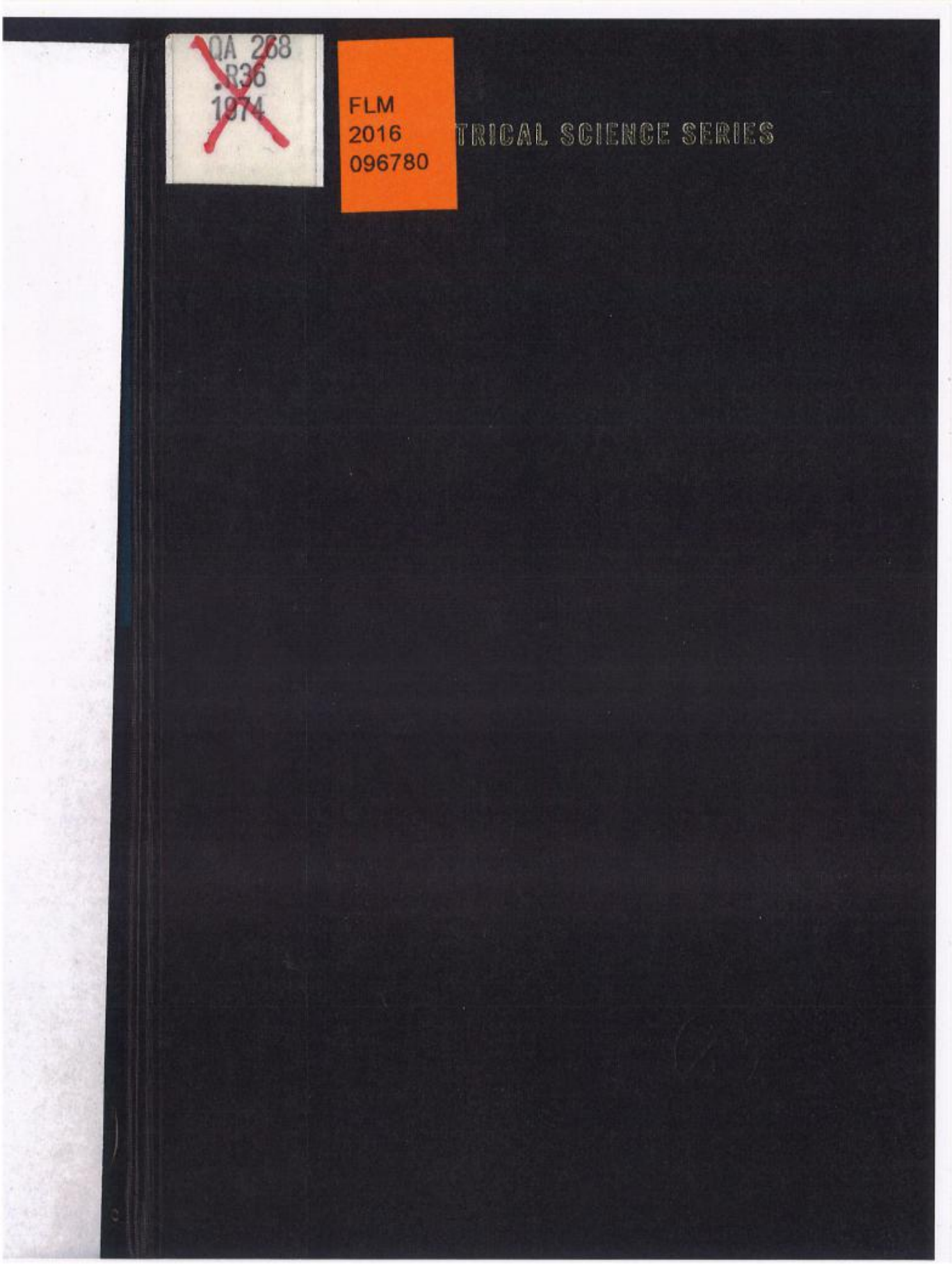
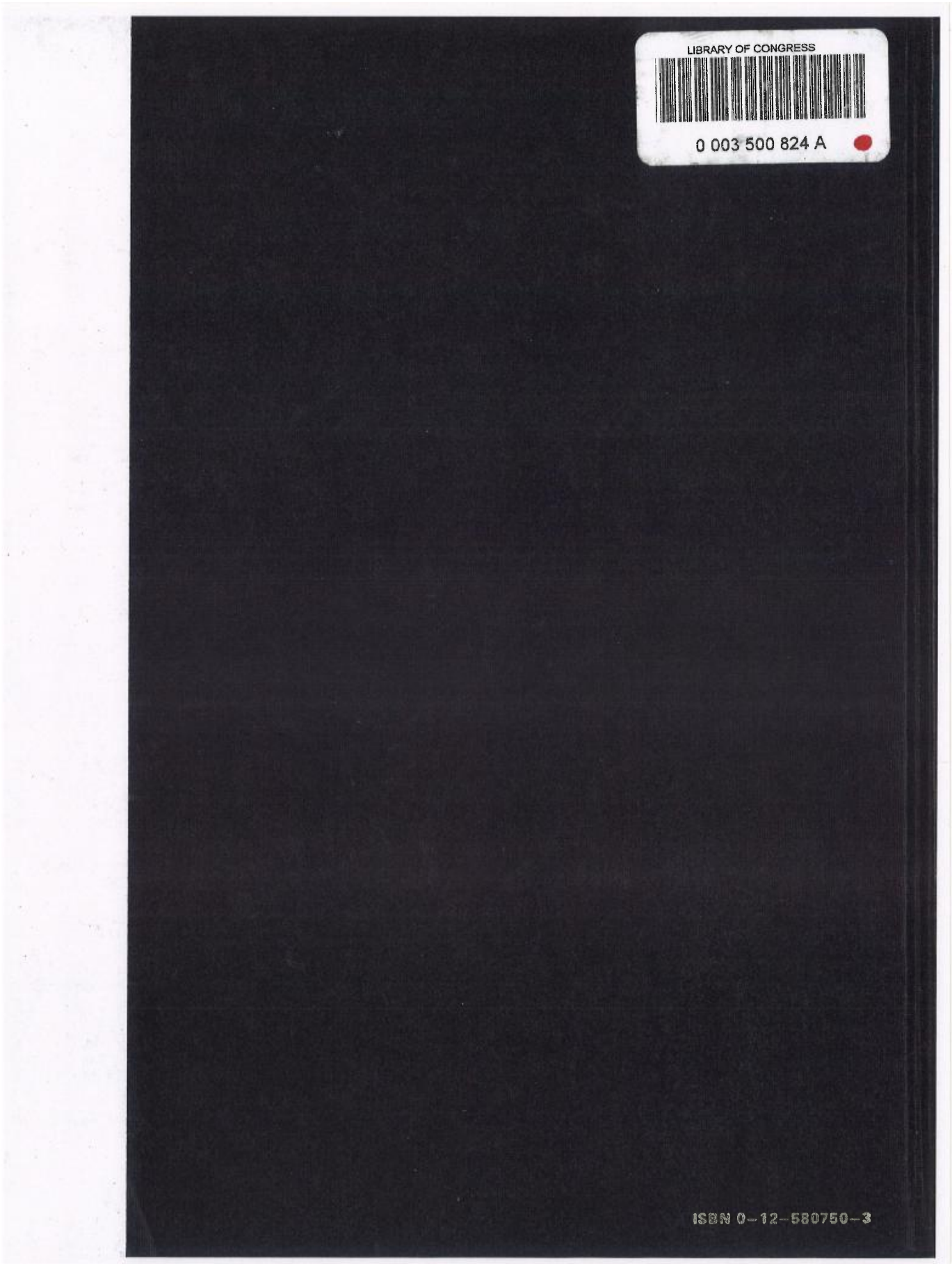
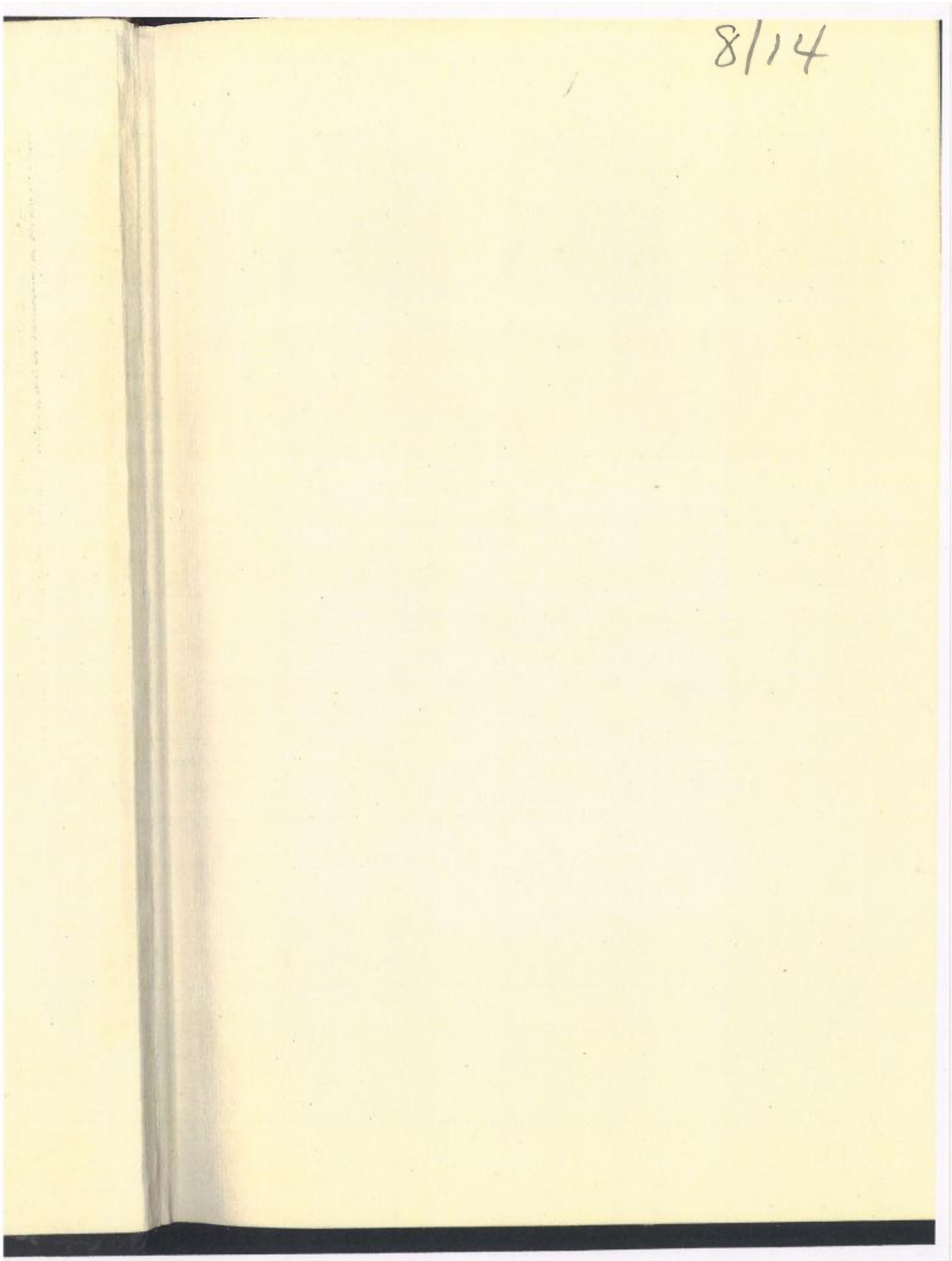



