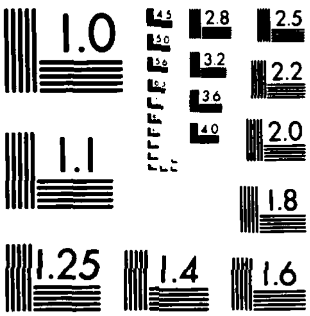
AD-A128 608   STATE-OF-THE-ART ASSESSMENT OF TESTING AND TESTABILITY   1/1
OF CUSTOM LSI/VLSI..(U) AEROSPACE CORP EL SEGUNDO CA
ENGINEERING GROUP M A BREUER ET AL. OCT 82
UNCLASSIFIED TR-0083(3902-04)-1-VOL-6 SD-TR-83-20-VOL-6 F/G 12/1   NL

END
DATE
FILMED
6-83
DTIC



MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

REPORT SD-TR-83-20

12

State-of-the-Art Assessment of Testing and Testability of Custom LSI/VLSI Circuits

Volume VI: Redundancy, Testing Circuits, and Codes

M. A. BREUER & ASSOCIATES
Encino, Calif. 91436

and

A. J. CARLAN
Technical Study Director

October 1982

Engineering Group
THE AEROSPACE CORPORATION
El Segundo, Calif. 90245

Prepared for

SPACE DIVISION
AIR FORCE SYSTEMS COMMAND
Los Angeles Air Force Station
P.O. Box 92960, Worldway Postal Center
Los Angeles, Calif. 90009

APPROVED FOR PUBLIC RELEASE;
DISTRIBUTION UNLIMITED

This final report was submitted by the Aerospace Corporation, El Segundo, CA 90245 under Contract No. F04701-82-C-0083 with the Space Division, Deputy for Logistics and Acquisitions, P.O. Box 92960, Worldway Postal Center, Los Angeles, CA 90009. It was reviewed and approved for The Aerospace Corporation by J. R. Coge, Electronics and Optics Division, Engineering Group. Al Carlan was the project engineer.

This report has been reviewed by the Office of Information and is releasable to the National Technical Information Service (NTIS). At NTIS, it will be available to the general public, including foreign nationals.

This technical report has been reviewed and is approved for publication. Publication of this report does not constit Air Force approval of the report's findings or conclusions. It is published only for the exchange and stimulation of ideas.

FOR THE COMMANDER

APPROVED

STEPHEN A. HUNTER, LT COL, USAF
Director, Speciality Engineering
and Test

Unclassified

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS BEFORE COMPLETING FORM

1. REPORT NUMBER
SD-TR-83-20

2. GOVT ACCESSION NO.
AD-A128608

3. RECIPIENT'S CATALOG NUMBER

4. TITLE (and Subtitle)
State-of-the-Art Assessment of Testing and Testability of Custom LSI/VLSI Circuits
Vol VI: Redundancy, Testing Circuits and Codes

5. TYPE OF REPORT & PERIOD COVERED
Interim

6. PERFORMING ORG. REPORT NUMBER
TR-0083(3902-04)-1

7. AUTHOR(s)
M.A. Breuer & Associates
and
A.J. Carlan, Aerospace Technical Director

8. CONTRACT OR GRANT NUMBER(s)
F04701-80-C-0081
F04701-81-C-0082
F04701-82-C-0083

9. PERFORMING ORGANIZATION NAME AND ADDRESS
M.A. Breuer & Associates
16857 Bosque Dr.
Encino, CA 91436

10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS

11. CONTROLLING OFFICE NAME AND ADDRESS
Space Division
Air Force Systems Command
Los Angeles, Calif. 90245

12. REPORT DATE
October 1982

13. NUMBER OF PAGES
91

14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office)
The Aerospace Corporation
El Segundo, Calif. 90245

15. SECURITY CLASS. (of this report)
Unclassified

15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

Redundancy constructs
Simplex structure
Self checking
Fault-tolerant architecture
Reliability
Exponential failure law
Quadding
Hybrid redundancy
Triple modular redundancy
STAR computer
SIFT computer
PRIME computer
Hamming code

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

The demands for higher system reliability and self checking required by the new fault tolerant computers have put new emphasis on the use of redundant circuits. Types of redundancy include parellel, triple modular redundancy, Quadd, standby, hybrid and software. Various computers employing one or more of these types are discussed. Generally, hardware, software and time redundancy required for error detection and correction, are interrelated. Mathematical modeling, when applied to fault tolerant systems, can be used to measure the systems reliability.

Unclassified

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

TABLE OF CONTENTS

Page

EXECUTIVE SUMMARY

PART I REDUNDANCY AND FAULT TOLERANT COMPUTER ARCHITECTURE

Part I of this chapter deals with redundancy and its framework. The framework of redundancy consists of (i) modeling and evaluation of the redundancy constructs, and (ii) the embodying of the constructs in fault-tolerant computer architecture.

Mathematical modeling of redundancy constructs permits their quantitative evaluation and provides a numeric basis for critical comparison.

Case histories of fault-tolerant computer architecture illustrate, by the design selection of particular redundancy constructs from the repertoire of constructs, the relative significance that the designer placed on specific redundancy constructs in relation to their functional environment in the architecture.

In general, a system if designed in such a manner that only the absolute minimum amounts of hardware is utilized to implement its function is said to be *non-redundant* or is said to have a *simplex structure*. If even after utilizing the finest components available the desired system reliability is not achieved or if failure-tolerance is desired as a system capability then *redundancy* as a design procedure is restored too, i.e., more system elements are used than were absolutely necessary to realize all the system's functions (excepting for the attributes of reliability and fault-tolerance). The additional system elements, referred to as the redundant elements, need not all necessarily be hardware elements but may also be additional software (software redundancy), additional time (time redundancy) and additional information (information redundancy). Examples of the latter are the application of error-detection and correction codes.

Naturally, the hardware, software, and time redundancy are often interrelated. Additional software requires additional memory storage and additional time is used to execute the added software. The term *protective redundancy* is often used to characterize that redundancy which has an overall beneficial effect on the system attributes since redundancy alone without proper application may well become a liability. Protective redundancy is utilized to realize *fault-tolerant digital systems and self-repairing systems* by such means as triple or N-tuple modular redundancy (TMR, NMR), quadded redundancy, standby-replacement redundancy, hybrid redundancy, software redundancy and the application of error-detection and correction codes.

Redundancy as a procedure, for designing more reliable system than allowed by the intrinsic reliability of the constituting components, is as old as the discipline of engineering itself. Examples of the use of redundancy in ancient times is provided in the civil engineering construction where more than the absolutely minimum redundancy were used as insurance against (i) the lack of accurate knowledge of underlying phenomena, and (ii) the lack of confidence in the available data on materials used. Redundancy as a procedure is even more basic. This is evidenced by the testimony of evolutionaly processes of life which make abundant use of it (e.g., in the human body there are two kidneys, two lungs, two cerebral hemispheres, etc.).

For the computer age, redundancy has been used at all levels of technology, from that of VLSI devices, circuitry, logic, subsystems, computers, and even to entire networks of digital systems.

Part I of this chapter spans the general area of fault-tolerant systems. The utilization of the various protective redundant structures as basic building blocks for fault-tolerant digital computing systems have been described and evaluated comparatively. A unifying notation for characterizing the most commonly used protective redundancy schemes has been presented. It has also been demonstrated that the k-out-of-N redundant model subsumes either directly or by composition a great number of other redundant structures.

By employing reliability analysis to these fault-tolerant systems, their overall reliability can be measured and compared.

PART II SELF-CHECKING CIRCUITS

Self-checking circuits by definition pertain to circuits whose outputs are encoded in an error-detecting code. In Part II of this chapter the underlying theory based on code spaces is developed to present the notions of self-checking circuits, partially self-checking circuits, totally self-checking circuits, and totally self-checking networks. An introduction to Morphic Boolean logic is also presented which is an aid to the design of self-checking checkers. These are presented with examples and illustrations.

PART III CODING TECHNIQUES

Part III of this chapter deals with coding techniques used to achieve concurrent diagnosis in digital computing systems. Coding theory is the body of knowledge dealing with the science of redundantly encoding data so that errors can be detected and with further encoding even corrected.

The fundamental principles underlying transmisssion codes as well as arithmetic codes are developed and illustrated by the use of short simple examples. Both error detection as well as error correction properties are treated and the tradeoffs between these are explained.

The use of residue codes for protecting instruction words in the JPL-STAR computer is given as a real example.

Coding theory is a very rich and by far the most developed branch of fault-tolerant computing. The theoretical basis, the functional limits of reliable communication for a given channel, and the mathematical tools and classification schemes are well established. This section does not attempt to be an exhaustive evaluation, the emphasis taken is to highlight the essential principles by means of short examples. For the more interested practitioner pointers are provided to the literature.

PART I REDUNDANCY AND FAULT TOLERANT COMPUTER ARCHITECTURE

1.0 INTRODUCTION

A fault-tolerant computer is a computer organized and structured such that it can perform its design specified functions even in the presence of hardware failures.

By the sheer force of necessity an important attribute in computer architecture is reliability and fault-tolerance. Historically, the early days of computers, because of the unreliability of thermonic devices, were extremely innovative and productive of fault-tolerance techniques, many of which we take for granted these days, e.g., parity checking, retrying of operations, duplexing of processors, etc. Then, with the advent of semiconductors and their greater inherent reliability fault-tolerance was no longer a pressing issue, and undiverted effort was allocated to the enhancement of computational architectures. Subsequently, the space age and computer-oriented national defense needs again shifted the equilibrium between intrinsic component reliabilities on the one hand and the sheer size and complexity and hazardous application environments of the fabricated structures on the other. These demands on reliability, continuous service, and hardware integrity spawned a new breed of computer architecture entitled "fault-tolerant computers."

This chapter surveys the various techniques employed in fault-tolerant architectures from the point of view of the protective redundancy structures utilized, and points out the significant features unique to the implementation of fault-tolerance in either hardware, microprogramming, or software.

The domain of reliability engineering involves considerations of all aspects of design, development and fabrication, so as to minimize the chance of equipment breakdown. Neglect of reliability considerations can prove to be very costly, from the loss of consumer accptance of the product to missions such as rocket launching of spacecrafts which depend heavily on reliability engineering. Failure of a single component could result in the total loss of the system.

Reliability in a qualitative sense can mean a host of different things relating to the confidence in the goodness of the equipment, and is closely connected, but often confused with the concepts of maintainability, availability, safety and even security of the system. Quantitatively reliability can be formulated mathematically as the probability that the system will perform its intended function over the stated duration of time in the specified environment for its usage.

As equipment becomes more complex the chances of system unreliability becomes greater, since the reliability of an equipment depends on the reliability of its components. The relationship between parts reliability and the system reliability can be formulated mathematically to varying degrees of precision depending on the scale of the modeling effort. The mathematics of reliability is based on parts failure rate statistics and probability theoretic relationships. The mathematical theory of reliability is used to model, simulate and predict the equipment's proneness to failure under expected operating conditions.

There have been two distinct and viable approaches taken to enhance system reliability. One is based on component technology, i.e., manufacturing the component as intrinsicly reliable as possible followed by parts screening, quality control, pretesting to remove early failures (infant mortality effects), etc. The second approach is based on the organization of the system itself, e.g., fault-tolerant architectures where the architecture makes use of protective redundancy to mask or remove the effects of failure, and thereby provide greater overall system reliability than would be possible by the use of the same components in a simplex or nonredundant configuration.

Fault-tolerance is the capability of the system to perform its functions to its design specifications even in the presence of hardware failures. If, in the event of faults, the system's functions may be performed but do not meet the design specifications with respect to the time required to complete the job or the storage capacity required for the job, then the system is said to be partial or quasi fault-tolerant. Since the number of possible hardware failures can be very large, in practice it is necessary to restrict fault-tolerance to prespecified classes of faults from which the system is designed to recover.

Faults may be classified as transient or permanent, deterministic or indeterminate, local or catastrophic. The first category refers to the duration of the fault, the second to its effect on the values of the system design parameters and the third to the propagation of the fault to its neighboring elements.

Fault-tolerance is provided by the application of protective redundancy — use of more resources so as to upgrade system reliabiltiy. These resources may consist of more hardware, software or more time or combination of all of these. Extra time is required to retransmit messages or to reexecute programs, extra software is required to perform diagnosis on the hardware, extra hardware is required to provide replication of units.

Hardware redundancy may be of the fault-masking or self-repair types or a hybrid of these two. In fault-masking, redundancy is of a static nature, faults are masked instantly and the operations of fault detection, location and correction are indistinguishable. In self-repair, redundancy is used dynamically, faults are selectively masked, and are detected, located and subsequently corrected by the replacement of the failed uni. by an unfailed replica. Examples of the former are Triple Modular Redundancy (TMR) and quadding, and of the latter standby-replacement (SR) systems and reconfigurable systems. Schemes using combinations of these two basic approaches are called hybrid or adaptive redundancy.

1.1 SOME FUNDAMENTAL PRINCIPLES

The fundamental principle of reliability is that reliability is not solely inherent to a component but is also a functior of how the component is used. Another fundamental principle of achieving reliability by means of protective redundancy is that redundancy be applied to the smallest level of complexity of the system in order to maximize gain in reliability. This is an idealized statement since, in practice, there are tradeoffs due to overhead required in utilizing redundancy techniques, e.g., providing voters in TMR systems and detection-switching requirements in standby-sparing systems. The application of mathematical thoery of reliability to model such systems provides quantitative design guidelines to make such tradeoffs and optimizations in practice.

If the above are the first and second principles of fault-tolerance, then the third principle states that a system may be made arbitrarily reliable provided that the degree or redundancy is made high, i.e., a sufficiently large number of replicas are provided. Again this principle holds only in an idealized situation; in practice, since the probability of detecting a failure and correctly switching in a spare is less than unity, this parameter, called coverage, limits the advantages postulated by the third principle.

A fourth principle concerns the problem of requiring the checking elements (those elements that are used for the diagnosis of the rest of the system and the subsequent reconfiguration of the system units) also to be checkable. This is the problem of "checking the checker." Thus, the fourth principle is formulated to state that any system utilizing protective redundancy will have major and minor "hardcores" (i.e., unprotected system elements) and that these cannot be totally eliminated from the system design, however, they may be made arbitrarily small by the judicious use of a mixture of different protective redundancy techniques.

1.2 MATHEMATICAL THEORY OF RELIABILITY

Some relationships between reliability parameters and the underlying probability theoretic relationships are as follows. If a fixed large number N_0 of identical items is being tested of which N_s is the number of items surviving after time t, N_f the number of items which failed during time t then, for all t, $N_0 = N_s+N_f$. Now, for a sufficiently large N_0, the reliability R(t) of an item is N_s/N_0. The failure rate $\lambda(t)$, which is defined to be the rate at which the population changes at time t, can be shown to be given by

$$\lambda(t) = -\frac{1}{R(t)}\frac{dR(t)}{dt} \qquad (1)$$

so that

$$R(t) = e^{-\int_0^t \lambda(\tau)d\tau} . \qquad (2)$$

The reliability function R(t) is often called the survival probability function since it measures the probability that failure of an item does not occur during the time interval [0,t].

1.2.1 Failure Rate

Statistical data on equipment failure yields a characteristic "bath tub" curve as shown in Figure 1. When the equipment is first put into service inherently weak components fail early; this stage is also called "infant mortality." Subsequently the failure rate stabilizes quickly to a relatively constant value; this period is called the useful life period. After much usage failure rate begins to increase rapidly due to deterioration and wearout.

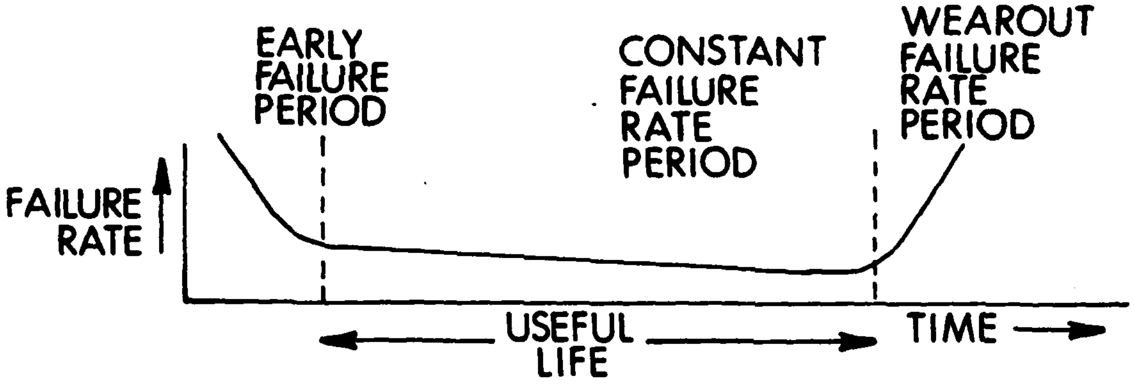


Figure 1. Bath-tub curve of failure rate.

1.2.2 Exponential Failure Law

In general the failure law of a component is the probability distribution obeyed from the moment at which a component enters service up to the moment of its failure. In practice the most commonly used failure law is the exponential law, which applies when a component is subject only to failures which occur at random intervals and the average number of failures is the same for equal time periods. These constraints are valid for a component which is no longer subject to infant mortality failures and whose failure rate is a constant within the "useful-life" span. Thus, for operating periods within the useful life, the component reliability over a period of time t can be expressed as $R(t) = e^{-\lambda t}$ where λ (usually expressed in failures per hour or per million hours) is the constant failure rate of the device. A characteristic of the exponential failure law is that, within the useful life period, the reliability of the device is the same for operating times of equal duration.

From the definition of R(t) it follows that the mean time between failures (MTBF) or the mean time to first failure (MTTF), usually expressed in hours, are given by $\int_0^\infty R(t)dt$, i.e., it is the area underneath the reliability curve R(t) plotted versus t. This result is true for any failure distribution. For the specific case of the exponential failure law the MTBF, m, is equal to $1/\lambda$. Further, when the product λt is small, the equation for R(t) may be approximated by $R(t) \approx 1 - \lambda t$. Thus, if $\lambda t = 0.01$, $R(t) = e^{-0.01} = 0.99$ or 99.0 percent. The product λt is often referred to as the "normalized" time, since $\lambda t = t/m$, i.e., the mission time t normalized with respect to the MTBF.

2.0 PRINCIPAL REDUNDANCY STRUCTURES AND THEIR MODELS

2.1 SERIES RELIABILITY

If a system is composed of elements in such a way that the failure of any one element causes a failure of the system, then these elements are considered to be functionally in series. For the system to survive each element must survive. The probability of survival for the system cannot be better than the element with the lowest probability of survival; e.g., a chain is no better than its weakest link. When these series elements are independent of each other then, by the probability multiplication law, the system survival probability is the product of the individual survival probabilities of the elements, i.e., $R_{system} = \sum_{i=1}^{n} R_i$ where R_i is the reliability of the i^{th} element of an n element system.

2.2 PARALLEL RELIABILITY

Parallel reliability is an illustration of protective redundancy. The system is composed of functionally parallel elements in such a way that if one of the elements fails the parallel unit will continue to do the system function.

The system reliability under the assumption of independence of failure of the elements is expressed by

$$R_{system} = 1 - (1-R)^n$$

which is the probability that not all the n elements have failed. The term (1-R), known as the unreliability of a unit, is the probability that a unit will fail.

2.3 TRIPLE MODULAR REDUNDANCY (TMR)

A TMR system is also known as the multiple-line voting system (see Figure 2). One of the earliest and most influential schemes was developed by J. von Neumann [1]. The simplex unit is triplicated and each of the three independent units feed into a majority voter which outputs the majority signal. The system fails if more than one unit fails in which case the failed units outvote the good one. This scheme is generalized to N-modular redundancy (NMR) where N is any odd number of units. Various schemes of protecting the voter are available and also various other variants of the basic TMR strategy have been developed. The TMR system reliability is expressed as

$$R_{system} = [R^3 + 3R^2(1-R)]R_v$$

which is the product of the reliability R_v, the voter reliability, and the reliability of the idealized TMR system. The idealized TMR system reliability is the sum of the probabilities of the two events that (i) all three units survive, R^3 and (ii) that at least any two units survive and at most one unit fails, $3R^2(1-R)$.

