error-control Coding for Computer Systems" Prentice-Hall Inc 1989

2.8 Timothy. May "Alpha-particle-induced Soft Error in Dynamic Memories" IEEE Transaction on Electron Devices, vol. ed-2, No. 1, Gin. 1979 pp 2-9

2.9 Robert J. McEliece " The Reliability of Computer Memory" Scientific American Vol. 252. No. 1, Gin. 1985, pp 88-92

2.10 Denndy. Tang, Ching-Te Chuang "A Circuit Concept for Reducing Soft Error in High-speed Memory Cells" IEEE of solid-state circuits, Vol. 23, No. 1, Feb. 1988, pp 201-203

2.11 Dimitris Nikolo "Theory and Design of t-error Correcting d-Error Detecting (d>t) and All Unidirectional Error Detecting Codes" IEEE Transaction on Computers 40, No. 2, Feb 1991, pp 132-141

2.12 Serban D. Constantin and T. R. N. Rao "On the Theory of Asymmetric Error Correcting Codes" Information and Control 40, (1979), pp 200-235

2.13 D. K. Prandhan, J. J. Stiffler "Error-correcting and Self-Checking Circuits" IEEE Computer, March 1980, pp 37-37

CHAPTER THREE LINEAR CODES

In this chapter we review the basic concepts of linear codes and introduce Hamming codes and BCH codes. The detailed discussions can be found in any text book on the coding theory. Some of them are listed in the end of the chapter.

3.1 Basic concepts of linear codes

We suppose that a message word which is to be stored in memory has a form of $\mathbf{u} = u_0u_1...u_{k-1}$. This is a k-tuple. For binary information (we only consider binary information in this chapter) the maximum number of message words is 2^k. By adding some redundant bits to the message word, we convert the k-tuple **u** to a n-tuple **x** $(= x_0x_1...x_{n-1})$. This process of converting is called encoding. The new n-tuple is accordantly called a codeword. The encoding is performed in a encoder. Every message word can be uniquely encoded into a codeword. All the codewords form a codeword set called code C (there are 2^k codewords in C). If we define the n-tuples as a vector space $\mathbf{V}_n$, then there are 2^n-tuple vectors in the space. The code C is a subspace of the vector space $\mathbf{V}_n$. This code is also denoted as an (n, k) code.

In a computer memory with coding, the message word **u** which is from a CPU data register is firstly encoded into a codeword **x**, and then the codeword is stored in the memory locations. Compared to the original message word of k bits, a codeword uses n bits to represent the same information. Thus r (= n - k) more bits are

used. Therefore a codeword needs more memory space to store the redundant bits. But we will see that these redundant bits make it possible to detect and correct errors when they occur in codewords. This way coding enables to improve the reliability of a memory with errors. As we have known, when a codeword is read out from memory, it may be different from what it was stored. Before the original message word **u** is recovered from the codeword, it is necessary to ensure the read codeword is error free. The process of correctly recovering the message from a read codeword is called decoding. Use of the redundancy, a decoder first checks if the read codeword is a valid codeword. If it is, the read codeword is accepted as correct, and the message word **u** is then extracted from the codeword. Otherwise the error(s) is said to be detected. The decoder can be designed to have ability of detecting, or correcting error(s) or both. Clearly, we hope using fewer redundant bits to detect and correct more errors in a codeword. The ratio of $R = k/n$ is defined as the rate of the code. Redundancy coding is the study of using redundancy to detect and correct errors effectively.

As an example we consider a memory, in which the probability of an error occurrence is $p = 10^{-6}$. We suppose a message word consists of four bits. In this memory any one or more than one erroneous bit(s) in a message word will spoil the message word and result in an erroneous word. Therefore the probability of reading a word that is in error in the memory is expressed as following

However if a (7, 4) code is applied to the memory, then every 4-bit message word is encoded to a 7-bit codeword. We suppose the

$$P_1 = \sum_{j=1}^{4} \binom{4}{j} p^j (1-p)^{4-j} = 4p + O(p^2) \doteq 4x10^{-6}$$

code is capable of correcting a single error occurring in the codeword. Thus any single bit error will no longer cause a erroneous word but any two or more than two bit errors do so. Therefore the probability of obtaining a erroneous codeword from the memory can be calculated as

$$p_2 = \sum_{j=2}^{7} \binom{7}{j} p^j (1-p)^{7-j} = 21p^2 + O(p^3) \doteq 21 \; x \; 10^{-12}$$

It can be seen that coding uses more bits but provides significant improvement of reliability.

There are systematic and nonsystematic codes. For a systematic code C, the first k bits of a codeword are a copy of message bits and the last r (= n - k) bits are the redundancy called parity check bits. In computer memories systematic codes are widely used. This is because these codes have simple implementations.

Generator matrix of code C

Since a code C is a subspace of $\mathbf{V}_n$, it is possible to find k linearly independent vectors $\mathbf{g}_0$ $\mathbf{g}_1 \ldots \mathbf{g}_{k-1}$ in C such that every codeword x in C is a combination of these k vectors. This can be generally expressed as

$$\begin{aligned} \mathbf{x} &= u_0\mathbf{g}_0 + u_1\mathbf{g}_1 + \ldots + u_{k-1}\mathbf{g}_{k-1} \\ &= \mathbf{u}.\mathbf{G} \end{aligned} \qquad (3.1)$$

where $\mathbf{u} = u_0u_1 \ldots u_{k-1}$, and $\mathbf{g}_i$'s can be arranged as a matrix as

following

$$G = \begin{vmatrix} g_0 \\ g_1 \\ . \\ . \\ g_k\text{-}1 \end{vmatrix} = \begin{vmatrix} g_{00} & \cdot & \cdot & \cdot & g_{o,n-1} \\ g_{1,0} & \cdot & \cdot & \cdot & g_{1,n-1} \\ . & . & . & . & . \\ . & . & . & . & . \\ g_{k-1,0} & \cdot & \cdot & \cdot & g_{k-1,n-1} \end{vmatrix} \qquad (3.2)$$

Eq. (3.1) means that when the matrix of **G** is given, for any message word **u** = $u_0u_1...u_{k-1}$, the corresponding codeword can be uniquely generated through the rows of the G matrix. Therefore the code C is the row space of the **G** matrix. For this reason the matrix **G** is called a generator matrix for code C.

A generator matrix **G** has the properties as following

i) For a (n,k) code the generator Matrix **G** is a (k x n) matrix.

ii) Every row of the **G** matrix is a codeword in C

iii) Any k linearly independent codewords in C can be used to form a **G** matrix.

iv) Elementary row operations performed on the **G** gives different generator matrices which also generate C.

v) One generator matrix may be more useful than another. For example, after some elementary row operations we can obtain a generator matrix which has the form as following

$$G = \begin{vmatrix} g_0 \\ g_1 \\ . \\ . \\ g_{k-1} \end{vmatrix} = \begin{vmatrix} 1 & 0 & 0 & . & . & 0 & p_{00} & p_{01} & \cdot & \cdot & p_{0,n-k-1} \\ 0 & 1 & 0 & . & . & 0 & p_{10} & p_{11} & \cdot & \cdot & p_{1,n-k-1} \\ . & . & . & . & . & . & . & . & . & . & . \\ . & . & . & . & . & . & . & . & . & . & . \\ 0 & 0 & 0 & . & . & 1 & p_{k-1,0} & p_{k-1,1} & \cdot & \cdot & p_{k-1,n-k-1} \end{vmatrix} = |I_k \cdot P| \qquad (3.3)$$

where **P** is a (k x r) matrix of 0's and 1's, and $\mathbf{I}_k$ is a (k x k) unit matrix. This is a generator matrix for systematic codes (note that row operations alone may not always result in a systematic form. Sometimes column permutation may be also required to obtain a systematic form[3.1]).

Example. A generator matrix **G** for a (7, 4) systematic code is given as following

$$G = \begin{vmatrix} g_0 \\ g_1 \\ g_2 \\ g_3 \end{vmatrix} = \begin{vmatrix} 1\ 0\ 0\ 0 & 1\ 1\ 0 \\ 0\ 1\ 0\ 0 & 0\ 1\ 1 \\ 0\ 0\ 1\ 0 & 1\ 1\ 1 \\ 0\ 0\ 0\ 1 & 1\ 0\ 1 \end{vmatrix} \qquad (3.4)$$

If the given message word is u = 1101, according to Eq. (3.1) the corresponding codeword is calculated as

$$\begin{aligned} \mathbf{x} &= 1\cdot\mathbf{g}_0 + 1\cdot\mathbf{g}_1 + 0\cdot\mathbf{g}_2 + 1\cdot\mathbf{g}_3 \\ &= (1000110) + (0100011) + (0000000) + (0001101) \\ &= (1101\ 000) \end{aligned}$$

The first four bits are message bits and the last three bit are the parity check bits. All $2^k = 2^4 = 16$ codewords generated same way are given in Table 3.1.

Parity check matrix of code C

In decoding, the first thing is to check the validity of a codeword read from the memory.

Since **G** matrix is a (k x n) matrix with k linearly independent

Table 3.1 A (7,4) code

message word	code word
0000	0000 000
0001	0001 101
0010	0010 111
0011	0011 010
0100	0100 011
0101	0101 110
0110	0100 100
0111	0111 001
1000	1000 110
1001	1001 011
1010	1010 001
1011	1011 100
1100	1100 101
1101	1101 000
1110	1110 010
1111	1111 111

rows, there exists an (r x n) matrix **H** with r linearly independent rows such that any vector in the row space of G is orthogonal to the rows of the **H** matrix. Or in other words, any codeword **x** in C is orthogonal to the rows of the **H** matrix. These relationships can be expressed in Eq. (3.5a) and (3.5b)

$$\mathbf{G}\,\mathbf{H}^T = 0 \qquad (3.5a)$$

$$\mathbf{x}\,\mathbf{H}^T = 0 \qquad (3.5b)$$

Such a **H** matrix can be used to verify the validity of a codeword in C. A codeword is a valid codeword if and only if it satisfies Eq. (3.5b), otherwise the codeword is in error. This **H** matrix is called the parity check matrix.

Given a generator matrix **G** shown in Eq. (3.2), the parity check matrix for a systematic code can be derived as

$$H = |P^T I_r| = \begin{vmatrix} p_{00} & p_{10} & \cdots & p_{k-1,0} & 1 & 0 & 0 & \cdots & 0 \\ p_{01} & p_{11} & \cdots & p_{k-1,1} & 0 & 1 & 0 & \cdots & 0 \\ \cdot & \cdot & \cdots & \cdot & \cdot & \cdot & \cdot & \cdots & \cdot \\ \cdot & \cdot & \cdots & \cdot & \cdot & \cdot & \cdot & \cdots & \cdot \\ p_{0,r-1} & p_{1,r-1} & \cdots & p_{k-1,r-1} & 0 & 0 & 0 & \cdots & 1 \end{vmatrix} \qquad (3.6)$$

The first k columns in the **H** matrix are corresponding to the message bits and the last r columns to the parity check bits.

The **H** matrix has following properties:

i) A **H** matrix for an (n, k) code is a (r, n) matrix.

ii) Any valid codeword in C is orthogonal to the rows of the **H** matrix. That is

$$\mathbf{x}\,\mathbf{H}^T = 0 \qquad (3.7)$$

According this property, we conclude that for any codeword **x**

and **y** in C, $(\mathbf{x}\mathbf{H}^T + \mathbf{y}\mathbf{H}^T) = (\mathbf{x} + \mathbf{y})\mathbf{H}^T = 0$. This means, in a linear code C, the sum of any two codewords is still a codeword in C.

iii) An **H** matrix also defines the code C. In fact, the **H** matrix is more convenient to describe a code than the generator matrix does.

Syndrome of a codeword

For a codeword x, if Eq. (3.7) is not satisfied, error(s) is said to be detected. But Eq. (3.7) does not provide any information about the position of the error(s), so correction can not be performed. To do so, a so called syndrome of a codeword is required.

Let us suppose that the codeword stored in the memory is $\mathbf{x} = x_0x_1\ldots x_{n-1}$, and the codeword read from the memory is $\mathbf{d} = d_0\ d_1\ \ldots d_{n-1}$. Due to the errors in the memory, **d** may differ from **x** in one or more bit positions. That is

$$\mathbf{d} = \mathbf{x} + \mathbf{e} \qquad (3.8)$$

where **e**, is an n-tuple called the error vector or error pattern. It has 0's in those positions where **d** and **x** agree and 1's in those positions where **d** and **x** disagree. In other words, the 1's in **e** mark the position where error have occurred.

When **d** is read from the memory, the decoder uses the **H** matrix to calculate an r-tuple vector **S**:

$$\begin{aligned}\mathbf{S} &= \mathbf{d}.\mathbf{H}^T = \mathbf{x}.\mathbf{H}^T + \mathbf{e}.\mathbf{H}^T \\ &= \mathbf{e}.\mathbf{H}^T = (s_0\ s_1\ldots s_{n-k-1}) \qquad (3.9)\end{aligned}$$

This is called the syndrome of **d**. If **S** = 0, then **d** is a valid

codeword. If **S** < > 0, then d is an invalid codeword, thus error(s) are detected. Furthermore, for different error patterns **e**, the syndrome **S** = (s_0 s_1...s_{n-k-1}) has different values. Therefore the syndrome can be used to determine the error patterns. Once the error pattern is specified, the correction of the codeword is achieved by bitwise XOR operations on **d** and **e**, namely, the correct codeword is obtained from (**d** + **e**). There are 2^r unique syndromes for an (n, k) code, hence a linear (n, k) code is capable of correcting up to 2^r error patterns.

As an example, we consider a (7, 4) code generated by Eq.(3.4). Eq.(3.10) is the **H** matrix of the code which is derived from Eq.(3.4) and Eq.(3.6)

$$H = \begin{vmatrix} 1 & 0 & 1 & 1 & 1 & 0 & 0 \\ 1 & 1 & 1 & 0 & 0 & 1 & 0 \\ 0 & 1 & 1 & 1 & 0 & 0 & 1 \end{vmatrix} \qquad (3.10)$$

Suppose a codeword which was stored in the memory is **x** = 1000110, and the read codeword is **d** = 1000111, in which the last bit position is in error. So the error pattern is **e** = 0000001. According to Eq.(3.9), the syndrome of the read codeword **d** can calculated as

$$S = d \cdot H^T = \begin{vmatrix} 1 & 0 & 0 & 0 & 1 & 1 & 1 \end{vmatrix} \cdot \begin{vmatrix} 1 & 1 & 0 \\ 0 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 0 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{vmatrix} = (0\ 0\ 1) \qquad (3.11)$$

or use the error pattern

$$S = \boldsymbol{e} \cdot H^T = |0000001| \cdot \begin{vmatrix} 1 & 1 & 0 \\ 0 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 0 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{vmatrix} = (0\ 0\ 1) \qquad (3.12)$$

Of cause, before the syndrome has been obtained the decoder does not know the error pattern, only Eq.(3.11) is performed in the decoder. Once the syndrome (001) is obtained, the error pattern $\boldsymbol{e}$ = 0000001 is specified. In the same way the other seven single-bit error patterns and their syndromes can be calculated and are shown in Table 3.2.

Unfortunately, the syndrome calculated from Eq.(3.9) does not specify an unique error pattern. For example, both 0000001 and 1010000 error patterns result in a syndrome 001. In Table 3.2 we have also provided two-bit error patterns which are sharing the same syndrome. They are denoted as error-pattern(I) and error-pattern(II), respectively. In fact, for a any codeword d, there are as many as 2^k error patterns that satisfy the Eq.(3.9). Only one of them is the real error pattern[3.2]. Therefore when a syndrome is obtained from Eq.(3.11), the decoder has to determine which error pattern is the real one. Strictly speaking, this is very difficult. Fortunately in computer memories, in which the 1

Table 3.2 Syndrome and error patterns

syndrome	error pattern I	error pattern II
001	0000001	1010000
010	0000010	0100001
100	0000100	0110000
101	0001000	0010010
111	0010000	1000001
011	0100000	1001000
110	1000000	0000110

error pattern I: all possible single errors.

error pattern II: some combinations of double errors.

-error and 0-error occur with the same probability, single error is dominant. Most probable error patterns are assumed to be ones that has the smallest number of 1's shown up in the column of error-pattern(I) in Table 3.2[3.3]. Otherwise multiple error correcting codes are required.

The capacity of error detection and correction

In coding we have another problem. If there are several errors occurring in a codeword, the codeword may be changed into another valid codeword in C rather than into an invalid codeword. In this case, neither the **H** matrix or the syndrome is impotent.

To reduce the occurrence of such decoding errors, it requires as many as possible bit positions in which the two codewords differ such that few errors are not able to change a valid codeword into another valid codeword. This idea can be accurately described by parameters of Hamming weight, Hamming distance and the minimum distance of a code.

The Hamming weight of a codeword or a vector x, denoted W(x), is the number of 1's in x.

The Hamming distance between two codewords x and y, denoted d(x,y), is the Hamming weight of (x - y). It also equal to the number of the positions in which the two codewords differ. That is,

$$d(x, y) = W(x - y) = W(y - x) \qquad (3.13)$$

The minimum distance of a code, denoted d_{min} is the minimum Hamming distance between all pairs of codewords in the code.

Clearly, the bigger the minimum distance of a code, the more errors are tolerant to change a valid codeword to another. Therefore the capability of error detection and error correction of a code is associated with the minimum distance of the code. A code of minimum distance d_{min} is able to detect $(d_{min} - 1)$ errors or correct $t =< [(d_{min} - 1)/2]$ errors in a codeword, where [Z] is the integer part of Z.

To create a code with required minimum distance d_{min}, we have following statement: A linear code with minimum distance d_{min} has a **H** matrix such that any $(d_{min} - 1)$ or fewer columns of the **H** matrix are linear independent [3.4]. This property can be used to construct linear codes of minimum distance d_{min}. In chapter four we will use this property to construct the modified Hamming codes.

3.2 Hamming codes[3.1]

In this section we will construct a very interesting class of single error correcting (SEC) codes. This codes are known as Hamming codes. Since the Hamming codes have capability of correcting single error in a codeword, the Minimum Hamming distance of the codes are at least $d_{min} >= 3$. According to the relationship between d_{min} and the **H** matrix, the **H** matrix satisfies: i) no column of the matrix is a zero vector, ii) every column vector is distinct, that is the sum of any two columns in the **H** matrix is not

a zero vector, iii) for systematic codes, the left most r columns form an (r x r) unit matrix (a Hamming code is not necessary a systematic code). Thus, for a given integer r, we can construct a (r x 2^r) **H** matrix of a Hamming code by using all r-tuples except the zero vector to be the columns of the H matrix. Therefor a hamming code has following parameters:

code length $n = 2^r - 1$
message word length $k = 2^r - r - 1$
parity bit number $r = n - k$
minimum distance $d_{min} = 3$
error correcting $t = 1$

A typical Hamming code is a (15, 11) code with distance-3. The **H** matrix of the code is given as follows

$$H = \begin{array}{c} p_0 p_1 p_2 p_3 d_4 d_5 d_6 d_7 d_8 d_9 d_{10} d_{11} d_{12} d_{13} d_{14} \\ \left|\begin{array}{ccccccccccccccc} 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 0 & 0 & 1 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 0 & 1 & 0 & 0 & 1 & 0 & 1 & 1 & 0 & 1 & 1 & 0 & 0 & 1 & 1 \\ 1 & 0 & 0 & 0 & 1 & 1 & 0 & 1 & 1 & 0 & 1 & 0 & 1 & 0 & 1 \end{array}\right| \end{array} \qquad (3.14)$$

A codeword in the (15, 11) code has a form of **x** = $p_0p_1p_2p_3x_4x_5x_6x_7x_8x_9x_{10}x_{11}x_{12}x_{13}x_{14}$, where p_i are parity bits and x_i are message bits. The parity bit are calculated through the parity check equations

$$\begin{aligned} p_0 &= x_4x_5x_7x_8x_{10}x_{12}x_{14} \\ p_1 &= x_4x_6x_7x_9x_{10}x_{13}x_{14} \\ p_2 &= x_5x_6x_7x_{11}x_{12}x_{13}x_{14} \\ p_3 &= x_8x_9x_{10}x_{11}x_{12}x_{13}x_{14} \end{aligned} \qquad (3.15)$$

The encoder for the code is then accordingly formed in

Fig.3.1.

The decoding of a read codeword

$$\mathbf{d} = d_0d_1d_2d_3d_4d_5d_6d_7d_8d_9d_{10}d_{11}d_{12}d_{13}d_{14}$$

uses the syndrome equations to identify the error position

$$\begin{aligned} s_0 &= d_3+d_8+d_9+d_{10}+d_{11}+d_{12}+d_{13}+d_{14} \\ s_1 &= d_2+d_5+d_6+d_7+d_{11}+d_{12}+d_{13}+d_{14} \\ s_2 &= d_1+d_4+d_6+d_7+d_9+d_{10}+d_{13}+d_{14} \\ s_3 &= d_0+d_4+d_5+d_7+d_8+d_{10}+d_{12}+d_{14} \end{aligned} \qquad (3.16)$$

If the syndrome $\mathbf{S} = (s_0s_1s_2s_3)$ is identical to the i^{th} column of the **H** matrix then a single error is assumed occurring at the i^{th} position of the codeword. The diagram of a decoder is shown in Fig.3.2.