LIBRARY OF CONGRESS
0 003 500 824 A
ISBN 0-12-580750-3


THE LIBRARY OF CONGRESS
1800


8/14

ERROR CODING FOR ARITHMETIC PROCESSORS

ELECTRICAL SCIENCE
A Series of Monographs and Texts

Editors: Henry G. Booker and Nicholas DeClaris
A complete list of titles in this series appears at the end of this volume

CIP 4 SEP 1974

ERROR CODING FOR ARITHMETIC PROCESSORS

T. R. N. Rao

Department of Electrical Engineering
University of Maryland, College Park

ACADEMIC PRESS *New York and London · 1974*

A Subsidiary of Harcourt Brace Jovanovich, Publishers

ACADEMIC PRESS, INC.
111 Fifth Avenue, New York, New York 10003

United Kingdom Edition published by
ACADEMIC PRESS, INC. (LONDON) LTD.
24/28 Oval Road, London NW1

Library of Congress Cataloging in Publication Data

Rao, Thammavarapu R N Date
Error coding for arithmetic processors.

(Electrical science)
Includes bibliographical references.
1. Error-correcting codes (Information theory)
2. Computer arithmetic and logic units. I. Title.
QA268.R36 1974 519.4 73-22381
ISBN 0-12-580750-3

PRINTED IN THE UNITED STATES OF AMERICA

To

Rajyalaxmi

Ramakrishna, Chandrika, and Radhika

"విబుధ జనుల వలన విన్నంత కన్నంత
తెలియవచ్చినంత తేటపరుతు"

బమ్మెర పోతరాజు

What I have heard or seen
from many a scholar,
a pious hope it has been
to render it crystal clear.

Translated from Bammera Pothanna's "Bhagavatham."

Contents

Foreword

Computation without error remains an illusive goal of considerable importance in certain critical applications which require sophisticated and extensive computation with a high degree of system reliability. Recent advances in solid state technology have provided individual devices with exceptional reliability. In some systems, this improvement in device reliability has obtained sufficient systems reliability. However, in others, the large number of devices required has negated the improvement in reliability at the systems level. Such problems can be solved by the unlikely development of a perfect device which never fails. In the absence of such a device, one can expect greater use of the techniques of fault-tolerant computing to obtain improved systems reliability. Such improvement is not obtained without cost in performance or equipment, but in some applications, the cost is justifiable.

The technology of fault-tolerant computing is, at the present time, in its infancy and much development is still needed. In particular, more research and development directed toward realizable implementations is needed. There exists a substantial amount of literature on the subject, but the number of successful applications remains limited. However, the need for fault-tolerant computing will probably increase rather than decrease in the future.

Professor Rao, in this book, considers arithmetically invariant

codes, which is an important technique of fault-tolerant computing. He has surveyed the extensive literature on the subject and organized it in a form which should be extremely useful to researchers in this area.

HARVEY L. GARNER
Moore School of Electrical Engineering
University of Pennsylvania
Philadelphia, Pennsylvania

Preface

Since the early work of Diamond (1955), there has been continuous progress in arithmetic coding theory. This field derives its strength and vitality from the classical works of Brown (1960), Peterson (1961), Chien (1964–1972), Massey (1964), Garner (1966), and others.

The purpose of this book is to combine the available knowledge on arithmetic codes and bring it under one cover for the benefit of students and researchers in this field of specialization. The presentation here includes the necessary mathematical background and error control preliminaries in order for this book to serve as a viable text in an advanced undergraduate or a graduate level course. The preliminary drafts of this book have been used for a graduate course in arithmetic coding and are presently being used in a new (modified version of the above) course entitled "Arithmetic Codes and Fault-Tolerant Computing" at the University of Maryland.

