Design and Analysis of Fault-Tolerant Digital Systems

Barry W. Johnson
University of Virginia, Charlottesville

ΒΙΒΛΙΟΘΗΚΗ Ι.Τ.Υ
LIBRARY C.T.I

ΠΑΝΕΠΙΣΤΗΜΙΟ ΠΑΤΡΩΝ
ΒΙΒΛΙΟΘΗΚΗ ΤΜΗΜΑΤΟΣ
ΜΗΧΑΝΙΚΩΝ - ΗΛΕΚΤΡΟΝΙΚΩΝ
ΥΠΟΛΟΓΙΣΤΩΝ & ΠΛΗΡΟΦΟΡΙΚΗΣ
Αρ. καταχώρησης
C.T.I LIBRARY

ddison-Wesley Publishing Company

eading, Massachusetts • Menlo Park, California • New York
on Mills, Ontario • Wokingham, England • Amsterdam • Bonn
ydney • Singapore • Tokyo • Madrid • San Juan

This book is in the ADDISON-WESLEY SERIES IN ELECTRICAL and COMPUTER ENGINEERING

Consulting Editor: Harold S. Stone

LIBRARY OF CONGRESS
Library of Congress Cataloging-in-Publication Data

Johnson, Barry W., 1957–
Design and analysis of fault-tolerant digital systems/Barry W. Johnson.
p. cm. —(Addison-Wesley series in electrical and computer engineering)
Includes bibliographies and index.
ISBN 0-201-07570-9
1. Fault-tolerant computing. 2. System design. 3. System analysis.
I. Title. II. Series
QA76.9.F38J64 1989
621.39'2—dc19 88-3955
CIP

Copyright © 1989 by Addison-Wesley Publishing Company, Inc.

All rights reserved. No part of this book may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, electronic, mechanical, photocopying, recording, or otherwise, without the prior written permission of the publisher. Printed in the United States of America. Published simultaneously in Canada.

ABCDEFGHIJ-HA-898

Foreword *by Harold S. Stone*

On October 19, 1987, the world's financial markets suffered an upheaval of a magnitude reached only once before in this century. On that day, behind the scenes at the New York Stock Exchange, a fault-tolerant computer system processed transactions at a rate deemed unimaginable by the governors of the exchange. Six hundred million shares changed hands that day, and another six hundred million the following day. The system capacity was rated at only three hundred million transactions per day—a rate believed to be unreachable for at least the next five years.

While it is somewhat amazing that the computer system successfully processed the transactions, what is more amazing is that the system performed beyond its limits in spite of sporadic failures that occurred during the day. The worst outages were limited to processing halts of a few periods of a few minutes each. During the deluge of orders, as successive waves passed through the system, orders started backing up awaiting service. Although the large backlog resulted in unusually long processing delays, transaction throughput was virtuously continuous at the maximum sustainable output rate, thus enabling the exchange to remain open at a time when market liquidity was essential to maintain public confidence in the financial system.

Why did the computer system function as well as it did? It was no accident. That computer-system design is fault tolerant. Individual disks and processors in the system can fail, but the system has redundant processors and disks that automatically take the place of units that fail. The failures that day came at a rate much higher than normal. On the whole, the failures were not actually failures of hardware components, but were system failures of one type or another caused by the excessive number of transactions being processed that day. When a disk filled, for example, requests to

store additional information on that disk were rejected. This is essentially the same response that is returned when a true hardware failure prevents a successful store operation from taking place. Thus, as various portions of the transaction system reached capacity, they rejected subsequent requests for service, and the rejection responses had much the same effect as hard failures. The apparent failure rate at the height of the transaction crunch was much higher than on a normal trading day. The massive processing load caused failures to occur precisely when the need for system availability was greatest.

Because of the inherent protection from failures in the design of the exchange's computer system, every failure point was backed up by another point. When a disk filled, transactions were routed to an alternate disk. When communications buffers filled, messages were directed to an alternate destination along an alternate route. This behavior cannot continue indefinitely, of course, because as a larger and larger fraction of system modules becomes inoperative, eventually some system module becomes the sole module of its type, and the system must shut down when this module fails. In a fault-tolerant computer system, failures force the system to reconfigure its activities. Reconfiguration pushes the system into a different operational state, possibly causing a monetary halt for some modules as well as some redoing of work that was lost because of the failure. The strategy for fault-tolerant operation of a system causes the operational state to be a stable state, in that when the system falls out of the operational state, it returns to an equivalent operational state almost immediately, unless failures have been sufficiently prevalent to prevent a return to operation.

The irony of events on October 19 is that two different systems were entwined that day—one the fault-tolerant computer system and the other the financial market system. The fault-tolerant computer system was designed for stability in the face of failure. The financial market demonstrated that it had an inherent instability. A slide in prices caused responses that day that further depressed prices and tended to exacerbate the fall rather than to correct it. On a normal trading day in an ideal market, a fall in price makes a stock more attractive to purchasers, who step in to buy the stock and thus create pressure for an upward move in price. This causes the stock price to stabilize at some level without experiencing wild gyrations in price. October 19 was not normal, however, so that downward price pressures were not compensated by upward price pressures, but instead produced responses that caused additional downward pressure. The inevitable result was a massive drop in the value of publicly held stock. So, the stable system, which was the computer system, continued to function through the day, even while the transactions reflected the inherent instability of the market system. Had the computer system been designed with instabilities equivalent to those of the market place, it would have ceased to operate early in the day. We can

only speculate about the panic that might have been triggered if nervous investors had suddenly and unexpectedly been unable to liquidate securities.

From the experience of October 19, we realize that computers have become an indispensable part of everyday life. In critical application areas, such as transaction processing for a major financial market, of which the stock exchange is only one example, a computer outage during normal operation can wreak havoc. A mild result of such an outage may be massive inconvenience, but the more serious results may include loss of life and habitat. At risk on October 19 was the economic vitality of the United States and free world.

How do we go about building fault-tolerant computers that can run without stopping? Barry Johnson sets forth the basic principles of fault-tolerant computer design in this text. Many techniques described here were used in the design of the computers that ran without stopping on October 19.

The development of the fault-tolerant techniques has paralleled the development of computers, but the wide application of the techniques lagged the development of computers by about twenty years. Von Neumann contributed to the literature on how to build reliable computers from unreliable components while he was engaged in the design and construction of the first stored-program computer. The actual practice of building fault-tolerant computers was not extensive in the early days of computing. The high cost of hardware precluded the use of extra hardware for reliability except possibly in the most critical applications. Since computers were very new, they had not found their way into many applications where reliability was a critical issue. Nevertheless, rudimentary protection from failures, in the form of parity checks, was used in the first generation of commercial systems and has been evolving ever since.

In the 1960s, the space program provided a whole new set of applications that both demanded computer implementation and forced the computers to operate in a hostile environment where maintenance is impossible. In short, fault-tolerant computers are essential in space. Reliability techniques advanced rapidly during this period, and many successful spacecraft voyages through the solar system had on-board fault-tolerant computers controlling them. Manned space flights stimulated fault-tolerant computer designs of high sophistication because the lives of the astronauts depend on the nonstop functioning of the computer system. The computer systems developed for the manned flights were highly successful and demonstrated that computer systems can be built to function over extended periods of time without maintenance even in the presence of failures.

As a classic example of such behavior, during the first lunar landing an anxious world listened intently to Neil Armstrong's reports of computer alarms, unable to comprehend their significance. Mounting tension was followed by relief and joy when Armstrong's famous words "Tranquility base

here" signaled a safe landing. The computers unexpectedly experienced a computational overload during the descent to Tranquility, but they continued to run without failure. The overload that day was quite analogous to the overload on October 19. Many of the principles put into practice in the Apollo flight found their way onto the floor of the New York Stock Exchange by 1987.

VLSI gave fault-tolerant techniques a tremendous boost. It greatly reduced the cost of redundancy and made replication of components a practical reality for all applications, not just the exotic ones. Meanwhile, with the drop in cost of computing power, computer technology became pervasive, finding applications in virtually all areas of science, business, and the arts. By the onset of the 1980s, computers had become critical components in the infrastructure of society. The stock exchange is only one of thousands of systems whose daily functioning is essential for maintaining the order of life.

One lesson of October 19 is that we cannot take the reliability of computers for granted. We must be able to design computer systems that function correctly in spite of failures. As every year brings greater reliance on computer systems, the need for fault tolerance becomes all the more important, while continual advances in device technology bring down the cost of achieving fault tolerance. This is the setting for Johnson's text.

Barry Johnson's treatment is quite thorough, carrying the reader through design techniques that describe what is possible to do, to evaluation techniques that enable the designer to determine how well the reliability goals have been achieved. Several case studies illustrate the principles as they have been put into practice. While these case studies make fascinating reading, the reader must understand that they represent the past and the present. The future will use the principles far more extensively than indicated in the case studies as new technology makes new practices feasible. But the underlying principles described in the text will remain unchanged.

To convey a better understanding of the impact of VLSI, Prof. Johnson concludes the text with material on VLSI design and testing. This technology is the enabling technology for fault tolerance, and the reader must be familiar with it to achieve the most innovative and effective fault-tolerant designs.

Today's reader who masters the material will not be the designer of a computer that guides the first lunar lander, nor of the machine that kept the transactions flowing during the crash of 1987. Today's reader, however, will be able to design machines that reach into every conceivable application, and if successful, the reader might well make a major contribution to the quality of life.