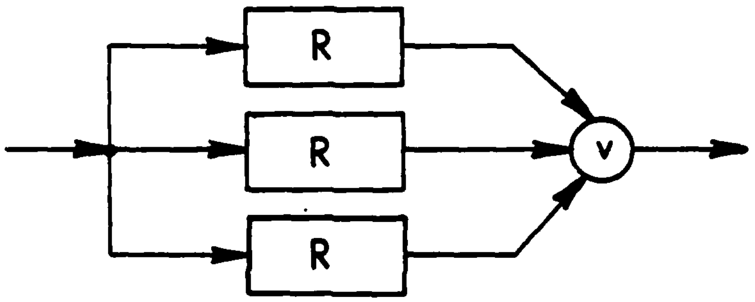


Figure 2. TMR system

2.4 QUADD REDUNDANCY

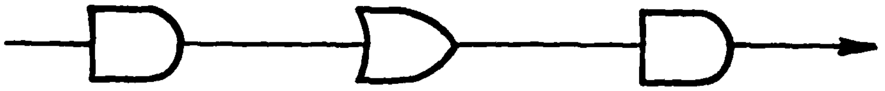
Quadding is an illustration of component redundancy and is similar in concept to TMR. The major difference is that the voting or restoration or fault-masking functions are distributed into the network and are not separable as in TMR. An example of quadding is shown in Figure 3 where the non-redundant logic circuit in Figure 3a is shown "quadded" in Figure 3b. The process of how an error downstream is subsequently corrected upstream is illustrated. In general the quadding procedure requires that each logic gate be quadriplicated and that each of the gates in a quadd stage will have twice as many inputs as the non-redundant gates replaced. The outputs of a stage are interconnected to the inputs of the succeeding stage by an interconnection pattern such that the effects of errors in earlier stages gets subsequently "restored" in the latter stages, i.e., the originally "good" signal is restored.

2.5 STANDBY REPLACEMENT REDUNDANCY

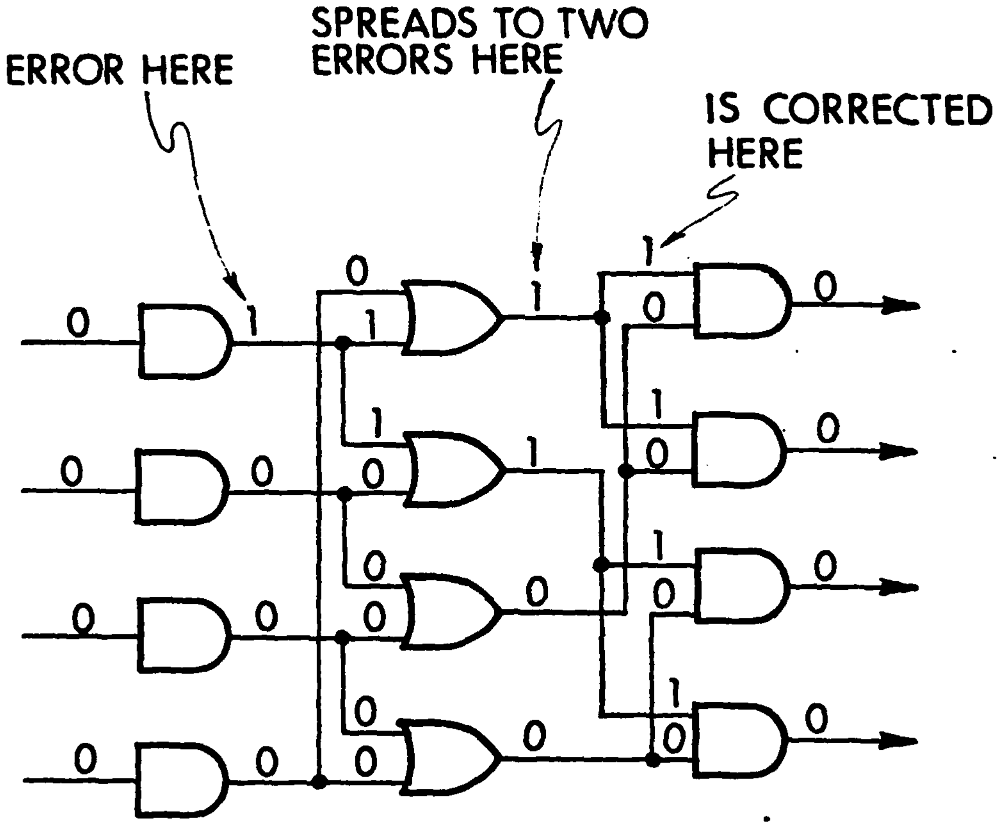
For standby replacement redundancy, unlike TMR, only one unit is operational at a time (see Figure 4). When the active unit fails this event is detected by additional circuitry and a spare unit from a reserve of spares is switched-in to replace the failed unit thereby restoring the system to its operational state. The reliability of this system is expressed as

$$R_{system} = 1 - (1-R)^{S+1}$$

which is the probability that not all units have failed.



(a)



(b)

Figure 3. An example of quadding. (a) Non-redundant circuit (b) circuit in (a) protected by quadding

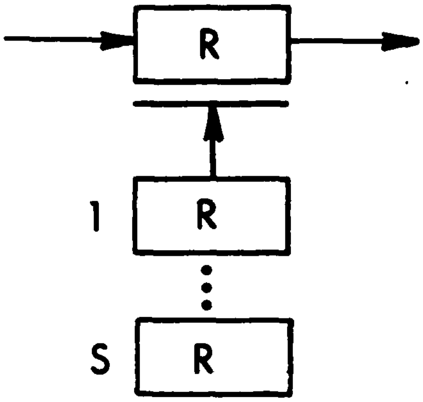


Figure 4. Standby replacement

2.6 HYBRID REDUNDANCY

Hybrid-redundancy is a synthesis of TMR and standby replacement redundancy (see Figure 5). It consists of a TMR system (or in general an NMR system) with a bank of spares such that when one of the TMR units fails, the failed unit is replaced by a spare unit. Failure detection is achieved by means of the disagreement detector which compares the individual outputs of each of the TMR'd units with the system output. Upon a disagreement the disagreement detector issues a signal to the switching network to replace the failed unit by a spare unit. At such time as all spares are utilized the hybrid redundancy system reduces to a TMR system. Variations of the hybrid or adaptive redundancy schemes are available. The system reliability in its simplest terms may be expressed as

$$R_{system} = 1 - [(1-R)^{S+3} + (S+3)(1-R)^{S+2} \cdot R]$$

which is the probability that not all S+3 units fail and that not any S+2 units fail with one not failing.

A comparison of reliability improvement and mean-life improvements of systems using no redundancy (simplex systems), TMR, standby sparing, and hybrid redundancy is presented by Mathur [27].

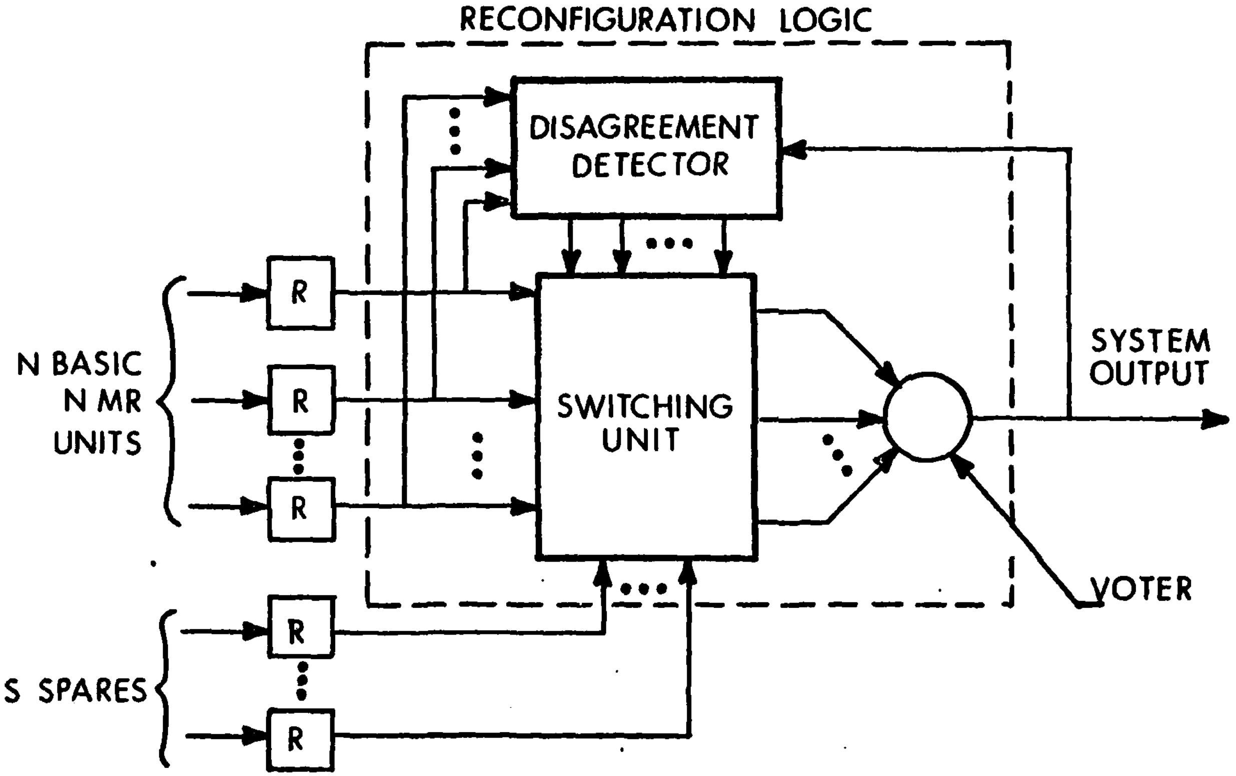


Figure 5. Hybrid redundancy

2.7 K-OUT-OF-N REDUNDANT ARCHITECTURE

This section gives a unified treatment of the various protective redundancy structures described in the preceding. It is contended that by and large all fault-tolerant computers are particular cases of the class of partitioned K-out-of-N redundant structures. The main differences being in (i) the degree of partitioning used and (ii) the means of error-detection employed.

The basic underlying structure of all Hardware Implemented Fault-Tolerant (HIFT) systems is the so-called K-out-of-N structure. It is composed of a total of N identical units. For the structure to function at least K of the N replicated units must remain operational. Hence its name. This structure can tolerate up to (N-K) independent failures, one in each of (N-K) out of the total N units. Thus, these structures exhibit a fault-tolerance, t equal to N-K. Here by fault-tolerance is meant the total number of replicas that the system can afford to have failed yet itself remain operational.

If r is the reliability (survival probability) or an individual replicated unit then the reliability of the K-out-of-N structure under the assumption that failures are independent events is given by the expression

$$R(\text{K-out-of-N}) = \sum_{i=1}^{N} \binom{N}{i} r^{i} (1-r)^{N-i}$$

where $$\binom{N}{i} = N!/(N-i)!i!$$

This reliability expression is simply the summation of all the successful events, i.e., the system survives provided K,K+1,K+2,...,N-1 or N units survive. The probability of exactly i units surviving is r^{i}. The probability of exactly (N-i) units having failed is $(1-r)^{N-i}$, and the number of ways in which this event can occur is N-combinatorial-i. The summation of all these events from i = K to N yields the above general expression. This powerful expression has a number of special cases which represent many of the commonly used protectively redundant structures. These special cases will now be described.

Case where K = N: Here all units need to survive for the structure to survive. This is the case when all units are in *series* reliability, and is representative of *simplex* (i.e., non-redundant) designs. Here the system reliability in terms of the unit reliability r is:

$$R(\text{N-out-of-N}) = r^N$$

and this structure exhibits zero fault-tolerance.

Case where K = 1: Here only one unit of the total N needs to survive for the structure to survive. This is typical of standby-spare redundancy, where one unit is active at any given time and the remaining are dormant as standbys.

$$R(\text{1-out-of-N}) = 1-(1-r)^N$$

The above reliability expression for standby-spares states that for the structure to be functional not all of the N units should have failed. Thus the case K = 1 represents a structure in *parallel* reliability, and exhibits a fault-tolerance of t = N-1.

Case where K = 2: The above two cases of K = 1 and K = N are the upper and lower bounds on the K-out-of-N structures. Now for the intermediate values of K. If K = 2, then the structure survives provided that at least two out of the N units are operative. This is the condition for a Hybrid redundant system having 3 units in triple modular redundancy (TMR) and the remaining units as standby-spares.

In the Hybrid redundant architecture (hybrid because it combines TMR and standby-spare redundant structures) three of the total N units are operated in TMR and the remaining N-3 units as backup units. Whenever one of the units composing the TMR structure fails it is replaced by one of the backup units. This process would continue until all backup units are exhausted at which time the hybrid structure reduces to a TMR structure. The TMR structure remains operative as long as at least 2 out of the three units, remain functional. Since only two units were required to remain operational throughout the system life of the structure and is equivalent to the hybrid redundant architecture. The system reliability is given by

$$R(2\text{-out-of-}N) = 1 - (1-r)^{N-1}[1+r(N-1)]$$

the fault-tolerance of a hybrid redundant system is equal to N-2.

Case where K = 2 and N = K+1: It is well known that hybrid redundant system H(3,S) with no spares is equivalent to a TMR configuration. Thus in the previous case if the total number of units N is three and K is still 2 we then have the classical von Neumann TMR system, i.e., for the system to remain operational at least 2 out of the three total units must remain operational. The reliability equation of the TMR system is given by

$$R(2\text{-out-of-}3) = r^3 + 3r^2(1-r)$$

has a fault-tolerance, t of one.

Case wehre K = (N+1)/2: The TMR structure can be generalized to an N-modular redundant (NMR) structure when N is any odd number of units operating in a majority configuration, i.e., an (N+1)/2-out-of-N configuration. The reliability expression for the NMR system is

$$R((N+1)/2\text{-out-of-}N) = \sum_{i=0}^{(N-1)/2} \binom{N}{i} (1-r)^{i} r^{N-i}$$

This system is capable of tolerating (n-2)/2 failures.

Case where K = (n+1)/2 with n odd: This case corresponds to the generalized hybrid redundant architectecture having a general nMR core and (N-n) spares. If the number of spares is zero then this case reduces to the previous one for NMR. The general hybrid redundant architecture can tolerate (N-n)+(n-1)/2 failures.

Composition of K-out-of-N Structures

The fact that hybrid redundancy is a combination of TMR and standby-sparing is readily seen from the *composition* of the following two K-out-of-N structures:

(i) 1-out-of-N .... standby-sparing

(ii) 2-out-of-3 .... TMR

(i) and (ii) are composed to yield:

(2-out-of-3)-out-of-N+3 .... Hybrid redundancy

the composition of (i) and (ii) is the Hybrid(3,N) system using a total of N+3 units.

Similarly for the generalized Hybrid redundancy case, NMR redundancy and standby-sparing can be composed thus:

(iii) 1-out-of-S .... standby-sparing

(iv) (N-1)/2-out-of-N .... NMR

The composition of (iii) and (iv) yields:

((N-1)/2-out-of-N)-out-of-S+N .... General hybrid

This composition represents the general hybrid redundant structure of H(N,S) having a total of N+S units.

Similarly other redundancy schemes can be shown to have a K-out-of N structure. The intent has not been to exhaustively list all equivalenҫes, the reader may readily try to represent some of the other redundant structures as K-out-of-N. The structures described here are summarized in Table I.

STRUCTURE	K	FAULT-TOLERANCE, t
Series	K = N	0
Parallel	K = 1	N - 1
TMR	K = 2; N = K+1	1
NMR	K = (N+1)/2; N = odd	(N-1)/2
Hybrid(3,S)	K = 2	N - 2
Hybrid(n,S)	K = (n+1)/2; n = odd	S + (n-2)/2

Table I: Summary of K-out-of-N Structures

It should be noted that in the above discussion no mention was made of the internal mechanisms by which errors are detected (errors imply failures in the system) and the means by which the system is reconfigured to remove the effect of these failures. These will be described in the next sections with reference to actual systems. However, in general, it may be stated that all HIFT structures are particular cases of (i) K-out-of N structures, (ii) self- or mutual-composition of these, (iii) partitioned K-out-of-N structures, (iv) compound (series) combinations of different K-out-of-N structures, and other combinations and permutations of these.

3.0 PARTITIONED AND BALANCED FAULT-TOLERANCE

As stated earlier, one of the fundamental principles of fault-tolerance is that redundancy ought to be applied to the smallest level of complexity of the system, in order to maximize gain in reliability. Naturally the overhead costs and associated unreliabilities involved in implementing too fine a level of partitioning dictate a compromise. Another factor in determining the partitioning resolution or the modularization level is the occurrence of natural interfaces in computer systems. Segmentation of a simplex computer design cannot be carried out arbitrarily but has to occur at the natural boundaries between the various functional subsets in order to (i) simplify the intercommunication between the modules, and (ii) to provide the necessary degree of isolation from failure propagation between one module and another.

Another effect of partitioning along natural boundary lines is that the resulting partitioned functional modules will have no two modules identical (idential in the reliability sense of having identical effective failure rates). The only exception being memory modules which are readily packageble to 4K or other standard size modules. Another possible exception is where the simplex computer has a highly uniform structured organization, e.g., that of, say, parallel processor array computers.

The net effect from a fault-tolerance view-point, of unequal modules is the task of balancing fault-tolerance over the entire system. Since a chain is no stronger than its weakest link, the architect has to strive to have all the sybsystems that comprise the system to be in fault-tolerance balance. In order then to balance unequally weighted subsystems, the degree of redundancy applied will vary from subsystem to subsystem. The two notions of level and degree of redundancy can now be formalized.

Level of Redundancy: The level of a system to which redundancy is applied refers to the size of the complexity of the unit to be replicated. The finer the system partitioning, the lower the level of redundancy.

Degree of Redundancy: At any level of a system the degree of redundancy refers to the number of replicas provided at that level.

Again, in striving for a balanced fault-tolerant architecture where all the subsystems are in fault-tolerance equilibrium the designer has to compromise. The perfect equilibrium cannot be reached but only approximated, since each subsystem having an arbitrary relationship to the other subsystems cannot be "fine-tuned" by simply adjusting the number of its discrete replicas, i.e., the reliability performance factors as a function of hardware are not continuous functions but discrete functions.

4.0 CASE HISTORIES

4.1 QUADDING AND THE OAC-PPDS

One fault-tolerant structure, applied at the component and the logical gate level rather than a module or subsystem level, which does not readily fit into the class of K-out-of-N structures is the fault-tolerance process of quadding. Quadding has been extensively and quite successfully applied in the design of NASA's Orbital Astronomical Observatory (OAO) satellite's on-board primary processor and data storage (PPDS) unit.

The PPDS employs extensive component level quadd redundancy. Its data storage buffer and data processor employ TMR, and its main memory is duplex redundant similar to the Saturn V LVDC (see later section on Saturn V LVDC). The PPDS receives commands to control the satellite's orientation, experiments, and data collection.

Quadding has three major advantages. First, it is applicable at the component level. Secondly, it is an autonomously redundant structure. By *autonomous redundancy* is meant the fact that no additional logic or circuits are necessary to implement error detection, location and reconfiguration. Thirdly, as a consequence of the second advantage it is applicable to truly real time systems with continuous availability. In contrast consider the self-repairing standby-spares system which requires an effective internal downtime to detect errors, program rollback, retry, locate source of error, and subsequently reconfigure.

Some of the disadvantages of the quaddnig approach are those of:

1. power consumption
2. fan-out
3. wide tolerances
4. difficult to test
5. difficult to evaluate its reliability
6. expensive
7. structure inflexible and unmodifiable.

Despite these disadvantages, quadding has been successfully applied and will continue to be applied in selective applications. It may be mentioned here that the Saturn V LVDC utilizes quadding to protect the decoupling capacitors in the power distribution system. Other applications of quadding are to protect component level hard-cores of self-reconfigurable computers.