The first two chapters provide the student with a minimal mathematical background in algebra, number theory, and error control techniques. Simple mathematical models for registers, arithmetic processors, and elementary arithmetic operations are introduced. Chapter 3 introduces arithmetic codes, definitions, and code classifications. Single-error detection using AN codes and separate codes is also presented. Chapters 4 and 5 cover single-error-correcting codes using

AN codes and separate codes, respectively. A section on byte-error correction using higher radix codes is included in Chapter 4. In Chapter 6 we introduce code conversion algorithms leading to a presentation of the large distance codes, namely, Barrows–Mandelbaum codes and Chien–Hong–Preparata codes. Chapter 7 begins with the systematic nonseparate category of codes and covers codes for burst errors and iterative errors. We conclude the book with a presentation of codes and their applications in Chapter 8.

I truly believe that I was very fortunate in having the opportunity to study under Professor Harvey L. Garner at the University of Michigan. I have benefited from and been profoundly influenced by his works, thoughts, and teachings. Many fundamental definitions and notations and much terminology used in this book reflect this background and influence.

This book contains a number of sections which report on the results of our research on "Residue Codes and Application to Arithmetic Processors" during the years 1967–1972. This research was made possible by grants from the National Science Foundation and the National Aeronautics and Space Administration. I acknowledge very gratefully the initial guidance and valuable suggestions on the subject matter of this book given by Professor Robert T. Chien of the University of Illinois. The comments and suggestions from Professor Oscar N. Garcia of South Florida University and Dr. Se June Hong of the IBM Corporation on portions of this book have been most helpful.

Finally I wish to thank my wife, Rajyalaxmi, for her understanding nature and constant encouragement without which this book would not have been possible.

T. R. N. RAO

2 ARITHMETIC PROCESSORS AND ERROR CONTROL PRELIMINARIES

This chapter introduces the subject of arithmetic processors and error control techniques. In the first section, we define arithmetic processors (AP), arithmetic operations, and a model to describe their functional behavior. In the second section, *logic faults*, which are the cause of errors (both numerical and logical), are discussed. The origin and nature of logic faults are discussed. Then *arithmetic weight* of errors (or error numbers) is introduced. In Section 2.3, some error control techniques, namely, triple modular redundancy (TMR) as well as duplication and switching, are introduced.

2.1 ARITHMETIC PROCESSORS AND DIGITAL COMPUTERS

A digital computer is divided, for convenience, into a number of functional units (or subsystems), such as arithmetic processor (AP), control unit (CU), memory unit (MU), input/output unit (I/O unit), program unit (PU), etc. (see Figure 2.1). These divisions are useful from the point of view of computer organization and design. The various units are connected together appropriately to enable processing of the information required of it. Stated briefly, the function of the

memory unit(s) is to store data, instructions, and intermediary results and to enable transfer of these to the arithmetic unit or I/O unit as required. The control unit handles the supervisory functions such as providing reference signals (or clock pulses) to initiate or terminate the various functions of these units. The I/O units provide the ability to communicate with the outside world. It may consist of tape readers or card readers and punch tape or punch card equipment. The arithmetic processor receives data (or operands) and operation commands (or instructions) from the memory and control units and performs the needed processing. The results of these operations are stored in the

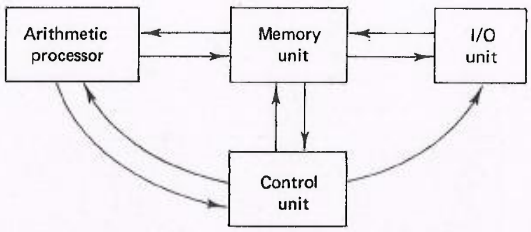


Figure 2.1 An organization of the functional units of a computer.

specified memory locations or retained in the arithmetic registers which are included as part of an AP. The nature and complexity of these various functional units depend on the class of problems they are to solve, and the instruction repertoire. Besides, the speed–cost trade-offs play a significant part in the makeup of each functional unit. No less important are factors such as the type of number system used, the arithmetic procedures (or algorithms) built in, the maintenance circuits (if any provided), and the required "reliability" or "dependability" of operation of the entire system. Therefore, any general characterization of any unit such as an AP will not be meaningful for any special-purpose computer operation but applies only in a rather general way. Since our interest here lies in (the error control coding for) arithmetic processors, we discuss their organization next.

Organization of an arithmetic processor

An arithmetic processor is the part of a computer that provides the logic circuits required to perform arithmetic operations such as ADD, SUBTRACT, MULTIPLY, DIVIDE, SQUARE-ROOT, etc. There are a number of other operations such as *complement*, *shift*, *rotate*, and *scale* which are classified at times as arithmetic operations and are performed by arithmetic units of large- and medium-sized computers. In order to perform these operations, an AP usually consists of a number of registers to store operands, intermediate results (partial sums or partial products), and logic blocks such as adders, division logic, overflow detection logic, shift–rotate logic, and in addition, several gating and decoding and control logic blocks.

A sample organization of a small arithmetic processor is shown in Figure 2.2. This setup shows four arithmetic registers, called *accumulator*, *addend*, *augend* (or *multiplicand*), and *multiplier–quotient* registers. Parallel adder, complement, and shift–rotate logic blocks are shown.

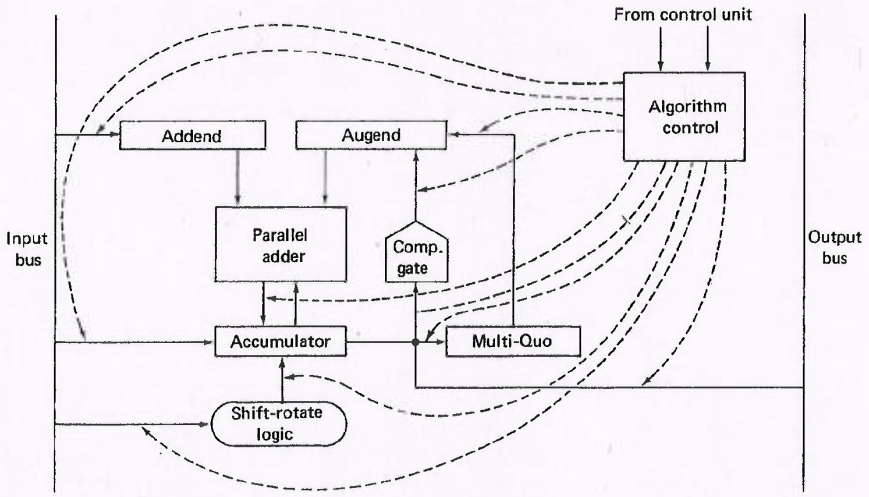


Figure 2.2 Sample organization of arithmetic processors.

Also indicated are an algorithm controller which receives the operation command from the control unit, and control signals to the logic blocks. These organizations come in a great variety and differ enormously in complexity. For the purpose of understanding the operation of an AP, we resort in the next section to a simple model which characterizes, in a rather general way, its operation from the point of view of error control coding logic. In this discussion one has to keep in mind the objective of this study, which is error control coding and not the design aspects of an AP. We discuss an organization only to enable a good understanding of the error control techniques presented later.

A simple model for arithmetic processors

An AP can be described by the inputs (operands and the operation command) and outputs (the sum or product, etc.). We can assume, without loss of generality, that one or more of the operands have already been supplied to the AP, and are stored in the appropriate registers; and the input required in such cases may be simply an operation command. "Shift right five times the contents of an accumulator" is an example of that type of instruction. For instructions such as ADD, generally one operand such as an augend is already available in the accumulator (or augend register), and another will be an input to the AP. Similarly, the outputs may simply be posted in one or more registers of the AP and can be retrieved from it on a separate operation command. Therefore, a block diagram of the type shown in Figure 2.3 can serve as a model for an AP.