However, if double errors occur, for instance, at position 2 and 6, the syndrome would be calculated as **S** = (0010) + (0101) = (0111). This will leads to a decoding error at position 8. To prevent decoding errors due to double error, a single-error-correcting/ double error detecting (shorted as SEC-DED) Hamming code can be easily obtained by adding a so called overall parity check to the **H** matrix of Eq.(3.14). The SEC-DED code has a H matrix as follows

$$\mathbf{H} = \begin{vmatrix} 0 & & \\ 0 & & \\ . & \mathbf{H} & \\ . & & \\ 1 & 1\ 1\ \ldots\ 1 & \end{vmatrix} \qquad (3.17)$$

The parity check equation corresponding to the last row of the new **H** matrix is an overall parity check

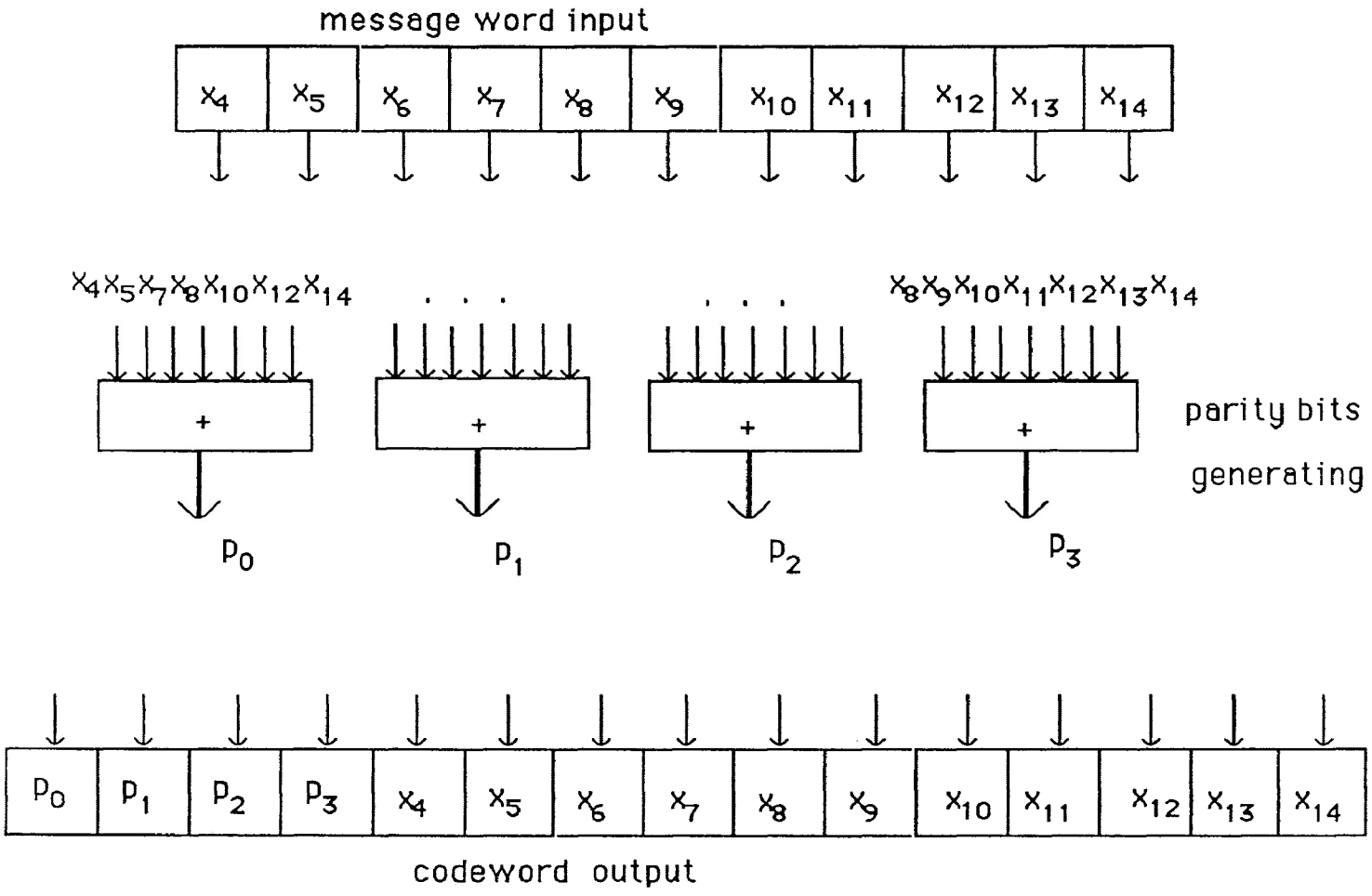


Fig. 3.1 An encoder for (15, 11) Hamming code

$$p_{0v} = p_0+p_1+p_2+p_3+x_4+x_5+x_6+x_7+x_8+x_9+x_{10}+x_{11}+x_{12}+x_{13}+x_{14}$$

The Minimum Hamming Distance of the SEC-DED code is then changed to $d_{min}=4$ and every codeword has an even Hamming Weight.

When decoding, an overall syndrome component s_{0v} is required in addition to the syndrome set (3.16)

$$s_{0v} = d_0+d_1+d_2+d_3+d_4+d_5+d_6+d_7+d_8+d_9+d_{10}+d_{11}+d_{12}+d_{13}+d_{14}$$

Then $\mathbf{S} = (s_{ov}s_0s_1s_2s_3)$ can be interpreted as

1) If $\mathbf{S} = \mathbf{0}$, the codeword is assumed error free.
2) If $s_{ov} = 1$, a single error is assumed and the error position is located by syndrome $(s_0s_1s_2s_3)$.
3) If $s_{ov} = 0$ and $(s_0s_1s_2s_3)$ not equal to zero, double errors (or even number of errors) are detected. These errors are uncorrectable.

By adding an overall parity check to the **H** matrix, we have obtained an even weight code which can correcting a single error and detect double errors simultaneously. On the other hand, by deleting some columns of **H** matrix of (3.17), we will obtain an odd weight Hamming code which can also correct single error and detect double errors. This odd weight Hamming code is called modified Hamming code and widely used in modern computer memories[3.5]. We will discuss the Modified Hamming codes in Chapter four in details.

3.3 Cyclic codes and BCH codes

In this section we only review the most important concepts of cyclic codes and BCH codes.

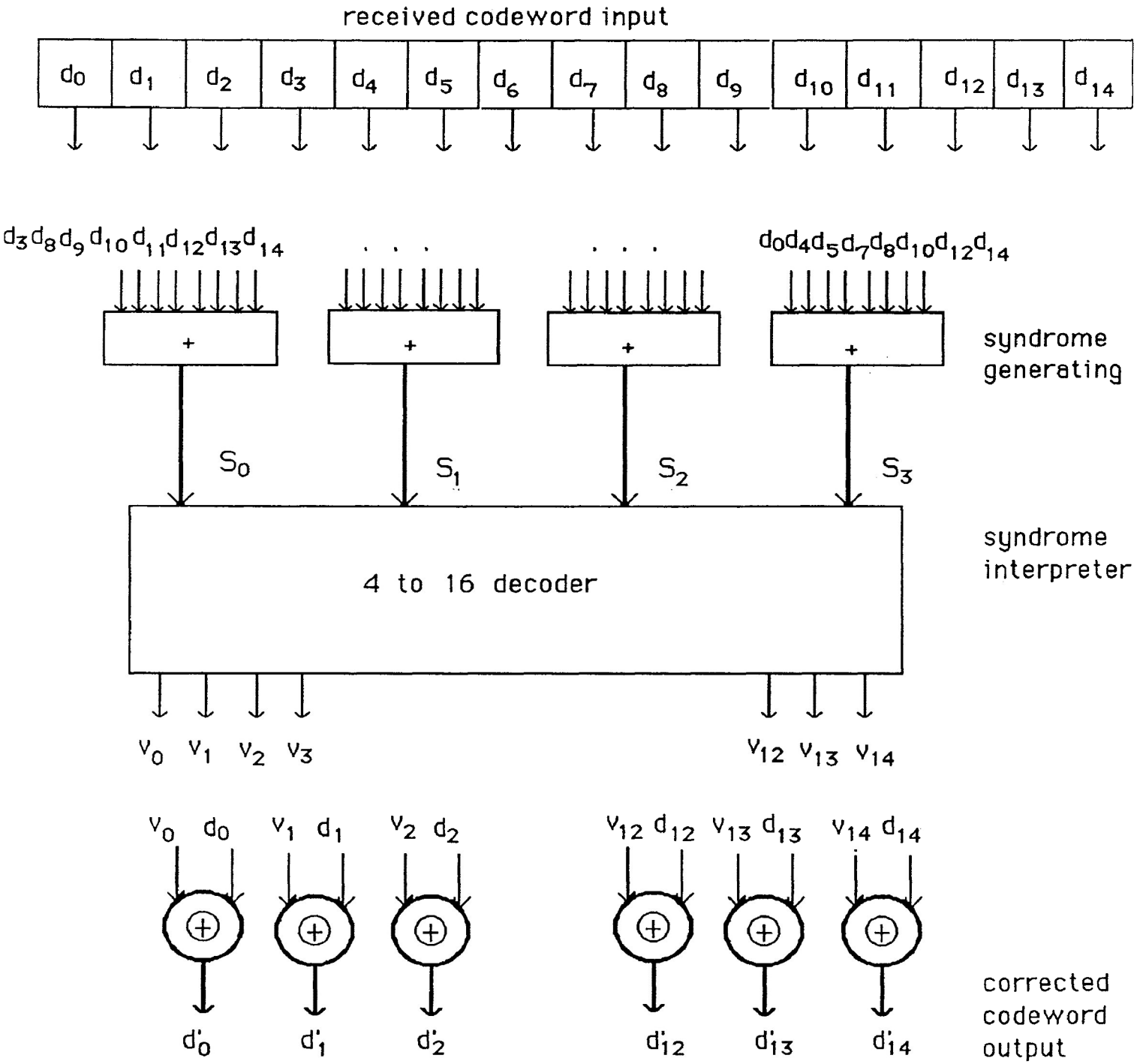


Fig. 3.2 A decoder for (15, 11) Hamming code

BCH codes are a class of cyclic codes.

Cyclic codes are well described by Galois field **GF**(q^m) of q^m symbols. For binary codes the fields are limited to **GF**(2^m).

Three representations of elements of field GF(2^m)

i) The elements of a field can be represented by the powers of its primitive element(s) u, that is

$$\mathbf{GF}(2^m) = \{0, u^0, u^1, \ldots u^i, \ldots u^{2^m-2}\}$$

Such a primitive element u of a field is the generator element of the field.

ii) The elements of the field can be represented by polynomials modula a so called primitive polynomial p(x) of degree m. Any polynomial representation of an element has degree at most m-1. The coefficients of the polynomial are over **GF**(2) or {0, 1}. The primitive polynomial P(x) has degree m and has u as its root. For different m, the primitive polynomials of field **GF**(2^m) can be easily found in literatures (see, e.g.[3.1]).

iii) The elements of the field can also be represented by m-tuple vectors. Each vector is corresponding to the coefficients of the polynomial described above.

The three representations are corresponded one by one. The power form representation is convenient to multiplication arithmetic and the polynomial representation is convenient to addition arithmetic.

Table 3.3 Three representations of elements of field GF(2^2)

powers of u	polynomials	vectors
0	0	0000
u^0	1	0001
u^1	x	0010
u^2	x^2	0100
u^3	x^3	1000
u^4	$x + 1$	0011
u^5	$x^2 + x$	0110
u^6	$x^3 + x^2$	1100
u^7	$x^3 + x + 1$	1011
u^8	$x^2 + 1$	0101
u^9	$x^3 + x$	1010
u^{10}	$x^2 + x + 1$	0111
u^{11}	$x^3 + x^2 + x$	1110
u^{12}	$x^3 + x^2 + x + 1$	1111
u^{13}	$x^3 + x^2 + 1$	1101
u^{14}	$x^3 + 1$	1001
$u^{15} = 1$		

Primitive polynomial $p(x) = x^4 + x + 1$

As a example, a field **GF**(2^4) represented in three forms are shown Table 3.3, where u is the primitive element and $p(x) = x^4 + x + 1$ is the primitive polynomial. Therefore $u^4 + u + 1 = 0$.

Minimal polynomials

For any element b ($=u^i$) of a field **GF**(2^m), there exists one and only one polynomial, $m_b(x)$, which has b as its root. This polynomial is called minimal polynomial of the smallest degree (m or less) of b. For element u, the minimal polynomial is $m_u(x)$.

Since the $m_u(x)$ has u, the primitive element, as its root, the $m_u(x)$ is also the primitive polynomial. Similarly, for element u^2, the minimal polynomial is $m_{u^2}(x)$ and so on.

Any minimal polynomial $m_j(x)$, which has u^j as its root, has u^{2j} as its roots as well, since $[m_j(u^j)]^2=m_j2(u^{2j})=0$.

Some elements of **GF**(2^m) may share a same minimal polynomial with some others. For example, $m_u(x)$ is the minimal polynomial of u, so is of u^2, u^4,... etc.

Cyclic codes

A linear code $C(=c_0c_1c_2...c_{n-1})$ is a cyclic code if every shift of a codeword in C is also a codeword in C.

A cyclic codeword of length n, $c=(c_0c_1c_2...c_{n-1})$, can be represented by a code polynomial $C(x)=c_0+c_1x..c_{n-1}x^{n-1}$. The degree of the polynomial is at most n-1.

A cyclic code C can be generated by a polynomial called generator polynomial. Suppose a(x) is the information

polynomial and g(x) is the generator polynomial, then the encoded codeword polynomial c(x) is obtained from

$$c(x) = a(x)g(x) \bmod x^n-1 \qquad (3.18)$$

The generator polynomial g(x) has following properties:

a) g(x) is a factor of $x^n - 1$, or say, $x^n-1 = g(x)h(x)$

b) The degree of g(x) is r = n-k

c) g(x) is the smallest degree monic polynomial in C, that is, if the general form of g(x) is

$$g(x) = g_r x^r + g_{r-1}x^{r-1} + \ldots + g_1 x + g_0 \qquad (3.19)$$

then g_r and g_0 have to be always 1.

To encode a systematic cyclic code, three steps are required:

1) multiple a(x) by x^{n-k}.

2) find the remainder polynomial b(x) resulting from dividing $x^{n-k}a(x)$ by g(x), that is, $x^{n-k}\, a(x) = d(x)g(x) + b(x)$.

3) Form a systematic cyclic codeword

$$c(x) = b(x) + x^{n-k}a(x) \qquad (3.20)$$

The decoding of a cyclic code C is based on

$$r(x)\ h(x) = 0 \bmod x^n-1 \qquad (3.21)$$

where r(x) is the received codeword polynomial and h(x) is the parity polynomial.

BCH codes

Given code length n, information length k and error correction number t, a BCH code has parameters as follows

$n = 2^m - 1$

$r = n - k <= mt$

$d >= 2t + 1$

The generator polynomial of the BCH code is constructed as follows:

$$g(x) = \text{LCM} \{ m_1(x)\ m_3(x)\ \ldots\ m_{2t-1}(x) \} \qquad (3.22)$$

where $m_j(x)$ is the minimal polynomial of u^j.

A BCH codeword is encoded as

$$c(x) = a(x)g(x)$$

The syndrome of a received codeword r(x) is a 2t-tuple

$$\mathbf{S} = (s_1\ s_2\ \ldots\ s_{2t})$$

where

$$s_1 = r(u) = e(u)$$
$$s_2 = r(u^2) = e(u^2)$$
$$\ldots \quad \ldots$$
$$s_{2t} = r(u^{2t}) = e(u^{2t})$$

where e(x) is the error polynomial: $e(x)=a(x)+r(x)$.

For double error correcting BCH codes the possible error combination is

$e(x) = 0$	no error
$= x^i$	single error at position i
$= x^i + x^j$	double error at positions i and j

The syndrome is interpreted as

a) if there is no error, then **S** = (0000)

b) if there is a single error, then

$$s_1 = e(u^i) = u^i$$
$$s_2 = e(u^{2i}) = u^{2i} = s_1^2$$
$$s_3 = e(u^{3i}) = u^{3i} = s_1^3$$
$$s_4 = e(u^{4i}) = u^{4i} = s_1^4$$

c) if there are two errors

$$s_1 = e(u^i + u^j) = u^i + u^j$$
$$s_2 = e(u^{2i} + u^{2j}) = u^{2i} + u^{2j} = s_1^2$$
$$s_3 = e(u^{3i} + u^{3j}) = u^{3i} + u^{3j}$$
$$s_4 = e(u^{4i} + u^{4j}) = u^{4i} + u^{4j} = s_1^4$$

Among these syndrome components, s_1 and s_3 are independent.

Let $s_1 = u^i + u^j = A + B$

Then $s_3 = u^{3i} + u^{3j} = (u^i + u^j)(u^{2i} + u^{2j} + AB)$
$= s_1(s_1^2 + AB)$

and $AB = s_3/s_1 + s^2_1$

A and B contain the information about error positions. To calculate A and B, form a equation

$$z(x) = (1 + Ax)(1 + Bx) = 1 + (A + B)x + ABx^2$$
$$= 1 + s_1x + (s_3/s_1 + s^2_1)x^2 = 0 \qquad (3.23)$$

The roots of the equation are $x_1 = A^{-1}$ and $x_2 = B^{-1}$. Therefore the roots of the equation locate the error positions. The equation of Eq.(3.23) is called error location equation.

To solve the Eq.(3.23) over field **GF**(2^m), it is necessary to try every element of **GF**(2^m). If there are two errors in the codeword, there must be two elements that satisfy the error location equation (or say there are two elements that will be the roots of the equation) respectively. Then the inverse of these two elements indicate the error positions.

Example Consider a (15, 7) double error correcting BCH code in which

$n = 15 = 2^4 - 1$ (so the code is defined on **GF**(2^m), $m = 4$)

$k = 7$ (message bits)

$t = 2$ (error correcting capability)

$r = n - k = 15 - 7 = 8$ $(= mt)$ (parity bits)

$d >= 2t + 1 = 5$ (minimum Hamming distance)

The elements of field **GF**(2^4), which has primitive polynomial $p(x) = x^4 + x + 1$, has been shown in Table 3.3

Suppose a received codeword polynomial is

$$r(x) = x^{10} + x^9 + x^8 + x^4 + x^2$$
$$= (0000\ 1110\ 0010\ 100)$$

The syndrome is calculated as $\mathbf{S} = (s_1\ s_2\ s_3\ s_4)$

$$s_1 = r(u) = u^{10} + u^9 + u^8 + u^4 + u^2 = u^{12}$$
$$s_3 = r(u^3) = (u^{10})^3 + (u^9)^3 + (u^8)^3 + (u^4)^3 + (u^2)^3$$
$$= u^{30} + u^{27} + u^{24} + u^{12} + u^6$$
$$= 1 + u^{12} + u^9 + u^{12} + u^6 = u^{10}$$

Thus the error location equation is, since $s_3/s_1 + s^2_1 = u^{10}$,

$$z(x) = 1 + u^{12}x + u^{10}x^2 = 0$$

Replace x by every element of **GF**(2^4) to find out the roots of the equation.

Since $z(u^{11}) = 1 + u^{12}u^{11} + u^{10}u^{22}$

$= 1 + u^8 + u^2$

$= 0001 + 0101 + 0100 = 0$

and

$$z(u^9) = 1 + u^{12}u^9 + u^{10}u^{18}$$
$$= 1 + u^6 + u^{13}$$
$$= 0001 + 1100 + 1101 = 0$$

the two roots are

$$x_1 = u^{11}$$
$$x_2 = u^9$$

the inverse of x_1 and x_2 indicate the error positions

$$(x_1)^{-1} = u^4$$
$$(x_2)^{-1} = u^6$$

therefore the error pattern is found out

$$e(x) = x^6 + x^4$$

the corrected codeword is then obtained by

$$r'(x) = r(x) + e(x)$$
$$= (x^{10} + x^9 + x^8 + x^4 + x^2) + (x^6 + x^4)$$
$$= x^{10} + x^9 + x^8 + x^6 + x^2$$
$$= (0000\ 1110\ \underline{1}0\underline{0}0\ 100)$$

where the underlined bits are bits that are corrected.

3.4 Summary

This chapter has presented basic concepts of linear codes. By adding some redundancy bits to the message bits, coding can detect and correct errors in the codeword. Therefore coding can be used to improve the reliability of computer memories.

A code is defined by either generator matrix or parity check matrix. A syndrome of a codeword provides the information of error positions in the codeword. Systematic codes are more suitable for computer memories. Generally, more redundancy in a code has more

error correcting ability. But as we have seen redundant bits require more storage space and longer encoding/decoding time. When a code is determined or selected for a computer memory, one has to consider these unfavourable factors. In most cases, the error in semiconductor memories affect single bits and are distributed randomly. Therefore, single error correcting (SEC) Hamming codes are widely used. If a single overall parity check bit is simply added to the SEC code, we can obtain a single error correcting and double error detecting (SEC-DED) code. On the other hand, by deleting some message bits from a SEC-DED code, we can obtain a code of different length without loosing the original ability of error correcting and detecting. This is very important to meet the requirement of computer message word length which usually contain 2^i bits.

In some situations, multiple error correcting ability may be required. Cyclic codes are designed for these purposes. An important subclass of the cyclic codes is BCH codes. BCH codes are well described by Galois fields $\mathbf{GF}(2^m)$. Like the generator matrix for Hamming codes, a BCH code is defined by its generator polynomial. When a syndrome of a BCH codeword is calculated, we still need to locate the positions of errors. In this chapter we have demonstrated the error locating process for double errors. If there are more than two errors in the codeword, the error locating process will be more complicated.

So far, the BCH codes are not very popular in computer memories. But, because of their multiple error correcting ability,

they are still attractive candidates.

In chapter four we will introduce an alternative multiple error correcting code, Orthogonal Latin Square codes which is more suitable for computer memories.