4.2 TMR AND THE SATURN V LVDC

Basically triple modular redundancy (TMR), consists of triplicating the simplex unit and deriving the system output by taking the majority signal (by means of a vote taker) of the three independent signals from the replicated units. The system can be partitioned and also the vote takers themselves can be triplicated. A major application of TMR techniques is exemplified by the design of the Saturn V launch vehicle guidance system. This guidance system is composed of two parts (i) the general purpose computer called the launch vehicle digital computer (LVDC) and (ii) the launch vehicle data adapter (LVDA). The LVDA is an input/output interface unit that buffers the computer to its launch vehicle environment.

The computer characteristics of the LVDC are given in Table II.

Type:	General purpose, serial, fixed point, binary
Clock:	512 kilobits per second
Speed:	Add - 82 microseconds, 26 bit Multiply - 328 microseconds, 24 bit Divide - 566 microseconds, 24 bit
Memory:	32K, 28 bits
Weight:	44 Kg.
Volume:	0.62 m^3
Power:	152 watts

Table II. Saturn V LVDC Characteristics

The reliability goal for the LVDC was established at 0.99 for a 250-hour mission. It was felt that a computationally equivalent simplex computer using conventional architecture would only be able to achieve a reliability performance of 0.63 for the 250 hours. This increase in target reliability from 0.63 to 0.99 was achieved by utilizing a combination of redundancy structures.

The computer central logic is TMR and is divided into seven modules, each with an average of thirteen voted outputs. A total of some 155 signals are voted on, by a total of 395 voters. The LVDA employs 237 voters in its TMR logic. The reason for the LVDC only using 395 voters and not 465 voters (= 155 x 3) is that many of the central logic outputs are supplied to duplex circuits in the memory and LVDA. Hence, only two voters are needed at these outputs.

Instructions are composed of a four-bit operation code and a nine-bit operand address. The nine-bit address allows 512 locations to be directly addressed. The instruction address is augmented by a pair of sector registers and a pair of module registers. Separate registers keep track of data and instructions are stored two to a word, one in syllable 0 and the other in syllable 1 of a memory word.

The memory of the LVDC is protected by means of duplex redundancy. The eight identical 4K memory modules may also be operated in simplex if additional storage capability is desired. Two methods of error detection are used. First, parity checking is performed by logic which is protected by TMR. Also, circuitry is provided in each memory module to detect (i) the absence or improper timing of X or Y half-select currents, and (ii) the presence of select currents in more than one X or one Y line. Any of these error detections will initiate memory switching.

The memory operation strategy is as follows: when memory units are operated in duplex only one of the two buffer register outputs (A or B) is used. If an error is detected in the memory being currently used, the memory select logic switches over to the alternate unit. The incorrect memory is then regenerated with the output of the replicated memory. Thus, a transient error will be corrected and both memories will be restored to proper operation. Switching from one memory to the other is virtually instantaneous and caused no interruption. The only type of failure, called a *systematic failure*, that can cause complete memory system failure is the simultaneous failure at the same storage location of both memories.

The only sub systems not redundantly protected in the LVDC/LVDA are (i) the clock oscillator and (ii) the telemetry logic. The reasoning used to justify this is that the oscillator only consists of 5 component parts which is less than 1% of total components used, hence the probability of a failure occurring in this area is negligible. However, in practice the designer is advised that not only should he take the laws of statistics into account but also the more perverse laws of Murphy. No explicit reliability model of this computer was developed, however extensive Monte Carlo failure simulation analysis was performed. The Monte-Carlo-generated estimate for the reliability of the computer logic for a 250-hour mission calculated from 20,000 simulated missions is 0.9994.

4.3 STANDBY-SPARING AND THE JPL-STAR COMPUTER

Standby-sparing structures (1-out-of-N) are exemplified by the Jet Propulsion Laboratory's self-test and repair computer (STAR). The STAR computer, like its architectural predecessor the Raytheon RAYDAC, is a good illustration of an architecture that used non-autonomous redundancy as its principle means of failure protection. This is in sharp contrast to the Saturn V LVDC which as described earlier used TMR predominantly, duplex redundancy for memory and power supplies, and quadding in an isolated instance (note that no standby-sparing was employed anywhere).

The principal difficulties in implementing standby-sparing are: (i) means for detecting errors, (ii) means for switching replicas, (iii) conditioning requirements of spares before switching them on-line (recovery strategy), (iv) isolation of the replicas from the instruction/data buses and also from the power bus, and (v) problems of checking the error detector (i.e., how to check the checker).

In the STAR, error detection is implemented by encoding all machine words with codes that are preserved under arithmetic operations, as well as transmission. In conjunction with information encoding, decoders to check the validity of information are provided and the decoders themselves are protected by a separate autonomous redundancy structure. Replicas are switched by means of power switching rather than information switching. The recovery strategy is implemented by means of software interrupts, and program rollback followed by retry. Isolation of the functional units is by means of component redundancy.

The principal architectural features of the STAR are:

1. All data words and address portion of instruction words are encoded for error-detection using modulo 15 residue coding. This permits error detection concurrent to program execution. A 4 bit check byte c(b) is appended to the 7 byte (28 bits) non-redundant binary number b, where c(b) is computed to be the byte-wise complement of the modulo 15 residue of b.

2. The computer is subdivided into a number of replaceable functional units, each containing its own operation code decoders, and sequence generators. This decentralization and replication of system control provides simple fault location procedures and also simplifies interfacing between units.

3. Information lines of the replicas are permanently connected to the buses through isolating circuits. Replacement of units is implemented by power switching of the unit.

4. Fault detection, recovery, and replacement are carried out by the monitor (TARP: test and repair processor).

5. Transient faults are identified by program retry. Repetitious errors are identified as a permanent fault and eliminated by replacement of the failed unit.

6. The monitor, which is the "hard-core" of the system, is protected by means of autonomous redundancy, specifically by hybrid-redundancy.

5.0 STANDBY REDUNDANCY VERSUS AUTONOMOUS REDUNDANCY

The advantages to autonomous redundancy, also known as fault-masking redundancy are:

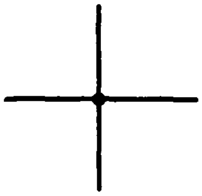
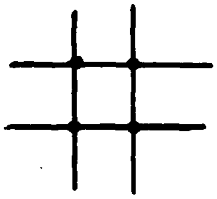
1. Corrective action is immediate and inherent to the structure.
2. During operation there is no need for separate error monitoring.
3. Machine words do not have to be encoded to provide error detection; consequently problems arising from encoding violations under arithmetic versus logical versus memory operations do not occur.
4. Impelmentation of such structures is relatively straightforward and can be applied to "off-the-shelf" subsystems such as microcomputres.
5. Coverage, the probability of detecting a failure given that there is a failure, is almost 100%, and is readily measurable.
6. Recovery strategy does not require "conditioning" of replicas.
7. The "hard-core" of the system is relatively small.
8. Internally truly real-time fault-tolerance and continuous system availability, since no program rollback and retry procedures are required.

Whereas, the advantages of standby redundancy are:

1. Power is required by only one replica at any time.
2. All replicas can be utilized.
3. Number of replica required can be easily tailored to a mission.
4. No increased "fan-out" problems arise.
5. System checkout is straightforward.
6. No synchronization between replicas is required.
7. System is less susceptible to externally induced transients.

6.0 PROTECTIVE ARCHITECTURE FOR THE "HARD-CORE"

The traditional means of protecting the "hard-core" of predominantly standby-spares redundant systems, namely the system's monitor and reconfiguration unit, is by means of TMR; with the voter of the TMR protected by componnet redundancy. It should be noted that the technique of quadding at the component level can be applied even to wires and connections. For example the connection:

 can be replaced by  thus

providing greater fault-tolerance. In this illustration the connection is provided by 1-out-of-4 redundancy and all 4 connections need to be impaired to violate the connection and at least two of the wires in any set of wires needs to open in order to violate continuity.

A more viable alternative to TMR is the class of redundancy known as hybrid redundancy. Hybrid redundancy combines the best features of both the autonomous TMR system and the more flexible standby-spares system. The principles of system operation are as follows: the replicated unit outputs are compared against the majority restored system output by means of the disagreement detectors (exclusive-or gates). The disagreement detector signals the switching-unit to replace the unit that disagrees with the majority. Thus the units in majority redundancy upon failure are continuously replaced by the spares, until all the spares are used up, at which instant the hybrid system reduces to the conventional majority voted system (NMR in general). In the next section we will consider the extra hardware required to implement the voter-disagreement-detector-switching-unit (V-D-S unit) for the TMR system.

6.1 IMPLEMENTATION OF THE V-D-S UNIT

A general implementation scheme utilizing iterative cell arrays for a V-D-S unit will now be described. First, two switching strategies are identified. One is *sequential* and the other *rotary*. In the first, the spares are ordered and utilized whenever a failure is detected in a first-available-first-used manner. In the rotary switching strategy, spares retain their ordering even after being switched-in to replace failed units. Thus, if the first spare (S1) initially replaced the third majortiy replica (M3) and subsequently the first majority replica (M1) where to fail, then S1 would rotate up to position M1 and a spare S2 would be called to fill in the vacancy at M3. The analysis on switch state requirements to implement these two strategies shows that if more than one spare is used then the rotary scheme provides fewer number of switch states (e.g., for 3 spares the rotary switch requires 20 states in contrast to the 34 required for sequential switching).

The basic characteristics of an iterative cell implementation, for a rotary switch, is shown in Figures 6 and 7 where an iterative cell array is a series of identical combinational logic cells that receive inputs from (i) outside the iterative cell network and (ii) from the cell immediately to its left via intercell leads. Each cell computes an output function that it transmits to (a) the switching network and (b) to a new intercell input for the cell immediately to its right. The output of a cell generates an output V_j^i assigning a module i to a voter position j. The output is generated as a function of (i) whether its corresponding module has disagreed or not and (ii) as a function of the number of prior modules found to be functional. The cell state and output as a function of previous state and disagreement input is shown for a typical call in Figure 7.

7.0 RECENT TRENDS IN FAULT-TOLERANT ARCHITECTURES

In the preceding sections we have discussed a number of HIFT architectures. One thing in common with all of them is that redundancy has been applied at the *intracomputer level* rather than at the *intercomputer* level.

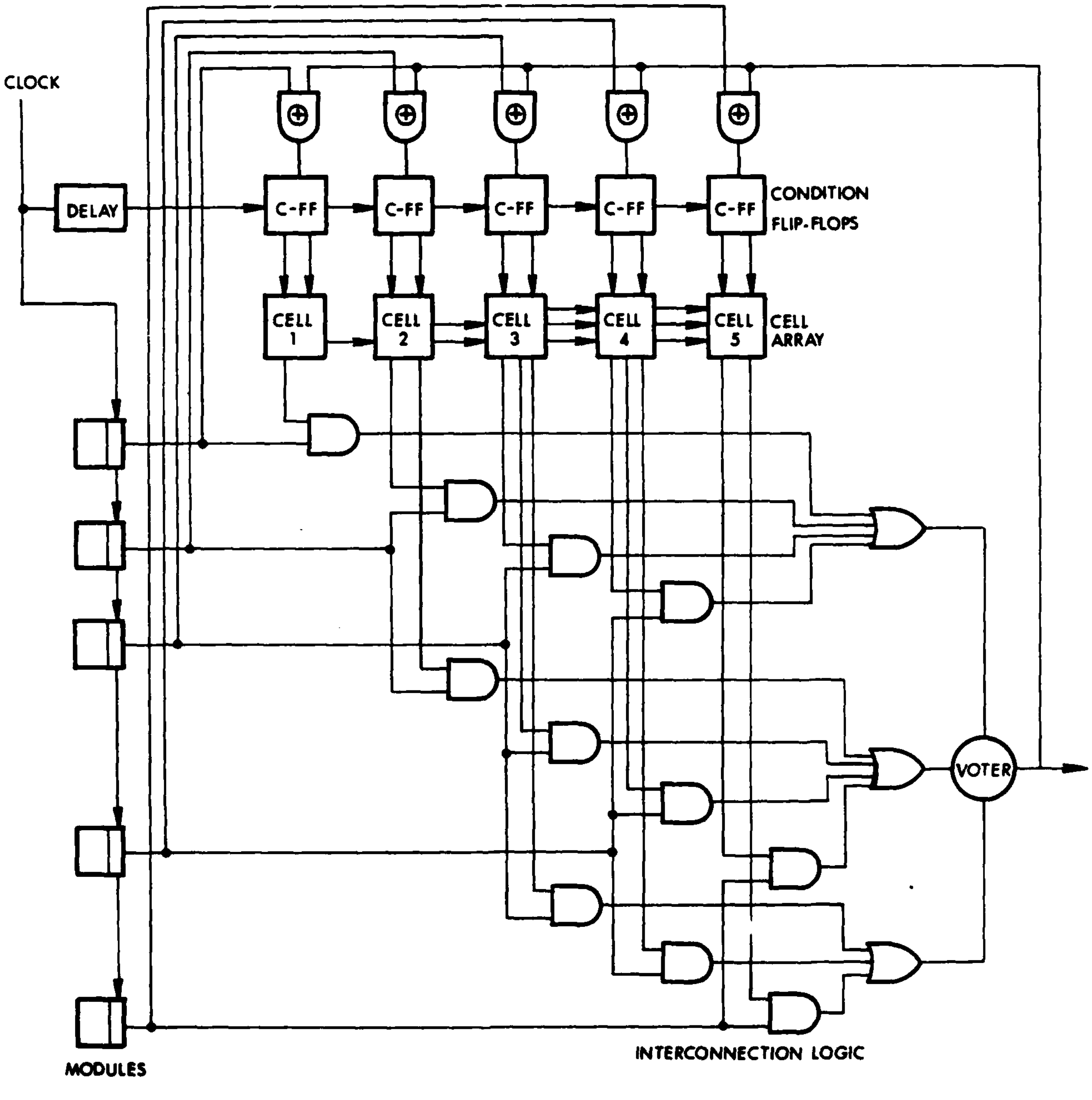
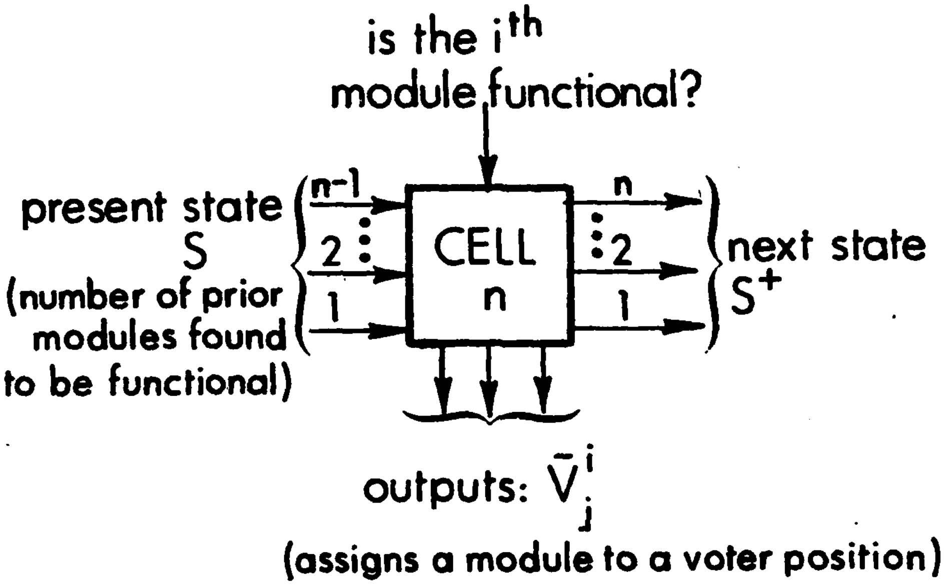


Figure 6. An iterative cell switch for H(3,2) system


is the i th module functional?
present state S (number of prior modules found to be functional)
n-1
2
1
CELL
n
n
2
1
next state S+
outputs: $\bar{V}_j^i$ (assigns a module to a voter position)

PRESENT STATE: S	NEXT STATE: S^+		OUTPUTS: $V_1^i V_2^i V_3^i$	
	$C_i = 0$	$C_i = 1$	$C_i = 0$	$C_i = 0$
A (zero)	A	B	0 0 0	1 0 0
B (one)	B	C	0 0 0	0 1 0
C (two)	C	D	0 0 0	0 0 1
D (three$^+$)	D	D	0 0 0	0 0 0

Figure 7. State and output table for an iterative cell.

There are three primary factors that are motivating the transition from intracomputer organizations to intercomputer configurations. First the extreme miniaturization of digital logic makes available complete CPU and memory units on one chip, which may have from 40 to 200 pins. If LSIs are custom specified then the state-of-the-art can allow many tens of thousands of gates per chip. This availability of subsystem performance capability at the level of the traditional discrete component motivates consideration of intercomputer level architectures.

Another factor is that the demand and volume production of LSI chips has drastically lowered haredware costs. Today's hardware designer equates an LSI CPU chip cost to the cost that existed seven or eight years ago for just one IC flip-flop. Thus the economics allow the designer to readily think in terms of *fault-tolerant networks of mini- or microcomputers.*

The third factor has been the increased reliability of semiconductors. Manufacturers have even started giving lifetime warranties for many devices. Of course, the failure rates of the latest LSI chips are not available to any level of confidence. It is a truism that by the time failure rates are available for any components to any satisfactory level of confidence, the components in question are obsolete! However, one can extrapolate increase in reliability of going from MSI to LSI by knowing the increase that was derived by going from ICs to MSI or from discrete transistors to ICs. This increase in reliability at the basic building block level is a factor in considering bigger structures and in considering fault-tolerance at higher hierarchical levels.

The foregoing does not mean that protective redundancy and fault-tolerance are not applied at the intrachip level or that it is not required at that level. On the contrary, as the chip becomes more complex the semiconductor manufacturing tolerances become stringent and result in poor yields of chips that can meet design specifications adequately. In fact one of the wa,s of achieving greater yields is to provide logic redundancy in the chip itself. The techniques of quadding mentioned earlier are readily applicable to this use. However, since the semiconductor industry is a highly competitive commercial enterprise the actual techniques employed by various manufacturers to achieve the *"black-chip"* specification (analogous to "black-box" specification) are closely guarded secrets. It is well known, and can be observed under a microscope, that more gates are etched or deposited on a wafer than are absolutely required, bonding of leads to the wafer can then be selectively performed to the best gates on the chip to meet the overall black-chip specification.

The factors outlined above make it feasible, both economically and engineeringwise, to design intercomputer fault-tolerant architectures. One such recent effort is the Software Implemented Fault-Tolerant (SIFT) computer proposed by Wensley for an avionics application [34].

7.1 THE SIFT COMPUTER

The SIFT architecture essentially embodies the TMR or NMR concepts in software. Majority voting is performed on the results of task segments of a job by software comparisons and decision making. Thus, the software majority voting is not at the hardware logic level but its finest resolution would be at the level of a single instruction (execution). This, in the limit, could approach the logic level resolution, provided that the machines are microprogrammed and allow task breakdowns to the microinstruction level. This operation is equivalent to partitioning in HIFT systems. Perhaps at the microinstruction level the SIFT system should be called a *"Micro-program Implemented Fault-tolerant"* (*MIFT*) system.

The integrity of the SIFT system is prevented from being violated by a failed and misbehaving processor by allowing the processors to only read from other processor's memory and never be allowed to write into them. Thus, essentially each memory unit has two sets of ports, (i) one that allows its companion processor to read and write into it, and (ii) a second set of ports that interfaces to the interprocessor bus and which allows other processors to only read information. To enable processors to communicate with one another a processor has to write a note into its companion memory and then subsequently, at some time the other processors would "look-in" to see if there were any messages lying around for them.

One advantage of this principle of organization is that the need for synchronized operation of replicated units is avoided. Each processor at the termination of a task would wait until the other processors assigned by the system executive dispatcher have completed the replicated software task step. The processors then read from each other the results of their tasks and effectively perform a majority agreement selection on the results. The disagreeing (faulty) replica need not necessarily have to be removed but can either be ignored or assigned to "void" tasks, i.e., tasks having no overall effect.