Without any loss of generality, the input operand B, the internal operand A (available initially in the register A), and the results R are each assumed to be of n binary digits (bits). The opcode for Φ may be k bits long, where k must be sufficiently large to accommodate all the possible operations of an AP. (Note that the number of different operation commands must be less than or equal to 2^k). Further, we could denote the results R by $R = \Phi(A, B)$. R is the result of the specified operation Φ (when the input operand is B and the internal operand is A).

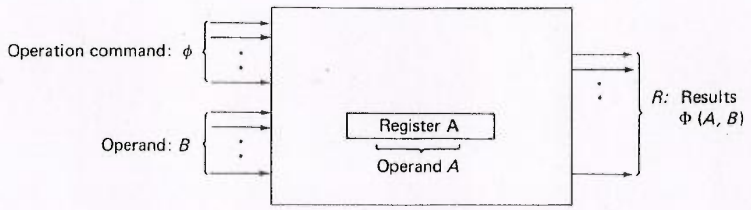


Figure 2.3 A model for an arithmetic processor.

The outputs R are numerical and/or logical results. The interpretation of the outputs as to the nature of their values is left to the control function. If Φ is the ADD instruction, then R may represent the sum, $A + B$ modulo m, denoted as $|A + B|_m$, where $m = 2^n$ for 2's complement binary logic, or $m = 2^n - 1$ for 1's complement binary logic.† If Φ denotes "cyclic shift left register A by one place," then R may represent the contents of register A after the execution of the instruction. If the register A represents the integer A (where $0 \leq A < 2^n$) before the cyclic shift, the integer value of A after the shift equals $|2A|_{2^n-1}$, which may also represent the numerical value of R. If Φ represents an operation, say "clear and add" (CLAD), then the register A will be replaced by the operand B. If Φ represents MULTIPLY, then R may represent the n most significant binary digits of the product of A and B. The least significant bits may be stored in one of the specified arithmetic registers.

The discussion above is intended as an overall view of the operations of the arithmetic units. A sample of operations has been given in Table 2.1. The seven operations listed first may be termed "elementary," and the rest "compound."

A compound operation can be performed by repeated use of one or more of the elementary operations; for example, MULTIPLY can be realized by repeated ADD, SHIFT operations. Therefore a number of medium-sized computers may not have any "built-in" compound

† For formulas which characterize these elementary operations and for examples the reader is advised to see Section 1.3 and in particular Table 1.4.

Table 2.1 Arithmetic operation commands

Elementary operations	
ADD	Φ_{ADD}
SUBTRACT (Complement and Add)	Φ_{SUB}
SHIFT RT[a]	Φ_{SHR}
SHIFT LT	Φ_{SHL}
CYCLE LT	Φ_{CYL}
CLEAR and ADD	Φ_{CLAD}
STORE ACC	Φ_{STA}

Compound operations	
MULTIPLY	FLOATING PT. ADD
DIVIDE	SCALE
SQUARE-ROOT	COMPARE

[a] We discussed previously in Chapter 1 two types of SHIFT RT operations. One is "Logical Shift Rt" and the other "Arith Shift Rt."

operations, but only a good set of elementary operations. Further, an elementary operation may be assumed to use a logic block or logic circuit just once. There are exceptions to this; for example, in the serial ADD operation, a single full adder stage is repeatedly used. This aspect of single-use or multiple-use of a logic block is an important concern for error control coding and is therefore discussed in detail in Chapter 7 in the section on iterative errors.

Further, if an effective error-correcting scheme is available for all elementary operations, then one could say that all arithmetic operations can be error controlled. Then the elementary operations that are more amenable to error control coding become an important part of our study.

2.2 NATURE AND ORIGIN OF ERRORS IN AP's

The words *errors*, *logic faults*, *component failures*, *malfunctions*, and *troubles* have been defined and used previously by different authors. Unfortunately there is no uniformity in their definitions. Before we

attempt some definitions of our own here, it may be appropriate to make a clear distinction between the words component, logic element, logic network, functional unit, etc. (See Figure 2.4.)

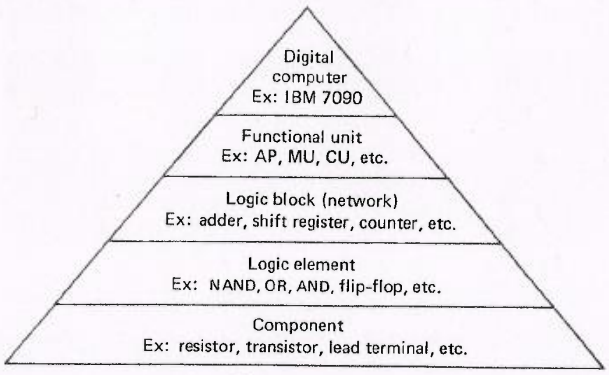


Figure 2.4 Subdivisions of a digital computer.

A *component* is the smallest building block of a digital computer, for example, a resistor, transistor, diode, or lead terminal. A *logic element* refers to a logic gate such as NAND, OR, AND, NOT, or a flip-flop. A *logic network* (or logic block) refers to a combinational network such as an adder, a complementer, an overflow detector, or a sequential network such as a shift register or a binary counter. Logic networks may vary in size considerably from one to another. A functional unit consists of a number of logic networks interconnected to perform any operation from a comprehensive set of operations. *Functional units* refer to large units such as an arithmetic processor, a control unit, or a memory unit.

Logic faults and their classifications

A *logic fault* or simply a *fault* is a deviation of logic variables from their specified values at one or more points of a logic network. Logic faults in an AP may be grouped broadly into two categories:

1. Control logic faults, which are due to failures† or malfunctions in the control logic. As an example, a logic fault in the operation command decoder lets a wrong algorithm (Φ' instead of Φ) be executed.
2. Arithmetic logic faults, which include logic faults in an adder, or division logic block, etc.

Both categories of faults relate to logic faults within the processor. If wrong inputs have been applied, that would constitute faults outside the premise of AP but are logic faults of the control processor or memory unit. Our concern is directed mainly toward obtaining error-free arithmetic processing through detection, location, and correction of errors originating in an AP. The techniques and codes studied for this purpose can also be used for error control of other functional units in an appropriate manner.

A logic fault was defined previously to be a deviation of logic variables from their specified values at one or more points of a logic network. A fault may invert a binary variable from 1 to a 0, from 0 to a 1, or force it to assume a constant logic value (such as stuck-at-1 or stuck-at-0). These faults are naturally caused by component failure(s) or malfunctions in a logic network. The origins of logic faults are related to component failures or malfunctions, electrical noise, overheating, etc., while the effects of logic faults are the errors in the outputs (or numerical results) of the AP.

The faults may be classified as *temporary* or *permanent*, *single* fault or *multiple* fault, *single-use* or *multiple-use*. This classification is based on the origin, nature, or use of the faulty network. *A temporary fault* is usually caused by a component malfunction or electromagnetic noise interference; its effects (the errors) cannot be reproduced under the program control. *A permanent fault is often caused by a component failure, and its effects are reproducible.* A *single fault* refers to one error-

† A component failure is the permanent destruction of a component such as a shorted or an open diode, or a grounded lead. A component malfunction is a temporary misbehavior due to outside noise interference, or overheating, or a marginal component.

inducing event, a *multiple fault* (double or triple, etc.) refers to several single faults occurring simultaneously. The effects of a multiple fault may be described as the cumulative effect of the individual single faults. A fault (either single or multiple) can be classified as *simple* (local) or *complex* (distributed) depending on its effect on the errors. A simple fault causes minimal damage on the outputs, or in other words, may produce an error of the type $\pm 2^j$; a complex fault causes extensive damage. Thus, a fault in a logic element, depending on the intricacy of the location of that logic element, is either simple or complex. By this definition, control faults are often classified as complex. This classification is based on the damage wrought on the results of an elementary operation or by the error value E, and more precisely, by its *arithmetic weight* $W(E)$, to be defined later.