Harold S. Stone
Chappaqua, NY

Preface

Outline

The purpose of this book is to provide the reader an *introduction* to the design and analysis of fault-tolerant systems. The word *introduction* is key because it is assumed that the reader has not been exposed previously to the terminology and techniques used in the fault-tolerant computing field. The book is intended for a senior-level undergraduate course, a first-year graduate course, or as a self-study guide for those interested in learning the many facets of fault-tolerant system design. The book is suitable for a one semester course, and it assumes that the reader is familiar with combinational and sequential digital circuit design.

The key features of the book are as follows:

1. The book is introductory and can be of immediate use to individuals with no exposure to fault-tolerant computing. Consequently, the reader needs only an interest in fault tolerance and a background in basic digital circuit design to understand the material presented.
2. Each chapter is supplemented with problems that can be used as an aid to learning the material. The author has used each problem as a homework assignment or a test question in a class taught at the University of Virginia. The problems at the end of Chapter 5 are small projects that allow students to design and analyze a fault-tolerant system.
3. Each chapter concludes with a summary of the most important concepts and terms presented in the chapter. The reader can refer to the summary for a quick refresher on terminology or as a guide to the important items within the chapter.
4. Each chapter is augmented with suggested additional reading that can be useful, particularly in a graduate course, for more detailed study of

each topic. A typical graduate course will require students to read and possibly present selected articles from the additional reading sections at the end of each chapter.

5. Preliminary versions of the book have been used for the past three years in an introductory course on fault-tolerant computing taught at the University of Virginia. The use of the material in the classroom environment has stimulated better ways of presenting each topic, as well as uncovered and corrected errors in the presentation.
6. The book devotes considerable time to the design process. The various phases of a typical design process are outlined and illustrated with an example. Once again, the projects at the end of Chapter 5 provide the reader an opportunity to practice the actual design of fault-tolerant systems.
7. The book includes descriptions of 13 sample fault-tolerant systems that have been designed previously. The important problems and aspects of each design are summarized so that the reader has a feel for what others have done in the past and the historical developments in the fault-tolerant computing field.

The chapters of the book cover definitions and basic terminology (Chapters 1 and 2), fault tolerance techniques and concepts (Chapter 3), analysis procedures (Chapter 4), system design and illustrative fault-tolerant systems (Chapter 5), fault tolerance techniques in a VLSI design environment (Chapter 6), and test techniques and design for testability (Chapter 7). The material is intended for presentation in the classroom in the same order presented in the book.

Chapters 1 and 2 set the stage for the complete book by presenting historical background information and terminology that is used throughout each of the remaining chapters. The first two chapters are extremely important because they provide the background necessary to prepare the reader for studying the remainder of the book and the literature available in the fault-tolerant computing field. Definitions of such key terms as fault, error, failure, reliability, dependability, safety, and availability are presented and illustrated in the first two chapters. In addition, the various types of fault models available are described in Chapter 2. The author has attempted to use the commonly accepted definitions used by researchers and practitioners in the fault-tolerant computing field.

Chapter 3 presents a collection of techniques that can be used to design systems that are either fault tolerant or that possess the ability to detect their own faults. Various redundancy techniques including hardware, information, time, and software redundancy are presented and illustrated with simple examples. Techniques covered include, for example, majority voting, standby sparing, duplication with comparison, software- implemented self-test techniques, and error detecting and correcting codes. The intent of

Chapter 3 is to introduce the techniques; subsequent chapters provide extensive details on the use and evaluation of the various approaches.

Chapter 4 introduces various techniques for analyzing both redundant and nonredundant systems. The intent of Chapter 4 is to provide the reader with the approaches necessary to compare one or more design techniques using both quantitative and qualitative comparison methods. Reliability, availability, safety, and maintainability modeling is presented and illustrated with examples. Modeling approaches using both combinatorial and Markov models are presented. Perhaps one of the most interesting features of Chapter 4 is the inclusion of a detailed analysis example that demonstrates the process of comparing one or more design candidates for the purpose of selecting a preferred approach. The analysis example demonstrates the importance of safety in many applications in addition to the traditional metrics of reliability and availability.

Upon completion of the first four chapters, the reader should have the tools necessary to design fault-tolerant systems. The intent of Chapter 5 is to present and illustrate the various aspects of a typical design process. One unique feature of Chapter 5 is a discussion of several basic fault avoidance techniques that can be used in conjunction with fault tolerance techniques to achieve the desired attributes of a system. A second key feature of Chapter 5 is the emphasis on the importance of system analysis as an integral part of the design process. The process typically undertaken in the design of a fault-tolerant system is illustrated in Chapter 5 with the high-level design of an example flight control system for an aircraft. Chapter 5 also includes discussions of 13 fault-tolerant systems to illustrate the approaches taken by those that have actually developed fault-tolerant designs.

An important part of Chapter 5 is the projects included at the end of the chapter. Each project has been used by the author as a class project normally conducted over the last two or three weeks of a semester. The projects provide a mechanism for students to practice the process of designing a fault-tolerant system. The projects are appropriate for students to perform in small groups of two or three and can be completed to varying levels of detail. For example, in short projects students can perform only high-level designs, whereas for longer, more-detailed projects the students can complete the designs to the gate- or circuit-levels.

Chapter 6 focuses on the impact that VLSI technology has had on the design of fault-tolerant systems. Many techniques that were previously impractical are now feasible because of the decreased power consumption and increased chip density available through VLSI technology. Chapter 6 discusses the opportunities presented by VLSI, the problems presented, and techniques that can be employed on VLSI chips. Examples include complementary logic, totally self-checking logic, redundancy in array structures, and the use of redundancy to improve yield.

Finally, Chapter 7 presents test pattern generation and design for testability techniques. Testing is a crucial component of courses on fault-tolerant

systems because of the impact that redundancy can have on testing. In addition, testing and design for testability must be considered throughout the design process to guarantee that a resulting design can be thoroughly tested. Chapter 7 covers such techniques as the *D*-algorithm test pattern generation technique, scan design as a means of design for testability, and testability analyzers.

It is intended that the individual completing this book will be thoroughly prepared to pursue more advanced studies in fault-tolerant computing, research in the field, or to practice the design of fault-tolerant systems. Also, it is intended that the book will be a valuable reference for people working in the field.

Acknowledgments

The successful completion of a project such as this book requires the coordinated efforts of many individuals. I would like to thank Paul M. Julich of Harris Corporaton who first provided me the opportunity to work on the design and analysis of fault-tolerant systems. My work with Paul formed the basis of much of the material contained within this book. I would also like to thank John Hadjilogiou of the Florida Institute of Technology (FIT) who allowed me to develop and teach a course on fault-tolerant computing within the Department of Electrical and Computer Engineering at FIT. Many of the methods of presenting the material were developed while working with John.

Edward A. Parrish, Jr., of Vanderbilt University and Robert J. Mattauch of the University of Virginia deserve special recognition for providing an excellent environment in which to teach, perform research, and write. Ed Parrish, who was my department chairman when the task of writing this book was first initiated, provided encouragement, support, and excellent facilities. Bob Mattauch, as my present department chairman, has been extremely supportive of this effort and has made the task of writing and working at the University of Virginia a pleasure.

I would like to give special thanks to James H. Aylor of the University of Virginia for his support. Jim has assisted me in numerous technical issues that have solidified the concepts presented in this book. In addition, Jim first stimulated my interest in fault-tolerant computing while I was a doctoral student studying under his direction. Finally, I would like to thank Jim for being a friend and supportive colleague during the difficult times associated with writing this book.

Thanks are also due to the many students that have used preliminary versions of this book during the past three years. Approximately 75 students have used the book in various stages and have provided numerous comments that I feel have extensively improved it. The true test of a book intended to be used in teaching is whether or not the students clearly un-

derstand the materal. I feel strongly that the many constructive comments provided by my students have significantly improved this book, and for this I am very thankful.

I would also like to thank the many reviewers, especially Lee Higbie and Dhiraj K. Pradhan, that have provided excellent ideas and comments. Particular thanks is due to Harold S. Stone of IBM whose careful reading of the manuscript led to improvements in both the wording and the technical aspects of the text. In addition, Harold has provided numerous words of encouragement that have made the writing much easier.

Tom Robbins and his associates at Addison-Wesley have been excellent and deserve significant praise. I appreciate their patience and understanding when I missed my deadlines. I am also very thankful for the friendly attitudes shown toward me by everyone that I have contacted at Addison-Wesley. I cannot imagine a better publisher with which to work on a book.

I would especially like to thank my parents, Raymond and Clara Johnson for their unending support and encouragement. Without them I might never have reached a point where writing a book of this type was possible. I would also like to thank my wife's parents, Oadie and Ruth Rowland, for their continued support throughout the years that I have known them.

Finally and most importantly, I want to thank my wife, Susan, and my daughter, Ashby, who have stood by me throughout this endeaver. Without their patience, concern, and efforts, this project would not have been possible. At times during the past two years, they have carried not only their own responsibilities but many of mine as well. It is to Susan and Ashby that this book is dedicated; I hope that in some way it can repay them for all they have done for me.