No special hardware except for the interfacing requirements are needed. The replicated buses are also considered as functional units or processors which have to be addressed and a link established. Subsequently this bus would then link up to the addressed processor memory and complete the "handshaking" between (i) the requesting processor, (ii) the available bus, and (iii) the addressed memory module. After a quantum of time the bus would release the "hand" and be available to shake another requesting processor's hand. Analogous to the hardware implemented TMR and NMR systems, the SIFT configuration can equivalently vary the degree of redundancy by allocating more than three processors to a job. Also, if the various tasks comprising a job have varying degrees of importance, then, equivalent to the HIFT approach, a fault-tolerance balancing action can be performed by allocating different degrees of replicas for the different tasks in the job.

The same approach that is used to segment jobs and allocate tasks is also used to protect the system executive. The executive system has two components, (i) a local executive which resides in every processor-memory module and is responsible for such fucntions as initiating new tasks (dispatching), reporting errors, loading tasks, and (ii) a system executive which resides in at least three modules (triplication) and has the functions of resource allocation, scheduling of work load and system reconfiguration. Each processor knows which processors have been designated as the system executives. If one of the system executive processors fails, then the other two will ignore the failed unit and assign another processor as system executive and inform all processors about this "ouster from the system executive troika."

7.2 THE PRIME COMPUTER

Another illustration of recent trends in fault-tolerant intercomputer architectures was exemplified by the PRIME project at the University of California - Berkeley. Here the objective is more to provide a fail-softly capability (quasi-fault-tolerance). The user is provided continuous service but a reduced levels of performance upon the occurrence of failures. The PRIME architecture uses off-the-shelf microprogrammable minicomputers (Digital Scientific Corporation META-4 microprocessors) to implement a multiprocessing time-sharing system with extensive secondary storage capabilities consisting of disk drives. Intercommunication between any processors or between any processor and any disk or between any processor and any external device is implemented by means of an interconnection network (IN) which is a distributed network. This network is partitioned and powered such that failures are always isolated to a very small part of the network. A failure in the IN is equivalent to a failure of the unit attached to the IN node that fails. Hence, this allows the system to run with an arbitrary variable set of the various units. The reader is referred to the PRIME literature for detailed description of the total multiprocessor time-sharing system [35-39].

The PRIME system does raise a very pertinent question,namely: can multiprocessing systems be considered to be in the same family lineage as fault-tolerant systems? One answer is that since quasi-fault-tolerance is the next lower hierarchical level (because it permits graceful performance degradation), and since software implemented fault-tolerant systems, as we have seen in some detail, are intrinsically multiprocessing systems (though however operating on replicated tasks) it follows that the distinction rests on the type of operating system (OS) that the multiprocessor operates under. Whether that OS implements software fault-tolerance, as in the SIFT proposal, or software quasi-fault-tolerance, as in the PRIME.

Thus we note that once the hardware is provided for "universal" connectivity and failure isolation, the fault-tolerance capability of these intercomputer configurations resides primarily in the software operating systems driving and managing the hardware.

Thus, although the HIFT architectures are here to stay,and will always be useful in selective applications such as avionics and aerospace, for the more general ground based commercial applications,such as computer utility type operations,we will see greater proliferation of software implemented fault-tolerant and quasi-fault-tolerant systems which will utilize not handcrafted processors, but commercially available off-the-shelf mini- and microcomputers.

8.0 AUTOMATION OF RELIABILITY MEASUREMENT PROCESSES

The large number of different redundancy schemes available to the designer of fault-tolerant systems, the number of parameters pertaining to each scheme, and the large range of possible variations in each parameter seek automated procedures that would enable the designer to rapidly model, simulate and analyze preliminary designs and through man-machine symbiosis arrive at optimal and balanced fault-tolerant systems under the constraints of the prospective application.

Such an automated procedural tool which can model self-repair and fault-tolerant organizations, computer reliability theoretic functios, perform sensitivity analysis, compare competitive systems with respect to various measures and facilitate report preparation by generating tables and graphs is implemented in the form of an on-line interactive computer program called CARE (for Computer-Aided Reliability Estimation) [40]. Essentially CARE consists of a repository of mathematical equations defining the various basic redundancy schemes. These equations, under program control, are then interrelated to generate the desired mathematical model to fit the architecture of the system under evaluation. The mathematical model is then supplied with ground instances of its variables and then evaluated to generate values for the reliability theoretic functions applied to the model.

The mathematical models may be evaluated as a function of absolute mission time, normalized mission time, non-redundant system reliability, or any other system parameter that may be applicable.

λ	=	Powered failure rate
μ	=	Unpowered failure rate
K	=	λ/μ = Dormancy factor
T	=	Mission time
T	=	Normalized mission time
R	=	Simplex reliability
R	=	Dormant reliability, exp(- T).
S	=	Number of spares
n	=	(N-1)/2 where N is the total number of multiplexed units
Q	=	Quota or number of identical units in simplex systems
C	=	Coverage factor, Pr(recovery/failure)
RV	=	Reliability of restoring organ or switching overhead
Z	=	Number of identical systems in series
W	=	Number of cascaded or partitioned units
P	=	Probability of unit failing to "zero"
TMR	=	Triple modular redundancy
TMR_p	=	TMR system with probabilistic compensating failures
(1,S)	=	Standby spare system
(N,S)	=	Hybrid redundant system
$(3,S)_{sim}$	=	Hybrid/simplex redundant system
MTF	=	Mean life
R(MTF)	=	Reliability at the mean life

Table III. Table of Abbreviations and Terms

8.1 UNIFYING NOTATION

A unifying notation, developed to describe the various system configurations using selective, massive or hybrid redundancy is illustrated in Figure 8.

N refers to the number of replicas that are made massivley redundant (NMR); S is the number of spare units; W refers to the number of cascaded units, i.e., the degree of partitioning; R() refers to the reliability of

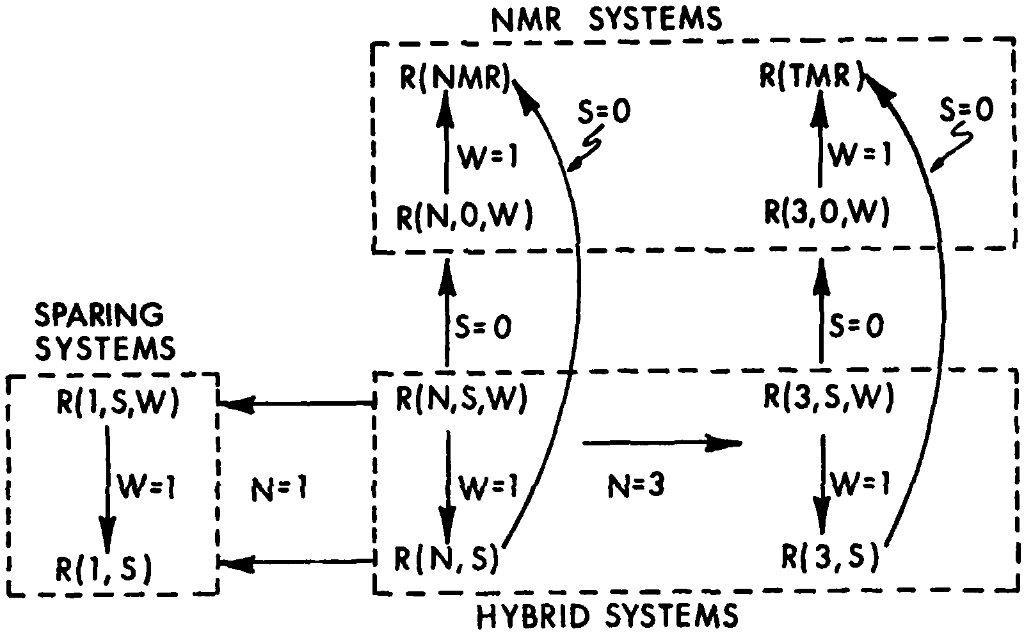


Figure 8. Unifying notation

the system as characterized in the parentheses; TMR stands for triple modular redundant system ($N = 3$); the NMR stand for N-tuple modular redundancy.

A hybrid redundant system H(N,S,W) is said to have a reliability R(N,S,W). If the number of spares is $S = 0$, then the hybrid system reduces to a cascaded NMR system whose reliability expression is denoted by R(N,0,W); in the case where there are no cascades, it reduces to R(N,0,1), or more simply R(NMR). Thus the term W may be elided if $W = 1$. The sparing system R(1,S) consists of one basic unit with S spares.

Furthermore, the convention is used that R* indicates that the unreliability ($1-R_V$) due to the overhead required for restoration, detection, or switching has been taken into account, e.g., $R^*(NMR) = R_V \cdot R(NMR)$; if the asterisk is elided then it is assumed that the overhead has a negligible probability of failure. This proposed notation is extendable and can incorporate a number of functional parameters in addition to those shown here by enlarging the vector or lists of parameters within the parentheses, e.g., R(N,W,W,...,X,Y,Z).

8.2 EXISTING RELIABILITY PROGRAMS

Some representative reliability evaluation programs are the RCP, RELAN, and REL70. RCP [41,42] is a program which can model a network of arbitrary series-parallel combinations of building blocks and analyzes the system reliability by means of probabilistic fault-trees. RELAN [43] is an interactive program developed by TIME/WARE and is offered on the Computer Sciences Corporation's INFONET network. RELAN, like RCP models arbitrary series-parallel combinations but in addition allows a wide choice (any of 17 types) of failure distributions. RELAN has concise and easy to use input formats and provides elegant outputs such as plots and histograms. REL70 [60] and its forerunner REL [61] are interactive programs developed in APL/360. Unlike RCP and RELAN, REL70 is more adapted for evaluating systems other than series-parallel, such as standby-replacement and triple modular redundancy. It offers a large number of system parameters, in particular, C, the coverage factor defined as the probability of recovering from a failure given that the failure exists and, Q, the quota, which is the number of modules of the same type required to be operating concurrently. REL70 is primarily oriented toward the exponential distribution though it does provide limited capabilities for evaluating reliability with respect to the Weibull distribution; its outputs are primarily tabular. Since APL is an interpretive language, REL is slow in operation; however, its designers have overcome the speed limitation by not programming the explicit reliability equations but approximate versions which are applicable to short missions by utilizing the approximation $(1\text{-exp}(-\lambda T)) \approx \lambda T$ for small values of λT.

The CARE program is a general program for evaluating fault-tolerant systems, general in that its relaibility theoretic functions do not pertain to any one system or equation but to all equations contained in its repository and also to compelx equations which may be formed by interrelating the basic equations. This repository of equations is extendable. Dummy routines are provided wherein new or more general equations may be placed as they are developed and become available to the fault-tolerant computing community. For example, the equation developed by Bouricius et al., for standby-replacement systems embodying the parameters C and Q has been bodily incorporated into the equation repository of CARE.

8.3 CARE'S REPOSITORY OF EQUATIONS

The equations residing in CARE, based on the exponential failure law, model the following basic fault-toleranct organizations:

(1) Hybrid-redundant (N,S) systems

(a) NMR (N,0) systems

(b) TMR (3,0) systems

(c) Cascaded or partitioned versions of the above systems

(d) Series string of the above systems.

(2) Standby-sparing redundant (1,S) systems

(a) K-out-of-N systems

(b) Simplex systems

(c) Series string and cascaded versions of the above.

(3) TMR systems with probabilistic compensating failures.

Series string and cascaded versions of the above.

(4) Hybrid/simplex redundant $(3,S)_{sim}$ systems.

(a) TMR/simplex systems

(b) Series string and cascaded versions of the above.

The equations for each of these systems are the most general representation of their systems, parameterizing mission time, failure rates, dormancy factors, coverage, number of spares, number of multiplexed units, number of cascaded units,and number of identical systems in series. The definitions of these parameters reside in CARE and may be optionally requested by the user. More complex systems may be modeled by taking any of the above listed systems in series reliability with one another.

9.0 REFERENCES

A. GENERAL REFERENCES

1. von Neuman, J., "Probabilistic logics and the synthesis of reliable organisms from unreliable components," Automata Studies, Princeton University Press, New Jersey, 1956, pp. 43-98.

2. Pierce, W.H., "Redundancy in computers," Scientific American, vol. 210, February 1964, pp. 103-111.

3. Teoste, R., "Digital circuit redundancy," IEEE Trans. on Reliability, vol. R-13, June 1964, pp. 42-61.

4. Short, R.A., "The attainment of reliable digital systems through the use of redundancy - a survey," IEEE Computer Group News, vol. 1, no. 3, March 1968, pp. 2-17.

5. Lyons, R.E. & Vanderkulk, W., "The use of triple-modular redundancy to improve computer reliability," IBM J. of Res. & Dev., vol. 6, no. 2, April 1962, pp. 200-209.

B. QUADDING

6. Jensen, P.A., "Quadded NOR logic," IEEE Trans. on Reliability, vol. R-12, no. 3,September 1963, pp. 22-31.

7. Tryon, J.G., "Quadded logic," in Redundancy Techniques for Computing Systems, R.H. Wilcox & W.C. Mann, eds., Spartan Books, Washington, D.C., pp.205-228, 1962.

8. Fasano, R.M. & Lemack, A.G., "A quad configuration - reliability and design aspects," Proc. 8th Symp. on Reliability and Quality Control, Washington, D.C., January 9-11, 1962.

9. Soremon, A.A.,"Digital-circuit reliability through redundancy," Electro-Technology, vol. 68, July 1961, pp. 135-140.

10. Tryon, J.G., "Redundant logic circuitry," U.S. Patent 2, 943, 193; filed July 30, 1958 (to Bell Telephone Laboratories, Inc.).

11. Creveling, C.J., "Increasing the reliability of electronic equipment by the use of redundant circuits," Proc. IRE, vol. 44, April 1956, pp. 509-515.

C. SATURN V LVDC AND THE OAO PPDS

12. Kuehn, R.E., "Computer redundancy: design, performance, and future," IEEE Trans. on Reliability, vol. R-18, no. 1, February 1969, pp. 3-11.

13. Anderson, J.E. & Macri, F.J., "Multiple redundancy applications in a computer," Proc. 1967 Ann. Symp. on Reliability, Washington, D.C., January 10-12, 1967, pp. 553-563.

14. Dickinson, M.M., Jackson, J.B. & Randa, G.C., "Saturn V launch vehicle digital computer and data adapter," AFIPS Conf. Proc. of the FJCC, San Francisco, California, October 1964.

15. Dickinson, M.M. et al., "Saturn V launch vehicle digital computer and data adapter," IBM Report No. 64-825-1179, IBM Federal Systems Division, Oswego, New York, September 1964.

16. McNeil, C.V. & Randa, G.C., "Self-correcting memory - the basis of a reliable computer," Electronic Design, vol. 13, August 1965, pp. 28-31.

17. Coffelt, R.B., "Automated system reliability prediction," Proc. 1967 Ann. Symp. on Reliability, Washington, D.C., January 10-12, 1967, pp. 302-305.

18. Rubin, D.K., "Triple modular redundant systems," Jet Propulsion Laboratory Report TM-34,Section 341, 1967.

D. RAYTHEON'S RAYDAC

19. Block, R.M., Campbell,R.V.D. & Ellis, M., "The logical design of the Raytheon computer," Proc. of the MTAC, vol. 3, pp. 286-295, 1948; also discussion notes, pp. 317-322.

E. THE JPL-STAR

20. Rohr, J.A., "STAREX self-repair routines: software recovery in the JPL-STAR computer," Digest of the 1973 Annual Symp. on Fault-Tolerant Computing, Palo Alto, California, 1973.

21. Avizienis,A. & Rennels, D.A., "Fault-tolerance experiments with the JPL-STAR computer," Digest of the 6th Annual Computer Conference, September 1972, pp. 321-324.

22. Avizienis, A., Gilley, G.C., Mathur, F.P., Rennels, D.A., Rohr, J.A. & Rubin, D.K., "The STAR (self-testing and repairing) computer: an investigation of the theory and practice of fault-tolerant computer design," IEEE Trans. on Computers, vol. C-20, no. 11, November 1971, pp. 1312-1321.

23. Mathur, F.P., "Reliability estimation procedures and CARE: the computer aided reliability estimation program," JPL Quarterly Technical Review, vol. 1, no. 3, October 1971, pp. 17-26.

24. Gilley, G.C., "Automatic maintenance of spacecraft systems for long-life, deep space missions," Ph.D. dissertation, University of California-Los Angeles, Department of Computer Science, September 1970.

F. HYBRID REDUNDANCY (Implementation & Evaluation)

25. Mathur, F.P., "Reliability modeling and analysis of a dynamic TMR system utilizing standby-spares," Proc. 7th Annual Allerton Conf. on Circuit and System Theory, IEEE Catalog No. 69C 48-CT, October 1969, pp. 243-252.

26. Mathur, F.P. & Avizienis, A., "Reliability analysis and architecture of a hybrid-redundant digital system," AFIPS Conf. Proc. of the SJCC, vol. 36, Atlantic City, New Jersey, May 1970, pp. 375-383.

27. Mathur, F.P., "On reliability modeling and analysis of ultra-reliable fault-tolerant digital systems," IEEE Trans. on Computers, vol. C-20, no. 11, November 1971, pp. 1376-1382.

28. Mathur,F.P., "Reliability modeling and architecture of ultra-reliable fault-tolerant digital systems," Ph.D. dissertation, University of California - Los Angeles, Computer Science Department, June 1970.

29. Mathur, F.P., "Automation of reliability evaluation procedures through CARE - the computer-aided reliability estimation program," AFIPS Conf. Proc. of the FJCC, vol. 41, Anaheim, California, December 1972, pp. 65-82a.

30. Siewiorek,D.P. & McCluskey, E.J., "Switch complexity in systems with hybrid redundancy," IEEE Trans. on Computers, vol. C-22, no. 3, March 1973, pp. 276-282.

31. Siewiorek,D.P. & McCluskey, E.J., "An iterative cell switch design for hybrid redundancy," IEEE Trans. on Computers, vol. C-22, no. 3, March 1973, pp. 290-297.

32. Ogus, R.C., "Fault-tolerance of the iterative cell array switch for hybrid redundancy," Digest of 1973 Int'l. Symp. on Fault-Tolerant Computing, Palo Alto, California, June 1973.

33. Brosius, D.B. & Jurison, J., "The design of a voter-comparator switch for redundant computer modules," Digest of 1973 Int'l. Symp. on Fault-Tolerant Computing, Palo Alto, California, June 1973.

G. SIFT

34. Wensley, J.H., "SIFT - Software implemented fault-tolerance," AFIPS Conf. Proc. of the FJCC, vol. 41, Anaheim,California, December 1972, pp. 243-253.

H. PRIME

35. Borgerson, B.R., "Spontaneous reconfiguration in a fail-softly computer utility," Digest of the DATAFAIR Conference, England, April 1973.

36. Borgerson, B.R., "Dynamic configuration of system integrity," AFIPS Conf. Proc. of the FJCC, vol. 41, Anahein, California, December 1972, pp. 89-96.

37. Baillin, G. & Borgerson, B.R., "A multipurpose-enhancement structure," Digest 1972 IEEE Computer Society Conference, San Francisco, California, September 1972, pp. 197-200.

38. Borgerson, B., "A fail-softly system for time-sharing use," Digest 1972 Int'l. Symp. on Fault-Tolerant Computing, Boston, Massachusetts, June 1972, pp. 89-93.

39. Baskin, M.B., Borgerson, B.R. & Roberts, R., "PRIME - a modular architecture for terminal-oriented systems," AFIPS Conf. Proc. of the SJCC, vol. 40, Atlantic City, New Jersey, May 1972, pp. 431-437.

I. RELIABILITY PROGRAMS

40. Mathur, F.P., "Automation of reliability evaluation procedures through CARE - the computer-aided reliability estimation program," AFIPS Conf. Proc. of the FJCC, vol. 41,Anaheim, California, December 1972, pp. 65-82a.

41. Chelson, P.O., "Reliability math modeling using the digital computer," Jet Propulsion Laboratory Report TR-32-1089,April 1967.

42. Chelson, P.O., "Reliability computation using fault-tree analysis," Jet Propulsion Laboratory Report TR-32-1542, December 1971.

43. Computer Sciences Corporation, "RELAN: reliability analysis package," CSC Sales Brochure No. 333, 1970.

44. Carter,W.C. et al., "Design techniques for modular architecture for reliable computer systems," IBM - Thomas J. Watson Research Center Report No. 70-208-0002, March 1970.