The types of errors produced by a fault also differ from one operation to another. If an operation does not require the use of a faulty network, then the fault is virtually inactive for the duration of that operation. Another operation may use that faulty network just once, and still another operation may require repeated use of the same faulty network. Thus a logic fault will have different levels of damage on the output results of elementary operations and complex operations. This fact should be borne in mind by the error control coding designer. As an example, a *simple fault* may be due to a component failure in the carry generation logic of the ith stage of a parallel adder, so that the error generated (in the addition operation) has a value $E = \pm 2^j$. (The error word may have several successive nonzero positions, but we call the error a single error. A later section has precise definitions of arithmetic weights and single and double errors.) A *complex fault* in the same stage of an adder could be a cause for both the carry and sum to be in error. In the latter case, the error value $E \equiv \pm 2^j \pm 2^{j-1}$ (E here may be a single error or double error, depending on its error magnitude $|E|$). An important aspect to be noted here is that a fault becomes simple or complex depending on the role of the logic element that contains the fault or the failed component. Another point of interest is the need for preventing complex faults by suitable logic design. This objective can often conflict with the natural, time-old objective of the

logic design, which is to realize logic circuits at minimal cost. An error control objective is to limit logic faults to simple and single-use categories, as far as possible, by design and by use of algorithms. This constraint is likely to limit the size of fanout of a logic element and increase the cost of hardware of the processor. This price, however, may be reasonable in view of the overall costs of alternative approaches such as the triple modular redundancy (TMR) or duplication and switching methods, which are described in Section 2.3.

Errors and arithmetic weight

Component failures or malfunctions produce logic faults. The logic faults generate erroneous results, or errors. Errors are thus the deviations in the outputs (numerical or logical) of the arithmetic processor. An error is said to occur in an operation Φ of AP whenever the actual output $\mathbf{R}' = (r'_{n-1}, r'_{n-2}, \ldots, r'_0)$ differs from the expected value $\mathbf{R} = (r_{n-1}, r_{n-2}, \ldots, r_0)$ specified by the designer. Therefore, the error word $\mathbf{E}$ (also called damage pattern or error pattern [1]) is

$$\mathbf{E} = (e_{n-1}, e_{n-2}, \ldots, e_0) = \mathbf{R}' - \mathbf{R} \tag{2.1}$$

where $e_i = r'_i - r_i$ for $i = 0, 1, 2, \ldots, n-1$. If we consider only binary outputs, r'_i and r_i can only be 0 or 1, and consequently e_i can be 0, 1, or -1.

EXAMPLE

Actual output

$$\mathbf{R}' = (110001), \qquad R' = \delta_I(\mathbf{R}')$$

specified output

$$\mathbf{R} = (101101), \qquad R = \delta_I(\mathbf{R})$$

error word

$$\mathbf{E} = (01\bar{1}\bar{1}00)$$

where $\bar{1}$ denotes -1 in the error word.

For each error word **E**, we define an error value E (or hereafter error) given by

$$E = \delta_I(\mathbf{E}) = \sum_{i=0}^{n-1} e_i 2^i \tag{2.2}$$

and

$$E = R' - R \tag{2.3}$$

For the example above, the error $E = \delta_I(\mathbf{E}) = \delta_I(01\bar{1}\bar{1}00) = 16 - 8 - 4 = 4$. This mapping δ_I of error words into errors is a conversion of the n-tuples into integer values, and is not a one-to-one correspondence. As examples, the error words $(01\bar{1}\bar{1}00)$, $(001\bar{1}00)$, and (000100) all correspond to the same error $E = 4$. The one most significant property of an error is its *arithmetic weight*, which is defined as follows.

The arithmetic weight† of an integer N, denoted $W(N)$ (in radix r), is defined as the *minimum* number of terms in an expression of the form

$$N = a_1 r^{j_1} + a_2 r^{j_2} + \cdots \tag{2.4}$$

where $a_i \neq 0$, $|a_i| < r$ (see Peterson [2, Chapter 15]).

Since binary arithmetic processors and binary codes are of special interest, we may restate the definition above for binary systems. The binary arithmetic weight of N is the *minimum* number of terms in an expression of the form

$$N = a_1 2^{j_1} + a_2 2^{j_2} + \cdots \tag{2.5}$$

where $a_i = 1$ or -1. For instance, the decimal number 31 has a binary representation 11111, but that is certainly not in a minimal form. Its minimal form is $10000\bar{1}$ ($\bar{1}$ denotes -1). Hence $W(31) = 2$, the number of terms in a minimal form. There may be more than one minimal form for a number N. For instance, $25 = 011001$ and $25 = 10\bar{1}001$. While the binary arithmetic weight of 31 is 2, its ternary arithmetic weight is 3, since $31 = 3^3 + 3 + 1$ is a minimal form. Thus the arithmetic weight of a number depends very much on the radix notation used. In order to simplify our notation, we observe the following. *Unless specifically stated otherwise, we will assume an arithmetic weight to refer to the binary arithmetic weight.*

† The arithmetic weight is also referred to as Peterson weight in some instances.

An important property of the arithmetic weight is that

$$W(N) = W(-N) \tag{2.6}$$

This can be very trivially observed from the expression (2.4) where each a_i can be negative or positive. If $N = \sum_{i=1}^{k} a_i r^{j_i}$, then $-N = \sum_{i=1}^{k} (-a_i) r^{j_i}$ and both N and $-N$ have exactly the same number of terms in their minimal forms. Another important property of the arithmetic weight is the triangular inequality given as

$$W(N_1 + N_2) \leq W(N_1) + W(N_2) \tag{2.7}$$

This is clear if we consider addition of the numbers N_1 and N_2 which are initially in their minimal forms. Cancellation of nonzero terms or carries may occur, but the number of nonzero terms of the sum cannot possibly exceed the sum of the number of nonzero terms of N_1 and N_2. As said before, the minimal form (or minimal weight form) for N is not unique. As another example, a decimal number $19 = 2^4 + 2^2 - 1 = 2^4 + 2 + 1$. Reitwiesner [3] has shown that if an integer is given in such a form that the coefficients $a_i a_{i+1} = 0$ for $i = 0, 1, \ldots, n-2$ in the expression

$$N = \sum_{i=0}^{n-1} a_i r^i, \qquad a_i = 0, \quad 1, \quad \text{or} \quad -1$$

then it is called the *nonadjacent form* (NAF) and it is also a minimal weight form. In Section 6.1 the interested reader can find further discussion on NAF and algorithms for conversion to these forms. An algorithm to determine by inspection the arithmetic weight of an integer expressed in binary form is available in the work of Garcia [4].

DEFINITION 2.1

Given integers N_1 and N_2, the arithmetic distance between N_1 and N_2, denoted $D(N_1, N_2)$, is given by $W(N_1 - N_2) = W(N_2 - N_1)$.

Let the specified (or correct) result and the actual (or possibly erroneous) result of an operation be N_1 and N_2, respectively. If the

arithmetic distance between N_1 and N_2 is d, then a d-fold arithmetic error (or an error of weight d) is said to have occurred. If E in (2.3) is such that $W(E) = d$, then a d-fold arithmetic error E is said to have occurred. When two numbers are added, failure in one of the flip-flops of the accumulator or in one of the carry stages of the parallel adder may affect several consecutive bit positions in the sum due to carry propagation; the arithmetic weight and distance are explicitly defined to treat such errors as single errors.