Barry W. Johnson
Charlottesville, Virginia

Contents

1 Introduction 1

1.1 Overview 1
1.2 Origins of Fault-Tolerant Computing 3
1.3 Goals of Fault Tolerance 4
1.3.1 Reliability 4
1.3.2 Availability 5
1.3.3 Safety 6
1.3.4 Performability 6
1.3.5 Maintainability 7
1.3.6 Testability 8
1.3.7 Dependability 8
1.4 Applications of Fault-Tolerant Computing 8
1.4.1 Long-Life Applications 9
1.4.2 Critical-Computation Applications 10
1.4.3 Maintenance Postponement Applications 11
1.4.4 High Availability Applications 13
1.5 Fault Tolerance as a Design Objective 15
Summary 16
References 18
Additional Reading 20

2 Fundamental Definitions 23

2.1 Introduction 23
2.2 Faults, Errors, and Failures 24
2.3 Causes of Faults 28
2.4 Characteristics of Faults 30

2.5 Fault Models 31
2.5.1 The Logical Stuck-Fault Model 32
2.5.2 Transistor Stuck-Fault Models 37
2.6 Error Models 37
2.7 Design Philosophies to Combat Faults 38
Summary 40
References 42
Additional Reading 43
Problems 45

3 Design Techniques to Achieve Fault Tolerance 47

3.1 Introduction 47
3.2 Primary Design Issues 48
3.3 The Concept of Redundancy 48
3.4 Hardware Redundancy 51
3.4.1 Passive Hardware Redundancy 51
Triple Modular Redundancy 52
N-Modular Redundancy 54
Voting Techniques 54
3.4.2 Active Hardware Redundancy 62
Duplication with Comparison 63
Standby Sparing 65
Pair-and-a-Spare Technique 67
Watchdog Timers 68
3.4.3 Hybrid Hardware Redundancy 69
N-Modular Redundancy with Spares 70
Self-Purging Redundancy 71
Sift-Out Modular Redundancy 75
Triple-Duplex Architecture 78
3.4.4 Summary of Hardware Redundancy 80
3.5 Information Redundancy 81
3.5.1 Parity Codes 84
3.5.2 *m*-of-*n* Codes 93
2.5.3 Duplication Codes 95
3.5.4 Checksums 98
3.5.5 Cyclic Codes 102
3.5.6 Arithmetic Codes 112
3.5.7 Berger Codes 123
3.5.8 Horizontal and Vertical Parity 125
3.5.9 Hamming Error-Correcting Codes 127
3.5.10 Error-Correcting Integrated Circuits 131

3.5.11 Code Selection Issues 133
3.6 Time Redundancy 134
3.6.1 Transient Fault Detection 135
3.6.2 Permanent Fault Detection 136
3.6.3 Recomputation for Error Correction 151
3.7 Software Redundancy 152
3.7.1 Consistency Checks 153
3.7.2 Capability Checks 154
3.7.3 *N*-version Programming 154
Summary 155
References 159
Additional Reading 160
Problems 162

4 Evaluation Techniques 169

4.1 Introduction 169
4.2 Quantitative Evaluation Methods 170
4.2.1 Failure Rate and the Reliability Function 170
4.2.2 Failure Rate Calculation 175
4.2.3 Mean Time to Failure 178
4.2.4 Mean Time to Repair 180
4.2.5 Mean Time Between Failure 180
4.2.6 Fault Coverage 182
4.3 Reliability Modeling 185
4.3.1 Combinatorial Models 185
Series Systems 186
Parallel Systems 189
4.3.2 Fault Coverage and Its Impact on Reliability 193
4.3.3 *M*-of-*N* Systems 197
4.3.4 Markov Models 199
4.4 Safety Modeling 214
4.5 System Comparisons 216
4.6 Availability Models 219
4.7 Maintainability Models 223
4.8 Redundancy Ratios 226
4.9 Qualitative Methods 227
4.9.1 Flexibility 228
4.9.2 Technology Dependence 228
4.9.3 Transparency to the User 228
4.9.4 Testability 229
4.10 Tradeoff Analysis Example 229
Summary 254

References 256
Additional Reading 257
Problems 258

5 The Design of Practical Fault-Tolerant Systems 263
5.1 Introduction 263
5.2 The Design Process 265
5.2.1 Problem Definition 266
5.2.2 System Requirements 267
5.2.3 System Partitioning 267
5.2.4 Candidate Designs 269
5.2.5 High-Level Analysis 270
5.2.6 Hardware and Software Specifications 271
5.2.7 Hardware and Software Design and Analysis 271
5.2.8 Testing 272
5.2.9 System Integration and Test 272
5.3 The Use of Fault Avoidance in the Design Process 273
5.3.1 Requirements Design Review 274
5.3.2 Conceptual Design Review 275
5.3.3 Specifications Design Review 275
5.3.4 Detailed Design Review 276
5.3.5 Final Review 276
5.3.6 Parts Selection 276
5.3.7 Design Rules 277
5.3.8 Documentation 277
5.4 A Sample Design 277
5.4.1 Problem Definition and Initial Partitioning 279
5.4.2 Requirements Definition 280
5.4.3 System Partitioning 282
5.4.4 Candidate Designs 284
5.4.5 High-Level Analysis 292
TTMR System Analysis 292
TMR System Analysis 295
TDTMR System Analysis 297
5MR System Analysis 297
5.4.6 Comparison of Approaches 300
5.5 Sample Fault-Tolerant Systems 304
5.5.1 Long -life Applications 304
Self-Testing and Repairing Computer 305
Fault-Tolereant Spaceborne Computer 310
Fault-Tolerant Bulding Block Computer 315

Space Shuttle 319
5.5.2 Critical-Computation Applications 319
Fault-Tolerant Multiprocessor 324
Software Implemented Fault Tolerance 329
August Systems CS-3001 Control Computer 332
Multi-Microprocessor Flight Control System 333
Agusta A129 Integrated Multiplex System 336
5.5.3 High-Availability Applications 345
The Tandem 16 NonStop System 346
The Stratus/32 System 348
Electronic Switching System 350
The Synapse N+1 Architecture 353
Summary 355
References 356
Additional Reading 358
Projects 361

6 Fault-Tolerant Design of VLSI Circuits and Systems **375**
6.1 Introduction 375
6.2 VLSI Technology 376
6.3 Failure Modes in VLSI Technology 378
6.3.1 Metal Systems 379
6.3.2 Diffusion 380
6.3.3 Foreign Material 381
6.3.4 Oxide 381
6.3.5 Package and Bonding 382
6.3.6 Mounting 382
6.3.7 Misapplication 382
6.4 Distributon of Faults in VLSI Technology 383
6.5 Opportunities Presented by VLSI 385
6.6 Problems Presented by VLSI 387
6.6.1 Common-mode Failures 387
6.6.2 Increased Design Mistakes 389
6.6.3 Increased Susceptibility to External Disturbances 390
6.7 Redundancy Techniques in a VLSI Design Environment 390
6.7.1 Duplication with Complementary Logic 391
6.7.2 Self-Checking Logic 394
6.7.3 Totally Self-Checking Checkers for *M*-of-*N* Codes 402
6.7.4 Reconfiguration Array Structures 404
Fabrication-Time and Compile-Time Reconfiguration 410
Real-Time Reconfiguration 421

6.7.5 Redundancy to Enhance Yield of VLSI Circuits 439
Summary 451
References 453
Additional Reading 454
Problems 457

7 Testing 463
7.1 Introduction 463
7.2 Fault-Testing 466
7.3 Test Pattern Generation 468
7.3.1 Fault Tables 468
7.3.2 Adaptive Experiments 477
7.3.3 Boolean Differences 481
7.3.4 Literal Propositions 488
7.3.5 Path Sensitization 491
7.3.6 The D-Algorithm 498
7.3.7 Fault Simulation for Test Pattern Generation 510
The Row Method 513
The Column Method 513
7.4 Random Testing 516
7.5 Signature Analysis 517
7.6 Design for Testability 520
7.6.1 Scanning as a Method of Design for Testability 524
Level Sensitive Scan Design (LSSD) 525
Scan Path 529
Scan/Set Logic 532
Random-Access Scan 535
7.6.2 Sample Design Comparing LSSD and Scan Path 538
7.6.3 Built-In Logic Block Observation 544
7.7 Testability Analysis 552
7.7.1 Important Definitions 553
7.7.2 Testability Analyzers 554
SCOAP 555
CAMELOT 559
7.7.3 Comparison of Testability Analyzers 562
7.7.4 Analysis of Circuits Containing Redundancy 564
Summary 565
References 568
Additional Reading 570
Problems 572

1

Introduction

1.1 Overview
1.2 Origins of Fault-Tolerant Computing
1.3 Goals of Fault Tolerance
1.4 Applications of Fault-Tolerant Computing
1.5 Fault Tolerance as a Design Objective
Summary
References
Additional Reading

1.1 Overview

This textbook is devoted to the study of techniques for designing and analyzing fault-tolerant and easily-testable systems. A **fault-tolerant system** is one that can continue to correctly perform its specified tasks in the presence of hardware failures and software errors. For example, the effect of a software "bug" in a fault-tolerant system is overcome so that the system continues correct operation. Likewise, the failure of a hardware component in a fault-tolerant system does not inhibit that system's ability to correctly execute its design-specified functions. **Fault tolerance** is the attribute that enables a system to achieve fault-tolerant operation. Finally, the term **fault-tolerant computing** describes the process of performing calculations, such as those performed by a computer, in a fault-tolerant manner.

An **easily-testable system** is one whose ability to perform correctly can be verified in a simple and straightforward manner. The complexity of

today's systems demands that special features be incorporated into the system to support testing. **Design for testability** is the process by which such features are included.