References

3.1 W. Wesley Peterson, E. J. Welson, JR. "Error Correcting Codes", MIT press 1986

3.2 Arnold M. Michelson, Allen H. Levesque "Error-control Techniques for Digital Communication", A Wiley-Interscience Publication Johnwiley & Sons, 1985

3.3 Shu lin, Daniel J. Costello, JR, "Error Control Coding Fundamental and Applications", Prentice-Hall, Inc., Englewood Cliffs New Jersey, 1983

3.4 Sccot A. Vanstone, Paul C. Van Orschot "An Introduction to Error-Correction Codes with Applications", Kluwer Academic Publishers, 1989

3.5 Bary W. Johnson, Design and Analysis of Fault Tolerant Digital System, Addison Wesley, 1989

3.6 T. R. N. Rao/E. Fujiwara, Error-control Coding for Computer Systems , Prentice-Hall Inc., 1989

CHAPTER FOUR CODES USED IN COMPUTER MEMORIES

In computer memories both error detecting and error correcting codes are used. Error detecting codes are simple in implementation compared to error correcting codes. But error correcting codes provide more improvements of memory reliability, which is essential for modern computer systems.

In various applications either of them respectively or both of them simultaneously are implemented. In this chapter several commonly used error control codes are discussed. They are classified as error-detecting codes, single-error-correcting/ double-error-detecting codes and multiple error correcting codes. Next chapter another type of codes which are used for detecting and correcting unidirectional errors will be discussed.

4.1 Criteria for code selection

Coding can dramatically improve the performance and the reliability of computer memories. But, it requires additional memory cells for storing check bits and extra circuitry for implementation of encoding and decoding logic. These not only consume additional memory space and VLSI area but also significantly increase the risk of potential hard errors and soft errors. The encoding and decoding processes will increase the memory access time. Even a delay of one microsecond in handling critical path information in a computer could be unacceptable for high speed computer systems. Hence, though it has many advantages

towards the reliability of computer memory, coding might also bring some liability to the performance of computer memories if not properly designed.

when making a decision to use ECC codes and to select a particular type of ECC code, several factors should be considered carefully. The first consideration is whether error detecting, error correcting, or both are really required. The second consideration is the number of bit errors that need to be detected and corrected. Generally speaking, the coding delay (including encoding and decoding delay), circuitry complexity, extra memory cells and cost will increase when more error detection or error correction capability is required. Sometimes the decision is really based on a kind of tradeoff between benefits and disadvantages. However, when a code is to be used in the memory following general criteria of design have to be considered.

i) Speed of performance

In a high speed memory the detection and correction process of a code should not unduly increase the memory access time. The delay should be less than five to ten per cent of the memory access time. For this reason, the codes which can accommodate parallel encoding and decoding, for example separable linear codes, are almost the first selection. Cyclic codes such as BCH codes are usually not suitable for semiconductor memories.

ii) Simplicity of Encoding and Decoding

Simplicity of implementation of ECC is essential for a large capacity memory due to the limited area in the VLSI memory. The simpler ECC scheme generally means less cost and more space for information data.

Since in semiconductor memories single errors are dominant error patterns, single error correcting and double error detecting codes are often preferable candidates in computer memory designs.

iii) Efficiency of Implementation

The efficiency means use of as little extra encoding and decoding circuitry as possible to perform error detection and correction. It is not only for economy, but also for eliminating potential source of circuit failures. It will be seen that modified Hamming codes are those very high efficiency codes.

Error detection codes provide faster decoding and simpler implementation than error correction codes, therefore, are preferable in small memories in which error detection is considered sufficient[4.1]. In large memories, however, since the probability of errors could be much higher than in small memories, error detection itself becomes not enough to maintain the high reliability. As a consequence, error correction ability has to be considered. On other hand, considering the error distributions in semiconductor memories, single error correcting codes are usually

sufficient for improvement of reliability. Moreover, if we add one more check bits to a single error correcting code, we can easily obtain a new code which has capability of single error correcting and double error detecting simultaneously (SEC-DED). Such SEC-DED codes are suitable and economical to large memories. Multiple error correcting codes provide even more reliability improvement, but as we have seen in chapter three, multiple error correction usually results in complicated implementation and longer decoding delay. Thus high speed computers can not afford it. However, in some special applications multiple error correction code might be required.

4.2 Error detecting codes

As mentioned earlier, due to its simplicity of implementations, error detecting codes are very suitable for small memories. In this section following error detecting codes will be discussed

i) Parity codes

ii) Duplication codes

iii) m-of-n codes

4.2.1 Parity codes

The simplest form of a error detecting codes are the parity codes. A single-bit parity code is constructed by adding an extra bit to the binary information data such that the resulting codeword has either odd number of 1's or even number of 1's. If the

information has n-1 bits then the code word has length of n. The code can be expressed as (n, n-1). Depending on odd or even number of 1's in each of its codeword, there are odd parity code and even parity code. In most implementation of computer memories, even parity codes are used, but both odd and even parity codes have exactly the same properties[4.2]. For the convenience of describing only even parity codes are included in the following discussion.

Encoding and decoding

The single-bit parity codewords have a typical construction of

$$i_1\ i_2\ \ldots\ i_k\ p$$

where i_1, i_2 and i_K are arbitrary information bits and p is the parity bit. The parity bit p is chosen in a way to satisfy the Eq. (4.1)

$$i_1 + i_2 + \ldots i_k + p = 0 \pmod 2 \qquad (4.1)$$

As a codeword is read from the memory decoding is performed. The decoding starts with syndrome generation. The syndrome is given by

$$S = d_1 + d_2 + \ldots + d_k + p_1 \pmod 2 \qquad (4.2)$$

where d_1, d_2 and d_k are the received data bits and p_1 is the parity bit read from the memory.

If S = 0, the codeword read from the memory is assumed correct;

Table 4.1 (4, 3) parity codes for BCD data

decimal digit	BCD	odd parity code	even parity code
0	0000	00001	00000
1	0001	00010	00011
2	0010	00100	00101
3	0011	00111	00110
4	0100	01000	01001
5	0101	01011	01010
6	0110	01101	01100
7	0111	01110	01111
8	1000	10000	10001
9	1001	10011	10010

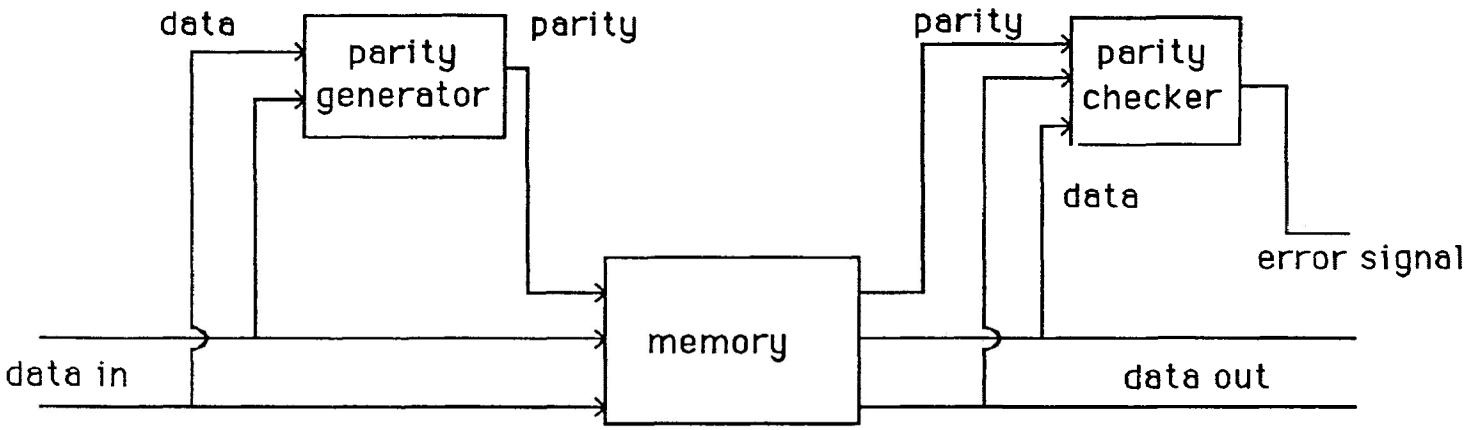


Fig. 4.1 An organization of a memory using single-bit parity code

if S =1, the decoder declares the codeword to be in error.

An example of single bit parity codes (both odd and even) are illustrated in the Table 4.1. The second column of the Table contains information data. The third column shows corresponding odd parity codes and the fourth column contains even parity codewords. The parity bits are underlined to make them distinct.

In Fig.4.1[4.3] the organization of a memory that uses single bit parity codes is shown. In the encoding process the parity generator creates a parity bit p_1. Then the parity bit and the original information bits are stored in the memory as a complete codeword. When the codeword is read out from the memory the decoding will be first performed. Based on the information data of codeword, a new parity bit is regenerated in the parity checker. The parity checker compares the new parity bit and the old parity bit read from the memory. If they agree, the codeword is assumed to be correct. If they are in disagreement, an error is then detected and an error signal is sent out the system.

Analysis of Error Detection

The parity generator and the parity checker used in the memory system are very simple and they have almost the same circuitry. In Fig.4.2 an even parity generator and parity checker for 4-bit information data are shown. It can be seen that single bit parity codes have minimum Hamming distance of 2. A codeword of minimum Hamming distance d_{min} is able to detect d ($=< d_{min}-1$) errors. Therefore the codes can detect a single error in a codeword but not

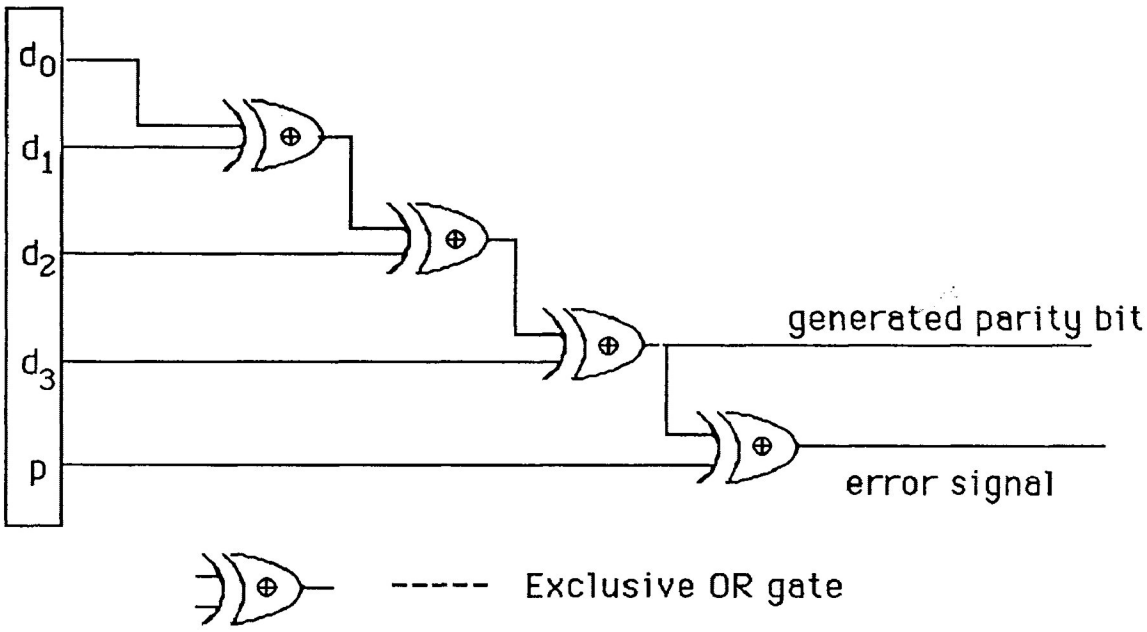


Fig 4.2 A 4-bit generation and checker circuit for even parity

correct it. For example, in a even parity code, a single error (either 1-error or 0-error) will change the codeword into another codeword which has odd number of 1's. This is not a valid codeword in the even parity code, hence the error is detected. But when a second error occurs in the same codeword, the total number of 1's will change back to even and the codeword then becomes a valid codeword in the even parity code. Thus the errors will not be detected. Any codeword that contains even number of errors will be incorrectly taken as a correct codeword.

For a random error memory, the probability of any particular pattern of j errors for a n-bit word is

$$\text{Prob (j-error)} = p^j (1-p)^{n-j} \tag{4.3}$$

where p is the probability of occurrence of bit-error in the memory. In most cases, $p << 1$.

The probability of zero errors in a codeword is then

$$\text{Prob(no-error)} = (1-p)^n \tag{4.4}$$

Using Eq.(4.3) and Eq.(4.4), the probability of any particular pattern of two errors (for which the parity code can not detect) can be derived as following:

$$\begin{aligned} Prob(2\text{-}errors) &= p^2(1-p)^{n-2} = \left(\frac{p}{1-p}\right)^2 (1-p)^n \\ &= Prob(0\text{-}errors)\left(\frac{p}{1-p}\right)^2 \end{aligned} \tag{4.5}$$

Since (1-p) >> p, the Prob(2-error) is much lower than the Prob(0-errors). That means the probability of correct decoding is much higher than that of incorrect decoding (when two errors occur).

In general, for single-bit parity codes, the probabilities of correct decoding, error detection and decoding failure are expressed as follows:

$$P(\text{correct decoding}) = (1-p)^n \qquad (4.6)$$

$$P(\text{error detection}) = \sum_{j=1,\, j=odd}^{n} p^j(1-p)^{n-j} \qquad (4.7)$$

$$P(\text{decoding failure}) = \sum_{j=2,\, j=even}^{n} p^j(1-p)^{n-j} \qquad (4.8)$$

Other parity codes

The concept of single-bit parity code can be extended to other forms which provide additional error detecting capability. Some of the forms are described as following:

a. Bit-per-word parity code

Bit-per-word parity code is the simplest type of parity codes. The form of a codeword is shown in Fig.4.3 (a). This code usually can detect odd number of errors. However, if a codeword including

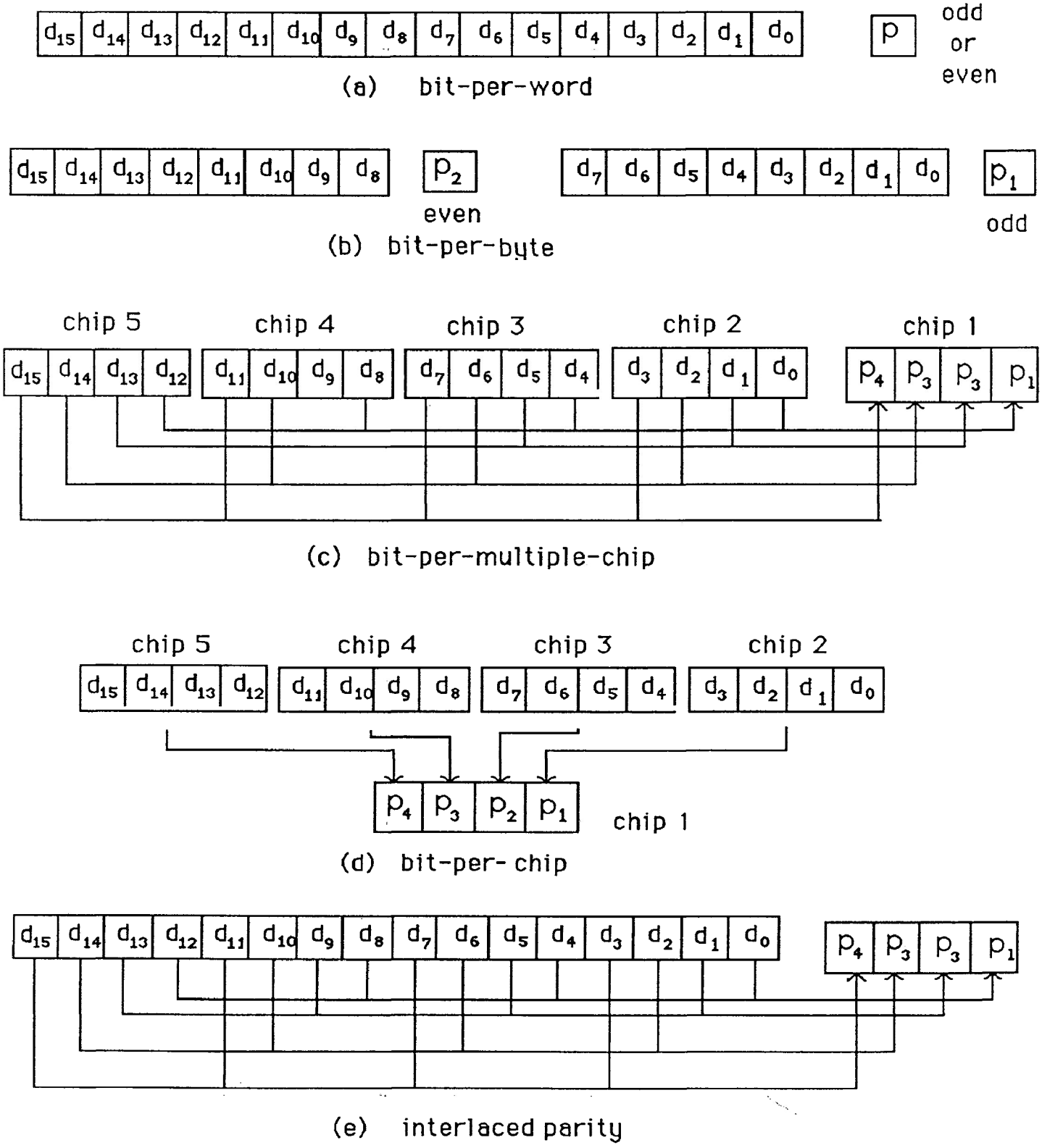


Fig. 4.3 The basic forms of parity codes

the parity bit becomes all 1's due to a complete failure of a bus or a set of data buffers, an odd parity code will fail to detect them because the faulty codeword would have odd number of 1's(supposing a codeword has a odd number length), similarly an even parity code will fail to the all 0's error.

b. Bit-per-byte parity codes

A bit-per-byte parity code is effective against the above failures. In coding, a word is segmented into two bytes. The parity bit of the first byte is formed by even parity rule and the parity bit of the next byte is formed by odd parity rule as shown in Fig.4.3 (b). One parity bit is for all 1's failure and another parity bit detects the all 0's failure.

c. Bit-per-multiple-chip parity codes

Many memories have multiple-bit-per-chip organization with each chip containing 4-bit, 8-bit or more. In these organizations if one chip fails, several bits in one codeword can be affected.

It is easy to see that both bit-per-word and bit-per-byte parity codes discussed above are ineffective for the situation. Bit-per-multiple-chip parity codes, shown in Fig.4.3 (c), can be used successfully for this type of errors. Since a faulty chip will corrupt every parity bit, a whole chip failure is detected.

d. Bit-per-chip parity code

If a bit-per-chip parity code is employed, it not only detects

errors but also locates the failed chip of failure. In this approach each parity bit is associated with one chip of memory, as in Fig.4.3 (d). For instance, p_1 is the parity bit for data group containing data bits 0, 1, 2, and 3. If a single bit becomes erroneous, the chip that contains the erroneous bit can be identified and the existence of the error is detected. This is very useful when the hardware maintenance scheme is employed in the memory.

e. Interlace parity codes

Interlace parity code is illustrated in Fig.4.3 (e). As can be seen from the figure, no two adjacent information bits are associated with the same parity group, therefore when any two adjacent bits are in error, two parity bits will be affected, and the errors are detected.

4.2.2 Duplication codes

The principle of duplication codes is very simple. It duplicates the original information bits to form a codeword. That is , there are two portions in a codeword, the first portion is the original information bits and the second portion is exactly the copy of the first portion. The decoding is done simply by comparing the two portions in a codeword. If the two portions do not agree, the errors are detected. The primary advantage of the code is its simplicity of encoding and decoding, but it occupies twice the storage space.

Types of duplication codes

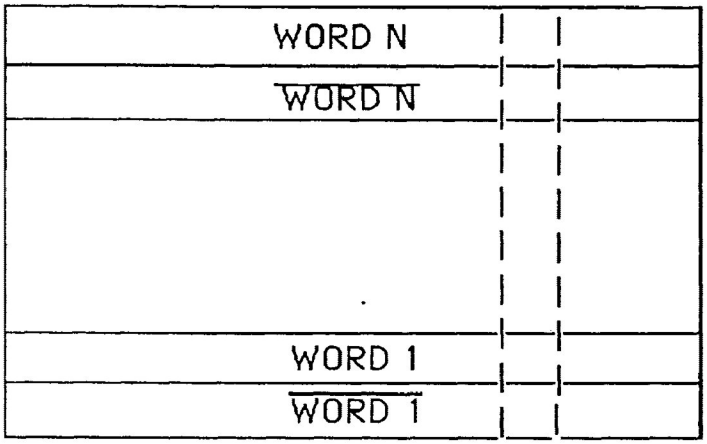
A variation of the basic duplication code is to take the complement of original information bits as its duplicated portion, as shown in Fig.4.4 (a).

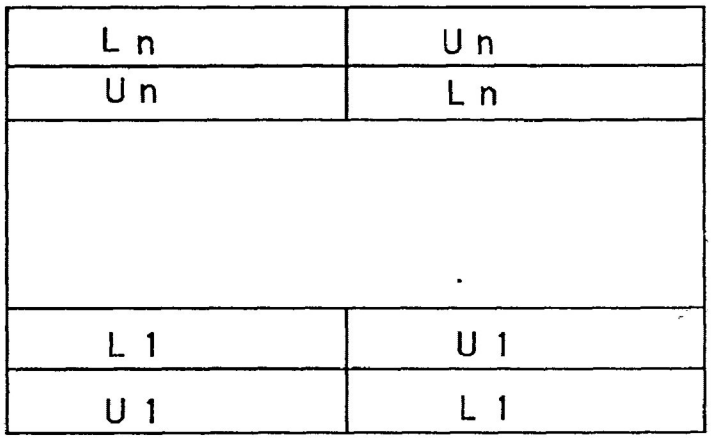
The effectiveness of this code can be analyzed as follows. Suppose that one bit slice of the memory becomes faulty such that every bit in that slice stuck at 1 regardless of what it should be. The complemented duplication code can detect this error. Because any two bits belonging to different portions (information portion and complemented portion) are supposed to be complement of each other.