Errors in finite ring arithmetic

Due to the finite size of the arithmetic registers, adders, and so on, the operands N_1 and N_2 are limited to a finite range of values. Let Z_m denote the finite ring of integers modulo m, namely $\{0, 1, \ldots, m-1\}$.† The additive inverse of N in Z_m is often called the complement of N, and is denoted by $\bar{N} = m - N$ and

$$\bar{N} \equiv -N \pmod{m}$$

The arithmetic weight as defined earlier satisfies $W(N) = W(-N)$. However, for N and $\bar{N}$ in Z_m, $W(N)$ may not be equal to $W(\bar{N})$. As an example, consider Z_{63}, where $\overline{32} = 63 - 32 = 31$. We have

$$W(32) = 1, \qquad W(\overline{32}) = W(31) = 2$$

This is an undesirable feature, since error magnitude does not uniquely specify its arithmetic weight. A shorted inverter in the jth stage of an adder may generate errors of $+2^j$ and -2^j at different times, and therefore cause errors of different weight. To alleviate this difficulty, Rao and Garcia [5] have introduced modular arithmetic weights for numbers in a finite ring.

† Since we are concerned only with the addition operation in $Z_m = \{0, 1, \ldots, m-1\}$, we need only to say that Z_m is an additive Abelian group. Algebraic ring properties will be of interest in consideration of both addition and multiplication operations.

DEFINITION 2.2

The modular weight of an integer $N \in Z_m$, denoted by $W_m(N)$, is given by

$$W_m(N) = \min(W(N), W(\bar{N}))$$

In the binary representation for $31 \in Z_{63}$, we have the following:

$$W(31) = 2$$
$$\overline{31} = 63 - 31 = 32, \qquad W(32) = 1$$

Therefore, $W_m(31) = W_m(32) = 1$.

Similarly, for $N_1, N_2 \in Z_m$, the *modular distance* between N_1 and N_2, denoted as $D_m(N_1, N_2)$, is given by $W_m(N_1 - N_2)$. Also

$$D_m(N_1, N_2) = W_m(N_1 - N_2) = W_m(N_2 - N_1)$$

In Z_m, it is not true in general that

$$W_m(|N_1 + N_2|_m) \leq W_m(N_1) + W_m(N_2) \tag{2.8}$$

The triangular inequality (2.8) is an essential property, as we shall see later, in establishing the correspondence between the "minimum distance" of a code and its error detection and correction capabilities.

EXAMPLE

Let $m = 51$, $N_1 = N_2 = 32 \in Z_{51}$. We have then

$$W_m(32 + 32) = W_m(|64|_{51}) = W_m(13)$$
$$= \min(W(13), W(51 - 13)) = 3$$

Also $W_m(32) = 1$. Therefore

$$W_m(|32 + 32|_m) > W_m(32) + W_m(32)$$

which contradicts (2.8). However, we have the following important theorem.

THEOREM 2.1

When $m = r^n$ (as in radix complement systems) or when $m = r^n - 1$ (as in diminished radix complement systems) the triangular inequality (2.8) holds in Z_m.

Proof

Let N_i' denote N_i or $-\bar{N}_i = N_i - m$, for $i = 1, 2$ correspondingly, whichever has the smallest arithmetic weight. This means that

$$N_1' + N_2' \equiv N_1 + N_2 \pmod{m}, \qquad |N_i| < m$$

and

$$W(N_i') = W_m(N_i) \qquad \text{for} \quad i = 1, 2 \tag{2.9}$$

Let the minimal form expansions of N_1' and N_2' be

$$\begin{aligned} N_1' &= \sum_{i=0}^{n-1} a_i r^i, \qquad |a_i| < r \\ N_2' &= \sum_{i=0}^{n-1} b_i r^i, \qquad |b_i| < r \end{aligned} \tag{2.10}$$

Let S represent the sum of N_1' and N_2' modulo m such that $|S| < m$. In other words,

$$S = \sum_{i=0}^{n-1} c_i r^i \equiv \sum_{i=0}^{n-1} (a_i + b_i) r^i \pmod{m} \qquad |c_i| < r \tag{2.11}$$

Thus the sum modulo m of N_1' and N_2' can be obtained by the propagation of carries or borrows as required. A carry or borrow from the most significant position will be of magnitude r^n, which equals m (for the radix complement case) or $m + 1$ (for the diminished radix complement case), and therefore can be discarded or propagated as an end-around-carry or end-around-borrow. Consequently, the number of nonzero c_i's in the expression for S can be no greater than the total number of nonzero terms in (2.10). Therefore the number of nonzero terms in S represents the modular weight of the sum $N_1' + N_2'$ and hence also equals the modular weight of $|N_1 + N_2|_M$. Therefore the inequality (2.8) holds. Q.E.D.

Massey and Garcia [6] show that the triangle inequality (2.8) holds for $m = 2^n \pm 1$. By arguments similar to those in the proof of Theorem 2.1, one can show that (2.8) holds also for $m = 2^n - 2^j \pm 2^i$ $(n-1 > j > i)$, i.e., for two end-around-carries or borrows. Therefore we formally state the following

COROLLARY 2.2

When $m = 2^n - 2^j \pm 2^i$ for $n-1 > j > i$, the triangle inequality (2.8) holds in Z_m.

A note on distance and metric

It is important to note that the arithmetic distance and the modular distance between integers are analogous to the well-known *Hamming distance* between vectors or codewords in algebraic linear codes (see, for instance [2, Chapter 1]). In the algebraic linear codes, the minimum Hamming distance of a code relates to its error correcting properties. In Chapter 4, we derive a similar relationship of *minimum arithmetic distance* of an *AN code* to its error control properties.

In the use and understanding of the word *distance* some caution is needed. Distance normally implies a real quantity satisfying the properties of a *metric* which are as follows:

$D(x, y) \geq 0$ equality holds iff $x = y$	(positive definite)
$D(x, y) = D(y, x)$	(symmetry)
$D(x, y) + D(y, z) \geq D(x, z)$	(triangular inequality)

While Hamming and arithmetic distances are invariably metrics, the modular distance is a metric only when the modulus m satisfies the carry properties as required by Theorem 2.1 or Corollory 2.2. The carry properties for a given m depend very much on the radix r of the number system. Thus in the concept of modular distance, the modulus m, the radix r, and the number of end-around caries are involved.

For the purpose of this book, we assume that our interest is mainly in such m and r that do not violate the metric properties. That means, the finite rings Z_m we are concerned with are assumed to be metric spaces. (Exceptions to this rule will be appropriately stated in the text.) In this context, we use the term *modular distance*.

Error sets

We consider R, R', and E in (2.3) to be integers from Z_m, and proceed to define the classes of errors in Z_m. We denote the set of all errors in Z_m of modular weight equal to d by $V(m, d)$, and the set of all errors of weight less than or equal to d by $U(m, d)$. The set of all errors of modular weight 1 is called the set of single errors; modular weight 2, the set of double errors, and so on. Also, $U(m, d)$ equals the union of the sets $V(m, 1)$, $V(m, 2)$, ..., $V(m, d)$.

EXAMPLES

$V(16, 1) = U(16, 1) = \{1, 2, 4, 8, 12, 14, 15\}$

$V(31, 1) = U(31, 1) = \{1, 2, 4, 8, 16, 15, 23, 27, 29, 30\}$

$V(31, 2) = \{3, 5, 6, 7, 9, 10, 11, 12, 13, 14, 17, 18, 19, 20, 21, 22, 24, 25, 26, 28\}$

$U(31, 2) = \{V(31, 1), V(31, 2)\}$ = all nonzero elements of Z_{31}

2.3 ERROR CONTROL TECHNIQUES

Logic faults in processors are caused by a number of possible events. Component failures and malfunctions are the most common; overheating, electromagnetic radiation, noise interference, and mechanical shocks are some of the other causes. Logic faults generate errors in the results. These errors have been characterized and classified in the preceding section according to their modular weight.