The concept of fault tolerance has become increasingly important during the past decade because of the increased use of computers in the vital aspects of almost everyone's life. Computers are no longer confined to use as powerful calculators where their incorrect performance can produce little more than frustration and lost time. Instead, computers are now integrated into commercial and military aircraft flight control systems ([Wensley et al. 1978], [Hopkins et al. 1978], and [Bosch and Kuehl 1977]), industrial controllers ([Ayache, Courtiat, and Diaz 1982] and [Wensley and Harclerode 1982]), space applications [Rennels 1978], and banking systems ([Katzman 1977], [Manual 1982], and [Herbert 1983]). In each application, erroneous computer performance can be devastating to financial records, environmental safety, national security, and even human life. In summary, fault tolerance has become more important simply because the functions of computers and other digital systems have become more crucial.

This book discusses the many aspects of designing and analyzing fault-tolerant systems. In addition, design for testability methods and techniques are presented. Specific topics covered include:

1. Fundamental terminology crucial to the understanding of fault tolerance and design for testability
2. Techniques for designing fault-tolerant systems
3. The use of fault tolerance to achieve design goals such as reliability
4. Measures of the quality of a fault-tolerant design
5. Practical examples of fault-tolerant systems
6. The impact of integrated circuit technology on the design of fault-tolerant systems
7. Techniques of design for testability

We begin our discussions in this chapter by examining some of the historical aspects of fault-tolerant computing. Also, we consider in more detail the design goals that might be satisfied through the use of fault tolerance, as well as the role of fault tolerance in the design process. Several key concepts presented in this chapter include:

1. Definitions of reliability, availability, maintainability, safety, dependability, performability, and testability
2. The distinction between fault tolerance and reliability
3. The applications in which fault tolerance is most frequently used
4. The role of fault tolerance in the design process

1.2 Origins of Fault-Tolerant Computing

Fault tolerance is certainly not a new field. The first digital computers made extensive use of error detection and fault tolerance techniques to overcome the low reliability of their basic components [Carter and Bouricius 1971]. Some of the early Bell Relay Computers (BRC), for example, had two central processing units [ERA 1950]; one unit would begin executing the next instruction when the other unit encountered an error. Later versions of the BRC used a retry mechanism to repeat an operation immediately after an error was detected. The IBM 650, UNIVAC, and the Whirlwind I computers [Weik 1955] incorporated parity to check the results of data transfers. The EDVAC computer [Carter and Bouricius 1971], designed in 1949, is generally considered to have been the first computer to completely duplicate the Arithmetic Logic Unit (ALU) and compare the results obtained by each unit; the processing continued as long as the two ALUs agreed.

The advent of the transistor, along with its increased reliability, led to a temporary decrease in the emphasis on fault-tolerant computing. For many designers, the major thrust was to increase computer performance and speed and to depend on the improved reliability of the transistor to guarantee correct computations. It was not until computers began performing much more critical tasks that fault tolerance again surfaced as a crucial issue. Perhaps the best examples are in the United States space program and in military applications. The increase in computational requirements in many of these applications mandated the use of digital computers, and the significant penalties for incorrect performance required that the computers perform their functions without error. As an example, the IBM Saturn V system [Kuehn 1969] used triplicated modules and parity checking to improve the fault tolerance capability of the system.

The first theoretical work in fault-tolerant computing is generally credited to John von Neumann [von Neumann 1956]. In 1952, von Neumann presented a series of lectures on the use of replicated logic modules to improve system reliability. von Neumann later developed an article entitled "Probabilistic Logics and the Synthesis of Reliable Organisms from Unreliable Components" [von Neumann 1956] in which he presented the concept of majority voting and analyzed the impact that such arrangements could have on the probability of a system producing erroneous results.

Since about 1970, the field of fault-tolerant computing has been rapidly developing. Several excellent journals such as *Computer*, *IEEE Micro*, the *Proceedings of the IEEE*, the *Journal of Design Automation and Fault-Tolerant Computing*, and the *IEEE Transactions on Computers* regularly present special issues that deal solely with fault-tolerant computing. In addition, the International Symposium on Fault-Tolerant Computing, commonly called

the Fault-Tolerant Computing Symposium (FTCS), has been held each year, beginning as the Symposium of Fault-Tolerant Computing in 1971.

Despite the apparent progress and the prolific dissemination of information, the field of fault-tolerant computing is still relatively immature, particularly when it comes to the application of the technology. Much of this should change, however, partly because of the advent of Very Large Scale Integration (VLSI). VLSI technology has made the implementation of many fault tolerance techniques not only feasible, but in many cases, extremely practical and cost effective. VLSI, however, introduces new problems in the design of fault-tolerant systems that previously did not have to be addressed. Consequently, the advantages of VLSI do not come to us free of charge. Subsequent chapters of this book address the relationship between fault tolerance and VLSI in significant detail.

1.3 Goals of Fault Tolerance

It is natural to ask at this point why fault tolerance is so important and why it is the concern of so many designers. Fault tolerance is an attribute that is designed into a system to achieve some design goal. Just as a design must meet many functional and performance goals, it must satisfy numerous other requirements as well. The most prominent of the additional requirements are reliability, availability, safety, performability, dependability, maintainability, and testability. Fault tolerance is one system attribute capable of fulfilling such requirements. This chapter provides an overview of each requirement; Chapter 4 describes techniques that allow the evaluation of each attribute.

1.3.1 Reliability

The **reliability** $R(t)$ of a system is a function of time, defined as the conditional probability that the system will perform correctly throughout the interval $[t_0, t]$, given that the system was performing correctly at time t_0. In other words, the reliability is the probability that the system will operate correctly throughout a complete interval of time. The reliability is a conditional probability in that it depends on the system being operational at the beginning of the chosen time interval. The **unreliability** $Q(t)$ of a system is a function of time, defined as the conditional probability that a system will perform *incorrectly* during the interval $[t_0, t]$, given that the system was performing *correctly* at time t_0. The unreliability is often referred to as the *probability of failure*.

Reliability is most often used to characterize systems in which even momentary periods of incorrect performance are unacceptable, or in which

it is impossible to repair the system. If repair is impossible, such as in many space applications, the time intervals being considered can be extremely long, perhaps as many as ten years. In other applications, such as aircraft flight control, the time intervals of concern can be no more than several hours, but the probability of working correctly throughout that interval can be 0.9999999 or higher. It is a common convention when reporting reliability numbers to use 0.9_i to represent the fraction that has i nines to the right of the decimal point. For example, 0.9999999 is written as 0.9_7.

It is important to understand the difference between fault tolerance and reliability. Fault tolerance is a technique that can improve reliability, but a fault-tolerant system does not necessarily have a high reliability. A system can be designed to tolerate any single hardware failure or software error that can occur, but the probability of such problems existing can be so high that the reliability is very low. Likewise, a highly-reliable system is not necessarily fault tolerant. A very simple system might be designed using extremely good components such that the probability of the hardware failing is very low, but if the hardware does fail, the system cannot continue its functions. In other words, the system can achieve a high reliability but not possess the attribute of fault tolerance.

In summary, fault tolerance can improve a system's reliability by keeping the system operational when hardware failures and software errors occur. For example, a computing system that has redundant processors can often be designed to continue the correct performance of its tasks, even when one or more of the processors becomes inoperable.

1.3.2 Availability

Availability is another design goal that we can achieve through the use of fault tolerance. **Availability** $A(t)$ is a function of time, defined as the probability that a system is operating correctly and is available to perform its functions at the instant of time t. Availability differs from reliability in that reliability depends on an *interval* of time, whereas availability is taken at an *instant* of time. A system can be highly available yet experience frequent periods of inoperability as long as the length of each period is extremely short. In other words, the availability of a system depends not only on how frequently it becomes inoperable but also on how quickly it can be repaired. The most common measure of availability is the expected fraction of time that a system is available to correctly perform its functions.

Availability is most often used as a design goal when the system's primary purpose is to provide its services as often as possible. Examples of high-availability applications include time-shared computing systems and certain transactions processing applications, such as airline reservation systems. The users of highly-available systems want those systems to possess a

high probability of performing correctly at the instant they are requested to do so.

Fault tolerance offers numerous ways in which to improve the availability of a system. For example, the use of spare processors in a computing system can allow the functions of the system to be performed by a spare processor in the event that the primary processor becomes inoperable. In other words, the spare processor can perform the functions of the system while the primary processor is being repaired, thus keeping the system available for use.

1.3.3 Safety

One attribute that is often overlooked is the safety of a system. **Safety** $S(t)$ is the probability that a system will either perform its functions correctly or will discontinue its functions in a manner that does not disrupt the operation of other systems or compromise the safety of any people associated with the system. Safety is a measure of the *fail-safe* capability of a system; if the system does not operate correctly, you at least want the system to fail in a safe manner. For example, a pilot can safely fly an airplane, even if the autopilot fails, as long as the failure does not inhibit the aircraft's normal flight modes. Likewise, if a control valve for a chemical process fails, you often prefer that the valve fail in the closed position. Safety is the probability that these safe actions will result.

Safety and reliability differ because reliability is the probability that a system will perform its functions correctly, whereas safety is the probability that a system will either perform its functions correctly or will discontinue the functions in a manner that causes no harm. Certain techniques can be used to improve safety by turning a system off if a failure of some sort is detected. For example, in a nuclear power plant the reaction process should be stopped if some discrepancy is detected; this is the safe course of action.

1.3.4 Performability

In many cases, it is possible to design systems that can continue to perform correctly after the occurrence of hardware failures and software errors, but the *level* of performance is somehow diminished. For example, in today's era of multiprocessors, the failure of a single processor might not render the complete machine inoperable, but instead might simply decrease the speed of operation or decrease the amount of memory available to any one user. The multiprocessor, in this example, can still perform its tasks, but the relative quality of the performance has been decreased.