Another variation is the swap and compare approach. The first portion of the codeword is divided into two parts called upper half and lower half. In the duplicated portion of the codeword, the upper half and lower half are swapped as shown in Fig.4.4 (b). A single bit slice that is faulty affects upper half and lower half,but the other two halves remain correct. By comparing the appropriate halves, the error is detected.

4.2.3 m-of-n codes[4.4, 4.5]

In a m-of-n code, every codeword of length n contains m 1's. Therefore it is a constant weight code. Any single-bit error will cause the codeword to have either m+1 or m-1 1's and a second bit error might change the erroneous codeword back to m 1's. The m-of-n code is of $d_{min} = 2$ and hence has the capability of detecting one error in a codeword.





L i --- Lower half of word i
U i --- upper half of word i

(a) Complement duplicate codes

(b) swaped duplicate code

Fig 4.4 duplicate codes

The easiest way to construct an m-of-n code is to take the original k information bits and append another k bits. The appended k bits are the bit by bit complement of the original information bits. This is a k-of-2k code with systematic construction. This code is also called balanced code since it has equal number of 0's and 1's in each of its codewords. However, most of m-of-n code are non-systematic codes.

The m-of-n codes can be used not only for single error detecting, but also for multiple-unidirectional error detecting[4.5] purposes. We will discuss the issue in the next chapter.

Now, in the following section error correcting codes will be discussed.

4.3 Single-error-correcting and double-error-detecting codes

In semiconductor memories, especially in the bit-per-chip organized memories, the single-bit errors are dominant error patterns[4.6]. The probability of soft errors show much higher than hard errors[4.7]. Therefore single error correcting codes are of significance for the reliability of large capacity and high speed semiconductor memories. In addition, the implementation of single error correcting codes are much simpler and the decoding speed is much faster than the multiple error correcting codes. Hence single error correcting codes are widely used in computer industries.

In this section two classes of single error correcting codes, H-V-parity codes and modified Hamming codes, are discussed.

4.3.1 H-V-parity codes

Single-bit parity codes detect errors occurring in a codeword but not correct them. This is because each information bit is associated with only one parity bit. If each of the information bits is allowed to appear in more than one parity groups (overlap parity check), the code will be able to correct errors as well.

The construction of the H-V-parity codes

The H-V-parity codes are constructed directly on a principle of overlap parity check. As shown in Fig.4.5, the information bits are arranged in a two-dimensional array in memory. Each row has a single parity bit calculated along the horizontal direction and each column has a single parity bit along the vertical direction. Each row data are called H-group data and each column data are called V-group data. In this array every information bit belongs to both a H-group data and a V-group data. For example, a horizontal parity bit (H-parity) P_1 is formed by data bits d_0, d_1,d_2 and d_3 in the first row of the array, and the vertical parity bit (V-parity) P_5 is formed by data bits d_0, d_4, d_8 and d_{12} in the left-most column etc.. Notice that P_9 is the parity bit of parity bits. It can be calculated on either H-parity bits or V-parity bits. In some cases this bit is not used at all.

Analysis of Error Correction/detection

In such a memory array, a single bit error in any row (or

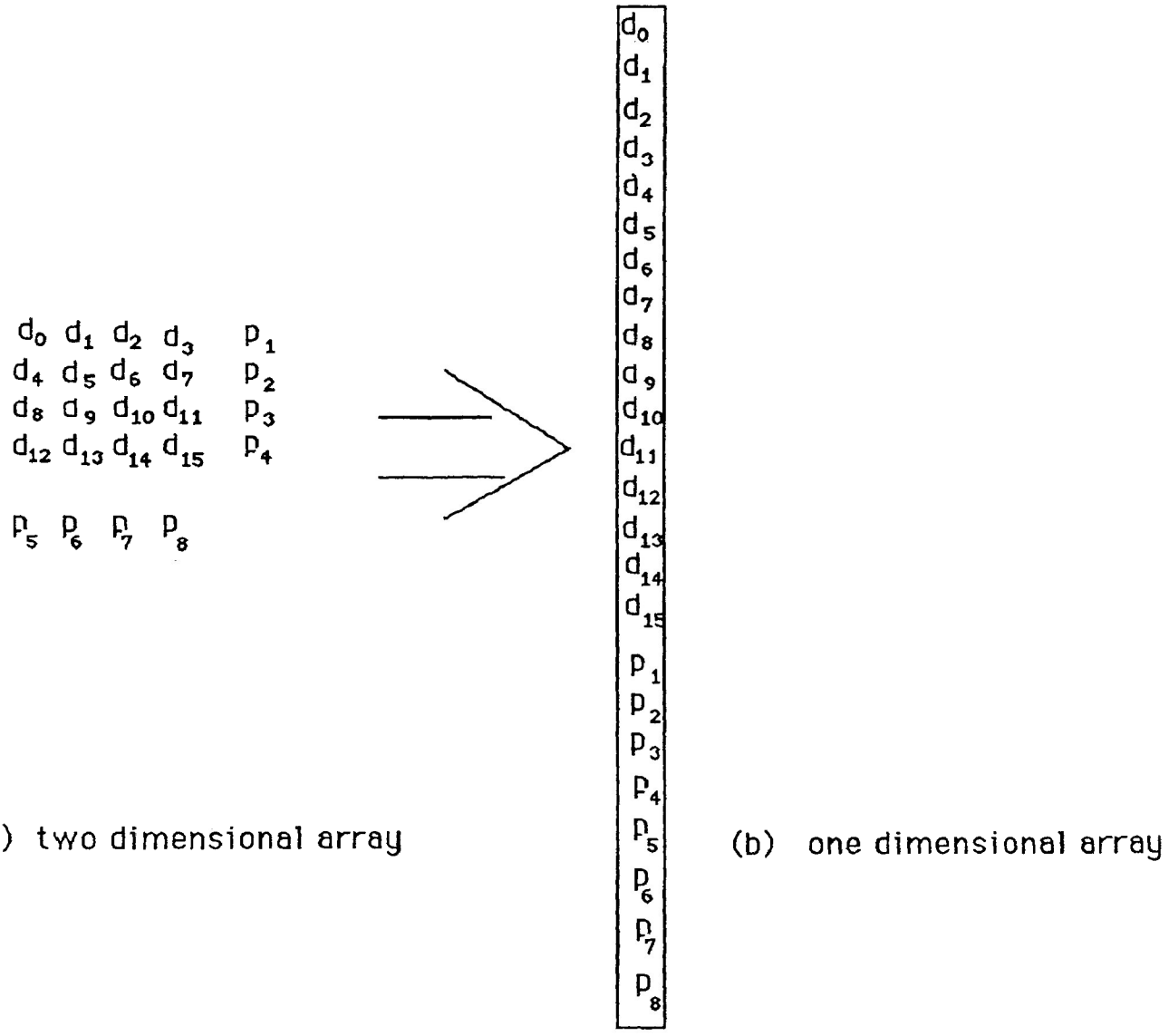


Fig 4.5 H-V- parity code

column) will cause both the H- and V-parity bits to declare error detected. The location of the error bit is then determined at the intersection of the row and column of the corresponding parity bits. Then the correction can be done. If two errors occur along a row, two V-parity bits associated with the errors will be in failure. However the H-parity bit, which is associated with the two errors in the row, fails to detect the presence of the errors. As a result, the locations of the errors can not be identified uniquely. Therefore, the H-V-parity code is a type of single-error-correcting and double-error-detecting code.

In fact, the H-V-parity code can be considered as a product of single-bit row parity code and single-bit column parity code. Both of them are linear codes. Therefore the minimum Hamming distance of the product code is the product of that of two single-bit parity codes[4.8], that is $d_{min}= 2 \times 2= 4$.

Due to its simple encoding and decoding logic and implementation, the H-V parity code is suitable for on-chip error correcting and error detecting scheme. In the following, we discuss a possible implementation of the H-V-parity decoder for computer memory[4.9].

The decoding of the H-V-parity codes

A block diagram of decoding logic is given in Fig.4.6. There are six main blocks in the diagram. H-group data selector, V-group data selector, H-parity generator, V-parity generator, correction circuitry and output multiplexer.

The data bits are organized into a one dimensional array from

a two dimensional array containing information bits and parity bits. Fig.4.5 is an example of such a transformation. It can see that each bit of the array is now connected with a word-line in the memory so that all of them can be accessed in parallel. During the error correction, according to the decoding logic, the H- and V-group data selectors transfer proper data bits into H-parity and V-parity generators respectively. In the parity generator, new H- and V-parity bits are generated. By comparing the new parity bits and the corresponding memory cell parity bits (the parity bits read from the memory), error(s) can be detected. If the error is detected as a single-bit error, the correction is then performed through a multiplexer and correction circuits.

It can be seen that the data selectors which feed in the desired data bits to corresponding parity generators and the multiplexer are the vital components for the whole process of error detecting and correcting.

An analysis of a decoder for a (19, 12) H-V-parity code

To have a better understanding of the process, a more detailed error correcting circuit is shown in Fig.4.7. This is a decoder for a (19, 12) H-V-parity code. Each codeword is transferred from a 4 x 5 array which contains twelve information bits, three H-parity bits and four V-parity bits. The twelve information data bits are denoted as d_1, d_2, ... d_{12}. The procedure of correction is divided into twelve subcycles. Each subcycle is able to check and correct one bit. In every subcycle, according to the decoding logic, data selectors feed in appropriate data bits to different

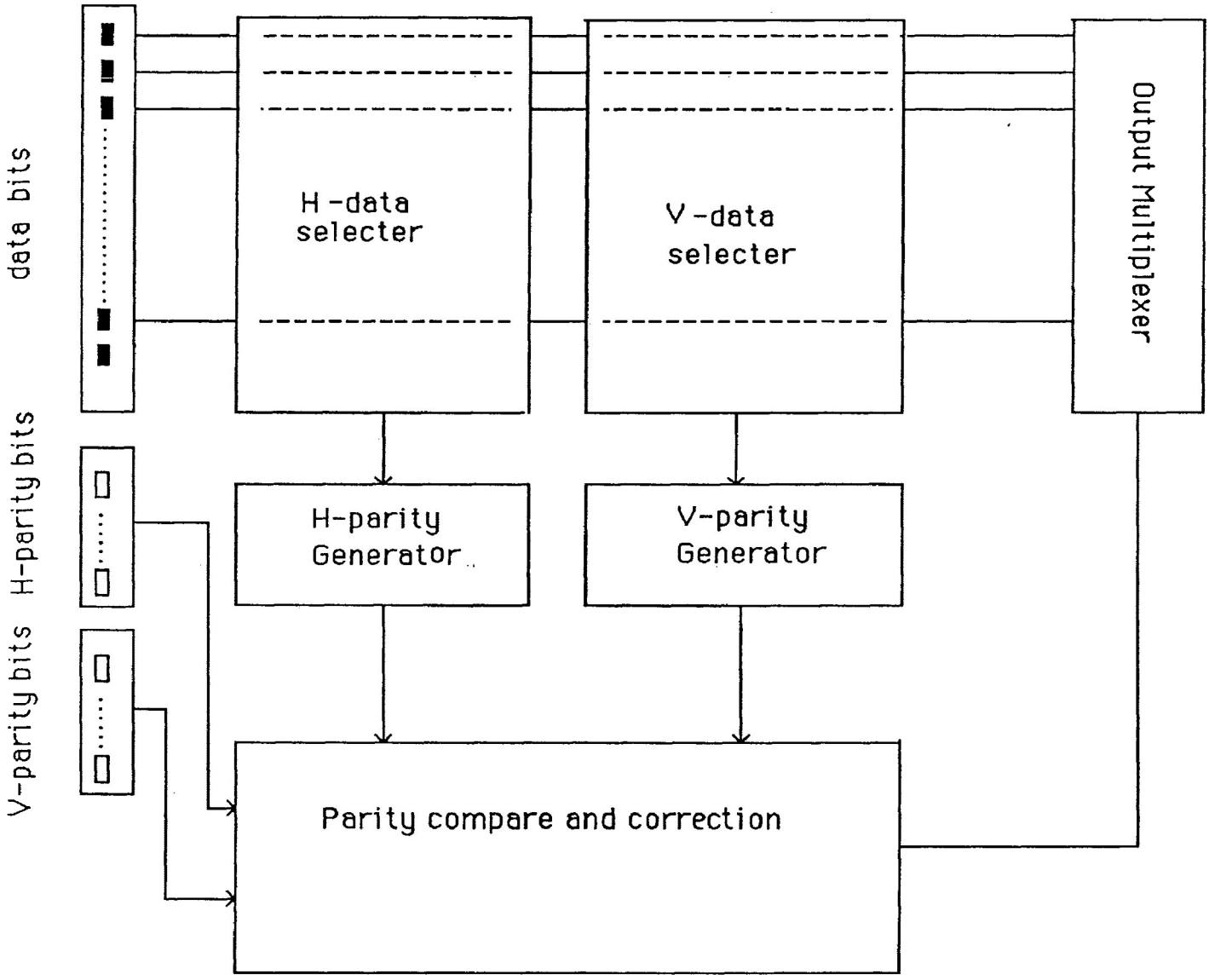


Fig. 4.6 A block diagram of decoding for H-V-parity code

blocks so that correcting decisions can be properly made. Table 4.2 shows the bit sequences that are transferred into H-parity generator, V-parity generator and output multiplexer in each subcycle.

Let us suppose that bit d_1 is in error, then during the second subcycle, bits d_0, d_1, d_2, and d_3 are fed in H-parity generator and bits d_1, d_5 and d_9 in V-parity generator. Consequently, the new H-parity bits h_0' and the new V-parity bit v_1' are generated. Since the new parity bits do not agree with the corresponding memory cell parity bits, the error is detected in both H- and V-direction at the same time. As a result, the correction circuit outputs a correcting signal '1', and correction is then made by toggling the output of multiplexer.

The whole decoding process has to be completed during one read cycle of the memory. That is, in this example, the time to correct one bit or one subcycle is less than 1/12 of the read cycle.

This ECC scheme using H-V-parity code has been applied to the design of 256k RAM[4.9]. In addition to fundamental 256k memory cells, the RAM employs 24k parity cells for H- and V-parity bits. In the scheme, each word line is connected to 512 (=16x32) memory cells and 48 (=16+32) parity cells. There are total 500 such word-lines. All of the memory cells can be checked within 4 seconds with each subcycle being 16 ns. The scheme has substantially improved the reliability of the memory compared to that of the memory without the ECC. The estimated improvement is reported up

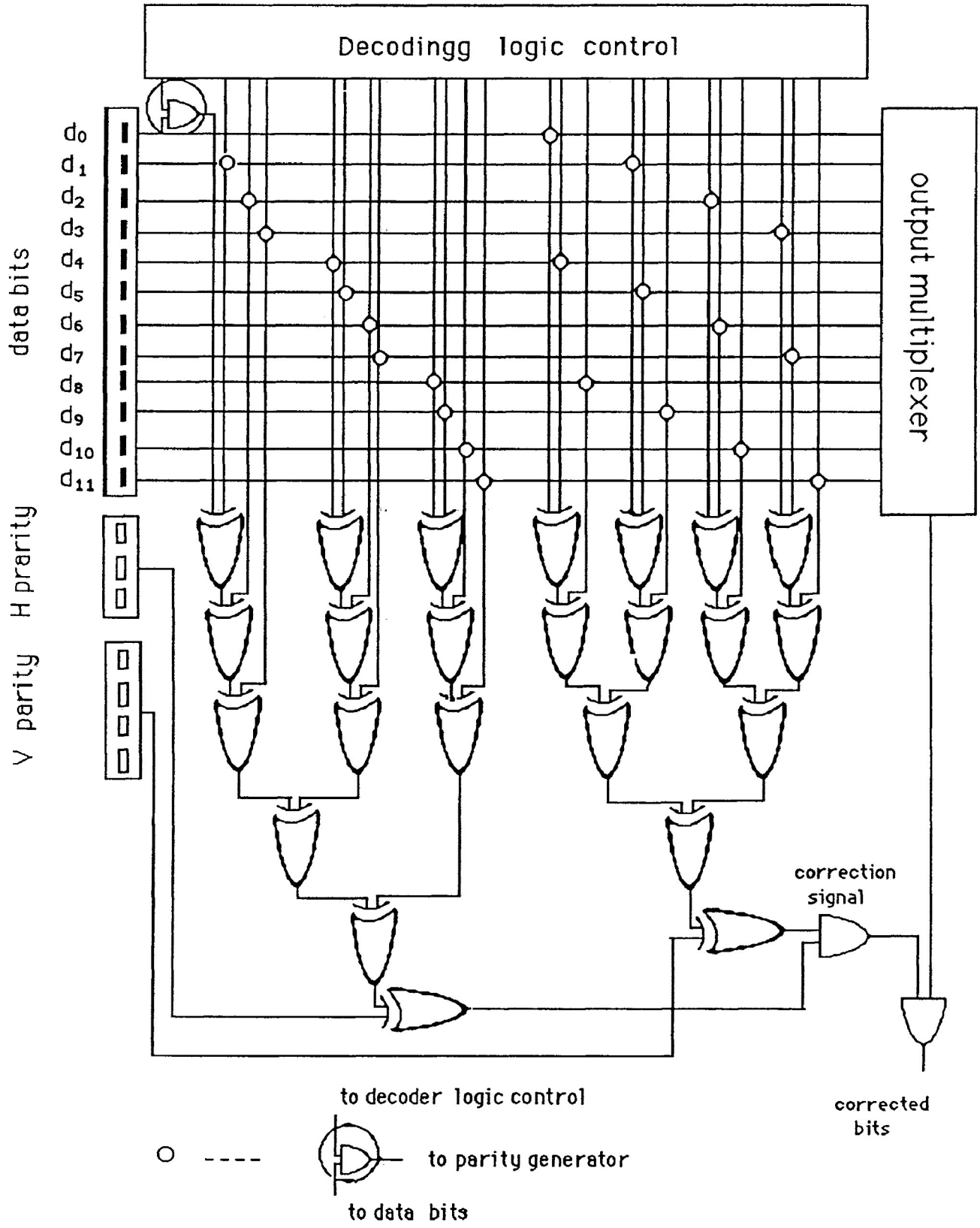


Fig 4. 7 A decoder circuit for H-V-parity code

Table 4.2 Decoding logic for a (19, 12) H-V-parity code

#	bits feed in H-parity generator	new H-parity bis	bits feed in V-parity generator	new V-parity bits	bits feed in multiplexer
1	d_0 d_1 d_2 d_3	h'_0	d_0 d_4 d_8	v'_0	d_0
2	d_0 d_1 d_2 d_3	h'_0	d_1 d_5 d_9	v'_1	d_1
3	d_0 d_1 d_2 d_3	h'_0	d_2 d_6 d_{10}	v'_2	d_2
4	d_0 d_1 d_2 d_3	h'_0	d_3 d_7 d_{11}	v'_3	d_3
5	d_4 d_5 d_6 d_7	h'_1	d_0 d_4 d_8	v'_0	d_4
6	d_4 d_5 d_6 d_7	h'_1	d_1 d_5 d_9	v'_1	d_5
7	d_4 d_5 d_6 d_7	h'_1	d_2 d_6 d_{10}	v'_2	d_6
8	d_4 d_5 d_6 d_7	h'_1	d_3 d_7 d_{11}	v'_3	d_7
9	d_8 d_9 $d_{10}d_{11}$	h'_2	d_0 d_4 d_8	v'_0	d_8
10	d_8 d_9 $d_{10}d_{11}$	h'_2	d_1 d_5 d_9	v'_1	d_9
11	d_8 d_9 $d_{10}d_{11}$	h'_2	d_2 d_6 d_{10}	v'_2	d_{10}
12	d_8 d_9 $d_{10}d_{11}$	h'_2	d_3 d_7 d_{11}	v'_3	d_{11}

to 10^6.

The concept of the product of two single-bit parity codes can be extended to any other linear codes. For example, the product codes that are obtained from any combinations of SEC-DED codes, DEC codes and even TEC (triple-error correction) codes will have more capacity of error correction and error detection, but accordingly need more complicated implementations of encoding and decoding.

4.3.2 Modified Hamming codes

It has been shown in Chapter three that an (n, k) linear Hamming code is uniquely specified by a parity check matrix H. This **H** matrix has r rows and n columns (among n columns, k of them correspond to information bit columns and r of them to the parity check bit columns) as shown in Eq.(4.9).

$$H = \begin{bmatrix} h_{0,0} & h_{0,1} & & h_{0,n-1} \\ \cdot & \cdot & & \\ h_{j,0} & h_{j,1} & & h_{j,n-1} \\ \cdot & \cdot & & \cdot \\ h_{r-1,0} & h_{r-1,1} & & h_{r-1,n-1} \end{bmatrix} = \begin{bmatrix} H_0 \\ \cdot \\ H_j \\ \cdot \\ H_{r-1} \end{bmatrix} \qquad (4.9)$$

where H_0, H_1, and H_{r-1} are row vectors of **H** matrix.

To determine the error patterns, a decoding syndrome $S = (s_0, s_1, s_2 \ldots s_{r-1})$ is necessary. The components of the syndrome can be calculated from the received codeword D and row vectors of the **H** matrix as given below.

$$s_0 = D\ H_0^T$$
$$s_1 = D\ H_1^T$$
$$\ldots\ldots\ldots$$

$$S_{r-1} = D\ H_{r-1}^{T} \qquad (4.10)$$

The codeword is accepted as error free if S is an all zero vector.