45. Roth, J.P., Bouricius, W.G., Carter, W.C. & Schneider, P.R., "Phase II of an architectural study for a self-repairing computer," (see section on the REL program), IBM Report SAMSO TR-67-106, November 1967.

PART II SELF-CHECKING CIRCUITS

1.0 INTRODUCTION

In 1968 Carter and Schneider [4] defined a self-checking circuit to be a circuit whose output is encoded in an error-detecting code. Anderson [2,3] further defined such circuits as having properties of self-testing and fault-secureness. Wakerly [6] introduced the concept of partially self-checking circuits.

Researchers have expanded upon these concepts and designed self-checking checkers and entire computers based on these properties. However, it is unfortunate that the initial historic definition by Carter and Schneider is rather narrow in that only circuits whose outputs are encoded in error detecting codes are considered to be self-checking circuits. As seen in the section on redundancy, hybrid and other system variants can also be self-checking without resorting to any error-detecting codes whatsoever.

This section on self-checking circuits presents an introduction to the fundamental underlying concepts and adheres closely to the literature and specifically is indebted to the notation developed by Wakerly [1].

Basic concepts of code space and detectable errors are explained and examples presented. Then the notions of fault-secureness and self-testing are introduced. Self-testing, partially self-testing, and totally self-testing circuits are then described.

Totally self-checking networks are then defined and an introduction to morphic Boolean logic is presented as a systematic methodology to the design of totally self-checking networks.

2.0 BASIC CONCEPTS OF CODE SPACE AND DETECTABLE ERRORS

Let U be the universe of all vectors of length n (n-tuples), then the subset S (called the *code space*) is an *error-detecting code* if the vectors in S are chosen such that every fault of interest affecting vectors in S (each vector in S is called a *code word*) will produce vectors that are not in S, i.e., in U-S (each vector in the *noncode space* U-S is called a *noncode word*). If a failure alters a code word x into another n-tuple x' then:

(i) if x' is also in S (the code space) then it is an *undetectable error*,

(ii) if x' is in U-S (the noncode space $\bar{S}$) then it is a *detectable error*.

Hence for the effect of a fault to be detectable it must produce an error such that some codeword (in code space) gets mapped onto some noncode word (in noncode space). For a code to be able to detect the set of failures of interest the code space S must be chosen so as to have this mapping property. These basic concepts are illustrated in Figure 1.

Example 1: Encode all 2-bit words (x_1x_2) with even parity (P_e). An encoded message (code word) is then of length 3.

All binary 3-tuples:

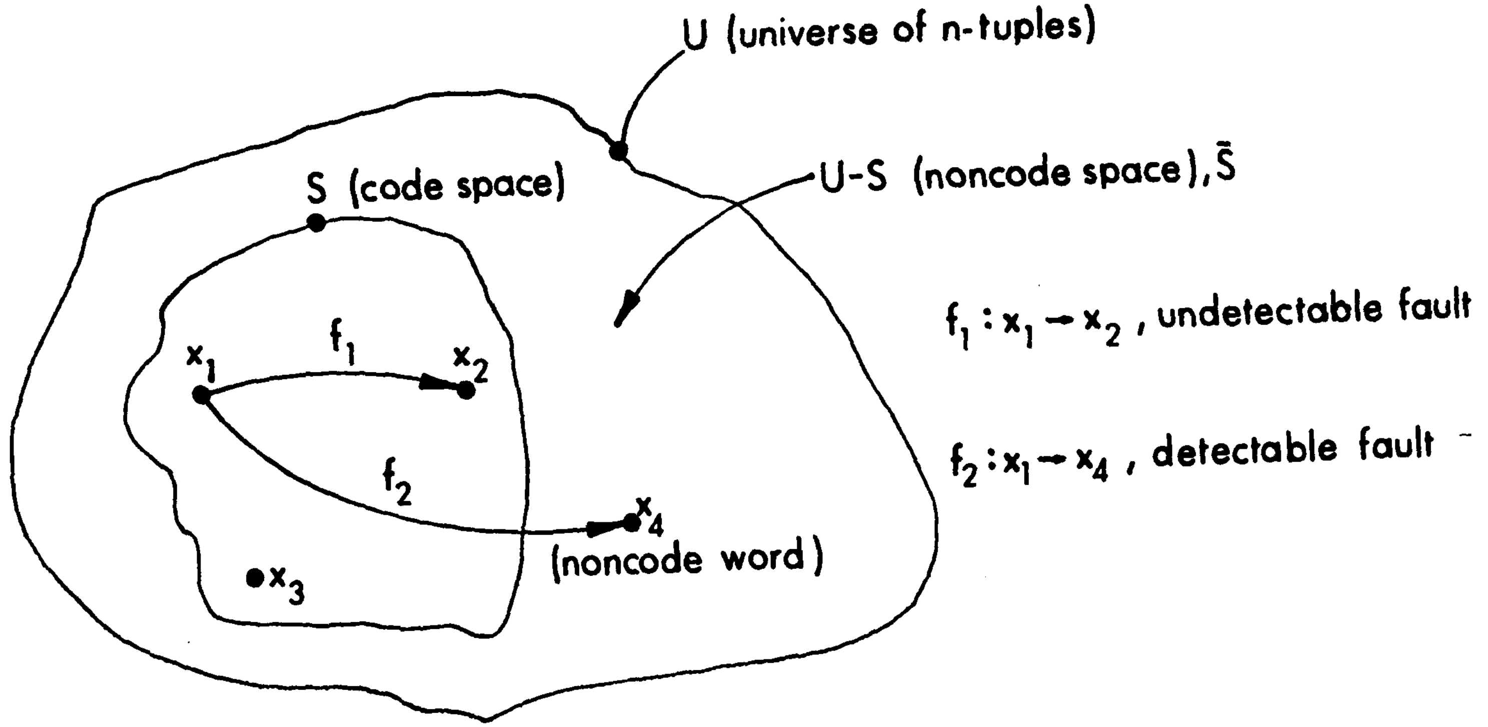
U =	0 0 0	1 0 0
	0 0 1	1 0 1
	0 1 0	1 0 1
	0 1 1	1 1 1

All uncoded messages:

x_2	x_1
0	0
0	1
1	0
1	1

All code words:
(with even parity)

S = x_2	x_1	P_e
0	0	0
0	1	1
1	0	1
1	1	0



U (universe of n-tuples)
S (code space)
U-S (noncode space), $\bar{S}$
$f_1 : x_1 \rightarrow x_2$, undetectable fault
$f_2 : x_1 \rightarrow x_4$, detectable fault
x_1
f_1
x_2
f_2
x_4
(noncode word)
x_3

Figure 1. Basic concepts of code space and detectable errors

All non-code words: U-S = 0 0 1
0 1 0
1 0 0
1 1 1

□

In Figure 2 we show how this simple form of parity encoding can be used in a self-checking circuit. The inputs x_1 and x_2 are used to drive a parity generator which generates P_e. The encoded data is then transmitted over a channel. The received data, consisting of $\hat{x}_1$, $\hat{x}_2$ and $\hat{P}_e$ is then again processed by a *checker*, and an odd parity (E = 1) indicates an error. Figure 3 indicates the possible I/O mappings.

In general, for all f_i in the fault-set the input code word could get mapped onto any 3-tuple in U. If f_i is such that an input is mapped into S then f_i is undetectable E = 0. If f_i is such that an input is mapped onto U-S, then f_i is detectable E = 1. Specifically, for the set of fault $\{f_i\}$ such that only an odd number of bits in the input code word are altered, the corresponding output words will be in the noncode space U-S, hence $\{f_i\}$ will be detectable, indicated by E = 1.

We will now examine and classify different fault sets by their corresponding mapping properties, and also consider input possibilities other than code words. For total generality we need to consider:

(i) the universe of inputs

(ii) the universe of outputs

(iii) the universe of faults

and the set of transfer functions $\{T_i\}$ associated with all possible fault-induced mappings onto the code space S and noncode space $\tilde{S}$. The absence of a fault, the *null fault*, will be denoted by λ. Thus,

$$\text{input: } x_1 \longrightarrow \overset{\text{fault-free, } \lambda}{\boxed{T(\text{input},\lambda)}} \longrightarrow \text{output:} x_1 \text{ in code space S}$$

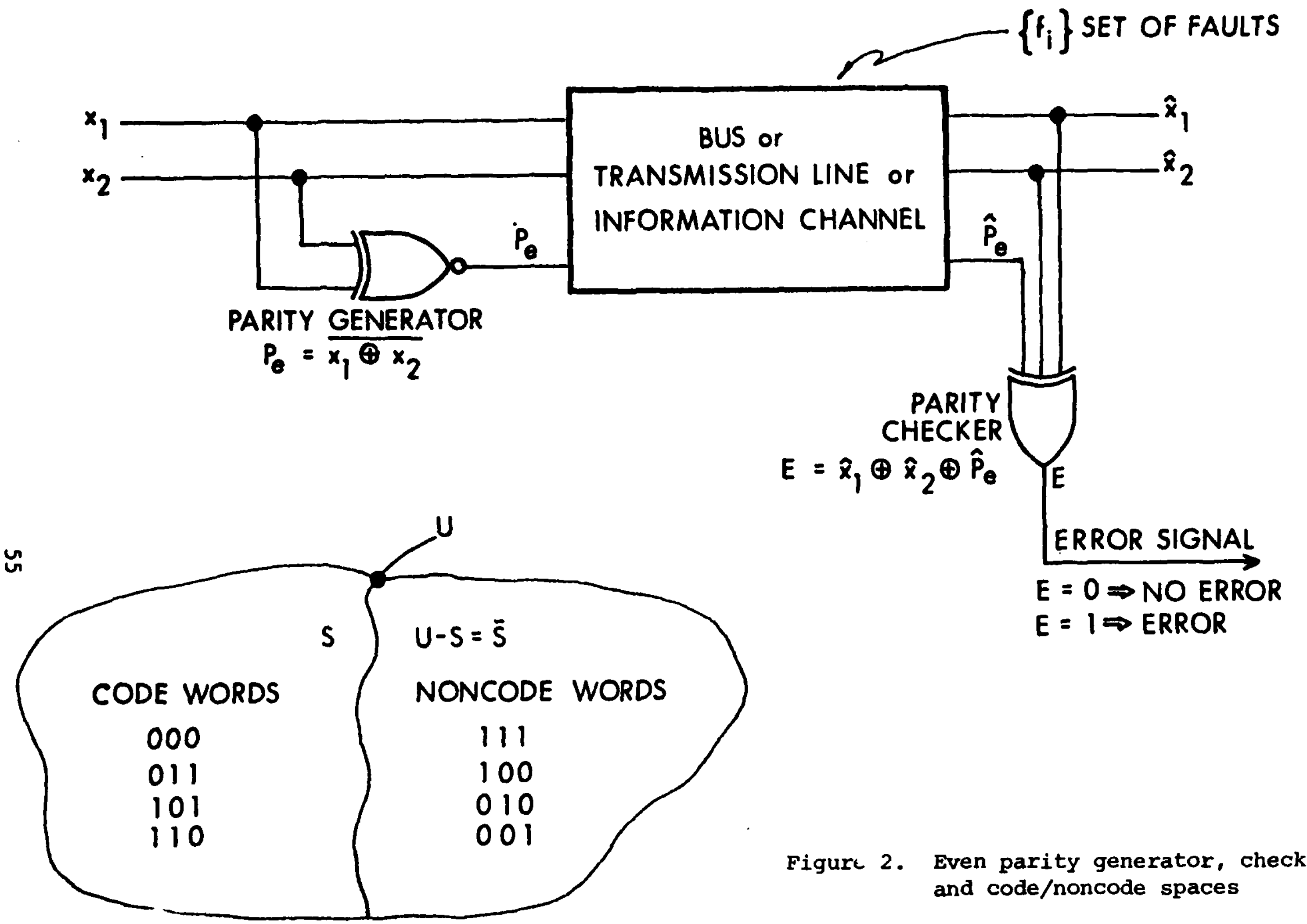


Figure 2. Even parity generator, checker and code/noncode spaces

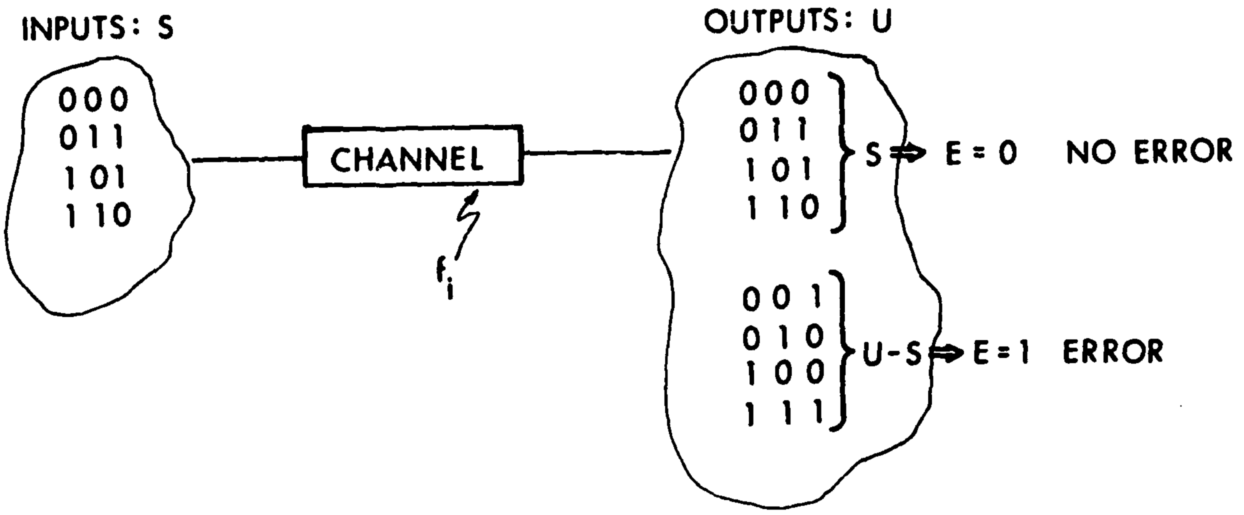
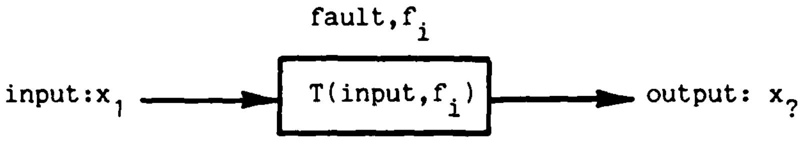


Figure 3. Input/output mapping: in general



For the fault-free case the output will always be in the code space S hence the checker will *correctly* indicate absence of any error.

In general for the faulty case the output $x_?$ may be anywhere in the output space U. Four cases can be identified:

Case I. Fault-free. Transfer function is $T(\text{input},\lambda)$.
Output is correct. Output is in code spaces.
No error is indicated.

Case II. Benign Fault. Transfer function is $T(\text{input},f_b)$
Output is correct. Output is in code space, S.
No error is indicated.

Case III. Detectable Fault. Transfer function is $T(\text{input},f_d)$.
Output is incorrect. Output is in noncode space, $\bar{S}$.
Error is indicated.

Case IV. Undetectable Fault. Transfer function is $T(\text{input},f_{ud})$.
Output is incorrect. Output is in code space,S.
Error is not indicated.

We summarize and illustrate these definitions in Figure 4 and Table I.

One of the goals of fault-tolerant design is to reduce the conditions under which Case IV can occur.

3.0 FAULT-SECURE CIRCUITS

If for a given design only the following transfer functions are possible $T(x,\lambda)$, $T(x,f_b)$, $T(x,f_d)$, i.e., Cases I, II and III, then the circuit is called a *fault-secure* circuit. (An alternative statement would be that

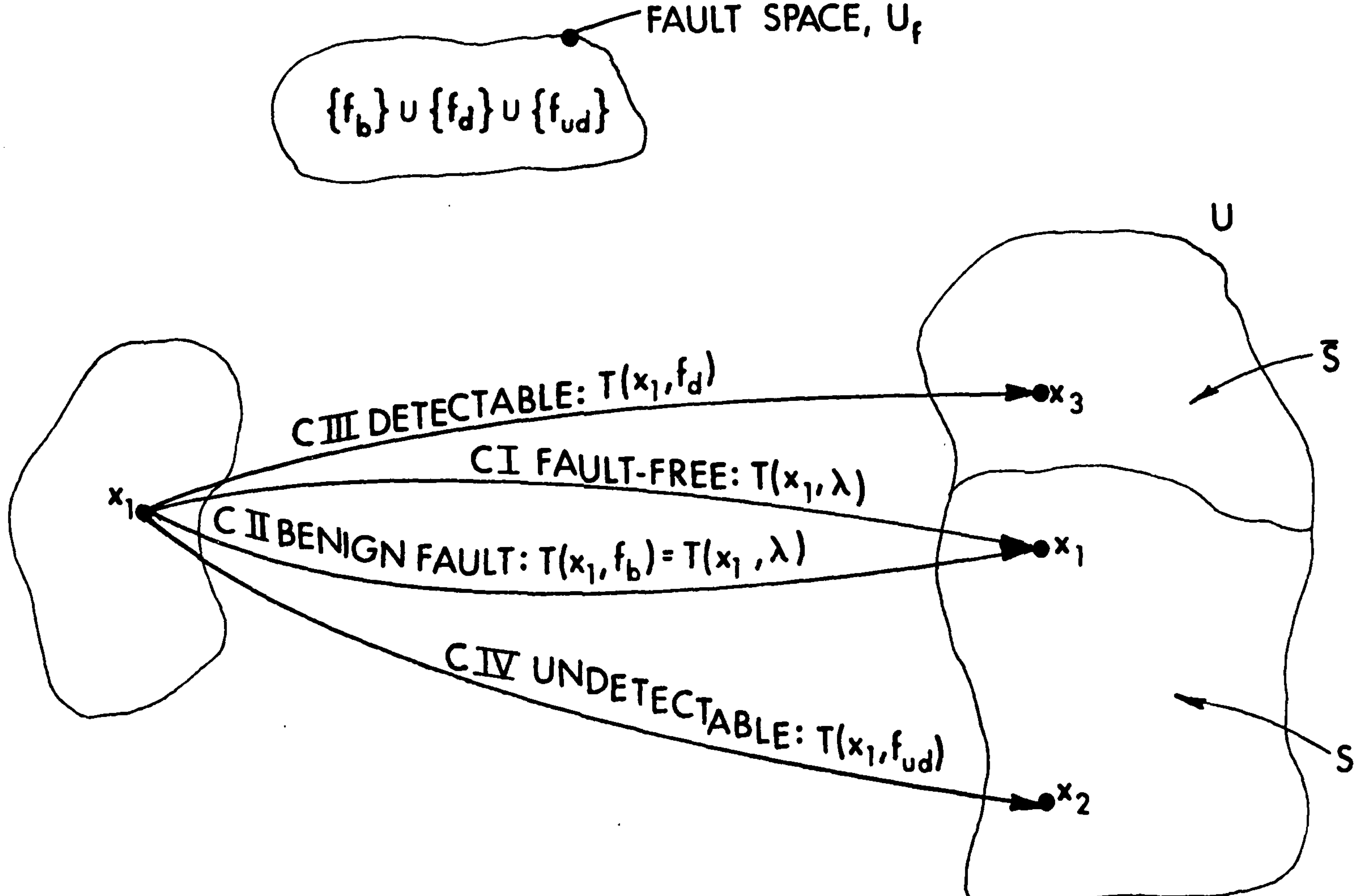

FAULT SPACE, U_f
$\{f_b\} \cup \{f_d\} \cup \{f_{ud}\}$
U
$\bar{S}$
S
x_1
x_3
x_1
x_2
C III DETECTABLE: $T(x_1, f_d)$
C I FAULT-FREE: $T(x_1, \lambda)$
C II BENIGN FAULT: $T(x_1, f_b) = T(x_1, \lambda)$
C IV UNDETECTABLE: $T(x_1, f_{ud})$

Figure 4. Four possible fault classifications: fault free benign, *detectable*, undetectable

CASE	FAULT CONDITION	TRANSFER FUNCTION	CORRECTNESS OF OUTPUT	OUTPUT SPACE	ERROR CAUSED	ERROR INDICATED
I	Fault-free	$T(x,\lambda)$	Correct (code word)	S	No	No
II	Benign fault	$T(x,f_b) = T(x,\lambda)$	Correct (code word)	S	No	No
III	Detectable fault	$T(x,f_d)$	Incorrect (noncode word)	$\bar{S}$	Yes	Yes
IV	Undetectable fault	$T(x,f_{ud})$	Incorrect (code word)	S	Yes	No

Table I. Fault classification cases and corresponding transfer functions and output attributes

$T(x,f) \in S$ implies that $f=f_b$ or λ. Figure 5 indicates the acceptable transfer functions for a fault secure circuit. When inputs are from the *secure input set*, U_{is} and the fault set is the *secure fault set*, U_{fs}, *fault secureness property* guarantees that no fault from the fault set will produce an undetectable incorrect output. It should be noted that in the above the set $\{f_d\}$ may be empty, i.e., all faults may be of the type f_b and hence undetectable. It is important to stress that, in general, a circuit is not fault secure with respect to all possible faults, but rather with respect to a given set of class of faults, e.g., all single stuck-at faults.