The **performability** $P(L, t)$ of a system is a function of time, defined as the probability that the system performance will be at, or above, some level

L at the instant of time t [Fortes and Raghavendra 1984]. If we relate performability to the multiprocessor example, the level of performance might simply be the number of processors available for computational use. Performability differs from reliability in that reliability is a measure of the likelihood that *all* of the functions are performed correctly, whereas performability is a measure of the likelihood that some subset of the functions is performed correctly.

Graceful degradation is an important feature that is closely related to performability. **Graceful degradation** is the ability of a system to automatically decrease its level of performance to compensate for hardware failures and software errors. For example, if an airplane's autopilot begins to perform incorrectly, the graceful degradation might consist of simply disabling the autopilot. The level of performance would be that coinciding with the loss of the autopilot, and the performability would be the probability of being at that level of performance at time t. Fault tolerance can provide graceful degradation and improve performability by eliminating failed hardware and software from a system, thereby allowing performance at some reduced level.

1.3.5 Maintainability

Almost every design has maintainability as a goal. **Maintainability** is a measure of the ease with which a system can be repaired, once it has failed. In more quantitative terms, maintainability $M(t)$ is the probability that a failed system will be restored to an operational state within a specified period of time t. The restoration process includes locating the problem, physically repairing the system, and bringing the system back to its operational condition. Maintainability is crucial in all systems, but it is particularly important when human lives, equipment, or the environment are placed in jeopardy while a system is repaired.

Many of the techniques that are so vital to the achievement of fault tolerance can be used to detect and locate problems in a system for the purpose of maintenance. Once the problem is located, maintenance personnel can then perform the necessary repairs. Automatic diagnostics can significantly improve the maintainability of a system because a majority of the time used to repair a system is often devoted to determining the source of the problem.

1.3.6 Testability

A **test** is a means by which the existence and quality of certain attributes within a system are determined. For example, if a computer is supposed to

execute one million instructions per second, you would probably want to design a test to verify that the computer could indeed run at that particular rate. **Testability** is the *ability* to test for certain attributes within a system. Measures of testability allow us to assess the ease with which certain tests can be performed. As we will discover in subsequent chapters, certain tests can be automated and provided as an integral part of the system to improve the testability. Many of the techniques that are so vital to achieving fault tolerance can be used to detect and locate problems in a system for the purpose of improving testability. Testability is clearly related to maintainability because it is important to minimize the time required to identify and locate specific problems.

1.3.7 Dependability

The term **dependability** encompasses the concepts of reliability, availability, safety, maintainability, performability, and testability. Dependability is the quality of service that a particular system provides [Laprie 1985]. Reliability, availability, safety, maintainability, performability, and testability are measures used to quantify the dependability of a system.

1.4 Applications of Fault-Tolerant Computing

The use of fault-tolerant computing has spread into a number of fields for several reasons. First, a better understanding of fault tolerance techniques exists today. The field has grown from a handful of researchers twenty-five years ago to the point where several companies concentrate solely on fault-tolerant systems. Examples include Tandem Computers, Stratus Computers, and August Systems. Both Tandem and Stratus are competitors in the transactions processing industry, whereas August Systems concentrates on the development of reliable and fault-tolerant systems for industrial process control. Second, the advent of Large Scale Integration (LSI) and Very Large Scale Integration (VLSI) has made many fault tolerance techniques practical for the first time. Previously, designers could barely get a single computer to fit within their size, weight, and power consumption budgets. Fault-tolerant systems that used multiple computers were simply not practical in any but the most crucial applications. Finally, many systems that were previously mechanical are now electronic because of the tremendous capability and flexibility offered by electronics. Fault tolerance is now required because the new electronic systems are often less reliable.

Existing applications of fault-tolerant computing can be categorized into four primary areas: long-life applications, critical computations, mainte-

nance postponement, and high availability. Each application presents differing design requirements and challenges.

1.4.1 Long-Life Applications

The most common examples of **long-life applications** are the unmanned space flight and satellites. The *Pioneer 10* spacecraft, for example, was launched on March 2, 1972, and became the first man-made object to pass beyond all known planets on June 13, 1983 [Lerner 1983]. During *Pioneer 10's* flight it has returned fascinating pictures of Jupiter and its moons, among other things. The information returned by *Pioneer 10* would have been severely restricted had not the electronics continued to function correctly throughout the time required for the spacecraft to reach its destination and perform its functions. The Mariner, Explorer, and Voyager missions are other examples of long-life space missions.

Satellites are also required to function correctly in space for extended periods of time. The cost of designing, building, and launching a satellite is much too high to allow electronic failures to render the satellite ineffective in space. Even though the space shuttle is now capable of retrieving satellites for repair, the cost of such repair is still extremely high, and many satellites are in orbits beyond the reach of the shuttle. Consequently, fault tolerance is required in satellite systems.

Typical requirements of a long-life application are to have a 0.95 probability of being operational at the end of a ten-year period. Unlike other applications, however, long-life systems can often allow extended outages as long as the system can eventually be made operational once again. For example, a one-week outage can be insignificant when you consider the five- or ten-year operational life of a satellite. In addition, long-life applications can frequently allow the system to be reconfigured manually by the operators. The Fault-Tolerant Space-borne Computer (FTSC) [Stiffler 1976], the Self-Testing And Repairing (STAR) computer [Avizienis et al. 1971], and the Fault-Tolerant Building Block Computer (FTBBC) [Rennels 1980] are examples of systems designed for long-life applications.

As an example of a fault-tolerant computer system intended for long-life applications, Fig. 1.1 shows a general block diagram of the electronics found on board the *Voyager* spacecraft ([Jones 1979] and [Pradhan 1986]). The system consists of eight primary elements including the flight data system, the attitude control system, the command and control system, the radio system, a telemetry modulator, a command detector, a receiver, and a tape recorder. Two identical copies of each element are provided: one copy, called the *primary* copy, performs all the operations under normal circumstances and the second copy serves as a *backup*. The system is designed so

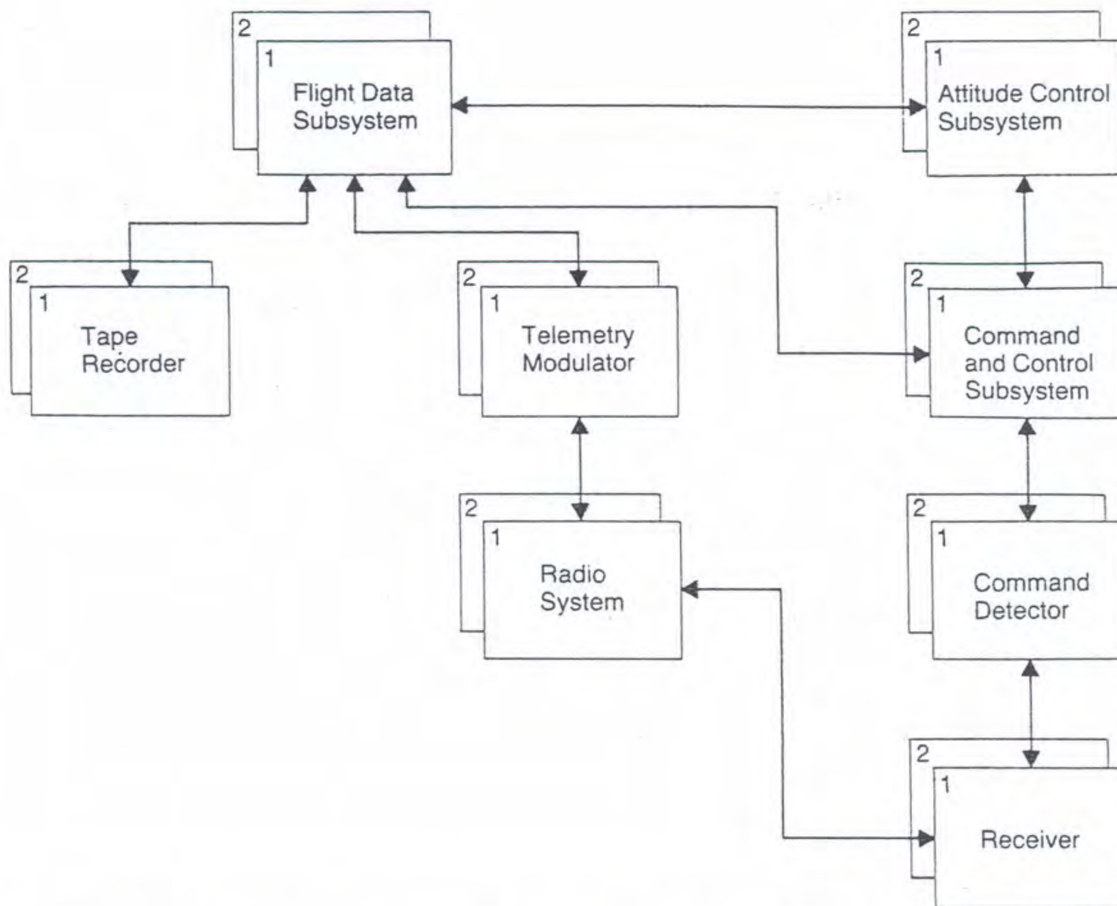


Fig. 1.1 The electronics on board the *Voyager* spacecraft achieve fault tolerance by using two identical copies of each major element.

that if the primary element fails, the backup element can be "switched in" to replace the primary and keep the system operational.