From the implementation point of view, the total number of 1's in each row vector of the **H** matrix is related to the number of logic level necessary to generate the syndrome bit corresponding to that row. Let N_i be the number of 1's in the i^{th} row vector of the **H** matrix, and L_i be the number of logic levels required to generate the i^{th} syndrome component. The relationship between L_i and N_i can be expressed as [4.10]

$$L_i = \lfloor \log_b N_i \rfloor \qquad (4.11)$$

where $\lfloor x \rfloor$ denotes the smallest integer equal to or greater than x, and b is the number of input of X-OR gate.

As can be seen, in order to minimize L_i, it is necessary to minimize N_i. Also, in order to achieve a least decoding delay, it is necessary to make the number of 1's in every row vector to be equal.

Based on above considerations, a class of optimal Hamming codes is derived, known as modified Hamming codes [4.12].

The construction of the H matrix of modified Hamming codes

The **H** matrix of a conventional Hamming code has 2^r-1 r-tuples as its columns, r columns for parity bits and k columns for information bits. By deleting l information bit column from the **H** matrix, it is possible to obtain a new matrix H_m which satisfies the following requirements:

1) Every column vector has odd number of 1's

2) The total number of 1's in the H_m matrix is minimum.

3) The number of 1's in each row vectors is equal to, or as close as possible to the average number (i.e., the total number of 1's in H_m divided by the number of rows)

The code specified by this new parity check matrix H_m is called a modified Hamming code.

The first requirement guarantees that the code has $d_{min} > 4$, (since the sum of any three columns of odd number 1's can not be a zero vector). Therefore the modified Hamming codes can be used as SEC-DED codes. The second and third requirements lead to a least logic levels in implementation and optimal delay in encoding and decoding.

According to above properties, the **H** matrix of the modified Hamming codes can be constructed by the following procedures[4.4]:

1) Take all $\binom{r}{1}$ weight-1 r-tuples as the most right r columns of **H** matrix which are corresponding to parity check bit columns.

2) If $\binom{r}{3}$ >= k, select k out of all $\binom{r}{3}$ weight-3 r-tuples as left k columns of the H_m matrix (thus the **H** matrix is completed). If $\binom{r}{3}$ < k, take all $\binom{r}{3}$ weight-3 r-tuples as the left columns of the H_m matrix,then keep going 3).

3) Select weight-5 r-tuples from all $\binom{r}{5}$ weight-5 tuples as the columns of **H** matrix and then select weight-7 r-tuples etc, until the **H** matrix is completed.

Only odd-weight r-tuples are used for the H_m matrix. The modified Hamming codes are sometimes called odd-weight-column

codes.

Fig.4.8 shows examples of **H** matrices of modified SEC-DED codes used in some IBM systems[4.11]. In Table 4.3 the parameters of some modified Hamming codes as well as the corresponding Hamming codes commonly used in computer memories are compared. It can be seen that modified Hamming codes are shortened Hamming codes. The shortening of the information bits makes it possible to meet the needs of data sizes in computer memory systems which are usually 4, 8, 16, 32, 64 etc. (i.e., the powers of 2).

Decoding of the codes

The modified Hamming codes are capable of correcting single errors and detecting double errors as well as detecting multiple odd number errors. The algorithm of the decoding of the codes is illustrated in Fig.4.9[4.11, 4.12].

i) According to the codeword R′ read from the memory, generate the syndrome vector $S = (s_1, s_2, \ldots, s_r)$.

ii) Test whether **S** = **0** or $(s_1 + s_2 + \ldots s_r) = 0$, where + denotes OR operation. If **S** = **0**, the codeword is assumed to be error-free.

iii) If **S** <> **0**, try to find a perfect match between the **S** and the i^{th} column in the **H** matrix. If it matches, the bit in the i^{th} position of the codeword is in error, hence the correction can be done.

iv) If **S** < > **0** and no match is found, then test if

$$s_0 \oplus s_1 \oplus \ldots \oplus s_{r-1} = 0 \qquad (4.12)$$

```
1 1 1 0 1 0 0 0
1 1 0 1 0 1 0 0
1 0 1 1 0 0 1 0
0 1 1 1 0 0 0 1
```

(a)

```
1 1 1 1 1 0 0 0  1 0 0 1 0 1 0 0  1 0 0 0 0 0
1 1 1 0 0 1 1 1  0 1 0 0 0 0 1 0  0 1 0 0 0 0
0 0 0 1 1 1 1 1  0 0 1 0 1 0 0 1  0 0 1 0 0 0
1 0 0 1 0 1 0 0  1 1 1 1 1 0 0 0  0 0 0 1 0 0
0 1 0 0 0 0 1 0  1 1 1 0 0 1 1 1  0 0 0 0 1 0
0 0 1 0 1 0 0 1  0 0 0 1 1 1 1 1  0 0 0 0 0 1
```

(b)

```
11111111 11111111 00010001 00010001 0010001( 00100010 00100010 00010100 10000000
10001000 10001000 11111111 11111111 00010001 00010001 10001000 10000010 01000000
01000100 01000100 10001000 10001000 11111111 11111111 01000100 01000001 00100000
00100010 00100010 01000100 01000100 10001000 10001000 00100010 00101111 00010000
00010001 00010001 00110011 00110011 01110111 01110111 00000000 00001111 00001000
00001111 00001111 00001111 00001111 00001111 00001111 11111111 00000000 00000100
00000000 11111111 00000000 11111111 00000000 11111111 11110000 11111111 00000010
11110000 00001111 11100001 00011110 11000011 00111100 00001111 11110111 00000001
```

(c)

```
11111111 00000000 00000000 11111111 10001110 10001110 10001110 10001111 10000000
11111111 11111111 00000000 00000000 01001101 01001101 01001101 01001101 01000000
00000000 11111111 11111111 00000000 00101011 00101011 00101011 00101011 00100000
00000000 00000000 11111111 11111111 00010111 00010111 00010111 00010111 00010000
10001110 10001110 10001110 10001110 11111111 00000000 00000000 11111111 00001000
01001101 01001101 01001101 01001101 11111111 11111111 00000000 00000000 00000100
00101011 00101011 00101011 00101011 00000000 11111111 11111111 00000000 00000010
00010111 00010111 00010111 00001011 00000000 00000000 11111111 11111111 00000001
```

(d)

Fig. 4.8 Parity matrice of some SEC-DED codes:

(a) a (8, 4) code
(b) a (22, 16) code (IBM System/3)
(c) a (72, 64) code (IBM 3033)
(d) a (72, 64) code (IBM 3081)

Table 4.3 Some parameters of Hamming codes and modified Hamming codes

Hamming codes			Modified Hamming codes			
n	k	r	n	k	r	l^*
31	26	5	13	8	5	18
63	57	6	22	16	6	41
127	120	7	39	32	7	88
255	247	8	72	64	8	183
511	502	9	137	128	9	374
1023	1013	10	266	256	10	757

l^* the number of eliminated bits

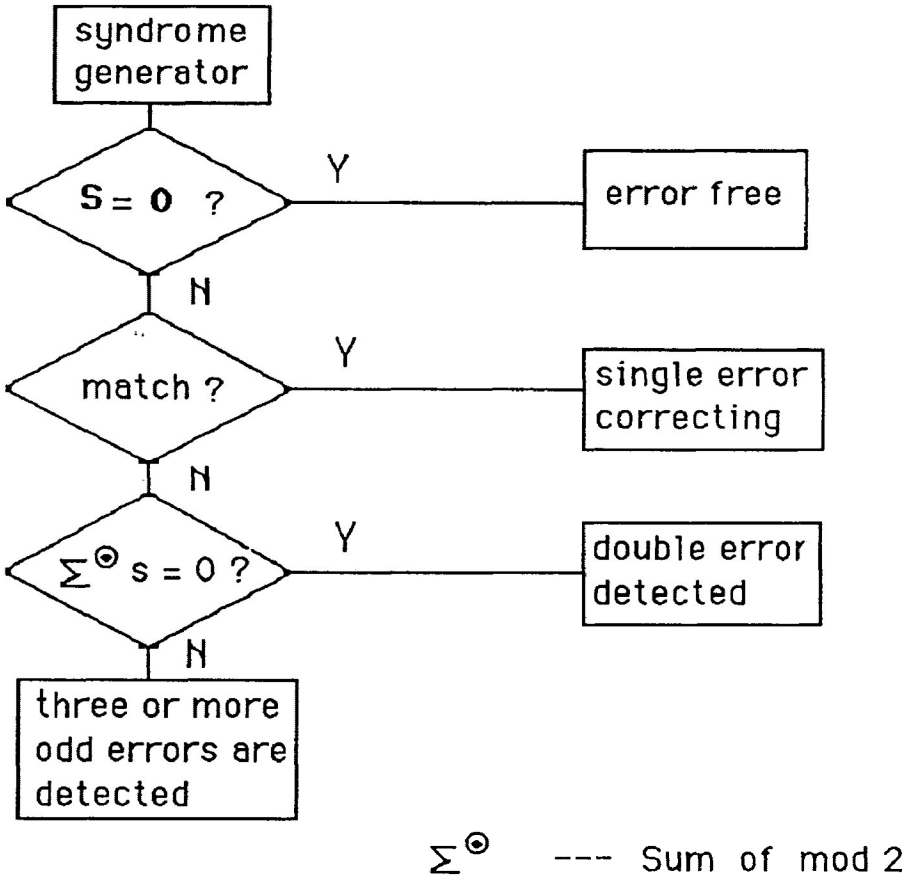


FIG. 4.9 Algorithm of decoding modified Hamming codes

If it does, a double error is then detected.

vi) If **S** <> 0 and no match is found and Eq.(4.12) does not hold, three or more odd number of errors are detected.

A decoding block diagram implementing the above algorithm is shown in Fig.4.10. The diagram consists of following blocks, syndrome-generator, syndrome-decoder, single-error-corrector, error-detector, double-error-detector and multiple-error-detector.

The syndrome-generator is composed of r n-input X-OR gates, calculating r syndrome bits according to Eq.(4.10). The output of the gates is a r-tuple syndrome.

The syndrome decoder checks whether the syndrome matches a column of the **H** matrix. It consists of n r-input AND gates. The inputs of each AND gate corresponds to a column of the H_m matrix. When a syndrome matches a column, the corresponding AND gate output a '1' so that the correction can be done in the error-corrector.

The error-detector is a r-input OR gate which test if **S** = **0**. An error detected signal is output if **S** < > **0**.

The double error-detector is composed of a r-input X-OR gate. If the output of the decoder equals to 0, a double error detected signal is given out.

The multiple-error-detector consists of a n-input NOR gate and an AND gate. If none of the n-output of syndrome-decoder is equal to '1', (which means no correction will be done) and the output of error-detector is '1', then the multiple-error-detector outputs '1', indicating that three or more odd number errors are detected.

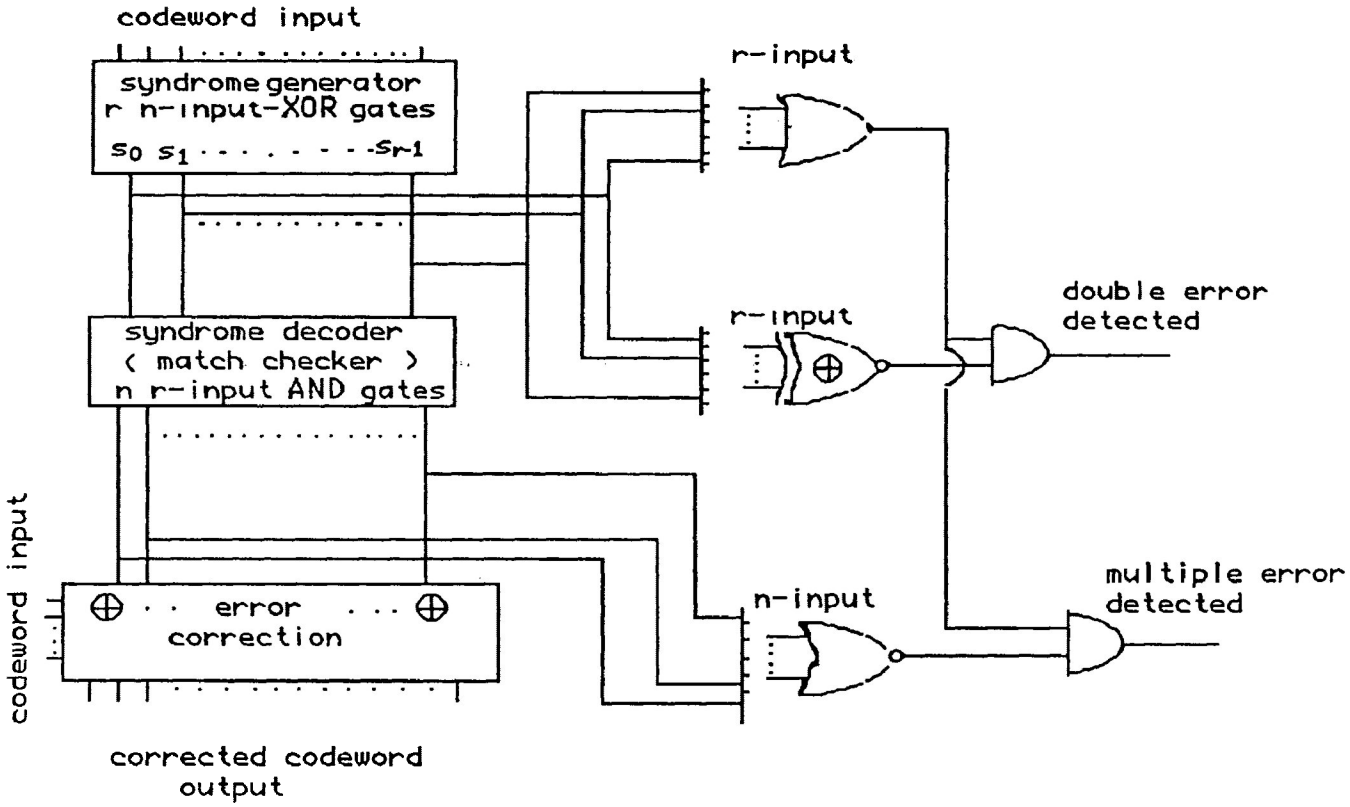


Fig. 4.10 A diagram of the decoder for modified Hamming code

To illustrate a whole decoding process, in the following we consider the circuit of decoder for a (8, 4) SEC-DED code.

Example: The **H** matrix of the (8, 4) code is as follows

$$H = \begin{vmatrix} d_0 & d_1 & d_2 & d_3 & c_0 & c_1 & c_2 & c_3 \\ 1 & 1 & 1 & 0 & 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 1 & 0 & 1 & 0 & 0 \\ 1 & 0 & 1 & 1 & 0 & 0 & 1 & 0 \\ 0 & 1 & 1 & 1 & 0 & 0 & 0 & 1 \end{vmatrix}$$

The syndromes can be calculated

$$s_0 = d_0 \oplus d_1 \oplus d_2 \oplus c_0$$
$$s_1 = d_0 \oplus d_1 \oplus d_3 \oplus c_1$$
$$s_2 = d_0 \oplus d_2 \oplus d_3 \oplus c_2$$
$$s_3 = d_1 \oplus d_2 \oplus d_3 \oplus c_3$$

Fig.4.11 gives the circuits of the decoder. It can be seen that the correction is performed in parallel, therefore has a high decoding speed.

Applications of Modified Hamming Codes

Many of the IBM system 370 models (e.g. 145, 155, 165, 158, and 168) use a (72, 64) SEC-DED modified Hamming code for error control in main memories.

In these applications encoder and decoder form additional circuitry. Recently Modified Hamming codes are also used in on-chip ECC schemes[4.13, 4.14, 4.15]. In [4.14], it has been demonstrated that combined use of memory cell redundancy and ECC scheme has

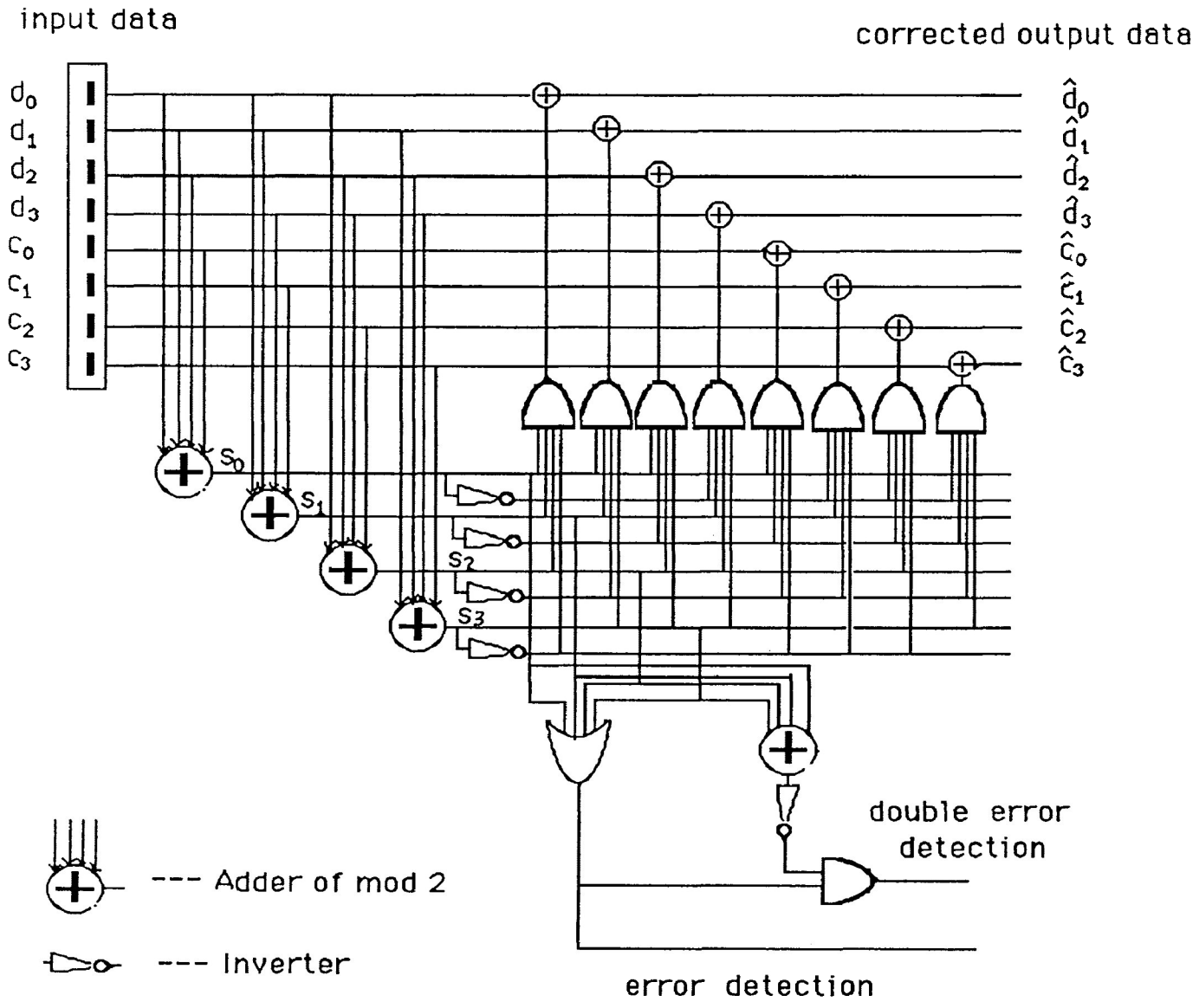


Fig 4. 11 A decoder for (8, 4) Modified Hamming code

enhanced the reliability significantly and brought effective improvement of yield of the memory chips. In Fig.4.12 the resulting yield as a function of the average number of (failing) single cells per chip is presented.

In a practical memory system with ECC, both of the parity codes and SEC codes are employed. For example, a (72, 64) modified Hamming code is used for memory error control while the parity code is used for validity checking of a data transferred in the data bus.

When a (72, 64) codeword is read from main memory, the decoding circuit generates a 8-bit syndrome. If the syndrome indicates no errors in the codeword, the 64 data bits are extracted and divided into 8 bytes. Then 8 parity bits are added to these bytes on the bit-per-byte basis as shown in Fig.4.13.

If a single error is detected in the (72, 64) codeword, the correction is performed and the corrected word is sent to CPU. In case a double error or a multiple error are detected, an interrupt signal will be generated. CPU then decides on further operations. In some systems the read instruction is retried. In other systems a duplication word (which is stored in the spare memory cells) is accessed. Or sometimes other outside operation may be required to deal with the errors[4.16].