4.0 SELF-TESTING CIRCUITS

If for every fault, f_i, in the set of faults (which are under consideration) there exists some input, say x_t (x_t is called a *test for f_i*) such that $T(x_t,f_i)$ belongs to the noncode space $\bar{S}$, then the circuit is called a *self-testing circuit*. Thus a self-testing circuit is one for which every fault is detectable by applying some input; the input is called a test for that fault. The set of f_i's is called the *tested fault set*. As shown in Figure 6 for some x_j, f_* may be undetectable, but for some x_t, f_* is detectable.

The input set for self-testing circutis is called the *normal input set*, N, and every input should occur during normal operation in order to detect the presence of a fault from the tested fault set.

The secure input set, N_s may or may not be a subset of N, but for all practical purposes can be assumed to be a subset of N since those inputs outside of N would never occur in normal operation (by definition).

Similarly the secure fault set, F_s, is assumed to be a subset of F_t, the tested fault set. The interrelationship and combined effect of the two properties of fault-secureness and self-testing are shown in Figure 7.

Now we can look at the relationship between N_s, the secure input set, and N, the normal input set. Three cases are of interest:

Case (i): If $N_s = N$ then the circuit is called *totally self-checking*, and is both self-testing and fault-secure. (Note: a *self-checking* circuit is defined as any circuit whose output is encoded in an error-detecting code.

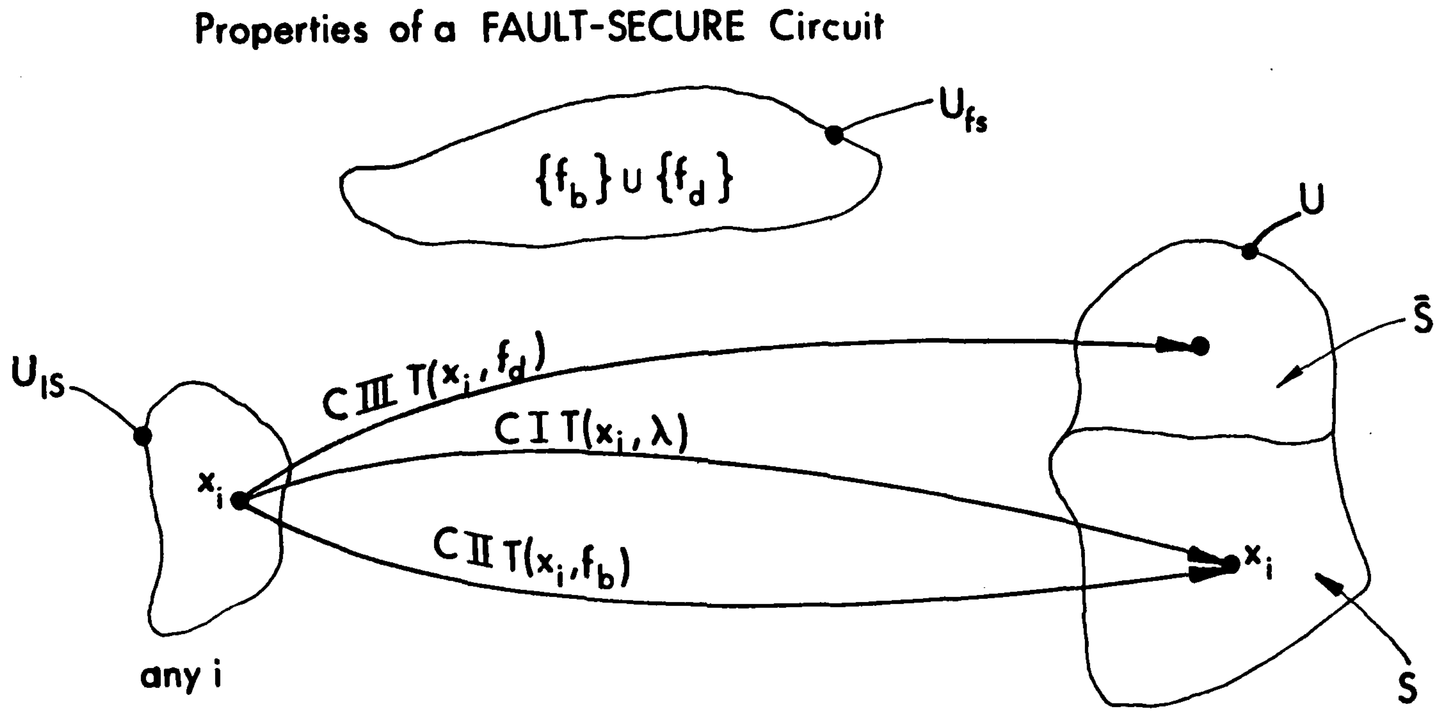


Figure 5. Fault-secure transfer functions

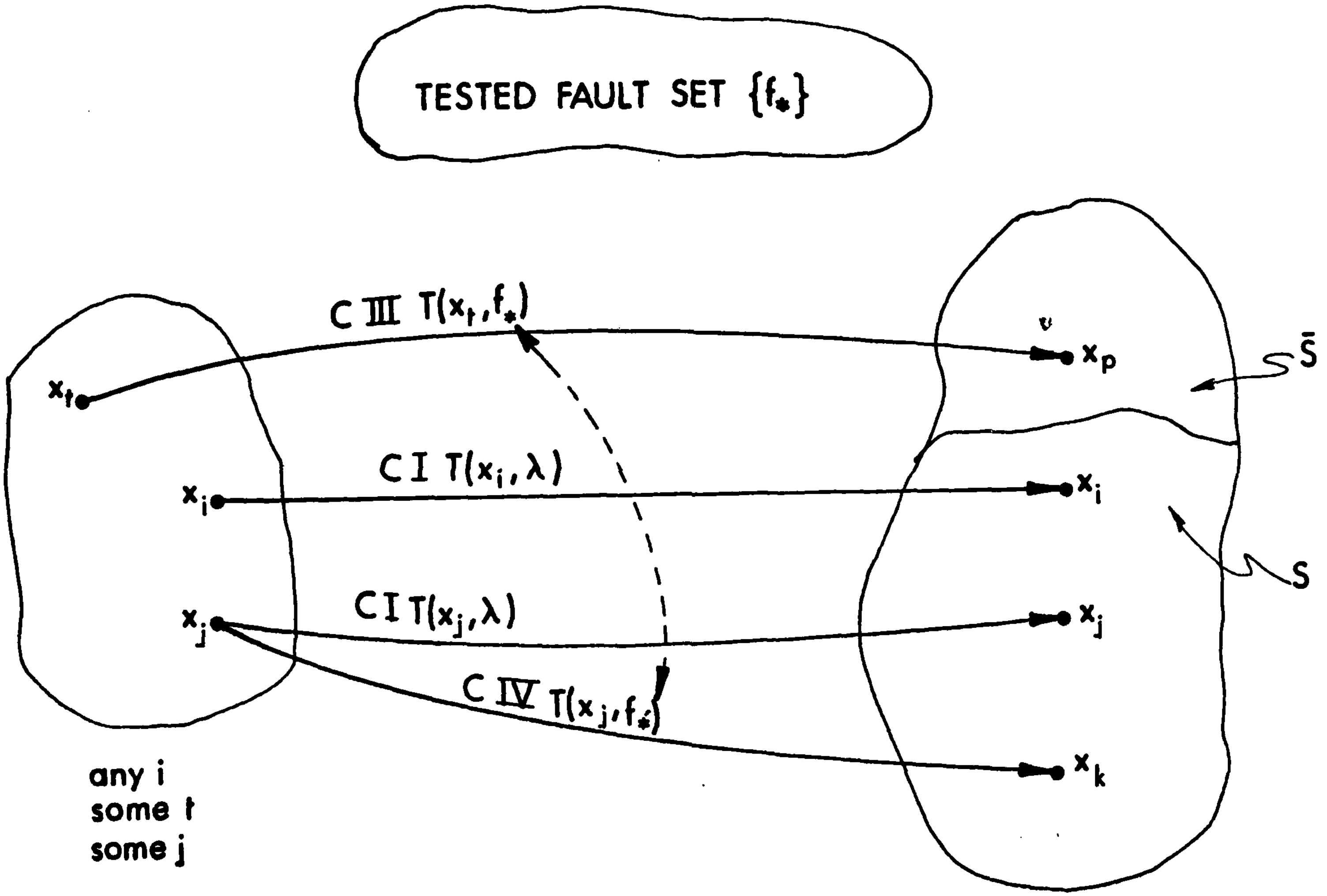


Figure 6. Self-testing transfer functions

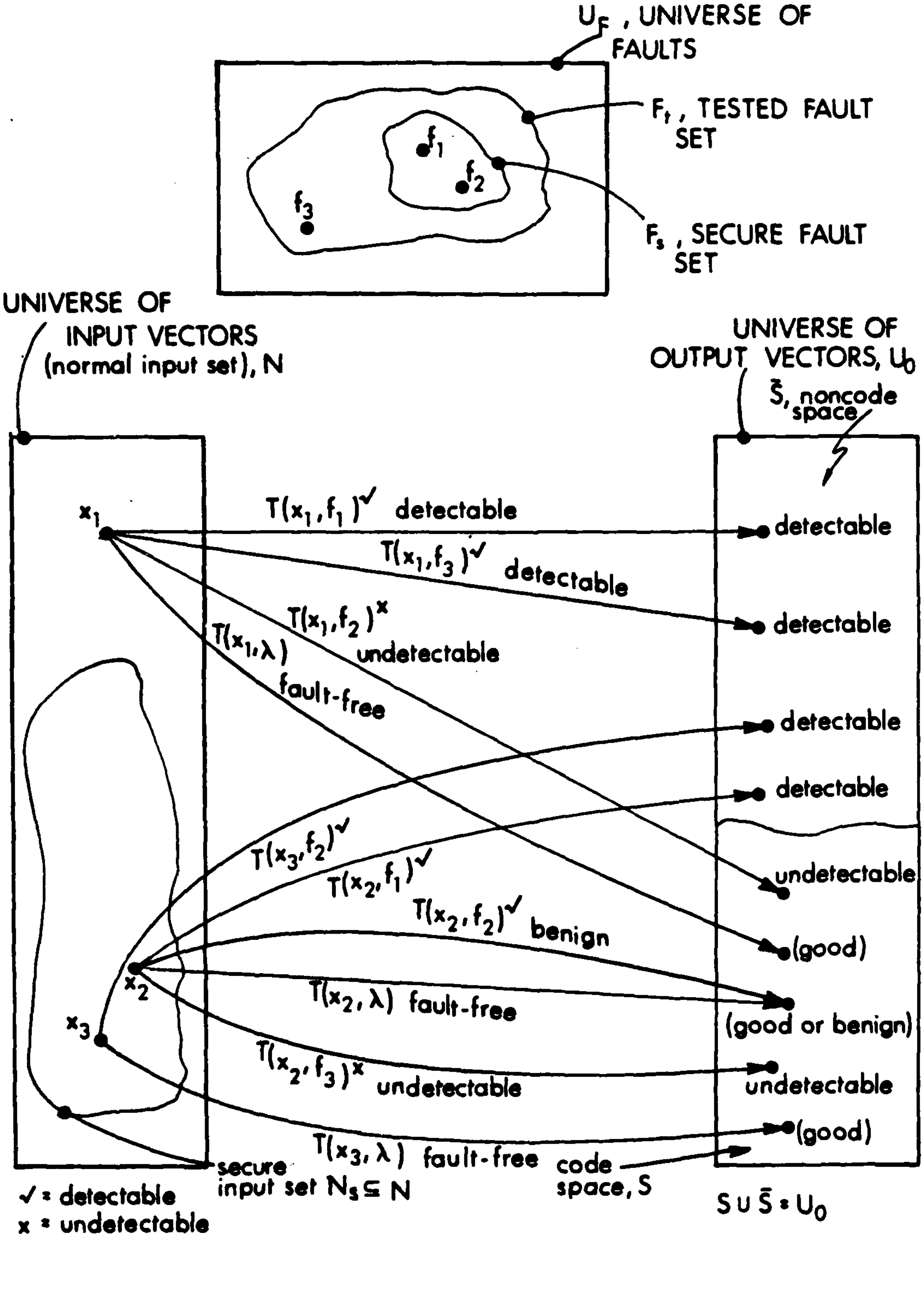

U_F, UNIVERSE OF FAULTS
F_t, TESTED FAULT SET
F_s, SECURE FAULT SET
f_1
f_2
f_3
UNIVERSE OF INPUT VECTORS (normal input set), N
UNIVERSE OF OUTPUT VECTORS, U_0
$\bar{S}$, noncode space
x_1
x_2
x_3
$T(x_1, f_1)^{\checkmark}$ detectable
$T(x_1, f_3)^{\checkmark}$ detectable
$T(x_1, f_2)^{x}$ undetectable
$T(x_1, \lambda)$ fault-free
$T(x_3, f_2)^{\checkmark}$
$T(x_2, f_1)^{\checkmark}$
$T(x_2, f_2)^{\checkmark}$ benign
$T(x_2, \lambda)$ fault-free
$T(x_2, f_3)^{x}$ undetectable
$T(x_3, \lambda)$ fault-free
detectable
detectable
detectable
detectable
undetectable
(good)
(good or benign)
undetectable
(good)
code space, S
secure input set $N_s \subseteq N$
✓ = detectable
x = undetectable
$S \cup \bar{S} = U_0$

Figure 7. Fault-secureness a subset of self-testability

Case (ii): If $N_s \subset N$ (a proper non-null subset) then the circuit is called *partially self-checking*.

Case (iii): If $N_s \cap N = \Lambda$(null) then the circuit is *self-testing-only*, and not fault-secure.

These definitions are summarized in Table II.

$N_s : N$	Properties of Self-Checking Circuits	References
$N_s = N$	Self-testing & fault secure (totally self-checking)	Anderson [2,3]
$N_s = \Lambda$	Self-testing & not fault secure	Carter [5]
$N_s \subset N$	Partially self-checking circuit	Wakerly [6]

TABLE II. Properties of Self-Checking Circuits

The principal advantage of partially self-checking circuits is that when using some simple codes, they may be used to perform logical operations and yet perserve proper encoding.

5.0 TOTALLY SELF-CHECKING NETWORKS

In order to have a totally self-checking network the checker must also be self-checking. A *totally self-checking network* is one that consists of a functional circuit and a checker where both the functional circuit and the checker are totally self-checking.

In Table III we summarize the main results dealing with self-checking circuits.

REFERENCE	RESULT
1. Toy [7] and Carter et al. [5]	General design of a self-testing only 1-out-of-n decoder
2. Anderson & Metze [3]	General procedure for designing totally self-checking checkers for k-out-of-2k codes for all k Also 1-out-of-n codes for all n except $n = 3$ and $n = 7$
3. Reddy [9]	Design of a totally self-checking 1-out-of-7 checker. Conjecture that no 1-out-of-3 checker exists.
4. Ashjaee & Reddy [10]	Conjecture that no totally self-checking equality checker with only three normal inputs exists.
5. Shedletsky [11]	Method for calculating a rollback interval for use with partially self-checking circuits.
6. Marouf & Friedman [12]	Algorithmic procedure for efficient design of general m-out-of-n checkers for $m \geqq 2$.
7. Carter et al. [13]	Design outline of an entire self-checking computer processor
8. Gay [14]	Markov models to compute the optimal testing strategy for a system with partial on-line error-detection.
9. Kolupaev [15]	Method to synthesize desired totally self-checking network by searching for an appropriate cascade of smaller generalized self-checking circuits. Method works for small networks but impractical for large nets.

TABLE III. Notable "Self-Checking" Results and Milestones

6.0 MORPHIC BOOLEAN FUNCTIONS AND THEIR IMPLEMENTATION AS SELF-CHECKING CIRCUITS

It is sometimes desirable to implement a Boolean variable x using a pair-of lines (x_1,x_2). Assume that $x = 1$ is represented sometimes as $x_1 = 1$, $x_2 = 0$, and at other times as $x_1 = 0$, $x_2 = 1$. Similarly if $x = 0$ is represented by both $(x_1,x_2) = (0,0)$ and $(1,1)$, then if either x_1 or x_2 is stuck at some value an error will be produced. We refer to codings of this type as *morphic* functions.

If $e_1,e_2 \in (0,1)$, then let the mapping M be defined as follows:

$$\begin{aligned} M:\ &((e_1,e_2),(\bar{e}_1,\bar{e}_2)) \mapsto 1 \\ &((\bar{e}_1,e_2),(e_1,\bar{e}_2)) \mapsto 0 \end{aligned}$$

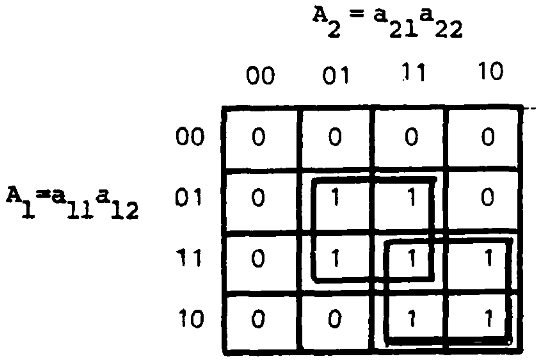
Since morphic logic functions (e.g.,AND,NAND,etc.) have to be implemented using the values taken by the pair of lines, a correspondence must be established between a ordinary Boolean function $g(a_1,a_2,\ldots,a_n)$ and the logic function whose inputs and outputs are pair of lines. The ordinary Boolean algebra is defined over the state space $\{0,1\}$, i.e., $B = \{0,1,\{*\}\}$, where $\{*\}$ is the usual set of logical operations. For the pair of lines we define the operators $\{*_M\}$ over the state space of the pair of lines, i.e., $\{\{(e_1,e_2),(\bar{e}_1,\bar{e}_2)\},\{(\bar{e}_1,e_2),(e_1,\bar{e}_2)\},\{*_M\}\}$. In order to establish a correspondence between the Boolean operator $\{*\}$ and the operator $\{*_M\}$ for the pair of lines, $\{*_M\}$ is defined to be a morphism (operation preserving property) between $\{0,1,\{*\}\}$ and $\{\{(e_1,e_1),(\bar{e}_1,\bar{e}_2)\},\{(\bar{e}_1,e_2),(e_1,\bar{e}_2)\},\{*_M\}\}$, i.e., between $\{*\}$ and $\{*_M\}$. If s_i and $s_j \in \{(0,0),(0,1),(1,0),(1,1)\}$ then $M(s_i *_M s_j) = M(s_i) * M(s_j)$.

As an example let $M: \begin{Bmatrix}01\\10\end{Bmatrix} \rightarrow 1, \qquad \begin{Bmatrix}00\\11\end{Bmatrix} \rightarrow 0$.