1.4.2 Critical-Computation Applications

Perhaps the most widely publicized applications of fault-tolerant computing are those in which the computations are critical to human safety, environmental cleanliness, or equipment protection. Examples include aircraft flight control systems, military systems, and certain types of industrial controllers. In **critical-computation applications,** the incorrect performance of the system will almost certainly yield devastating results. A typical requirement for a critical-computation application is to have a reliability of 0.9_7 at

the end of a three-hour period. Requirements can vary, however, depending on the particular function that the system is performing.

The most publicly visible critical-computation application of fault-tolerant computing has been the space shuttle. A malfunction in the shuttle's flight control system during either ascent or descent can result in the loss of the shuttle. Consequently, extreme care has been taken to ensure that the system performs its tasks dependably. In fact, the shuttle can continue its mandatory flight control functions after as many as three computer failures [Sklaroff 1976].

Industrial control systems also perform critical computations. For example, chemical reactions may have to be precisely controlled to prevent explosions or other unwanted effects. The goal in almost all critical-computation applications is to prevent the electronics from being the *weak point* in the system; fault tolerance is a means of accomplishing this design goal.

As an example of a fault-tolerant system used in a critical-computation application, consider the architecture of the X-29 aircraft flight control system, as shown in Fig. 1.2 [Anderson 1983]. The forward swept wing technology that the X-29 uses to maximize aerodynamic benefits requires a stabilizing control system. If the control system fails to perform correctly, the airplane will not be flyable. A hybrid analog/digital fly-by-wire flight control system is used to provide closed-loop control of the aircraft. The term *fly-by-wire* simply means that there are no mechanical connections between the pilot's stick and the control surfaces (for example, the ailerons, elevators, and rudder). Instead, an electronic system samples the position of the pilot's stick, calculates the desired position of the control surfaces, and commands a motor to move the control surfaces. The connection between the pilot's stick and the control surfaces is completely electrical; consequently, the loss of the electronics implies the loss of the ability to fly the aircraft.

As shown in Fig. 1.2, the control system uses three identical computers performing the same operations. The results from each computer are examined, and the output from the system is formed via a majority vote of the three results. Consequently, a single computer performing incorrectly will be overruled by the two computers that are performing correctly. Each computer within the system consists of both a digital computer and an analog computer. The analog computer is used as a backup that can assume the functions of the system if the digital computer fails. The analog backup provides protection against software errors that could simultaneously affect all the digital computers.

1.4.3 Maintenance Postponement Applications

Maintenance postponement applications appear most frequently when maintenance operations are extremely costly, inconvenient, or difficult to

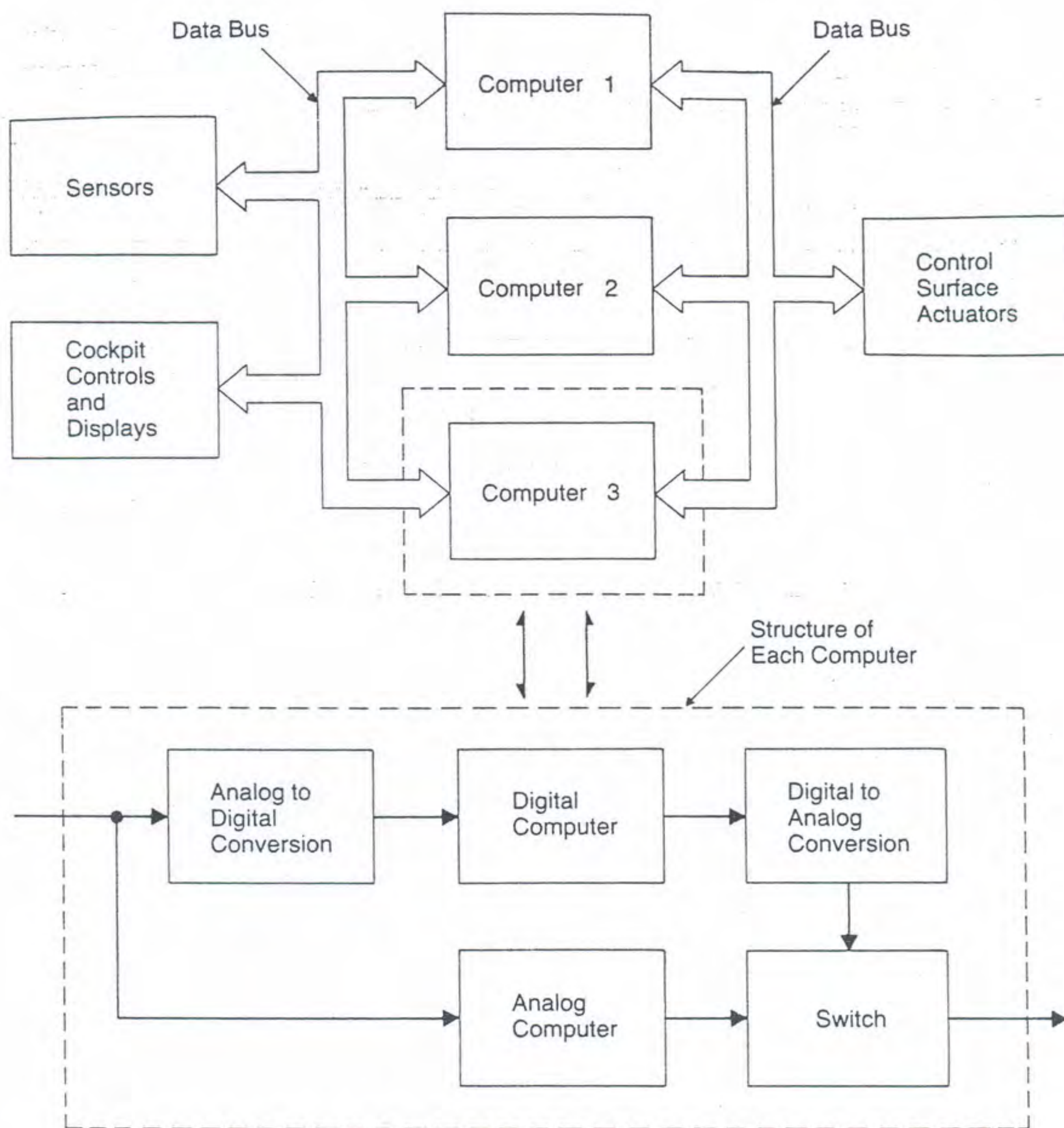


Fig. 1.2 The X-29 flight control system contains three complete computer systems. In addition, each computer system contains both an analog and a digital computer.

perform. Remote processing stations and certain space applications are good examples. In space, maintenance can be impossible to perform; at remote sites, the cost of unexpected maintenance can be prohibitive. The main goal is to use fault tolerance to allow maintenance to be postponed until a more convenient and cost-effective time. Maintenance personnel can visit a site monthly and perform any necessary repairs. Between maintenance visits, the system uses fault tolerance to continue to perform its tasks.

A telephone switching system [Toy 1978] is an example of a system that could require maintenance postponement. Many telephone switching systems are located in remote areas where it is necessary to provide telephone service, but it is costly to perform the maintenance and service operations. The primary objective is to design the system such that unscheduled maintenance can be avoided. Therefore, the telephone company can visit the facility periodically and repair the system or perform routine maintenance. Between maintenance visits, the system handles failures and service disruptions autonomously.

Figure 1.3 shows the block diagram of the 3B20D processor used in the Electronic Switching System (ESS) developed by AT&T Bell Laboratories [Serlin 1984]. Each element of this system is completely duplicated. One set of elements can be used to perform all the system functions, whereas the duplicate set serves as a backup in the event of a hardware failure. The duplicate set of elements can allow the system to remain functional while waiting for a repair to occur. Note in Fig. 1.3, for example, that either processor can access either storage disk, so the failure of a disk does not render the system inoperable.

1.4.4 High-Availability Applications

Availability is rapidly becoming a key parameter in many applications. Banking and other time-shared systems are good examples of **high-**

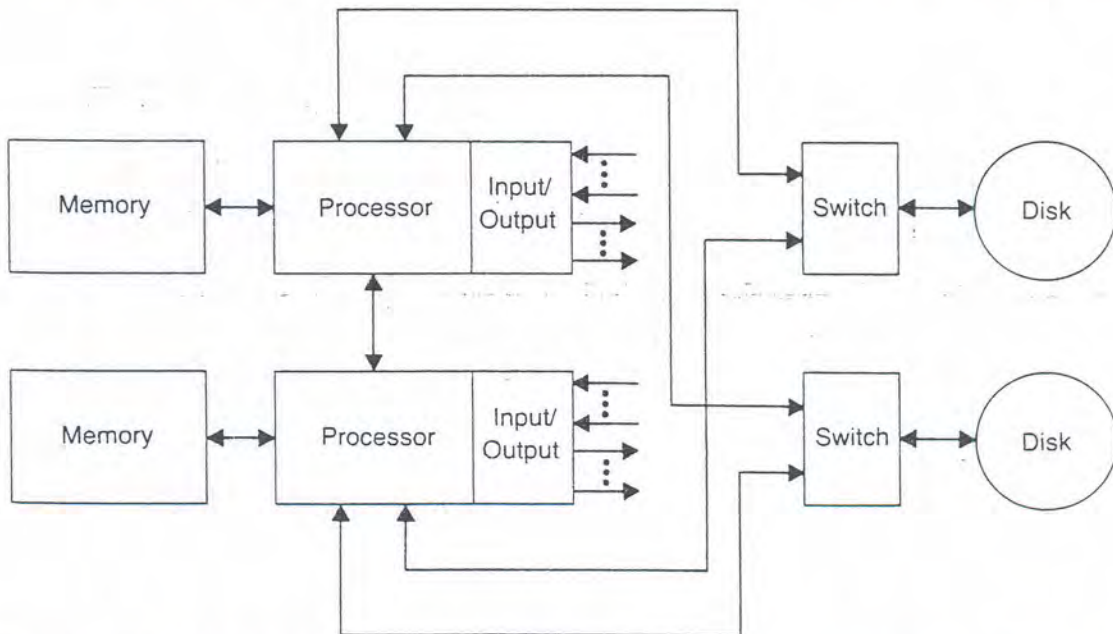


Fig. 1.3 Block diagram of the 3B20D processor used in the Bell Electronic Switching System. All critical components in this system are duplicated. (From [Serlin 1984] © 1984 IEEE)

availability applications. Users of these systems want to have a high probability of receiving service when it is requested. The Tandem NonStop transaction processing system [Katzman 1977] is a good example of one designed for high availability. A major competitor of the Tandem computer is the Stratus system [Herbert 1983]. Both the Tandem and the Stratus computers are designed to achieve a high probability of being operational when their services are required.