When a data word (which has the form shown in Fig.4.13 from the data bus) is to be stored in the memory, the parity-encoded codeword (which has the form of Fig.4.13) is first checked. If it is a valid data word, the 8 parity bits are removed and the 64 data

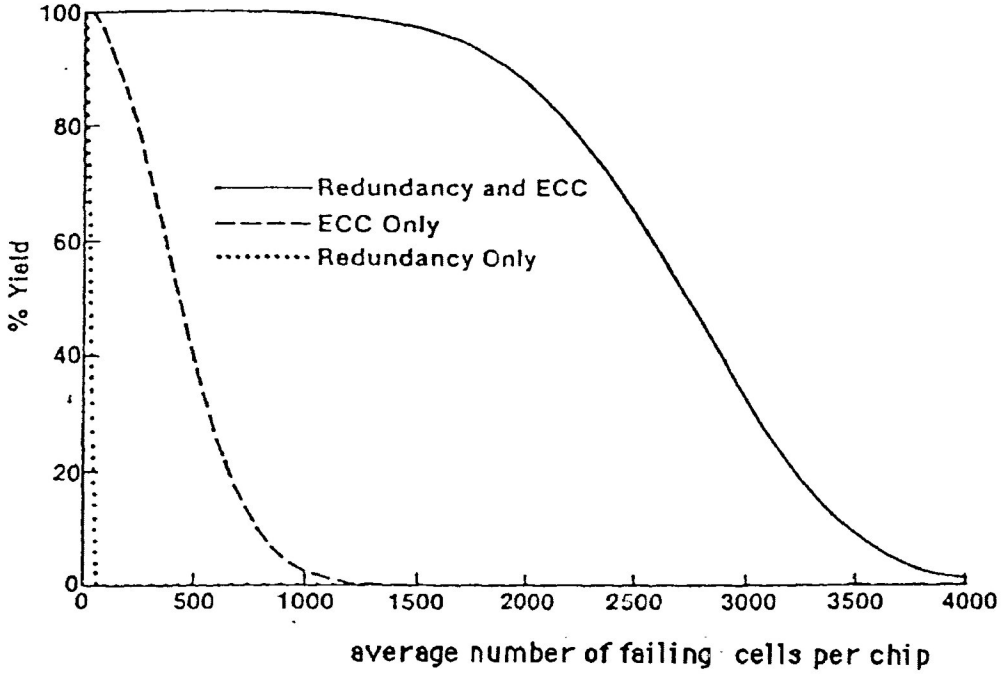


Fig 4.12 Yield curves for ECC and bit-line redundancy

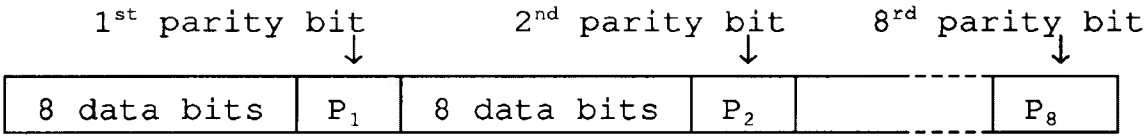
bits are then encoded into a (72, 64) SEC-DED Hamming codeword and then is stored in the memory.

4.4 Multiple error correcting codes

In some cases when a memory cell has been permanently defective, a soft error caused by alpha-particle radiation will result double errors in a codeword. Modified Hamming codes will fail to correct the double errors. And the probability of double errors might be high under certain conditions. In order to maintain a high memory reliability, double error correcting or multiple error correcting codes are required. Usually, in practice, hardware maintenance strategies are used for repairing the defective memory chips in regular intervals. Use of multiple error correcting codes allows the maintenance strategy to accumulate the hard failures up to certain threshold, thus defer the replacement and reduce the down time of the systems.

In theory there are several multiple error correcting codes, e.g., BCH codes, R-S codes, majority logic codes etc [4.8]. But these codes are cyclic codes, which as we discussed in chapter 3, require relatively complicated decoding logic and longer decoding time. In this section, we introduce a type of fast decoding codes, majority orthogonal Latin square codes which can be used as SEC codes as well as multiple error correcting codes. These codes are linear codes.

We first consider a parity check matrix **H** of a SEC code as follows



(a) data format from/to bus

(b) data format in storage

Fig.4.13 Data format in/not-in storage

$$H = \left|\begin{array}{ccccccccccccccc}
d_0 & d_1 & d_2 & d_3 & d_4 & d_5 & d_6 & d_7 & d_8 & p_1 & p_2 & p_3 & p_4 & p_5 & p_6 \\
1 & 1 & 1 & & & & & & & 1 & & & & & \\
 & & & 1 & 1 & 1 & & & & & 1 & & & & \\
 & & & & & & 1 & 1 & 1 & & & 1 & & & \\
1 & & & & & & 1 & & & & & & 1 & & \\
 & 1 & & & 1 & & & 1 & & & & & & 1 & \\
 & & 1 & & & 1 & & & & & & & & & 1
\end{array}\right| \qquad (4.13)$$

From the **H** matrix, a set of parity equations can be derived as in Eq.(4.14)

$$\begin{aligned}
p_1 &= d_0 + d_1 + d_2 \\
p_2 &= d_3 + d_4 + d_5 \\
p_3 &= d_6 + d_7 + d_8 \\
p_4 &= d_0 + d_3 + d_6 \\
p_5 &= d_1 + d_4 + d_7 \\
p_6 &= d_2 + d_5 + d_8
\end{aligned} \qquad (4.14)$$

When a codeword $D=(d_0\ d_1\ \ldots d_7\ d_8\ p_1\ \ldots\ p_6)$ with a single-bit error is read from the memory, the decoding might be processed on majority voting basis. For instance, in the decoder, three copies of bit d_0 are regenerated. Two of them are derived from parity equations p_1 and p_4, that is

$$\begin{aligned}
d_0 &= p_1 + d_1 + d_2 \\
d_0 &= p_4 + d_3 + d_6
\end{aligned} \qquad (4.15)$$

and the third one is d_0 itself received from the memory. The correct d_0 is then recovered from a three majority voting gate. The output of a majority vote gate depends on the majority values

of its input. If there is a single-bit error, data d_0 can be recovered correctly, because any single-bit error affect at most one vote of the majority gate. In fact, as long as the parity equations p_1 and p_4 given in Eq.(4.14) are orthogonal to the data bit d_0, the error correction by majority voting is always valid.

A set of parity equations are said to be orthogonal on d_j, if for all the equations, d_j is the only common variable involved. For example, in Eq.(4.15), p_1 and p_4 are orthogonal on d_0, p_2 and p_5 are orthogonal on d_4, etc.

If there is a set of 2t parity equations orthogonal on d_j, then d_j can be decoded correctly by majority decoding logic (majority voting among 2t parity equations and d_j itself, totally 2t+1 votes are involved). In the above example, the code is designed as SEC code, so every two equations are orthogonal on each data bit.

In the following, according to Hsiao [4.17], we will discuss a type of majority-logic codes for multiple-error correcting. These codes are defined on a set of so called Orthogonal Latin Squares, hence are called Orthogonal Latin Square (OLS) codes. The codes are a class of one-step decodable majority logic codes which can be decoded in parallel manner.

The most attractive feature for one step majority decodable codes is that they can be decoded at an exceptionally high speed. For the OLS codes, there is another noticeable unique advantage. That is, the decoder of the code can be built in a modular form such that each additional modular adds a further error correcting

capability without affecting the existing modulars.

Orthogonal Latin Squares

A Latin square of order m is an m x m square array of the digits 0, 1, ..., m-1, with each row and column a permutation of the digits. Two Latin squares are orthogonal if one Latin square is superimposed on the other, every ordered pair of elements appear only once.

Following are three (4 x 4) Latin squares L_1, L_2, and L_3.

0 1 2 3	0 2 3 1	0 3 1 2
1 0 3 2	1 3 2 0	1 2 0 3
2 3 0 1	2 0 1 3	2 1 3 0
3 2 1 0	3 1 0 2	3 0 2 1
L_1	L_2	L_3

The result of a superposition of the first two Latin squares is as follows

0,0	1,2	2,3	3,1
1,1	0,3	3,2	2,0
2,2	3,0	0,1	1,3
3,3	2,1	1,0	0,2

In this array, each of the 4^2 possible ordered pairs occur exactly once. This holds for all other squares as well, and hence these Latin squares are orthogonal.

Given a integer m, there exists a set of h orthogonal Latin squares, where h is the number of orthogonal Latin squares that m elements can generate. The relationship between h and m is expressed as $h = \min(p_i^{ei} - 1)$, where p_i^{ei} are integer powers of the prime factors of the integer m. The method of constructing

Orthogonal Latin squares is included in Hsiao [4.17], or it can be found from tables of Fisher and Yates[4.18].

The construction of the H matrix

A OLS code which can correct t errors in a codeword is defined on a set of orthogonal Latin squares. The codes have following parameters[4.19]

$k = m^2$	the number of data bits
$r = 2tm$	the number of check bits
$n = m^2 + 2tm$	the length of the codeword
$d = h + 3$	the minimum Hamming distance

Let m^2 data bits be denoted by a vector

$$\mathbf{D} = (d_0, d_1, d_2, \ldots, d_{m^2-1}) \qquad (4.16)$$

Then the 2tm parity check equations for t-error correcting can be obtained from the following parity check matrix H:

$$H = \left| \begin{array}{c|c} M_1 & \\ M_2 & \\ . & I_{2mt} \\ . & \\ M_3 & \end{array} \right|_{2tm \times (m^2+2tm)} \qquad (4.17)$$

where I_{2tm} is an identity matrix of order 2tm and $M_1, M_2, \ldots, M_{2t}$ are submatrices of size m x m^2. These submatrices are expressed as following

$$M_1 = \begin{vmatrix} 11111000000000000000000000 \\ 00000111110000000000000000 \\ 00000000001111100000000000 \\ 00000000000000001111100000 \\ 00000000000000000000011111 \end{vmatrix} \quad (4.18)$$

$$M_2 = \quad [\ I_m\ I_m\ I_m\ \ldots\ I_m\ ]_{m \times m^2} \quad (4.19)$$

The matrices M_3, M_4, ..., M_{2t} are derived from the existing set of orthogonal Latin squares L_1, L_2, ..., L_{2t-2} of the size m x m. Denote the Latin squares as

$$\begin{aligned} L_1 &= [\ l^1_{i,j}\ ]_{m \times m} \\ L_2 &= [\ l^2_{i,j}\ ]_{m \times m} \\ &\ \ .\qquad\qquad .\qquad\qquad (4.20) \\ L_{2t-2} &= [\ l^{2t-2}_{i,j}\ ]_{m \times m^2} \end{aligned}$$

where l and $i, j \in \{1, 2, \ldots, m\}$.

For any given Latin square having m elements, there exists an incidence matrix defined on one of its elements as follows.

Let $L = [\ l_{ij}\ ]$ be a Latin square; then an incidence matrix defined with respect to the element c ($1 <= c <= m$, integer), denoted by $Q_c = [q^c_{ij}]$, is defined by the rules

$$\begin{aligned} q^c_{ij} &= 1, \quad \text{if } l_{ij} = c \\ &\quad 0, \quad \text{if } l_{ij} <> c \end{aligned} \quad (4.21)$$

For each Latin square of m element, there are m incidence matrices Q_1, Q_2, ..., Q_m. Each incidence matrix is concatenated into a vector form,

$$V_c = [q^c_{11} \ldots q^c_{1m} \quad q^c_{21} \ldots q^c_{2m} \quad \ldots \quad q^c_{m1} \ldots q^c_{mm}] \qquad (4.22)$$

If submatrix M_i is derived from a unique Latin square L_i, then V_1, V_2, ..., V_m are derived from L_i

$$M_i = \begin{vmatrix} V_1 \\ V_2 \\ . \\ . \\ V_m \end{vmatrix} \qquad (4.23)$$

Examples of constructing OLS codes

Given m = 5, then k = m^2 = 25

Example-A Let t = 1, then r = 2tm = 10, n = m^2 +2tm = 35. This is a (35, 25) SEC OLS code. The **H** matrix of the code has the form as follows

$$H_1 = \left|\begin{array}{c:c} M_1 & \\ M_2 & I_{10} \end{array}\right| \qquad (4.24)$$

where, according to Eq.(4.18) and Eq.(4.19) M1 and M2 are derived

$$M_1 = \left|\begin{array}{ccccccccccccccccccccccccccccccccccc} 1&1&1&1&1&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0 \\ 0&0&0&0&0&1&1&1&1&1&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0 \\ 0&0&0&0&0&0&0&0&0&0&1&1&1&1&1&0&0&0&0&0&0&0&0&0&0&0&0 \\ 0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&1&1&1&1&1&0&0&0&0&0&0&0 \\ 0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&1&1&1&1&1&1&1 \end{array}\right| \qquad (4.25)$$

$$M_2 = \begin{vmatrix} 1\,0\,0\,0\,0\,1\,0\,0\,0\,0\,1\,0\,0\,0\,0\,1\,0\,0\,0\,0\,1\,0\,0\,0\,0 \\ 0\,1\,0\,0\,0\,0\,1\,0\,0\,0\,0\,1\,0\,0\,0\,0\,1\,0\,0\,0\,0\,1\,0\,0\,0 \\ 0\,0\,1\,0\,0\,0\,0\,1\,0\,0\,0\,0\,1\,0\,0\,0\,0\,1\,0\,0\,0\,0\,1\,0\,0 \\ 0\,0\,0\,1\,0\,0\,0\,0\,1\,0\,0\,0\,0\,1\,0\,0\,0\,0\,1\,0\,0\,0\,0\,1\,0 \\ 0\,0\,0\,0\,1\,0\,0\,0\,0\,1\,0\,0\,0\,0\,1\,0\,0\,0\,0\,1\,0\,0\,0\,0\,1 \end{vmatrix} \quad (4.26)$$

The whole **H** matrix of the code is given in Fig.4.14(a). For ECC codes, Orthogonal Latin squares are not needed.

Example-B Let t = 2, then r= 2tm = 20, n = m^2 +2tm =45. This is a (45, 25) DEC OLS code. The H_2 matrix has the form,

$$H_2 = \begin{vmatrix} M_1 \\ M_2 \\ M_3 \\ M_4 \end{vmatrix} \quad (4.27)$$

where M_1 and M_2 are identical to Eq. (4.25) and (4.26). To construct M3 and M4, it is necessary to find (5 x 5) Latin squares. The existing Latin squares are given in follows

0	1	2	3	4
1	2	3	4	0
2	3	4	0	1
3	4	0	1	2
4	0	1	2	3

L1

0	1	2	3	4
2	3	4	0	1
4	0	1	2	3
1	2	3	4	0
3	4	0	1	2

L2

0	1	2	3	4
3	4	0	1	2
1	2	3	4	0
4	0	1	2	3
2	3	4	0	1

L3

0	1	2	3	4
4	0	1	2	3
3	4	0	1	2
2	3	4	0	1
1	2	3	4	0

L4

From L1, the incidence matrix Q_0, Q_1, Q_2, Q_3 and Q_4 can be derived as following

(a)

$d_0 \cdots d_5 \cdots d_{10} \cdots d_{15} \cdots d_{20} \cdot d_{24}\, c_1 \cdots\cdots c_{10}$

M_1

11111 00000 00000 00000 00000
00000 11111 00000 00000 00000
00000 00000 11111 00000 00000
00000 00000 00000 11111 00000
00000 00000 00000 00000 11111

M_2

10000 10000 10000 10000 10000
01000 01000 01000 01000 01000
00100 00100 00100 00100 00100
00010 00010 00010 00010 00010
00001 00001 00001 00001 00001

I_{10}

(b)

$d_0 \cdots d_5 \cdots d_{10} \cdots d_{15} \cdots d_{20} \cdot d_{24}\, c_1 \cdots\cdots c_{20}$

M_1
M_2

M_3

10000 00001 00010 00100 01000
01000 10000 00001 00010 00100
00100 01000 10000 00001 00010
00010 00100 01000 10000 00001
00001 00010 00100 01000 10000

M_4

10000 00010 01000 00001 00100
01000 00001 00100 10000 00010
00100 10000 00010 01000 00001
00010 01000 00001 00100 10000
00001 00100 10000 00010 01000

I_{20}

(c)

$d_0 \cdots d_5 \cdots d_{10} \cdots d_{15} \cdots d_{20} \cdot d_{24}\, c_1 \cdots\cdots c_{30}$

M_1
M_2
M_3
M_4

M_5

10000 00100 00001 01000 00010
01000 00010 10000 00100 00001
00100 00001 01000 00010 10000
00010 10000 00100 00001 01000
00001 01000 00010 10000 00100

M_6

10000 01000 00100 00010 00001
01000 00100 00010 00001 10000
00100 00010 00001 10000 01000
00010 00001 10000 01000 00100
00001 10000 01000 00100 00010

I_{30}

(a) Single error correcting code (35, 25)
(b) Double error correcting code (45, 25)
(c) Triple error correcting code (55, 25)

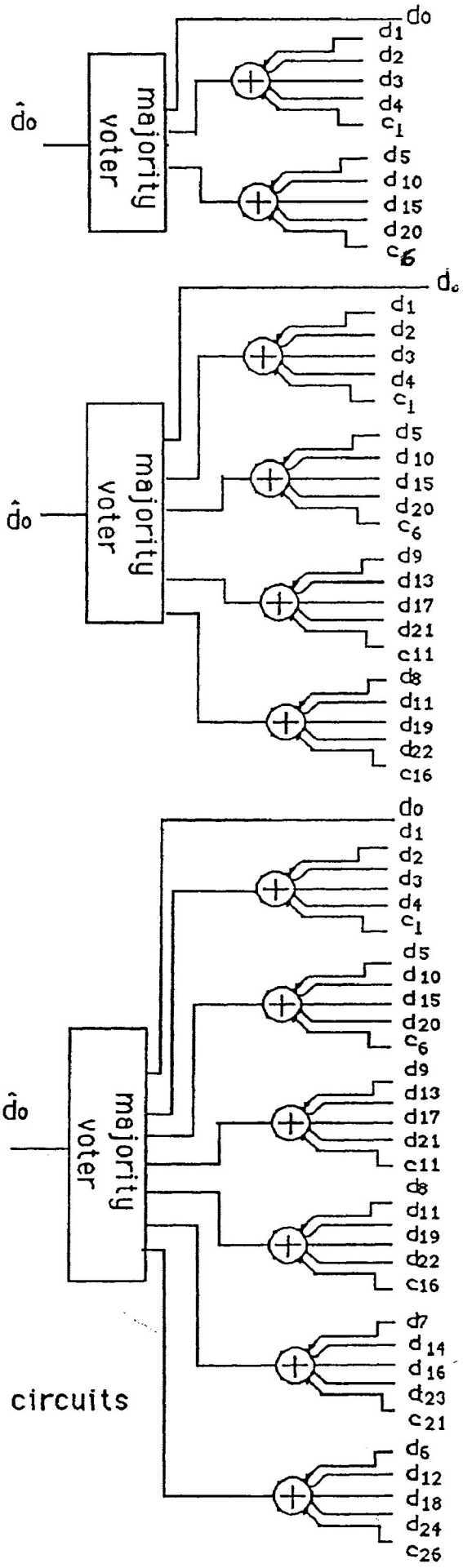


Fig. 4. 14 H-matrices for OSL codes and decoder circuits for d_0.

$$Q_0 = \begin{vmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \end{vmatrix} \qquad Q_1 = \begin{vmatrix} 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 \end{vmatrix} \qquad Q_2 = \begin{vmatrix} 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 \end{vmatrix}$$

$$Q_3 = \begin{vmatrix} 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{vmatrix} \qquad Q_4 = \begin{vmatrix} 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 \end{vmatrix}$$

Q_j is obtained by replacing all j's in L_1 for 1's and replacing all values rather than j for 0's. Then concatenate Q_j row by row into a vector V_j, as follows

$$M_3 = \begin{vmatrix} V_0 \\ V_1 \\ V_2 \\ V_3 \\ V_4 \end{vmatrix} = \begin{vmatrix} 10000 & 00001 & 00010 & 00100 & 01000 \\ 01000 & 10000 & 00001 & 00010 & 00100 \\ 00100 & 01000 & 10000 & 00001 & 00010 \\ 00010 & 00100 & 01000 & 10000 & 00001 \\ 00001 & 00010 & 00100 & 01000 & 10000 \end{vmatrix} \qquad (4.28)$$

In the similar way, M_4 can obtained from Latin square L2, as follows

$$M_4 = \begin{vmatrix} 10000 & 00010 & 01000 & 00001 & 00100 \\ 01000 & 00001 & 00100 & 10000 & 00010 \\ 00100 & 10000 & 00010 & 01000 & 00001 \\ 00010 & 01000 & 00001 & 00100 & 10000 \\ 00001 & 00100 & 10000 & 00010 & 01000 \end{vmatrix} \qquad (4.29)$$

The whole **H** matrix of the double error correcting OLS code is

2shown in Fig.4.14 (b). In this example, Latin squares L_1 and L_2 were used to derive M_3 and M_4.

Example-C Let t = 3, then r = 2tm = 30, n = m_2 2tm = 55. This is a (55,25) triple error correcting OLS code. The H_3 matrix has the form

$$H_3 = \left| \begin{array}{c|c} M_1 & \\ M_2 & \\ M_3 & \\ M_4 & I_{30} \\ M_5 & \\ M_6 & \end{array} \right|$$

where M_1, M_2, M_3, M_4, are completely the same as Eq. (4.25), (4.26), (4.28) and (4.29). In the same way, M_5 and M_6 can be derived from Latin squares L_3 and L_4 respectively. The whole H_3 matrix of the codes is shown in Fig.4.14 (c). In fact this code is a result of extension of H_2. Since there are no more Latin squares that can be used, the code of m=5 has a maximum error correcting capability of three.

In Fig.4.14, the decoders of data bit d_0 for three codes are also provided. It is seen that, whenever higher bit error correcting capability is required, the decoding logic allows to add a modular to the existing circuitry to perform the additional error correcting without interfering the original mechanisms of the decoder. The added modulars are identical in form to existing ones. That is, all modulars are composed of mod 2 adders with the same inputs and (2t + 1) voting gates. The simplicity and

regularity are useful for extending the error correcting capability for an existing system.

It can be noticed that the maximum error correcting capability of a OLS code is related to m, the size of corresponding Latin squares. For an existing set of h (m x m) orthogonal Latin squares ($h <= m - 1$), the maximum error correcting t is

$$t = [h / 2] + 1 \qquad (4.30)$$

In most cases of m, the number of Orthogonal Latin squares are (m - 1) with the exception of m =6 and m = 10. In these two cases, the SEC codes can not be extended to multiple error correcting codes.