The morphic AND function (MAND) can be defined as follows:

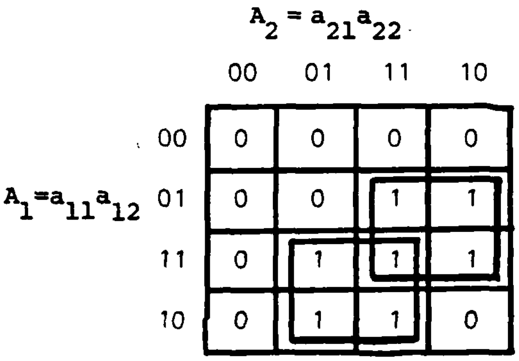
MAND	01	10	00	11
01	$\begin{Bmatrix}01\\10\end{Bmatrix}$	$\begin{Bmatrix}01\\10\end{Bmatrix}$	$\begin{Bmatrix}00\\11\end{Bmatrix}$	$\begin{Bmatrix}00\\11\end{Bmatrix}$
10	$\begin{Bmatrix}01\\10\end{Bmatrix}$	$\begin{Bmatrix}01\\10\end{Bmatrix}$	$\begin{Bmatrix}00\\11\end{Bmatrix}$	$\begin{Bmatrix}00\\11\end{Bmatrix}$
00	$\begin{Bmatrix}00\\11\end{Bmatrix}$	$\begin{Bmatrix}00\\11\end{Bmatrix}$	$\begin{Bmatrix}00\\11\end{Bmatrix}$	$\begin{Bmatrix}00\\11\end{Bmatrix}$
11	$\begin{Bmatrix}00\\11\end{Bmatrix}$	$\begin{Bmatrix}00\\11\end{Bmatrix}$	$\begin{Bmatrix}00\\11\end{Bmatrix}$	$\begin{Bmatrix}00\\11\end{Bmatrix}$

Using two K-maps we can determine how to implement the MAND function. Let the morphic variables be A_1 and A_2, and let A_1 be implemented as (a_{11},a_{12}) and A_2 as (a_{21},a_{22}). Then a judicious choice of implicants would lead to the following realization (shown below).



$A_1=a_{11}a_{12}$ \ $A_2=a_{21}a_{22}$	00	01	11	10
00	0	0	0	0
01	0	1	1	0
11	0	1	1	1
10	0	0	1	1

f_1



$A_1=a_{11}a_{12}$ \ $A_2=a_{21}a_{22}$	00	01	11	10
00	0	0	0	0
01	0	0	1	1
11	0	1	1	1
10	0	1	1	0

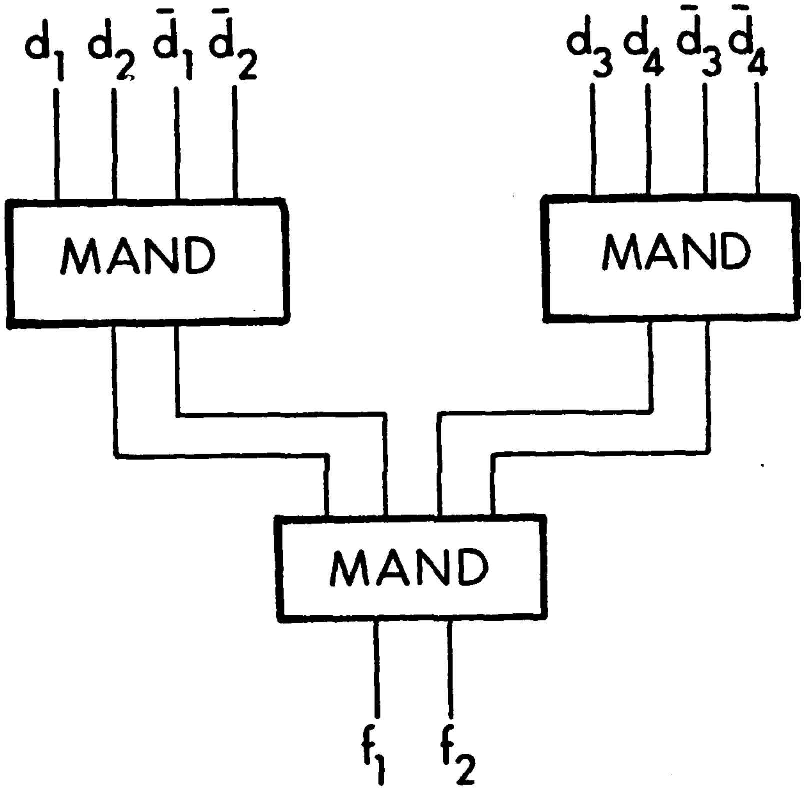
f_2

Hence $A_1 \wedge_M A_2 = ((a_{11}a_{21}+a_{12}a_{22}),(a_{11}a_{22}+a_{12}a_{21}))$.

Other functions $(\vee_M,\oplus_M)$ can be generated similarly. The implementation of MAND function shown above is not unique.

In summary a correspondence is set up between the ordinary Boolean function defined over $\{1,0\}$ and the morphic Boolean function over the state space $\{\{(e_1,e_2),(\bar{e}_1,\bar{e}_2)\},\{(e_1,\bar{e}_2),(\bar{e}_1,e_2)\}\}$.

The use of self-checking circuitry in the hardcore part of the system enhances the reliability of the error handling portion of the system. It has been shown that the Morphic Boolean function can be used as self-checking operators and can be interconnected to provide self-checing implementation of logic circuits. An example of a self-checking circuit would be the MAND function derived earlier. This operator can be used to implement a self-checking comparator as follows. If d_1, d_2, d_3, and d_4 are four signal lines, then to ensure reliability of the signal on these lines the four lines are complemented and the pairs of lines are compared using the self-checking morphic comparator as follows in Fig. 8.



d_1 d_2 $\bar{d}_1$ $\bar{d}_2$
d_3 d_4 $\bar{d}_3$ $\bar{d}_4$
MAND
MAND
MAND
f_1 f_2

Figure 8. Self-checking checker

7.0 CONCLUSION

Part II of this chapter has described the basic concepts of self-checking circuits. Self-checking circuits were defined to be circuits whose outputs are encoded in an error-detecting code. The underlying theory based on code and noncode space was developed with illustrative examples. The notions of fault-secureness and self-testing were introduced. Self-testing, partially self-testing, totally self-testing circuits, and totally self-checking networks were described and defined.

An introduction to Morphic Boolean logic as a systematic aid to the design of totally self-checking nteworks was also provided. This section forms an important theoretical basis for the design of totally self-checking computers.

8.0 REFERENCES

1. Wakerly, J., Error Detecting Codes, Self-Checking Circuits and Aplications, Elsevier North-Holland, Inc., New York, 1978.

2. Anderson, D.A., "Design of self-checking digital networks using coding techniques," Tech. Report R-527, Coordinated Science Laboratory, University of Illinois, Urbana, 1971.

3. Anderson, D.A. and Metze, G., "Design of totally self-checking check circuits for m-out-of-n codes," IEEE Trans. on Computers, vol. C-22, no. 3, pp. 263-269, March 1973.

4. Carter, W.C. and Schneider, P.C., "Design of dynamically checked computers," IFIP Conf. Proc., vol. 2, pp. 878-883, 1968.

5. Carter,W.C. et al., "Implementation of checkable acyclic automata by Morphic Boolean functions," Symp. on Computers and Automation, Polytechnic Institute of Brooklyn, pp. 465-482,April 1971.

6. Wakerly, J.F., "Partially self-checking circuits and their use in performing logical operations," IEEE Trans. on Computers, vol. C-23, no. 7, pp. 658-666, July 1974.

7. Toy, W.N., "Modular LSI control logic design with error detection," IEEE Trans. on Computers, vol. C-20, no. 2, pp. 161-166, February 1971.

8. Carter,W.C., Duke, K.A. and Jessep, D.C., "A simple self-testing decoder checking circuit," IEEE Trans. on Computers, vol. C-20, no. 11, pp. 1413-1414, November 1971.

9. Reddy,S.M., "A note on self-checking checkers," IEEE Trans. on Computers, vol. C-23, no. 10, pp. 1100-1102, October 1974.

10. Ashjaee, M.J. and Reddy, S.M ., "On totally self-checking checkers for separable codes," IEEE Trans. on Computers, vol. C-26, no. 8, pp. 737-744,August 1977.

11. Shedletsky, J.J., "A rollback interval for networks with an imperfect self-checking property," Dig. 1976 Int'l. Symp. on Fault-Tolerant Computing, pp. 163-168.

12. Marouf, M.A. and Friedman, A.D., "Efficient design of self-checking checkers for m-out-of-n codes," Dig. 1977 Int'l. Conf. Fault-Tolerant Computing, pp. 143-149.

13. Carter, W.C., Putzolu, G.R., Wadia, A.B., Bouricius, W.G., Jessep, D.C., Hsieh, F.P. and Tan, C.J., "Cost-effectiveness of self-checking computer design," Dig.1977 Int'l. Conf. Fault-Tolerant Computing, pp. 117-123, 1977.

14. Gay, F.A., "Reliability of partially self-checking circuits," Dig. 1977 Int'l. Conf. Fault-Tolerant Computing, pp. 135-142.

15. Kolupaev, S.G., "Cascade structure in totally self-checking networks," Dig. 1977 Int'l. Conf. Fault-Tolerant Computing, pp. 150-154.

PART III CODING TECHNIQUES*

1.0 INTRODUCTION

Part III of this chapter discusses the use of concurrent diagnosis techniques based on codes in digital computing systems. More specifically, it deals with a subset of such concurrent techniques, since massive redundancy methods such as circuit triplication and voting or circuit duplication (with diagnostic checks in case of disagreement) have already been considered in an earlier section (see Part I of this chapter). The two types of coding methods or concurrent diagnosis considered in this section are:

(a) partial circuit duplication using a checking algorithm to detect or correct errors.

(b) use of encoded operands such that errors can be detected by means of a subsequent checking algorithm.

The distinction between the two types of diagnosis will become more clear when we consider both separate and non-separate methods of coding.

Concurrent diagnosis means the "immediate" and "local" checking or correcting of information being transmitted or processed. The meaning of "immediate" and "local" may be debatable, but here immediate means either during a minor computation or just after it; and local will mean that the additional hardware requirements are built into the processor itself.

This section is based primarily on the papers by Kautz [1], Avizienis [2] and Armstrong [3], but additional references are cited where further information was found in various particular areas. A quick survey of the literature in the field shows a fairly heavy emphasis on the theory of codes, with little being written about the usage of such codes in practice. However, the advent of smaller and cheaper logic circuits due to the use of integrated circuit techniques and particularly VLSI (very large scale

*This section is an edited abbreviated version of an unpublished paper by G. Cole.

integration), has made the extra hardware requirements of checking techniques more feasible. Also the more complex problem solving capabilities of modern computers has made the use of error detection or correction more necessary. One can readily envision the use of computers in problems of such size that error-free solutions are otherwise impossible.

The IBM 7030 (STRETCH) computer is a notable example of a design which utilized extensive error detection and correction capabilities. The problem requirements for this machine were of such magnitude that both fast and error free computation were required. Each memory word had 8 check bits in addition to the 64 data bits for automatic correction of any single bit errors. Other operations were also checked by the use of parity checks, duplication or computation, and "casting out threes." When errors were detected, they were corrected and/or the error was recorded on a special maintenance output device. An estimated 14% of the entire computer was used solely for checking purposes [15]

2.0 TRANSMISSION CODES

For ease of presentation, one can divide the subject of error detection and correction into two groups, namely transmission codes and arithmetic codes. The transmission codes check for errors during the transmission of information, such as between processor units, during memory accesses, etc. The arithmetic codes can also detect errors in such information transfers, but more importantly, they can check the correctness of arithmetic operations. The obvious question is then, "why don't we always use arithmetic codes instead of transmission codes?" The answer is twofold; (1) the arithmetic codes often require more check bits than the transmission codes, and (2) the arithmetic codes are not nearly as well known as the transmission codes, e.g., the use of parity bit(s).

2.1 PARITY BITS

The work-horse of the error checking world is the parity bit. The use of this one extra bit to detect the presence of any single bit position error has been adopted to such an extent, that it is unusual when it is not used in information transfers. The most common type of parity check is based on the selection of a 1 or 0 for the parity bit such that the total number of 1s in the word is odd. The choice of an even number of 1s would work equally well, of course. Indeed, the use of even parity is consistent with Garner's generalized theory of parity checking in which the parity digit is the modulo b sum of the digits of the number, where b is the base of the number system [9].

The parity bit will allow the detection of any single bit error (or an odd number of errors) but will not detect any combination of two errors or any other even number). Let us consider what further detection, or perhaps even correction,.capability that we could have at the cost of additional parity bits. If instead of selection the parity bit for an odd number of 1s, we use multiple parity bits such that there are always some multiple of three 1s in each word. Such a method would require either two or three parity bits depending on the elimination of the use of an all 0s word. Even this rather trivial example illustrates the close ties between the machine design and the redundancy techniques to be used. The use of such multiple parity bits, would reduce the number of undetected error conditions, although it would not catch all combinations of two errors.

Another extension of the use of multiple parity bits is to use a matrix data configuration with parity checks on both the rows and the columns. Any single bit error will result in one row parity failure and one column parity failure. These two parity failures will represent the "coordinates" of the incorrect bit location, and therefore, will allow the correction of such a single error. Checks of this type are often made in magnetic tape data blocks, and are shown in Figure 1 for 5 rows and 6 columns.

$$\begin{array}{ccccc|c}
1 & 1 & 0 & 1 & 0 & 1 \\
1 & 0 & 0 & 0 & 1 & 0 \\
1 & 0 & 1 & 1 & 0 & 1 \\
1 & 0 & 0 & 0 & 0 & 1 \\
1 & 1 & 1 & 1 & 1 & 1 \\
0 & 1 & 0 & 0 & 1 & 0 \\
\hline
1 & 1 & 0 & 1 & 1 & 0
\end{array}$$

Figure 1: Even parity checks for matrix data.

If even parity is used for the row and column checks, the corner parity bit is useful as a "check on the checks,", i.e., the corner bit should be the proper parity bit for both the row parity bits and the column parity bits. The use of odd parity bits does not, in general, result in a correct corner check bit. The use of even parity is successful because it essentially forms the modulo two sum of all the bits in the block. This sum is the same, regardless of whether the sum is taken over the rows or over the columns. The above does bring out a subtle difference between the use of even and odd parity. The two are often considered to be (and usually are) quite interchangeable.

2.2 HAMMING CODES

Since three bits can define eight unique states, one might consider attaching three parity bits to a group of eight data bits with the idea of detecting the location of an erorr. Further investigation reveals that one of the eight states is required for the "no error" condition, and hence only seven data bits could be used. One possible assignment of the parity checks would be as shown in Figure 2.

Bit Locations

		1	2	3	4	5	6	7
Parity Bits	P_1	*		*		*		*
	P_2		*	*			*	*
	P_3				*	*	*	*

* = Bit locations which are included in the individual parity checks

P_1 P_2 P_3 1 2 3 4 5 6 7

Parity Bits Information Bit Locations

Figure 2: One possible parity check arrangement

One problem with this parity check scheme is the lack of any check on the check bits. Since a sizeable fraction of the bits are used as parity checks, there is a high probability that some of those bits would eventually cause errors rather than merely correct other errors. However, the scheme can detect the error location for any of the seven data bits, since each location is checked by a unique combination of parity checks. For example, if partiy checks #1 and #3 indicate an error but #2 does not, the error must be in bit location 5.

The Hamming code utilizes a similar parity checking scheme, except that the partiy bits are included in the group of seven protected bits. This leaves four bits for data, and the code is often called a (7,4) code indicating the total number of bits and the number of data bits. The Hamming (7,4) code is shown in Figure 3.

Figure 3 also shows the Hamming (7,4) code representation for the decimal numbers from zero to nine and demonstrates error correction by means of an example in which a ONE is lost in the 6th bit location. The parity error pattern (110) points to the location in error and hence correction can be made by simply complementing that bit. We will find later that correction in arithmetic processes is not as simple as merely complementing one bit since errors may propagate due to carries or borrows. However, for the transmission of data, the bits are considered to be independent.

The notions of "distance" and "weight" are also shown in Figure 3. Since the terms are not uniquely defined, we should call these the Hamming weight and the Hamming distance, to distinguish between the Hamming and arithmetic distance and weight. The Hamming weight is the number of non-zero digits appearing in the code symbol, and the Hamming distance is the number of digit positions in which two code symbols differ. For error detection, the Hamming distance must be at least two, while for correction of a single error or detection of two errors, the distance must be at least three. In general, the distance between any two allowable code symbols must be at least 2n-1 to correct n errors or to detect 2n errors. Richards [16] states that error detection can be traded for error correction since one cannot generally have the full amount of each. This trade-off can be seen by a simple example of a distance three code. We could detect up to two errors by such a code, but we could not distinguish between one error and two errors. Hence, we would run the risk of falsely "correcting" a bit position if we tried to correct it when two errors had actually occurred. We must either assume single errors

	Bit Locations						
	1	2	3	4	5	6	7
Parity Bits	P_1		*		*		*
		P_2	*			*	*
				P_3	*	*	*

(a) Parity bit locations and bits which are included. in each

$$P = \begin{Bmatrix} 1 & 0 & 1 & 0 & 1 & 0 & 1 \\ 0 & 1 & 1 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 \end{Bmatrix}$$

(b) Parity matrix for Hamming code

$$\underset{\text{correct word}}{1\ 0\ 1\ 1\ 0\ 1\ 1} \xrightarrow{\text{transmission}} \underset{\text{erroneous word}}{1\ 0\ 1\ 1\ 0\ \overset{\text{error}}{0}\ 1}$$

P_1 checks alright, 0

P_2 does not check, 1

P_3 does not check, 1

pattern is 110 = 6, 6 is the error location

(c) Error detection and correction

	Hamming Code
0	000 0 000
1	110 1 001
2	010 1 010
3	100 0 011
4	100 1 100
5	010 0 101
6	110 0 110
7	000 1 111
8	111 0 000
9	001 1 001

$$\begin{array}{r} 110\ 1\ 001 \\ \oplus\ \underline{010\ 1\ 010} \\ 100\ 0\ 011 \end{array}$$

weight equals the number of ONEs, equals three

(d) Hamming code representation for decimal 0 through 9

Figure 3. The Hamming (7,4) code and examples of its use.

and correct any error, or we would do no error correction but could detect any pattern of one or two bits in error. For the simultaneous correction of t or fewer errors and the detection of d or fewer errors, one needs a distance of at least t-d-1 [4].

The parity matrix of Figure 3b indicates by a 1 the bits which are involved in each parity check. Note that no distinction is made as to which bit is the actual parity bit. The arrangeement of 1s in the parity matrix is such that the pattern of partiy errors "points" to the location which is in error. Correction is by merely complementing that particular bit.

Larger Hamming coded words can be built up by the use of k check bits and $n=2^k-1$ total bits. Some of these values are listed in Table 1. Note the advantage of using long words rather than several shorter words, e.g., by coding bytes separately.

Code	Total No. of Bits, n	No. of Check Bits, k	No. of Information Bits, (n-k)
(7,4)	7	3	4
(15,11)	15	4	11
(31,26)	31	5	26
(63,57)	63	6	57

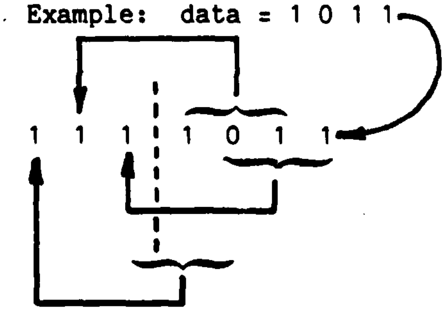
Table 1. Various Hamming Codes

2.3 CYCLIC CODES

The arrangement of the 1,0 pattern in the Hamming code parity matrix was chosen for each of locating the defective bit position. Another arrangement of interest is that of the (7,4) cyclic code, namely:

$$P = \begin{Bmatrix} 0 & 0 & 1 & 0 & 1 & 1 & 1 \\ 0 & 1 & 0 & 1 & 1 & 1 & 0 \\ 1 & 0 & 1 & 1 & 1 & 0 & 0 \end{Bmatrix}$$

Example: data = 1 0 1 1



The check digits are chosen to be the leftmost three bits and, as can be seen by the parity matrix, they check the three bits which occur after an intermediate location. A particularly simple encoding scheme for this arrangement is shown in [1].