Intel's 432 processor system [Siewiorek 1982] is an example design developed to support high availability in many general-purpose processing applications. Intel's 432 system employs a number of techniques that support fault-tolerant operation. For example, Fig. 1.4 shows the structure of the central processing unit (CPU) and illustrates how two CPUs can be operated as a pair. If CPU 1 in Fig. 1.4 is enabled, the system's outputs will come from CPU 1, and CPU 2 will check those outputs with its own. Similarly, CPU 2 could be enabled, and CPU 1 would serve as the checker.

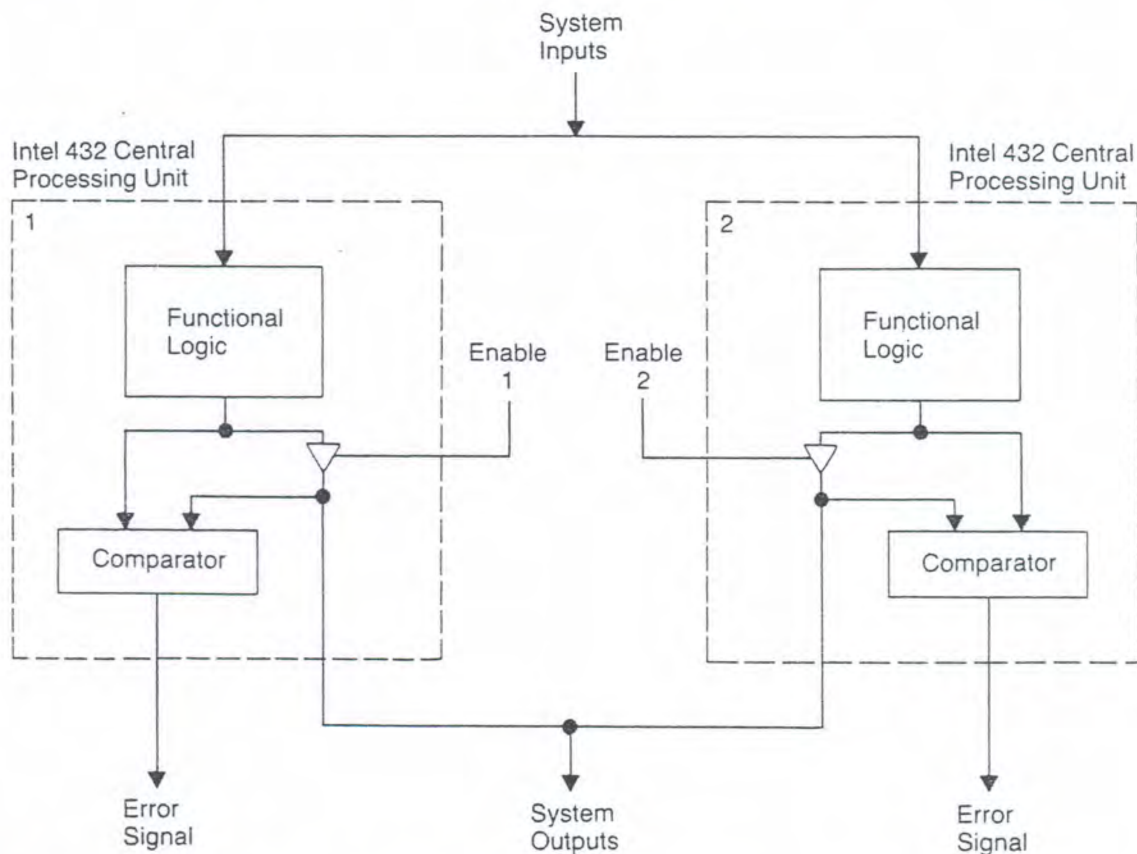


Fig. 1.4 The Intel 432 Central Processing Unit (CPU) supports operating two CPUs as a pair. The outputs of each CPU are compared. (From [Serlin 1984] © 1984 IEEE)

1.5 Fault Tolerance as a Design Objective

The design of a system is often conducted with many goals in mind. Certainly, we want the system to perform its intended function, but we also want the system to be dependable, cost effective, efficient, easy to test, and easy to repair. To accomplish our design goals, we begin with specific requirements. We may mandate that the cost and power not exceed certain values. Likewise, our design must meet specific reliability, availability, or maintainability goals. Fault tolerance plays an important role in attaining these goals, but it is not the only key part of the design process. Fault tolerance is a means of achieving our goals, but it must be coupled with other design techniques to be successful.

Figure 1.5 shows a top-level view of the design process. The system requirements, such as reliability, are achieved through two primary means: system design and system evaluation. The system design includes both fault avoidance and fault tolerance techniques. **Fault avoidance** techniques are performed to help prevent hardware failures and software errors. Examples include selecting high-quality components, enforcing design rules, and reviewing the designs periodically. Fault tolerance techniques, on the other hand, handle hardware failures and software errors when they occur. Examples include the use of redundant hardware, voting, and reconfiguration techniques.

The evaluation of a system is often overlooked as an integral part of the design process. Evaluation must be used in parallel with the design process, if the design is to be successful. System evaluation can uncover problems with a design early enough to allow corrections to be implemented. For example, if a design problem is discovered before the design is committed to hardware, the problem usually can be corrected easily. However, if problems remain in a system after it is built, correction can be impossible, or significant performance degradations may have to be accepted to allow the correction to be made. Numerous evaluation methods are available to analyze systems. Examples include Markov reliability models, system repair models, combinatorial reliability models, availability models, and maintainability models. In addition, evaluation techniques allow us to locate areas within a system that are prone to failure or where failure can be catastrophic. Each evaluation technique is crucial to a quality system design.

The primary purpose of this textbook is to study fault avoidance, fault tolerance, and system evaluation techniques that can be used in the design and analysis of fault-tolerant systems. This book presents the techniques that are available to achieve fault tolerance. More importantly, however, this book shows practical examples of how fault tolerance can be used to achieve design goals such as reliability, availability, and maintainability.

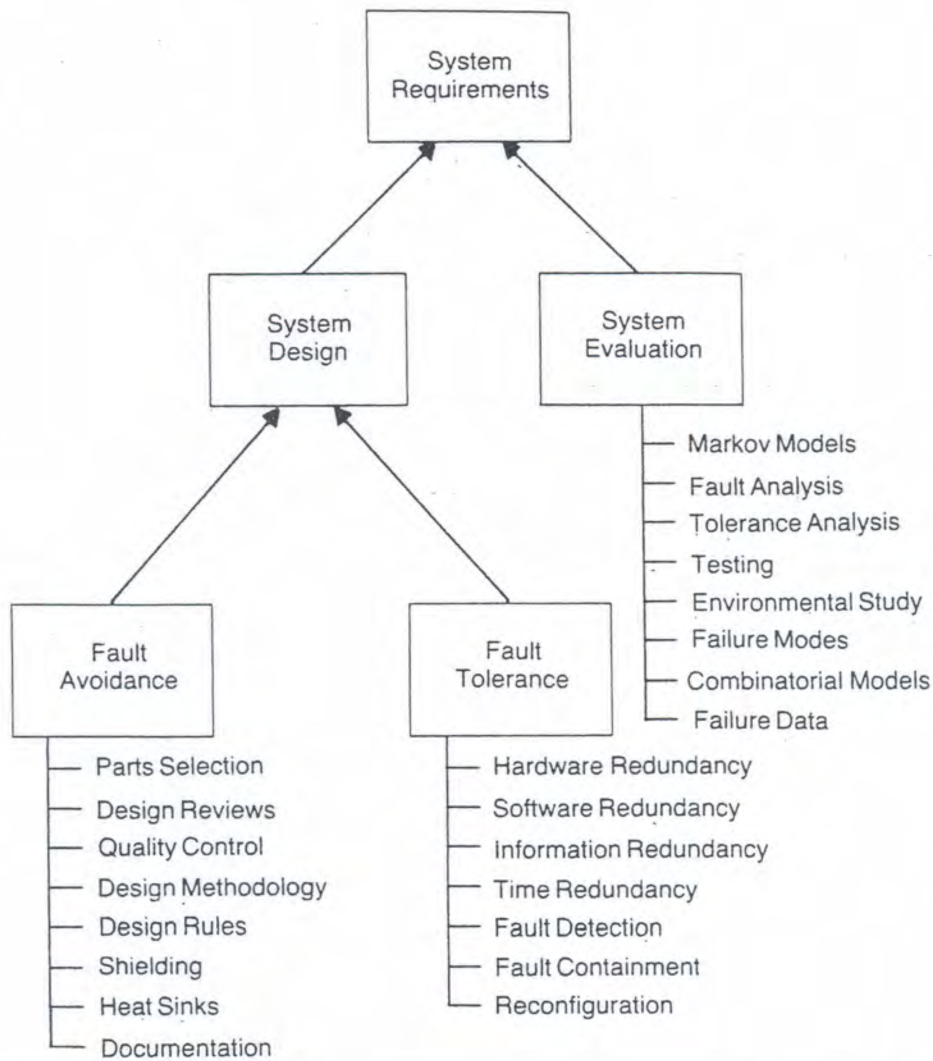


Fig. 1.5 A top-level view of the system design process illustrating the importance of fault avoidance, fault tolerance, and system evaluation.