We discuss here briefly some of the techniques that are employed to detect and correct these errors. These are also referred to as *fault-tolerance* techniques [6] or error control techniques. The objective of these techniques is to obtain a processor that is capable of performing accurately despite the occurrence of logic faults, and thereby increase the reliability, availability, or dependability of the system.

Reliable performance of digital processors is attainable by the systematic application of two techniques.

The first technique involves selection of long-life, highly reliable components; provision of liberal margins between component ratings and the actual operating conditions; and application of proven methods for the interconnection and packaging of these component parts.

The second technique is termed *protective redundancy*, and as the name suggests, it requires the use of redundant equipment (hardware and/or software). The redundant equipment is so designed to detect and/or correct, "bypass," or "mask" the effects of the logic faults. The protective redundancy techniques may be further divided into two classes: *massive redundancy* and *selective redundancy*.

In the massive redundancy approach, the logic faults of a module, such as a component, network, or processor, are masked by permanently connected and parallel operating (fault-free) replicas of the faulty module. As an example, a triplicated processor with the outputs applied to majority vote-takers would mask any output errors due to a faulty processor (see Figure 2.3). This approach is commonly known as triple modular redundancy (TMR) [7, 8], and is discussed in the next section. Other massive redundancy techniques are *quadded logic* as discussed by Tryon [9], *recursive nets* [10], and *adoptive logic elements* [11], among others.

In the category of selective redundancy we include all those that require only a limited (or fractional) increase in hardware or software. Error detection followed by fault diagnosis, and error detection and correction through coding come under this category. Errors are detected by the provision of error-detecting codes, or special monitoring circuits, or by a periodic check through *fault-recognition* or *diagnostic* programs. Use of hardware circuits for error detection has the advantage of

instantaneous or *concurrent diagnosis* which may prevent system deterioration or propagation of errors by calling for an appropriate corrective action. *Periodic diagnosis* requires an elaborate corrective action, a "rollback" of the program, and interruption of the system operation for considerable periods of time. When faults are detected by the fault detection circuits, or programs, a corrective action which eliminates the errors is followed. This corrective action will be in the nature of one of the following:

1. error correction by use of error-correcting codes and associated special-purpose error-decoding hardware and/or software;
2. diagnosis to determine the faulty element or module and its replacement by a standby spare [12, 13];
3. reorganization or reconfiguration of the system to enable the necessary computation of functions, if not in a normal mode, in a degraded mode (with loss of precision or speed). This method of operation is often called *graceful degradation*.

There are some good books [7, 14, 17] and papers available on the many different fault-tolerance techniques. This book is intended as a summary of the recent advances in the theory and application of error control codes for arithmetic processors. The arithmetic codes described here are most appropriate for arithmetic operations such as ADD, SHIFT, COMPLEMENT, MULTIPLY, etc. They can also be useful in the control of errors in transmission or in communication channels. The arithmetic codes have a number of properties that parallel and are analogous to those of such communication codes as Hamming codes [15] and cyclic codes [2], but they have many distinct properties of their own, and differ significantly in their algebraic structures.

Triple modular redundancy (TMR)

A TMR system is constructed from a nonredundant system S_0, replacing each logic block in S_0 by three identical blocks and combining their outputs in one or more majority vote-takers. (See Figure 2.5.)

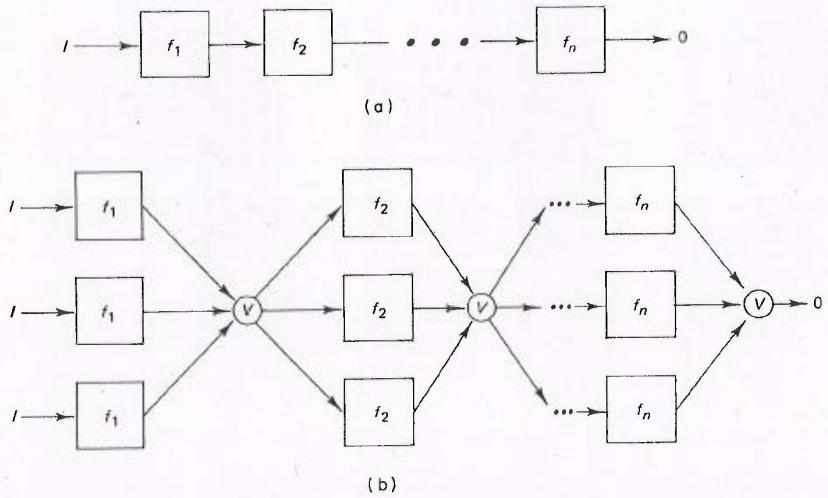


Figure 2.5 (a) Nonredundant system S_0. (b) TMR version of (a).

Figure 2.5a represents a series-connected system with n functional units $f_1, f_2, \ldots, f_n$. Let us assume that the reliability of the unit f_i (for $i = 1, 2, \ldots, n$), which is the probability that the unit f_i is functioning at a given time t, is P_i. The nonredundant system reliability R_0 is given by

$$R_0 = \prod_{i=1}^{n} P_i \tag{2.12}$$

In the redundant system, if we assume the voters to be perfect (i.e., their reliability to be 1), then the reliability of the ith voted triplet is

$$P_i^3 + 3P_i^2(1 - P_i) = 3P_i^2 - 2P_i^3$$

The reliability of the redundant system of Figure 2.5b is

$$R = \prod_{i=1}^{n} (3P_i^2 - 2P_i^3) \tag{2.13}$$

If we assume each functional unit to be of the same reliability, i.e., $P_i = P$ for all i, then

$$R_0 = P^n, \qquad R = (3P^2 - 2P^3)^n$$

If the reliability of the redundant system is to be better than that of the nonredundant system, then

$$R > R_0$$

or

$$(3P^2 - 2P^3)^n > P^n$$

or

$$3P^2 - 2P^3 > P$$

Since P is positive, it means that

$$3P - 2P^2 > 1$$

or

$$(2P^2 - 3P + 1) < 0, \quad \text{i.e.,} \quad (2P - 1)(P - 1) < 0$$

or

$$P > 0.5$$

As long as the reliability of each functional unit is better than 0.5, we could expect improvement in reliability by TMR. This statement is made, of course, under the unrealistic assumption that the voters are perfect. With imperfect voters, where each voter V has a reliability q, system reliability R^* is given by

$$R^* = q^n(3P^2 - 2P^3)^n \qquad (2.14)\dagger$$

Reliability improvement results if

$$q^n(3P^2 - 2P^3)^n > P^n$$

or

$$q(3P^2 - 2P^3) > P$$

or

$$q > P\left(\frac{1}{3P^2 - 2P^3}\right), \quad \text{i.e.,} \quad q > \frac{1}{3P - 2P^2}$$

† This formula as well as others in this section is only approximate and does not take into account the cancellation of errors due to failures in successive units. Therefore the actual reliability is slightly better than that given by the formula.

Since $3P^2 - 2P^3$ represents the reliability of a triplet, we have $0 < (3P^2 - 2P^3) < 1$. This leads us to conclude that q must be sufficiently greater than P to obtain reliability improvement. The voters that are imperfect contribute to the unreliability of the system by a factor q^n. If this factor is to be improved, a TMR version with triplicated voters is adopted. (See Figure 2.5c.)