Cyclic codes are based on the algebra of polynomials. For example, the (7,4) cyclic code is developed from a 3rd degree (7,4) polynomial generating function,

$$g(x) = 1 + x^2 + x^3$$

which is a factor of 1-x, (we use x^7 since $n = 7$). The preceding parity matrix can be found as follows [4]:

$$h(x) = (1-x) \cdot g(x) = 1+x^2+x^3+x^4$$

$$\begin{aligned} h(x) &= 1\ 0\ 1\ 1\ 1\ 0\ 0 \\ xh(x) &= 0\ 1\ 0\ 1\ 1\ 1\ 0 \\ x^2h(x) &= 0\ 0\ 1\ 0\ 1\ 1\ 1 \end{aligned} \qquad H = \left\{\begin{matrix} 0 & 0 & 1 & 1 & 1 & 0 & 1 \\ 0 & 1 & 1 & 1 & 0 & 1 & 0 \\ 1 & 1 & 1 & 0 & 1 & 0 & 0 \end{matrix}\right\}$$

The H matrix is formed from $h(x)$, $xh(x)$ and $x^2h(x)$ with the order of the elements reversed. Our P matrix is the same as H except for the reversal of the elements, since we wanted the parity bits on the left to be consistent with the previous parity and Hamming code usage.

The use of such generating polynomials has been extended to a number of such codes which can detect and correct either burst or random errors. Burst errors are errors which affect several adjacent bit positions and hence are of definite practical concern. Such codes are of special interest since considerably less redundancy is required for the correction of a given number of errors if such errors are confined to a burst [1 , 4 , 8].

Any (n,m) cyclic code can detect a burst of length n-m or less, which is a considerably larger capability than when the errors are randomly dispersed.

2.4 CODES FOR ASYMMETRIC ERRORS

Some types of logic devices may have definite failure modes which allow one to assume that all errors are 1s becoming 0s (or vice versa for other types of devices). This limitation might allow one to develop more efficient error detection and correction techniques.

One such technique is the Berger code which uses an extra k bits to represent the number of 0s in the data word (or the number of 1s). The number of bits in the check symbol is:

$$k = 1 + \log_2 m \qquad m = \text{no. of data digits}$$

The above code does not correct errors, but can detect all combinations of errors with the limitation that only 1s becoming 0s are possible (or vice versa).

2.5 FIXED WEIGHT CODES

The Hamming weight of a binary code symbol is its number of 1s. A fixed weight code has the same number of 1s in each code symbol. One such example is the "two out of five" code, in which two of the five bit positions are 1s, and the other three are 0s, such as 01010.

3.0 ARITHMETIC CODES

Those coding schemes which have been discussed in Section 2.0 are primarily for error control during the transmission of information. When one has to perform arithmetic operations on the information, those code bits do not perform any useful check and may, in some cases, make the arithmetic operations more difficult. For example, if the information is coded in the two-out-of-five code, a normal binary arithmetic unit could not be used without requiring a conversion to binary code. Some other codes, such as the use of a single parity bit or a Hamming code, allow the separation of the check bits and the information bits. However none of these codes provide a check on the arithmetic process itself.

Some codes will now be considered which provide protection during information transmission and also provide a check on the arithmetic process. As one would expect, this additional check will cost us something. In the cases of interest this cost is both additional check bits and a somewhat longer check method.

3.1 RESIDUE OPERATIONS

Before considering how residues can be used for checking in arithmetic operations, let us review a few properties of residue arithmetic.

The residue of an integer number x modulo m is the remainder of the division x/m. As examples:

26/3 = 8 with a remainder 2; 26 module 3 = 2
5/3 = 1 with a remainder 2; 5 modulo 3 = 2
17/8 = 2 with a remainder 1; 17 modulo 8 = 1

or in general;

x modulo m = r where: $x = q \cdot m + r$,
x & q are integers
m & r are positive integers

The residue can be indicated by various notations including

x modulo m
x mod m
$m|x$
$|x|_m$

The latter notation will be used in this report to indicate the residue of x modulo m.

Several residue operations are of interest in error detection and correction including the following identities (see [13] for proofs).

1. Residue of multiples of m; $|km|_m = 0$ for k = integer
   E.g.:* 15 modulo 5 = 0

2. Addition of multiples of m; $|(x+km)|_m = |x|_m$
   E.g.: (9+15) modulo 5 = 9 modulo 5 = 4

3. Addition and subtraction; $|(x=y)|_m = |(|x|_m \pm |y|_m)|_m$
   E.g.: (9±7) modulo 5= (9 modulo 5) ± (7 modulo 5) $_m$ = (4±2) modulo 5=1 or 2

4. Multiplication; $|(x\cdot y)|_m = |(|x|_m \cdot |y|_m)|_m$
   E.g.: (9·7) modulo 5=[(9 modulo 5)·(7 modulo 5)] modulo 5 = [3]
   modulo 5 = 3

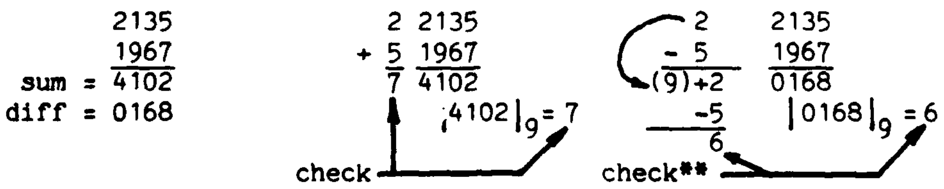
Another property of interest is the "casting out 9s" for decimal operations, or in general, casting out (b-1)s for the number base b, and n an integer (except for b=2 and n=1). For example, the residue modulo 9 of a decimal number can be found by "casting out 9s" from the digits of the number, or stated otherwise, adding the digits in modulo 9 arithmetic.

$$|19672|_9 = 1\cancel{9}67\cancel{2} = 7 \quad \{\text{casting out 9s}\}$$

$$|19672|_9 = (1+9+6+7+2) = 7.$$

3.2 THE USE OF RESIDUES FOR ARITHMETIC ERROR DETECTION

Just as the parity bit is concatenated,i.e., attached to one end of a data word, the residue could also be used as such an attached error check. For example, the residue for 1967 modulo 9 is 5,and this number could be written in error coded form as 5 1967. Suppose that we want to add two such numbers,as shown below.



[*] For examples, consider m = 5, k = 3, x = 9 and y = 7.

[**] Note that the residue can never be negative. In this example a 9 was added to the residue to make the final residue positive.

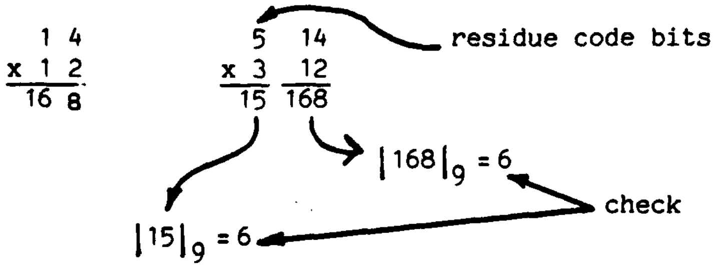
The check on the addition is made by comparing the residue of the sum with the sum (modulo 9) of the residues of the two operands. If the comparison is not "true," i.e., not the same values, then an error has occurred. If the comparision is "true," the computation is assumed to be correct. However, certain errors will not be detected when their net effect adds up to some multiple of the modulus, e.g., in the above example, if the sum were 4192 rather than 4102. The larger the modulus, the less the probability of such missed errors. Paal [11] utilized a combination of modulo 31 error coding and a repetition of the algorithm to ensure that large magnitude errors were not passed over due to this undetectable, multiple of the modulus, type of error. In the second execution of the algorithm, only 10 bit accuracy was utilized, which was adequate to ensure that the magnitude of any "missed errors" would be less than 0.1%. The use of the modulo 31 residue required that 5 check bits be used on each word, which in itself would detect some 97% of all error patterns, i.e., the fraction 30/31 of all errors.

The STAR computer [17] utilizes a residue code for instruction words in which a four bit check symbol is attached to each 28 bit address word. The check symbol is (15 modulo 15 residue), i.e., the 1s complement of the residue so that the sum of the check residue symbol and the instruction residue should be zero (represented as 1111 due to the use of the 1s complement). This approach eliminates the need for a separate comparision operation between the two residues. The residue calculation is by "casting out 15s" through the use of a four bit adder.

The residue check method of error detection can also be applied to multiplication by use of the identity

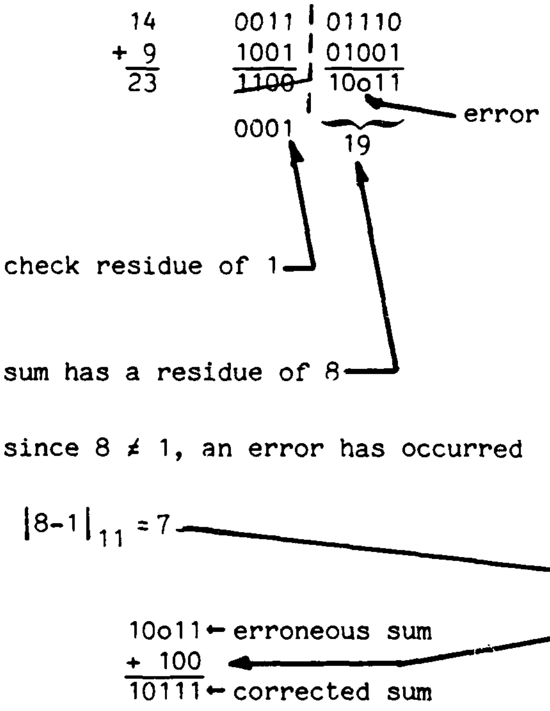
$$|x \cdot y|_m = ||x|_m \cdot |y|_m|_m$$

An example of the checking of a multiplication is:



3.3 THE USE OF RESIDUES FOR ERROR CORRECTION

In an n bit binary word there are 2n possible error "states," since each bit position can be in error by either +1 or -1. Since one additional state is required for the "no-error" condition, a total of 2n+1 states are required to be able to correct an error in the n bit word. This requirement puts a constraint on the lower bound of the modulus,namely the modulus must now be at least 2n+1 to have 2n+1 unique residue values (including zero). As an example, consider a 5 bit adder for which n = 5, and m = 2n+1 = 11.



Correction Code Table

Residue	Correction
0	no error
1	-2^0
2	-2^1
3	$+2^3$
4	-2^2
5	-2^4
6	$+2^4$
7	$+2^2$
8	-2^3
9	$+2^1$
10	$+2^0$

The error correction table used in the previous example can be built up by assuming certain errors and seeing what the resulting residue is. For example, if during the addition of zero plus zero, a 1 occurs in the 2^0 column, the residue will be 1, and the correction should be to subtract 2^0. Hence -2^0 occurs opposite the residue 1. The rest of the table can be built up in a similar manner.

One can see that as the number of bits in a data word gets longer, the number of distinct residues must also increase and hence, the modulus must become larger.

3.4 THE AN PRODUCT CODES

The basic notion of a product code for error detection is quite simple, namely, that if we multiply both integer operands in an addition (or subtraction) operation by some other integer A, then the sum (or difference) should also be an integral multiple of A. That is:

$$AX \pm AY = A(X \pm Y)$$

We can check for this "integral multiple" by a repeated division by A until the remainder is either zero or at least less than A. The remainder should be zero (if an integral multiple). We say that an error has occurred if it happens to not be equal to zero; otherwise we assume that the answer is correct even though some undetectable error may actually have occurred. We will see a little later that in some cases we can use the remainder value to determine which bit position was actually in error,and hence can have error correction as well as detection. The similarities between the AN product codes and the previously discussed residue codes can readily be seen in the above comments, and will be demonstrated again in subsequent paragraphs.

Several practical considerations influence the choice of the multiplier A, and the way in which the remainder is found. Division is a time consuming computer operation and some alternative method would be of great value in determining the remainder. We will see that a good choice for A is

$$A = b^n-1 \qquad \begin{array}{l} n = \text{some integer,} \\ b = \text{number base} \end{array}$$

A's of this form allow the remainder search to be performed by a "casting out" procedure such as the casting out 9s of decimal arithmetic checking. In binary arithmetic checking, we can cast out 3s, 7s, 15s, etc., corresponding to a choice of A equal to 3, 7, 15, etc., respectively. As we make

A larger, we reduce the probability of accepting an erroneous answer as correct, but at the cost of additional bits in the coded representation of the numbers. These relationships are:

Fraction of all errors which are detectable = $\frac{A-1}{A}$

Extra bits for the redundancy $\cong \log_2 A$.

3.5 THE AN+B CODES

The AN+B or augmented product codes are used when ease of complementing is desired, e.g., when a 9s complement representation is desired by merely complementing each bit position of the binary representation. One example of such a code is the 3N+2 code. The complementing action is shown below.

Example of 3N+2 code:

Decimal 4 becomes 3(4)+2 = 14 → 01110

Decimal 5 becomes 3(5)+2 = 17 → 10001

Note that these two numbers have complementary 1 & 0 positions and that 4 & 5 are 9s complements

The choice of the values of A and B are determined by two factors; (1) the A must be selected, as for the regular AN codes, as a relatively prime number with respect to the radix, and (2) the B must be some integer solution of the equation for the 1s complement:

$$[r^n-1]-[AN+B] = A[b-1-N]+B$$

where

n = highest exponent of r

r = radix of the number system

b = number of states that a digit can assume

$b-1-N$ = 9s complement of n (if $b = 10$)

Solving for B we obtain

$B = \frac{1}{2}[(r^n-1)-A(b-1)]$. For $A=3$, $r=2$, $b=10$, and $n=5$,

$$B = \tfrac{1}{2}[(32-1)-3(9)] = 2.$$

Hence 3N+2 is a valid solution of AN+B for these values since 2 is an integer solution for B. A 3N+2 coded BCD is shown in Table 2.

N	3N+2 BCD	N	3N+2 BCD
0	00010	5	10001
1	00101	6	10100
2	01000	7	10111
3	01011	8	11010
4	01110	9	11101

Table 2. 3N+2 BCD Code

The possible occurrence of some value of A for which one could have a self-complementing code with $B=0$ compels one to try to solve the above equation towards this goal.

$$[r^n-1]-A(b-1) = 0 \qquad r^n = Ab.$$

$$\text{For } r = 2, \quad 2^n = Ab.$$

Due to the constraint that A must be an odd number (relatively prime with the radix 2), there can be no solution for integer values of A and b. Hence, we must use an AN+B type code if we are to have the ease of complementing feature.

The added difficulty due to the plus B term is not only the nuisance of having to add it at each encoding, but also due to the need to correct each addition and subtraction which otherwise end up of the form An+2B and AN+0B respectively. Multiplication and division are also made more difficult.

3.6 SUM CODES

The sum code utilizes a separable check code with a check modulus. This check code has k bits such that

$$\text{check code} = \left|(-x)r^k\right|_\alpha$$

where x = number being coded, r = radix (2 for binary), k = number of check bits, and α = check modulus.

The sum code is placed at the right (least significant digit) part of the coded number. The original data and the check symbol are processed separately and a checking algorithm is used to check for proper residue values after the computation.

4.0 OTHER CONSIDERATIONS AND CONCLUSIONS

4.1 A SUMMARY OF CODE TERMINOLOGY

In this section we have discussed codes in which the check bits were quite distinguishable from the information bits (called separate codes) and also codes, such as the AN codes, in which the bits are nonseparable.

The parity bits of various codes are examples of "systematic" codes since there is a functional relationship between the check bits and the information bits. When such a relationship does not exist, the code is said to be non-systematic.

Errors can either be randomly dispersed throughout the word, or they may be neighboring bits. The latter are called "burst" errors and are an important practical consideration, since errors can often occur as bursts due to an interference or transient fault which affects a string of bits.

The weight of a code symbol is the number of non-zero digits in the symbol. For the "arithmetic weight" the number must be represented in minimal form, i.e., using a minimum number of digits. For example, 01111 becomes 10001 in minimal form.

The Hamming distance differs from the arithmetic distance since the Hamming distance is the (Hamming) weight of the modulo sum of the two numbers, while the arithmetic is the (arithmetic) weight of the difference of the two numbers.

4.2 FAULT PROPAGATION

There are several ways in which one fault can propagate throughout an information word causing more than one damaged bit location. One simple example is a defective carry in an addition which could propagate for some distance. However, this error is not really a multiple error since one correction, via the necessary borrows or carries could restore the correct result. In other applications, the damage pattern may not be corrected so easily. The two major ways in which a fault can propagate in a damaging manner are in byte organized processors and in multiple "cycle" algorithms, such as for multiplication and division, in parallel processors, or even for addition and subtraction in serial machines. One method of checking (and perhaps correction) would be to perform a check at the end of each byte or cycle in the algorithm. Unfortunately, this might increase the overall time for such operations (as multiplication and division) to some intolerable value. The close tie between the processor design and the type of redundancy and checking is shown by such difficulties. For a more complete description of fault propagation and the use of "error magnitudes" the reader is referred to Avizienis [2].

4.3 FURTHER STUDIES

Coding theory is a very rich and by far the most developed branch of fault-tolerant computing. For a simplified least mathematical treatment the interested reader is referred to Lin [18]. For an encyclopedic treatment the reader should consult Hamming [20]. A recent interesting work by a pioneer of coding theory is Peterson [19] which interrelates coding and information theory.

5.0 REFERENCES

1. Kautz, W.H., "Codes and coding circuitry for automatic error correction within digital systems," Redundancy Techniques for Computing Systems, Spartan Press, 1962, pp. 152-195.

2. Avizienis, A., "Concurrent diagnosis of arithmetic processors," Dig. 1st Annual IEEE Computer Conf., 1967.

3. Armstrong, D.B., "A general method of applying error correction to synchronous digital systems," Bell System Technical Journal, March 1961, pp. 459-465.

4. Peterson, W.W., Error Correcting Codes, John Wiley & Sons, New York, 1961.

5. Brown, D.T., "Error detecting and correcting codes for arithmetic operations, IRE Trans. on E.C., vol. EC-9, 1960, pp. 333-337.

6. Garner, H.L., "The residue number system," IRE Trans. on E.C., June 1959, pp. 140-147.

7. Peterson,W.W., "On checking as adder," IBM Journal, vol. 2, 1958, pp. 166-168.

8. Peterson, W.W. and Brown D.T., "Cyclic codes for error detection," Proc. of IRE, January 1981, pp. 223-235.

9. Garner, H.L., "Generalized parity checking," IRE Trans. on E.C., September 1958, pp. 207-213.

10. Garner, H.L., "Error codes for arithmetic operations," IEEE Trans. E.C., October 1966, pp. 763-770.

11. Paal, F.F., "Automatic Correction of Arithmetic Errors in Digital Computing Machines," M.S. Thesis, University of California, Los Angeles, 1966.

12. Mauriello, R., "Application of Separate Check Symbols in Arithmetic Error Detection," M.S. Thesis, University of California, Los Angeles, 1965.

13. Szabo, N.S. and Tanaka, R.I., Residue Arithmetic and its Applications to Computer Technology, McGraw-Hill, New York, 1967.

14. Kautz, W.H., "Automatic Fault Detection in Combinational Switching Networks," Stanford Research Institute, 1961.

15. Buchholz, W.H. (ed.), Planning a Computer System, McGraw-Hill, New York, 1962.

16. Richards, R.K., Arithmetic Operations in Digital Computers, Van Nostrand, New York, 1955.

17. Avizienis,A., "Design of fault tolerant computers," FJCC Proc., 1967, pp. 733-743.

18. Lin, S., An Introduction to Error-Correcting Codes, Prentice-Hall, Englewood Cliffs, New Jersey, 1980.

19. Peterson, W.W. and Weldon, E.J.,Jr., Error Correcting Codes, 2nd Ed., MIT Press, Cambridge, Massachusetts, 1972.

20. Hamming, R.W., Coding and Information Theory, Prentice-Hall,Englewood Cliffs, New Jersey, 1980.