Shortening of the OLS codes

The maximum length of data for the OLS codes are m^2, but in computer memory systems, commonly used data size are 8, 16, 32, 64 etc. To meet the needs for memory systems, the procedure of shortening a regular code, like in modified Hamming codes, can also be performed on OLS codes. That is, choose and delete l columns from m^2 data bit columns of the OLS **H** matrix. The deletion of column neither affects the orthogonality of the set of parity equations nor reduces the number of orthogonal parity equations, hence does not affect the validity of the majority logic. Therefore as many columns as required can be deleted to meet different applications. For instance, in example-C mentioned above, there are $k=m^2=25$ data columns in the H_3 matrix. To meet a need for 16 bit data size, 9 columns from 25 have to be deleted.

Obviously, there are several deleting approaches to be chosen.

1) Delete the left-most 9 columns

In this case, among the 9 deleted columns, the first m columns are included, thus the parity p1 is now independent of any data bits, hence can be deleted. On the other hand because of the deletion, some of the adders in the decoder need less inputs than they did in original decoder. For instance, the adders corresponding to the parity p6, p7, p8 etc. now need three inputs instead of five. But some of the others (p2, p3, p4, and p5) still need five input. This means the deleting does not improve the speed of decoding. The shortened **H** matrix thus obtained is given in Fig.4.15 (a).

2) Delete the first m (= 5) columns and then columns d5, d10, d15 and d20.

As a result of the deleting scheme, p1 and p5 become independent of any data bits, hence can be deleted. Moreover, the maximum number of inputs that adders need reduces to 4, therefore this scheme also enhances the decoding speed. The shortened **H** matrix thus obtained is shown in Fig.4.15 (b).

It seems that, when a shortened OLS code is required, by proper choice of deleted columns, it is possible to improve the decoding speed as well. If l < m, the deleting can be done randomly. If l = m, each deleted column is chosen from each m-bit group (columns d0 to d4 is the first m-bit group, d5 to d9 the second m-bit group etc), we call this per-column-per-m-bit-group.

$d_0 \ldots d_4$	$d_5 \ldots d_9$	$d_{10} \ldots d_{14}$	$d_{15} \ldots d_{19}$	$d_{20} \ldots d_{25}$
11111	00000	00000	00000	00000
00000	11111	00000	00000	00000
00000	00000	11111	00000	00000
00000	00000	00000	11111	00000
00000	00000	00000	00000	11111
10000	10000	10000	10000	10000
01000	01000	01000	01000	01000
00100	00100	00100	00100	00100
00010	00010	00010	00010	00010
00001	00001	00001	00001	00001
10000	00001	00010	00100	01000
01000	10000	00001	00010	00100
00100	01000	10000	00001	00010
00010	00100	01000	10000	00001
00001	00010	00100	01000	10000
10000	00010	01000	00001	00100
01000	00001	00100	10000	00010
00100	10000	00010	01000	00001
00010	01000	00001	00100	10000
00001	00100	10000	00010	01000
10000	00100	00001	01000	00010
01000	00010	10000	00100	00001
00100	00001	01000	00010	10000
00010	10000	00100	00001	01000
00001	01000	00010	10000	00100
10000	01000	00100	00010	00001
01000	00100	00010	00001	10000
00100	00010	00001	10000	01000
00010	00001	10000	01000	00100
00001	10000	01000	00100	00010

H matrix from H_3

d_9	$d_{10} \ldots d_{14}$	$d_{15} \ldots d_{19}$	$d_{20} \ldots d_{25}$
0	00000	00000	00000
1	00000	00000	00000
0	11111	00000	00000
0	00000	11111	00000
0	00000	00000	11111
0	10000	10000	10000
0	01000	01000	01000
0	00100	00100	00100
0	00010	00010	00010
1	00001	00001	00001
1	00010	00100	01000
0	00001	00010	00100
0	10000	00001	00010
0	01000	10000	00001
0	00100	01000	10000
0	01000	00001	00100
1	00100	10000	00010
0	00010	01000	00001
0	00001	00100	10000
0	10000	00010	01000
0	00001	01000	00010
0	10000	00100	00001
1	01000	00010	10000
0	00100	00001	01000
0	00010	10000	00100
0	00100	00010	00001
0	00010	00001	10000
0	00001	10000	01000
1	10000	01000	00100
0	01000	00100	00010

(a) H matrix of shortened code

.. d_9	.. d_{14}	.. d_{19}	.. d_{25}
0000	0000	0000	0000
1111	0000	0000	0000
0000	1111	0000	0000
0000	0000	1111	0000
0000	0000	0000	1111
0000	0000	0000	0000
1000	1000	1000	1000
0100	0100	0100	0100
0010	0010	0010	0010
0001	0001	0001	0001
0001	0010	0100	1000
0000	0001	0010	0100
1000	0000	0001	0010
0100	1000	0000	0001
0010	0100	1000	0000
0010	1000	0001	0100
0001	0100	0000	0010
0000	0010	1000	0001
1000	0001	0100	0000
0100	0000	0010	1000
0100	0001	1000	0010
0010	0000	0100	0001
0001	1000	0010	0000
0000	0100	0001	1000
1000	0010	0000	0100
1000	0100	0010	0001
0100	0010	0001	0000
0010	0001	0000	1000
0001	0000	1000	0100
0000	1000	0100	0010

(b) H matrix of shortened code

Fig. 4.15 H matrices* of shortened codes

* only information portions are given

If l > m, the first m columns deleted can be any m-bit group, then delete other (l - m) columns on per-column-per-m-bit-group basis.

4. 5 Erasures

Another method for coping with multiple errors in RAM memories without introducing excessive decoding delays or excessively complex hardware is to take advantage of the erasure correcting capability of transfer error-control codes[4.20, 4.21, 4.22]. The following is a brief review of erasure correcting scheme.

An erasure is an error for which the location of the error in memory block is known but the magnitude is not. Usually hardware defections cause erasures. The task of a decoder is to restore (or fill) the erasure position.

Erasure correction has two major advantages:

1) A code with minimum Hamming distance d can be used to correct up to (d -1) erasures (compare to correct up to (d-1)/2 random errors).
2) Erasures can be corrected more quickly than random errors.

The erasure correction procedure is briefly described as follows:

step1 locate the defective positions in memory

step2 store the syndromes associated with the various combinations of erasures

step3 generate the syndrome of the codeword read from the memory

step4 compare the generated syndrome and stored syndromes

step5 if a match is found, the error is then determined

step6 if no match is found, a new erasure is occurring. Use step1 to find the erasure location and add the new syndrome to the stored syndrome set for further use.

The correction is based on a kind of 'look-up-table', hence provides a fast decoding speed.

There are several approach to locate the erasure positions in step1[4.22]. Following is one of them

a) write an arbitrary data word d (= d_1 d_2...d_k) into a location of the memory

b) read the same location, obtaining a memory word y_1

c) repeat step a) and b) with data word d', where d' is bitwise complement of d, obtaining a memory output word y_2.

e) the positions of the erasure are 1's in the n-bit word

$$\mathbf{E = y_1 \oplus y_2}$$

Example, let us write a data word d=00010110 into a specific memory block. Suppose when the word is read out from the memory, it becomes y_1=00010111, in which the 8^{th} bit is stuck at 1. To find out the erasure position, we rewrite a data word d' (=11101000) into the same memory block, where d' is the bitwise complemented of d. When the word d' is read out, it becomes y_2= 11101001. By calculating $E = y_1 \oplus y_2$ =(00010111) $\oplus$ (11101001)=00000001, the erasure position is located at 8^{th} position.

4.6 Summary

In this chapter four classes of error control methods, namely error detection, single error correction, multiple error correction and erasure were presented. As has been seen, the error detection and correction capability is directly related to the distance of a code, and so does the redundancy requirement. A decision of using and selecting an error control code depends on several considerations as descbribed in 4.1.

Error detection codes are the simplest codes with respect to implementation of encoding and decoding. These codes can be used in small memory systems and some applications in which error detection is normally sufficient e. g., those with the capability of repeated transmission or re-configuration.

Single error correcting and double error detecting codes are widely used in computer memories. This is because most memory errors caused by hard failure or soft failures are single bit errors. H-V-parity codes discussed in section 4.2 using bidirectional parity checking has a simple decoding circuit structure, hence suitable for on-chip ECC schemes. There are several reports on the implementations of the H-V-parity codes used in computer memories [4.23, 4.24, 4.25], and some modified H-V-parity codes[4.26]. Modified Hamming codes are a type of optimal Hamming codes from the practical point of view. With the same coding efficiency, the modified Hamming codes provide improvement over Hamming codes in speed, cost and reliability. The decoding circuit of the codes can also be built in memory chips, such that the

memory system can be made more compact, high yield and high reliability [4.13].

Multiple error correcting codes combined with memory maintenance strategies make it possible to delay the replacement of defective memory cells thus reduce the system down time. In section 4.3, the OLS codes which can correct multiple errors are presented. These codes are attractive because of their exceptional decoding speed and flexible circuit features. Even though the OLS codes require higher redundancy, they are becoming a strong candidate for multiple error correcting applications.

References

4.1 Len Levine and Ware Meyers, "Semiconductor Memory Reliability with Error Detecting and Correcting Codes", Computers, October 1976, pp 43-49

4.2 Arnold M. Michelson, Allen H. Levesque, Error-Control Techniques for Digital Communications, John Wiley & Sons 1985

4.3 Bary W. Johnson, Design and Analysis of fault tolerant digital system, Addison Wesley, 1989

4.4 T. R. N. Rao/E. Fujiwara, Error-Control Coding for Computer Systems, Prentice-Hall Inc., 1989

4.5 Bella Bose, "On Unordered Codes", IEEE Transactions on Computers, vol. 40, No. 2, Feb. 1991, pp 125-131

4.6 Mario Blaum, Rodney Goodman, and Robert Mceliece, "The Reliability of Single-Error Protected Computer Memories", IEEE Transactions on Computers, vol. 37 No. 1, Jan. 1988, pp 114-119

4.7 Loe Cohen, Robert Green, Kent Smith and J. Leland Seely, "Single-Transistor Cell Makes Room for More Memory on An MOS Chip", Electronics, Aug. 2, 1971, pp 69-76

4.8 W. Wesley Peterson, E. J. Welson, JR. "Error Correcting Codes", MIT press 1986

4.9 Tsuneo Mano, Junzo Yamada, "Circuit Techniques for a VLSI Memory", IEEE Journal of Solid-State Circuits No. 5, Oct. 1983,pp 463-469

4.10 Shu lin/Daniel J. Costello, JR, "Error Control Coding Fundamental and Applications", Prentice-Hall, Inc., Englewood Cliffs New Jersey, 1983

4.11 C. L. Chen and M. Y. Hsiao, "Error-Correcting Codes for Semiconductor Memory Application: A State-of-the-Art Review", IBM Journal of Research and Development, 28 March 1984, pp 124-134

4.12 D. C. Bossen and M. Y. Hsiao, "A System Solution to the Memory Soft Error Problem", IBM Journal of Research and Development, 24 May 1980, pp 390-397

4.13 Kiyohiro Furutani, "A Built-in Hamming Code ECC Circuit for DRAM's", IEEE Journal of Solid-State Circuits No. 1, Feb. 1989, pp 50-55

4.14 Howard 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 No. 5, Oct. 1990, pp 1118-1127

4.15 A. Field, C.H. Stapper, "High-Speed On-chip ECC Synergistic Fault-Tolerant Memory Chips", IEEE Journal of Solid-State Circuits, vol. 26, No. 10, Oct. 1991, pp 1449-1452

4.16 William C. Carter an Charles E. McCarthy, "Implementation of an Experimental Fault-Tolerant Memory System", IEEE Transaction on Computers, c-25, June 1976, pp 557-568

4.17 M.Y. Hsiao and D. C. Bossen and R. T. Chen, "Orthogonal Latin Square Codes", IBM Journal of Research and Development, 14 July 1970, pp 390-394

4.18 R. A. Fisher and F.Yates, "Statistical Tables for Biological Agricultural and Medical Research", Hafner Publishing Co., 1957

4.19 Mu. Y. Hsiao, Douglas C. Bossen, "Orthogonal Latin Square Configuration for LSI Memory Yield and Reliability Enhancement", IEEE Transactions on Computers, vol. c-24, No. 5, May 1975, pp 512-516

4.20 D. K. Pradhan, J. J. Stiffler "Error-Correcting Codes and Self-Checking Circuits" IEEE Computer March 1980, pp 27-37

4.21 J. J. Stiffler "Coding for Random-Access Memories" IEEE Transactions on Computers, vol. c-27, No. 6, June 1978, pp 526-531

4.22 Carl-Erik W. Sundberg "Erasure and Error Decoding for Semiconductor Memories" IEEE Transactions on Computers, vol. c-27, No.8, August 1978, pp 696-705

4.23 T. Mano et al., "Circuit Technologies for 16M-bit DRAM's", in ISSCC Dig. tech. Papers, Feb. 1987, pp 22-23

4.24 J. Yamada, "Selector-line Merged Built-in ECC Technique for DRAM's ", IEEE J. Solid-state Circuits, vol. sc-22 pp 868-873 Oct. 1987

4.25 J. Yamada et al., "A 4-Mbit DRAM with 16-bit Concurrent ECC", IEEE J. Solid-state Circuits, vol.23 pp 20-26 Feb. 1988

4.26 Sang H. Han and Miroslaw Malek, "A New Technique for Error Detection and Correction in Semiconductor memories", IEEE International Test Conference Papers, pp 864-870, 1987

CHAPTER FIVE UNIDIRECTIONAL ERROR CONTROL CODES

Most of the codes discussed in the previous chapters are effective against random errors. In computer memories there are other types of errors as well. They are asymmetric errors and unidirectional errors[5.1]. According to the predominant occurrences of the error types computer memories are referred as symmetric, asymmetric and unidirectional error memories.

By random error memories, we mean that the probability of 1 to 0 errors (1-error) is identical to the probability of 0 to 1 errors (0-error). While in symmetric error memories the probability of 1-errors is significantly greater than that of 0-errors (or vice versa), an ideal asymmetric error memory allows only 1-errors (or 0-errors, but not both) occurring. In the unidirectional error memories, both 1-error and 0-error are possible but in any particular codeword all the errors are of the same type (1-errors or 0-errors). Generally speaking, the asymmetric error memories are a subset of the unidirectional error memories.

In some recently developed LSI/VLSI ROM and RAM memories the most likely errors or faults are unidirectional errors. Also the number of symmetric errors, which are random errors, are relatively limited [5.2]. In these situations unidirectional error control codes may be more effective.

In this chapter some codes which are used for unidirectional error control in computer memories will be discussed. First, in section 5.1 unidirectional error detecting codes will be analyzed. Following that in section 5.2 t-random error correcting and all

unidirectional error detecting codes are discussed.

5.1 Unidirectional error detecting codes

Let X and Y be two binary n-tuples, we define N(X, Y) as the number of 1 to 0 crossovers from X to Y. N(X, Y) and N(Y,X) can be calculated as

$$N(X,Y)=\sum_{i=0}^{n} x_i\overline{y}_i \qquad N(Y,X)=\sum_{i=0}^{n} y_i\overline{x}_i \tag{5.1}$$

Using these parameters, the Hamming distance of the code can be then expressed as

$$d(X,Y) = N(X,Y) + N(Y,X) \tag{5.2}$$

A codeword X =(x_1, x_2, ..., x_n) is said to cover another codeword Y = (y_1, y_2, ..., y_n), if for all y_i = 1, where i=1, 2, ..., n, the corresponding values of x_i are x_i = 1[5.3]. In this case we say X >= Y. It can be seen that if X >= Y, then N(Y, X) = 0.

Unordered codes

A binary code C is called an unordered code if there is no codeword that covers another codeword, that is, for any pair X and Y in C, N(X, Y) >= 1 and N(Y, X) >= 1. For example, let

X = (1011) Y = (1101)

X_1= (1011) Y_1= (1001)

Then N(X,Y) = 1 and N(Y,X) = 1, hence X and Y are unordered

codewords. On the other hand, $N(X_1,Y_1) = 1$ and $N(Y_1,X_1) = 0$, therefore $X_1 > Y_1$ and they are not unordered codewords. The Hamming distances of the two codes are $d(X,Y) = N(X,Y) + N(Y,X) = 2$ and $d(X_1,Y_1) = N(X_1,Y_1) + N(Y_1,X_1) = 1$, respectively.

According to [5.3, 5.4], a code C is capable of detecting all unidirectional errors (AUED) occurring in a codeword if and if only the code is unordered, i.e.

$$\forall X, Y \in C, \quad N(X,Y) \geq 1 \ \wedge \ N(Y,X) \geq 1$$

The above theorem can be proved by the following argument. Let X be a valid codeword in unordered code C and X_1 is an erroneous codeword containing any number of unidirectional errors (1-errors 0-errors, but not both) from codeword X. It is easy to see that for m 1-errors occurring in X, $N(X,X_1) = m >= 1$ and $N(X_1,X) = 0$ and for m 0-errors occurring in X, $N(X,X_1) = 0$ and $N(X_1,X) = m >= 1$. In each case X_1 is not a valid codeword in the set of unordered codewords, hence can be detected as an erroneous codeword.

In chapter 3 we have stated that a linear code of minimum Hamming distance of d_{min} is capable of detecting up to $d_{min}-1$ errors (random errors). Now we see that an unordered code is capable of detecting all unidirectional errors in a codeword. In this sense unordered codes are more effective in unidirectional error cases.

Systematic and nonsystematic unordered codes

There are two classes of unordered codes, one is systematic

and the other is nonsystematic. Generally, nonsystematic codes need fewer check bits and provide a higher code rate, but need complicated decoding logic. Systematic codes can be encoded and decoded in parallel, but need more check bits.

As an example of nonsystematic unordered codes, we will discuss a balanced code, which is one of optimal m-out-of-n codes. Unlike most of other nonsystematic unordered codes, this code has a high code rate as well as a simple decoding logic.

As an example of systematic unordered codes, we will discuss Berger codes, which is the fundamental form of systematic unordered codes.

m-out-of-n codes

As we have discussed in section 4.1 that m-out-of-n codes are a class of single random error detecting codes. These codes can also be used for all unidirectional error detecting purposes.

All codewords in the code are of length n and have m 1's. For any X and Y in the code, if X < > Y, X does not cover Y. Therefore the code is unordered and capable of detecting all unidirectional errors.

For given n and m, the number of the codewords can be calculated as

$$|C| = \binom{n}{m} = \frac{n!}{m!\,(n-m)!} \qquad (5.3)$$

Freiman [5.4] and Leiss [5.6] have shown that the number |c| is

maximized when m = [n/2]. The [n/2]-out-of-n codes are optimal codes in terms of maximum number of codewords. Every codeword in the code contains equally number of 1's and 0's. So they are also called balanced codes. In the following we will develop an approach to constructing a class of efficient balanced codes which can be decoded in parallel [5.7].

Construction of efficient balanced codes

To obtain a balanced codeword, some of the information bits need to be modified so that there are equal numbers of 1's and 0's in the resulting word. The resulting word is a balanced information word. Meanwhile some check bits (called check word) which carry the information about how many message bits has been complemented are added to the balanced word. A complete codeword is then composed of a balanced information word and balanced check word. In the following, we will show that for any message word it is always possible to obtain a balanced word and decode them.

Let X be information word of k bits, W(X) be the weight of X, W(j) be the weight of the first j bits of X and X^j be the new word obtained from X by complementing the first j bits of X. Therefore $W(X^j)$ can be expressed as

$$W(X^j) = W(X) - W(j) + (j - W(j))$$

$$= W(X) + j - 2W(j) \qquad (5.4)$$

where $0 =< j =< k$. It can be seen that whenever j increases by one, the weight of the new word X^j changes by +1 or -1, called

random walk'. Suppose $W(X) = i$, then $W(X^0) = i$ and $W(X^k) = k - i$. This means, when complementing the first j bit of X, the weight of new word X^j are changed between i and k-i. Hence there exist at least one j such that $W(X^j) = [k/2]$, that is, for any information word, we can always find a j such that by complementing the first j bits the information word can be changed to a balanced word. For example,

if $X = 01110\ 01101$

then $X^5 = 10001\ 01101$

and $X^9 = 10001\ 10011$

There are two j's that make X to be balanced, they are j = 5 and j = 9. Only one j is taken in a given code.

In order to indicate the j, suppose a check word of r bits is required. Then the balanced r bits have to be sufficient to indicate the maximum possible j which is =< k. Therefore r has to satisfy the following relationship,

$$\binom{r}{\lfloor r/2 \rfloor} \geq j_{max} = k \qquad (5.5)$$

In the balanced code any number of unidirectional errors (1-errors or 0-errors) that occur in a codeword will affect the balance of the codeword, hence all unidirectional errors can be easily detected.

As can be seen, even though the codes are nonsystematic (some of the information bits are modified), the parallel decoding can be achieved. For example, the values of j's (the number of the bits

complemented) can be determined by a table-lookup scheme. The table provides all values of j's corresponding to the check words. The size of the table is limited up to k. When the value of j has been determined the original information word will be recovered using $X = (X^j)^j$.

Berger codes [5.4]

Another class of commonly used unordered codes are Berger codes. The construction of the Berger codes can be described as follows.

Let $X = (x_1\ x_2\ \ldots\ x_k)$ be an information word, the number of 0's in X be k_0 and p(X) be the check word for X, then the codeword has the form:

$$Xp(X)$$

where p(X) is the representation of k_0 in binary.

For example, if X = 00010110, then $k_0 = 5$ and p(X) = 101. The codeword is therefore 00010110 101.

Here the information words are separated from the corresponding bits and no information bits are modified. Therefore Berger codes are systematic codes.