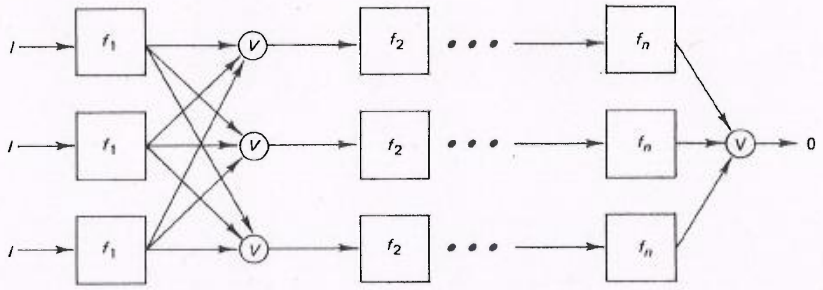


Figure 2.5c

The reliability R^{**} of the TMR system of Figure 2.5c can be obtained as

$$R^{**} = (3P^2 - 2P^3)(3P^2q^2 - 2P^3q^3)^{n-1}q \qquad (2.15)$$

Equations (2.13)–(2.15) can be compared to obtain the conditions as to how P and q must be related to obtain the necessary improvements in reliability. These have been worked out in great detail with charts and figures in the literature [11, 16].

Duplication and switching

Error detection is often a first step in error control coding. Duplication of the essential functional units and matching (or comparing) outputs provide not only the needed error detection, but could also provide the means of establishing an operational system in the presence of logic faults. In case of a mismatch of the outputs from the duplicated

units, a corrective action is initiated. The corrective action may be in the nature of a program interrupt, followed by a "system recovery" or a "reconfiguration process" and a diagnosis of the faulty unit. A reconfiguration process involves the location of the faulty unit or network and its replacement by means of "switching" with a standby spare. If a standby spare is not provided, then the reconfigured mode of operation will not provide for matching of outputs, and therefore will be unprotected from future errors. Such a mode of operation is termed *graceful degradation*. Graceful degradation sometimes means that the operational mode is now somewhat degraded in terms of speed, accuracy, or precision.

In summary, the corrective procedures that follow the error detection and eliminate the effects of the fault may be grouped as follows:

1. correction by the replacement of the faulty unit or network by a standby spare, or an active spare [13];
2. system recovery through reconfiguration and switching. The recovered system may operate in a degraded mode until such time as the required maintenance on the failed unit is carried out and it is returned to the system;
3. correction by the application of error-correcting codes such as Hamming [15] or residue [2, 5] codes.

Error-correcting codes have been found to be useful in the detection and correction of errors. These codes can be divided into two major categories:

1. parity-based codes or communication codes;
2. residue codes or arithmetic codes.

The communication codes [2, 15] are by far better known and well developed relative to the arithmetic codes. They are well suited for control of errors during transmission of data over a communication channel or for transmission of data from one section of a computer to another, but not for arithmetic operations. The arithmetic codes,

on the other hand, are ideally suited for arithmetic operations and arithmetic processors and can also be used to control errors during data transmission. In the next chapter we introduce fundamentals of arithmetic codes.

PROBLEMS

1. Obtain the arithmetic weight of the following numbers in radix (a) $r = 2$, (b) $r = 3$, (c) $r = 10$ systems:
   (i) 29, (ii) 199, (iii) 100.
2. Given $m = 61$, obtain the modular weight ($r = 2$) of numbers 29, 13, and 53.
3. Obtain the error sets $V(61, 1)$, $V(63, 2)$, and $U(63, 2)$. (Use $r = 2$.)
4. Show that the triangular inequality given by (2.8) holds for $m = 2^n - 2^k - 2^j$ for positive integers j, k, n such that $j < k < n - 1$.
5. Given $P_i = P = 0.8$, for $i = 1, 2, \ldots, 5$, $n = 5$, and voter reliability $q = 0.9$, calculate the system reliability for the three cases given by Figures 2.5a–c.
6. Given $n = 2$ for Figures 2.5b and c, derive a condition that makes $R^{**} < R^{*}$.
7. A metric function d is a real-valued function satisfying
   (i) $d(x, y) \geq 0$, $d(x, y) = 0$ iff $x = y$ (positive definite),
   (ii) $d(x, y) = d(y, x)$ (symmetric),
   (iii) $d(x, y) \leq d(x, z) + d(z, y)$ (triangle inequality).
   Show that the arithmetic distance (D) and the modular distance (D_m) are metrics. Assume $m = 2^n - 2^j \pm 1$ $(n - 1 > j > 1)$.

REFERENCES

1. A. Avizienis, Arithmetic Error Codes: Cost and Effectiveness Studies for Application in Digital Systems. *IEEE Trans. Comput.* C-20, 1322–1330 (November 1971).

2. W. W. Peterson and E. J. Weldon Jr., "Error-Correcting Codes," 2d Ed., MIT Press, Cambridge, Massachusetts, 1972.
3. G. H. Reitwiesner, Binary Arithmetic, *Advances in Comput.* 1, 232–308 (1960).
4. O. N. Garcia, Error Codes for Arithmetic and Logical Operations. Ph.D. Thesis, Dept. of Elec. Engrg., Univ. of Maryland, College Park, 1969.
5. T. R. N. Rao and O. N. Garcia, Cyclic and Multi-residue Codes for Arithmetic Operations, *IEEE Trans. Information Theory* IT-17, 85–91 (January 1971).
6. J. L. Massey and O. N. Garcia, Error correcting codes in computer arithmetic, Chapter 5, in "Advances in Information System Sciences" (J. L. Tow, ed.), Vol. 4, pp. 273–326. Plenum Press, New York, 1971.
7. W. H. Pierce, "Failure-Tolerant Computer Design." Academic Press, New York, 1965.
8. J. Von Neumann, "Probabilistic Logics and the Synthesis of Reliable Organisms from Unreliable Components" (Automata Studies, Ann. Math. Studies, No. 34). Princeton Univ. Press, Princeton, New Jersey, 1956.
9. J. G. Tryon, Quadded Logic, *in* "Redundancy Techniques for Computing Systems" (R. H. Wilcox and W. C. Mann, eds.). Spartan Books, Washington, D.C., 1962.
10. S. Levy, Reliability of Recursive Triangular Switching Networks Built of Rectifier Gate, *in* "Redundancy Techniques for Computing Systems" (R. H. Wilcox and W. C. Mann, eds.). Spartan Books, Washington, D.C., 1962.
11. W. H. Pierce, Adaptive Vote-taxers Improve the Use of Redundancy, *in* "Redundancy Techniques for Computing Systems" (R. H. Wilcox and W. C. Mann, eds.). Spartan Books, Washington, D.C., 1962.
12. A. Avizienis, Design of Fault-Tolerant Computer, *Proc. Fall Joint Comput. Conf., 1967*, pp. 733–743.
13. R. W. Downing, *et al.*, No. 1 ESS Maintenance Plan, *Bell System Tech. J.* pp. 1961–2019 (September 1964).
14. "Redundancy Techniques for Computing Systems" (R. H. Wilcox and W. C. Mann, eds.). Spartan Books, Washington, D.C., 1962.
15. R. W. Hamming, Error Detecting and Error Correcting Codes, *Bell System Tech. J.* 29, 147–160 (April 1960).
16. D. K. Rubin, The Approximate Reliability of Triply Redundant Majority-voted Systems, *Proc. of First Annu. IEEE Comput. Conf., Chicago, Illinois, September, 1967.*
17. F. F. Sellers, Jr., M. Y. Hsiao, and L. W. Bearnson, "Error Detecting Logic for Digital Computers." McGraw-Hill, New York, 1968.