On the other hand, if X and Y are two information words such that W(X) > W(Y), then p(Y) > p(X) and N(X,Y)>1. Since it is obvious that N(X,Y) > 1, the codeword encoded from X and Y are unordered. If W(X) = W(Y) and X < > Y, then they are already unordered. Therefore the Berger codes are unordered codes.

When unidirectional errors occur in a codeword (without loss

of generality, only 1-errors are considered) they may affect the codeword in three ways.

1) errors are in the information part X, in which k_0 will be increased and p(X) remains unchanged.
2) errors are in the check word p(X), in which k_0 will remain the same and p(X) will decrease.
3) errors are in both of information part X and check word part p(X), in which both k_0 and p(X) will change, but k_0 increases and p(X) decreases.

In all the above cases, the consistency between k_0 and p(X) is affected, hence any number of the unidirectional errors occurring in a codeword can be detected.

In the Berger codes, The maximum number of k_0 in a codeword is k. To represent such a k_0 in binary, the check bits required are at least $[\log(k + 1)]$, where [x] denotes the smallest integer greater than or equal to x. This number has been proven to be optimal for systematic unordered codes [5.4].

The decoding of the Berger codes is as simple as separating the information word from the check bits. So it is easy to achieve a parallel decoding.

It needs to be emphasized that both m-out-of-n codes and Berger codes are the basic forms for constructing unordered codes in the unidirectional error control and combinational errors control.

tED-AUED codes

Unordered codes can effectively detect all unidirectional errors but fail to detect random errors. To prevent some random errors from passing undetected , it is necessary to add some ability of random error detecting to the unordered code. These types of codes are tED-AUED codes that can detect t random errors as well as all unidirectional errors. In the following we introduce a general technique of constructing the tED-AUED codes[5.8].

Let k be the number of information bits, t be the required random error detecting capability. The construction of the codes is divided into two steps.

Step-1 Select an (n_1, k) code C_1 with $d_{min} = t+ 1$. Then encode the k information bit into a (n_1, k) codeword of code C_1.

Step-2 Select an (n, n_1) Berger code C_2, where n_1 bits are a codeword of C_1. Then encode the n_1 bits into a codeword of Berger code.

The length of the final code is $n = n_1 + [\log(n_1+1)]$. The code is capable of detecting t or fewer random errors and all unidirectional errors. In the implementation, the two steps mentioned above can be interchanged. The parameters shown in Table 5.1 for a set of tED-AUED codes where t = 2 and t = 3, are derived by using distance-3 and distance-4 Hamming codes as C_1.

The applications of this codes can be seen in ROM control stores of PDP11/40, IBM 370/168, and Nanodata QM1. The sizes of control words in these machines, respectively, are 56, 108, and 360 bits.

5.2 tEC-AUED codes

By tEC-AUED codes we mean the codes that correct t or fewer random errors and detect (t + 1) or more errors if all of the errors are unidirectional errors.

It is given in [8] that for all distinct codeword X and Y in code C, if N(X,Y) >= t + 1 and N(Y,X) >= t + 1, then C is capable of correcting t or fewer random errors and detecting all unidirectional errors.

This is because of the fact that N(X,Y) >= t + 1 and N(Y,X) >= t + 1, the code is unordered, hence is capable of detecting all unidirectional errors. On the other hand, the Hamming distance d(X,Y) is calculated as d(X,Y) = N(X,Y) + N(Y,X) >= 2t + 2, hence is capable of correcting t or fewer random errors.

As can be seen, all the m-out-of-n codewords with minimum Hamming distance (2t + 2) is a tEC-AUED code [5.9]. For example a SEC-AUED code can be constructed by selecting the subset of all its codewords of m weight with d=4. This is a nonsystematic code. However, there is no known efficient decoding scheme for it.

In the following we first discuss a general technique for constructing tEC-AUED codes, then introduce some other construction methods of tEC-AUED codes. Among them, SEC-AUED codes (t = 1) are the most interesting ones for computer memories.

General technique for constructing tEC-AUED codes[5.8]

The following technique is based on the concept of the product of two codes. This concept was also previously discussed in H-V-

parity codes in section 4.2.

Let k be the number of the information bits and not a prime number. Then it can be represented as $k = k_1 \times k_2$ for some k_1 and k_2 and be rearranged to a matrix form of $k_1 \times k_2$. The code construction is then divided into two steps.

Step-1 Encode k_1 rows using an (n_1, k_2) linear error correcting code C1. This C1 code must have $d_{min} >= t + 1$. The resulting matrix will have a size of $k_1 \times n_2$.

Step-2 Encode the n_2 columns into Berger codewords, The final matrix will be an $n_1 \times n_2$ matrix where $n_1 = k_1 + [\log(k_1 + 1)]$.

The bits in the $n_1 \times n_2$ matrix represent the codeword while the bits in $k_1 \times k_2$ are information bits. This is a systematic code. An illustration of constructing a SEC-AUED code using above steps are shown in the following example.

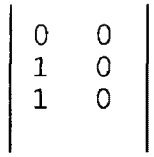
Example: Let k = 6, be arranged into a 3 x 2 matrix as given in Fig 5.1 (a). Since the single-bit parity code has a minimum Hamming distance of 2, it is used as row code. The matrix of this row code is shown in Fig 5.1 (b). The columns of Fig 5.1 (b) are encoded to Berger codewords. The final matrix is shown in Fig 5.1 (c).

The algorithm for decoding for SEC-AUED is as follows,

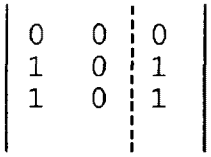
a) Check the top k_1 rows. Let the number of rows found in errors be e_1.

b) Check all of n_2 columns. Let the number of columns found in errors be e_2.

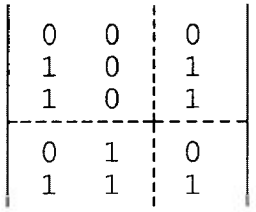
c) If $e_1 = e_2 = 0$, no errors are assumed.



(a) Information bits



(b) Row code matrix



(c) The final codeword matrix

Fig.5.1 Encoding of a SEC-AUED code

d) If $e_1 = e_2 = 1$, there is a single bit error in the information block located at the intersection of the row and the column where errors were found. This error bit is then corrected.

e) If e_1 and e_2 are not the value mentioned in c) or d), then multiple errors are detected.

Some specific tEC-AUED codes

Bose-Pradhan (BP) codes [5.10]

The codes are systematic codes. The codewords typically have the following form:

$$X\ B_1\ B_2 \ldots B_{t+1}$$

Here X represents the encoding of the given information bits into a codeword in a systematic code C^* where C^* has a minimum Hamming distance of $(2t + 2)$. $B_1\ B_2\ \ldots B_{t+1}$ are Berger type checkbits defined as follows.

The decimal value of B_1 is equal to the number of 0's in X and B_j is equal to the number of 0's in $(X\ B_1 \ldots B_{j-1})$ for $j = 2, 3, \ldots, t+1$.

Suppose X = 000 110 100 101 010 is a codeword in C^* with $d_{min} >= 2t+2 = 6$. The checkbits are computed as bellow

X = 000 110 100 101 010 there are nine 0's in X,

B_1= 1001 there are eleven 0's in $(X\ B_1)$

B_2= 1011 there are twelve 0's in $(X\ B_1\ B_2)$

B_3= 1100

Then the entire codeword is expressed as

$$\underbrace{000110100101010}_{X}\ \underbrace{1001}_{B_1}\ \underbrace{1011}_{B_2}\ \underbrace{1100}_{B_3}$$

The well-known Berger codes can be considered as a special case (t = 1) of these codes. The number of the check bits is ($t\log_2 n$) where n is the length of codeword of C^*. It can be seen that the coding rate of this type of codes is much higher than those constructed by general technique discussed above.

The decoding algorithm of the BP codes can be described as follows.

Let $R = X\ B_1B_2...B_{t+1}$ be an error free codeword that is to be stored in memory and $R' = X'B'_1...B'_{t+1}$ be a codeword read from the memory which is corrupted by random errors and unidirectional errors.

1) Compute B_1" (= The number of 0's in X') and B_j" (= the number of 0's in $(X'\ B_1'...B'_{j-1})$), and then compute the syndromes

$$s_j = |\ B_j" - B_j'\ |$$

where |Z| denote the absolute value of Z

2) If all of s_j's have values of greater than t, then more than t errors are detected but they are uncorrectable.

3) If at least one of the s_j's has value less than t, then there are t or fewer errors in the codeword.

4) Apply the error correction procedure of code C^* to X'. The resulting word is now X".

5) Recompute B_j" to recover the complete codeword R. In some applications, recovering X is sufficient, so the step 5) can

be omitted.

NGP codes [5.11]

These codes are also systematic codes which have the same codeword form as BP codes discussed above, but need fewer checkbits, hence provide higher coding rate. the codewords of the code have the form

$$XB_1B_2..B_{t+1}$$

where X is the systematic parity check codeword of code C^* with length n and minimum Hamming distance $d_{min} >= 2t + 1$, and $B_1\ B_2...B_{t+1}$ are the binary representations of $(B_1)\ (B_2)\ ...(B_{t+1})$, where (B_j)'s are related to the number of 0's in X. Let k_0 be the number of 0's in X, then (B_j) can be generated from the following

$$(B_j) = \lfloor \frac{k0}{2j-1} \rfloor \qquad (5.6)$$

Comparing with BP codes , this codes have higher coding rate.
Decoding algorithm

The detection and correction algorithm for the codes is described as follows.

Let $R = X\ B_1\ B_2...B_{t+1}$ be the error free codeword which is stored in the memory, $R' = X'\ B_1'\ B_2'...B'_{t+1}$ be the codeword read from the memory which is corrupted by random errors or unidirectional errors.

1) According to the decoding procedure for the parity check code C^*, compute the syndrome S of X'.

2) According to the rules of generating B_j's described above, compute the values of check symbols D_1 D_2 ...D_{t+1} which correspond to X', and set

$$Q1 = W(B_1' \; B_2' \; \ldots \; B'_{t+1} \oplus D_1 \; D_2 \; \ldots D_{t+1})$$

3) If S = 0 and Q1 = 0, no error has occurred. The received codeword R', is assumed to be correct. Otherwise some errors have occurred.

4) Let the syndrome S correspond to q multiplicity error. If q > t then the errors are detected but uncorrectable. Other wise set Z = k.

5) Correct X' by using the correction procedure in the parity check code C^* and obtain X" as the resulting word.

6) Recompute the values of check symbols D_1' D_2' ...D'_{t+1} which correspond to X" and set

$$Q = Z + W(B_1' \; B_2' \ldots \; B'_{t+1} \oplus D_1' \; D'_2 \ldots D'_{t+1}).$$

7) If Q =< t, the word X"$D'_1D'_2 \ldots D'_{t+1}$ is the correct word. Otherwise the errors are only detectable.

Among the above seven steps, steps 1) and 2) are independent, hence can be implemented in parallel.

Examples of SEC-AUED NGP codes and their decoding

In computer memories SEC-AUED codes are special interest. In the following an example of encoding and decoding of a SEC-AUED code is described.

Let X = 11101000 be a (8,4) SEC modified Hamming code with the parity check matrix

$$H = \begin{bmatrix} 1 & 1 & 1 & 0 & 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 1 & 0 & 1 & 0 & 0 \\ 1 & 0 & 1 & 1 & 0 & 0 & 1 & 0 \\ 0 & 1 & 1 & 1 & 0 & 0 & 0 & 1 \end{bmatrix}$$

To form a SEC-AUED codeword, we generate B_1 and B_2 according to Eq.(5.6)

$B_1 = 0100$ $B_2 = 01$

The complete codeword is therefore R = 11101000 0100 01.

Random error correction:

Let us assume that random error occurred at position 3 of the codeword R, thus the erroneous codeword becomes

R′ = 11<u>0</u>01000 0100 01

where the underline bit is in error and $B'_1 = 0100$, $B'_2 = 01$. The decoding is as follows

1) According to Eq.(4.10) $S = H X'^T = [1011]^T$

2) Regenerate $D_1 = 0101$, $D_2 = 01$ and set

$$Q1 = W(B'_1, B'_2 \oplus D_1, D_2) = 1$$

3) Since S < > 0, some errors have occurred.

4) Since S = (1011) which matches to the third column of the H matrix, a single error is assumed at the third bit. Therefore set Z = 1. After the correction is done. The resulting word is now X" = 11101000.

5) Recompute the check bits $D_1' = 0100$, $D_2' = 01$ and set

$$Q = Z + W (B_1', B_2' \oplus D_1', D_2') = 1 + 0 = 1$$

6) Since $Q =< t = 1$, the corrected codeword is taken as correct. That is R" = R = 11101000 0100 01.

Unidirectional error correction:

We suppose that there are five 1-errors occurring at position 1, 3, 5, 10 and 14 of the same codeword R = 1̲1̲0̲1000 0̲100 0̲1̲ where the underline bits are to be in error. Thus the erroneous codeword becomes

$$X' = 01000000 \qquad B'_1 = 0000 \qquad B'_2 = 00$$

1) Compute $S = H X'^T = [\ 1101\ ]^T$

2) Regenerate $D_1 = 0111$, $D_2 = 10$ and set

$$Q1 = W (B_1', B_2 \oplus D_1, D_2) = 4$$

3) Since $S < > 0$, errors have occurred.

4) Since S (= [1101]) matches to the second column of the H matrix, a single error at the second bit is assumed. Therefore set Z = 1. After the correction, the resulting word becomes X" = 00000000.

5) Recompute the checkbits $D_1' = 1000$ and $D_2' = 10$ and set

$$Q = Z + W (B_1', B_2' + D_1', D_2') = 1 + 2 = 3$$

6) Since $Q = 3 > t = 1$, the errors are only detectable.

5.3 Summary

In this chapter unidirectional errors control codes have been

discussed. The most important concept for all unidirectional error detection is that the codes have to be unordered. There are two basic types of unordered codes, nonsystematic and systematic unordered codes. An m-out-of-n code is a typical nonsystematic unordered code. The optimal m-out-of-n codes are [n/2]-out-of-n codes. This is also called balanced codes. We discussed a class of efficient balanced codes which can be decoded in an efficient way.

Well-known Berger codes are the most common type of systematic unordered codes. Many other codes used for unidirectional error control are constructed based on Berger codes and the like.

In section 5.2 some tEC-AUED codes are discussed. The general way to generate a tEC-AUED code is to append some Berger type checkbits to a tEC systematic parity check code C^*. The product codes have a simple encoding and decoding concept but need more redundancy. The BP code are much better than the product codes in term of coding rate. The NGP codes provide even more improvement with respect to the redundancy. In recent years many proposals regarding methods of constructing and decoding tEC-AUED codes have come forward[5.7, 5.8, 5.9, 5.10, 5.11].

Since the decoding complexity and delay tend to increase rapidly with the number of errors to be corrected, in practice only single or double error correcting codes are used. That is the main reason for why we only illustrated the encoding /decoding of SEC-AUED codes in the chapter.

Finally it is necessary to note that the methods for constructing tEC-AUED codes discussed in the section 5.2 can be extended to constructing t-EC/d-ED/AUED codes with $d > t + 1$ by selecting a systematic code C^* with capacity of t-random error correcting and d-random error detecting[5.9]. The codes that can correct t random errors and detect d random errors and also detect all unidirectional errors (t-EC/d-ED/AUED) are presented.

However, we also notice that the unidirectional error codes we discussed in this chapter have not yet found widespread application in computer memories[5.12]. There are two reasons: The requirement of higher redundancy than that of the linear parity codes and the incompatibility with parity-check codes which have been used widely in computer memories fields for many years. Even though, there is significant potential for high performance computers in future with the advancement even larger capacity memory technology.

References

5.1 Serban D. Constantin and T. R. N. Rao, "On the Theory of Asymmetric Error Correcting Codes" Information and Control 40, 20-36 (1979) pp 20-35

5.2 D.K. pradhan and J.J. Stiffler, "Error Correcting Codes and Self-checking Circuits" IEEE Transactions on Computers, (special issue on fault tolerant computing) pp 27-37 Mar. 1980

5.3 Bella Bose, "On Unordered Codes" IEEE Transactions on Computers, vol. 40, No.2, Feb. 1991 pp 125-131

5.4 J.M Berger, "A Note on Error Detecting Codes for Asymmetric Channel" Infor. contr., vol. 4, pp 68-73 Mar. 1961

5.5 C. V. Freiman "Optimal Error Detecting Codes for Completely Asymmetric Binary Channel" Inform. contr., vol. 5, pp 64-71 Mar. 1962

5.6 E.L. Leiss, "Data Integrity in Digital Optical Disks" IEEE Transactions on Computers, vol. c-33, No. 9, Sept. 1984 pp 818-827

5.7 Mrio Blaum, Rodney Goodman, and Robert Mceliece, "The Reliability of Single-Error Protected Computer Memories" IEEE

Transactions on Computers, vol. 37, No. 1, Jan. 1988, pp 114-119

5.8 Dhiraj K. Pradhan, "A New Class of Error-correcting /detecting Codes for Fault-tolerant Computer Applications" IEEE Transactions on Computers vol. c-29, No. 6, June 1980, pp 471-481

5.9 Bella Bose, Thammavaram R. N. Rao, "Theory of Unidirectional Error Correcting/detecting Codes" IEEE Transactions on Computers, vol. c-31, No. 6, June 1982, pp 521-530

5.10 B. Bose and D.K. Pradhan, "Optimal Unidirectional Error Detecting/correcting Codes" IEEE Transactions on Computers, vol.c-31, No. 6, June 1982, pp 564-568

5.11 Dimitris Nikolos, Nicolas Gaitanis, and George Philokyprou, "Systematic t-error Correcting/All Unidirectional Error Detecting Codes" IEEE Transactions on Computers, vol. c-35, No. 5, May 1986, pp 394-401

5.12 D. K. Pradhan, J.J. Stiffler, "Error-Correcting Codes and Self-checking Circuits" IEEE Computer, March 1980, pp 27-37

CHAPTER SIX CONCLUSION

In computer memories there exist hard errors and soft errors. Hard errors are caused by permanent failures occurred in memory during manufacture or service time. Soft errors are environmentally induced and not permanent. For example, the radiation of alpha particles are known as the most likely source of soft errors.

For hard errors, commonly used scheme to reduce their consequences are such as spares of memory cells, periodical maintenance of memory devices and replacement for those failed memory components. Proper memory organization is also very effective to decrease the effects caused by such errors. Per-bit-per-chip organization, for example, enables memory to disperse several errors into several memory words so that every word could be contaminated by only one bit. In this organization the dominant error patterns are usually single error patterns.

For soft errors, since they occur randomly not only in positions but also randomly in time, it is very difficult to predict them before their happening and is very difficult to replace them when they have happened. Therefore some on line or real time protection have to be employed. Error control coding techniques are very effective for these purposes. In this thesis we have discussed several error control codes used in computer memories. Some of them are practically implemented in computer industry while others have not yet been widely used. For the latter we have pointed out their potential uses in future

applications.

Error detecting codes are simple in terms of implementation of encoding and decoding. As an example, parity codes, which have one bit redundancy attached to message bits, are of ability of detecting one bit error or odd number of errors in a codeword. Error detecting codes do not correct any errors. Once an error is detected there is no way to recover the original codeword. Computer memory has to employ other methods to get rid of the error word and obtain correct one. A request for repeat transmission of the codeword until correct one is received is one of the methods which was frequently used in early computers. Error detecting codes are usually used in small memory systems.

To achieve higher reliability of computer memory, error correcting ability is required. Hamming codes are among firstly selected codes for error correcting and error detecting purposes. Hamming codes provide with capability of single bit error correcting or double bits error detecting. It has simple encoding and decoding logic. When a syndrome of a particular codeword is calculated and the syndrome is found out matched to one of the columns of the H matrix, the location the error bit is then determined. If there is no such a match at all, double errors are detected without indicating the locations of the errors. Modified Hamming codes have higher decoding speed. This is because the decoder of the code is optimized in terms of the levels the XOR gates used in decoder. A report shows that combined use of memory cell redundancy and modified Hamming code could enhance the

reliability and improve the yield of the memory by thousand times(see figure 4.12). As an alternative of single error correcting and double error detecting code, in the thesis we have introduced a so called H-V-parity code. Due to the simple circuit structure of its decoder, it is well suited for on-chip error control schemes. The improvement of reliability by using this code is reported to be up to 10^6.

For multiple error correcting codes, we found that some codes which are usually used in communication systems , such as BCH code, are not very convenient for computer memories. This is because the decoding of BCH code is performed serially and therefore is relatively slow in speed. Modern high speed computers can not afford its decoding delay. We have studied a new class of multiple error correcting codes, orthogonal latin square codes (OLS). The OLS codes are one step decidable codes. The most attractions of these codes are their high decoding speed and flexible circuit features that allows to add more error correcting capability by adding corresponding circuit modules without changing the existing circuit structure. This is essential for computer memory to expand its error correcting capability. The disadvantage of the OLS codes is its high redundancy requirement. However, with the development of VLSI technology they are becoming a very strong candidate for multiple error correcting applications.

In some recently developed LSI/VLSI ROM and RAM memories the most likely error are of unidirectional characteristics. In this special case a class of unidirectional error detecting /correcting

codes could be more effective. We have pointed out that an unordered code is capable of detecting all unidirectional errors in a code word. Well known Berger codes are a class of typical systematic unordered codes. Many other unidirectional codes are based on Berger codes. Unfortunately, unidirectional codes have not been widely used in computer memory systems. One of the reasons for this is that the encoding and decoding of unordered codes are more complicated than Hamming codes and modified Hamming codes. In most cases Hamming codes and modified hamming codes are believed to be effective enough. Unidirectional error control codes are incompatible with parity codes while the parity codes have already been implemented in practice. However, unidirectional codes are very interesting and promising in large, high reliable computer memory systems.