Summary

This chapter has introduced the basic concept of fault tolerance and has illustrated the design goals often achieved via fault tolerance. The applications that require fault tolerance have been discussed, and examples from each application area have been presented. Perhaps the most important concept presented in this chapter is that fault tolerance is one aspect of a system design. To achieve design goals such as reliability or availability, a designer must use not only fault tolerance but fault avoidance and system evaluation as well.

The following list summarizes the important terminology and concepts that have been presented in this chapter.

Availability $A(t)$—the probability that a system is operating correctly and is available to perform its functions at the instant of time t.

Critical-Computation Application—an application in which the incorrect performance of computations can create devastating results.

Dependability—the quality of service provided by a particular system.

Design for Testability—the process of including special features to make a system easily testable.

Easily-Testable System—a system whose ability to perform correctly can be verified in a simple and straight forward manner.

Fault Avoidance—the process of attempting to prevent hardware failures and software errors from occurring in a system.

Fault-Tolerant Computing—the process of performing calculations, such as those performed by a computer, in a fault-tolerant manner.

Fault-Tolerant System—a system that can continue the correct performance of its specified tasks in the presence of hardware failures and software errors.

Fault Tolerance—the quality or attribute that enables a system to behave in a fault-tolerant manner.

Graceful Degradation—the ability of a system to automatically decrease its level of performance to compensate for hardware failures and software errors.

High-Availability Application—an application in which availability is the crucial design requirement.

Long-Life Application—an application in which the longevity of operation is the crucial design requirement.

Maintainability, $M(t)$—the probability that an inoperable system will be restored to an operational state within the time t.

Maintenance Postponement Application—an application in which it is desired to delay the process of repairing a system until the most convenient times.

Performability, $P(L, t)$—the probability that a system is performing at or above some level of performance L at the instant of time t.

Reliability, $R(t)$—the conditional probability that a system performs correctly throughout an interval of time $[t_0, t]$, given that the system was performing correctly at time t_0.

Safety, $S(t)$—the probability that a system will either perform its functions correctly or will discontinue its functions in a well-defined, safe manner.

Test—a means by which the existence and quality of certain attributes within a system is determined.

Testability—the ability to test for certain attributes within a system.

Unreliability, $Q(t)$—the conditional probability that a system will perform *incorrectly* during the interval of time $[t_0, t]$, given that the system was performing *correctly* at time t_0.

References

1. Anderson, D. "X-29 Forward swept wing flight control system," *Proceedings of the Joint AIAA-IEEE Fifth Digital Avionics Systems Conference*, Washington, D.C., December 1984, pp. 1–8.
2. Avizienis, A., Gilley, G.C., F.P. Mathur, D.A. Rennels, J.A. Rohr, and D.K. Rubin. "The STAR (Self-Testing And Repairing) computer: An investigation of the theory and practice of fault-tolerant computer design," *IEEE Transactions on Computers*, Vol. C-20, No. 11, November 1971, pp. 1312–1321.
3. Ayache, J., J. Courtiat, and M. Diaz. "REBUS: A fault tolerant distributed system for industrial real-time control," *IEEE Transactions on Computers*, Vol. C-31, No. 7, July 1982, pp. 637–647.
4. Bosch, J.A., and W.J. Kuehl. "Reconfigurable redundancy management for aircraft flight control," *Journal of Aircraft*, Vol. 14, No. 10, October 1977, pp. 966–971.
5. Carter, W.C., and W.G. Bouricius. "A survey of fault tolerant computer architecture and its evaluation," *Computer*, Vol. 4, No. 1, January 1971, pp. 9–16.
6. Engineering Research Association. *High Speed Computing Devices*, McGraw-Hill, 1950.
7. Fortes, J.A.B., and C.S. Raghavendra. "Dynamically reconfigurable fault-tolerant array processors," *Proceedings of the 14th International Conference on Fault-Tolerant Computing*, Kissimmee, Fla., June 20–22, 1984, pp. 386–392.
8. Herbert, E. "Computers: Minis and mainframes," *IEEE Spectrum*, Vol. 20, No. 1, January 1983, pp. 28–33.
9. Hopkins, A.L., T.B. Smith, and J.H. Lala. "FTMP: A highly reliable fault-tolerant multiprocessor for aircraft," *Proceedings of the IEEE*, Vol. 66, No. 10, October 1978, pp. 1221–1239.
10. Jones, C.P. "Automatic fault protection in the Voyager spacecraft," AIAA Paper No. 79–1919, American Institute of Aeronautics and Astronautics.
11. Katzman, J.A. "System architecture for nonstop computing," *Proceedings of the 14th Computer Society International Conference (Compcon)*, San Francisco, February 1977, pp. 77–80.

12. Kuehn, R.E. "Computer redundancy: Design, performance, and future," *IEEE Transactions on Reliability*, Vol. R-18, No. 1, February 1969, pp. 3–11.
13. Laprie, J.C. Dependable computing and fault tolerance: Concepts and terminology, *Proceedings of the 15th Annual International Symposium on Fault-Tolerant Computing*, June 19–21, 1985, Ann Arbor, Michigan, pp. 2–11.
14. Lerner, E.J. "Crossroads in space," *Spectrum*, Vol. 20, No. 9, September 1983, pp. 28–55.
15. Manual, T. "New architecture cuts redundancies in fail-safe processing," *Electronics*, Vol. 55, No. 17, August 25, 1982, pp. 45–46.
16. Pradhan, D.K. *Fault-Tolerant Computing Theory and Techniques*, Prentice-Hall, Englewood Cliffs, N.J., 1986.
17. Rennels, D.A. "Architectures for fault tolerant spacecraft computers," *Proceedings of the IEEE*, Vol. 66, No. 10, October 1978, pp. 1255–1268.
18. Rennels, D.A. "Distributed fault-tolerant computer systems," *IEEE Computer*, Vol. 13, No. 3, March 1980, pp. 55–64.
19. Serlin, O. "Fault-tolerant systems in commercial applications," *Computer*, Vol. 17, No. 8, August 1984, pp. 19–30.
20. Siewiorek, D.P., and R.S. Swarz. *The Theory and Practice of Reliable System Design*, Digital Press, Bedford, Mass., 1982.
21. Sklaroff, J.R. "Redundancy management technique for the space shuttle computers," *IBM Journal of Research and Development*, Vol. 20, No. 1, January 1976, pp. 20–28.
22. Stiffler, J.J. "Architectural design for near-100% fault coverage," *Proceedings of the International Symposium on Fault Tolerant Computing*, 1976, pp. 134–137.
23. Toy, W.N. "Fault-tolerant design of local ESS processor," *Proceedings of the IEEE*, Vol. 66, No. 10, October 1978, pp. 1126–1145.
24. von Neumann, J. "Probabilistic logics and the synthesis of reliable organisms from unreliable components," *Automata Studies, Annals of Mathematical Studies*, Princeton University Press, No. 34, pp. 43–98, 1956.
25. Weik, M.H. "A survey of domestic electronic digital computing systems," Report #971, Commerce Department, Ballistic Research Laboratories, Aberdeen Proving Grounds, Md., December 1955.
26. Wensley, J.H., L. Lamport, J. Goldberg, M.W. Green, K.N. Levitt, P.M. Melliar-Smith, R.E. Shostak, and C.B. Weinstock, "SIFT: Design and analysis of a fault tolerant computer for aircraft control," *Proceedings of the IEEE*, Vol. 66, No. 10, October 1978, pp. 1240–1255.
27. Wensley, J.H., and C.S. Harclerode. "Programmable control of a chemical reactor using a fault tolerant computer," *IEEE Transactions on Industrial Electronics*, Vol. IE-29, No. 4, November 1982, pp. 258–264.

Additional Reading

For the reader interested in learning more about the history and development of fault-tolerant computing, the following list of suggested references is provided. This list primarily includes tutorial and survey articles that give the reader a good feel for the development of the technology, terminology, and applications of fault-tolerant computing.

Avizienis, A. "Fault tolerance: The survival attribute of digital systems," *Proceedings of the IEEE*, Vol. 66, No. 10, October 1978, pp. 1109–1125.

Avizienis, A. "Fault tolerant computing: An overview," *Computer*, Vol. 4, No. 1, January 1971, pp. 5–8.

Avizienis, A. "Architecture of fault-tolerant computing systems," *Proceedings of the 1975 International Symposium on Fault-Tolerant Computing*, Paris, June 1975, pp. 3–16.

Baechler, D.O. "Aerospace computer characteristics and design trends," *Computer*, Vol. 4, No. 1, January/February 1971, pp. 45–57.

Cooper, A.E., and W.T. Chow. "Development of on-board space computer systems," *IBM Journal of Research and Development*, Vol. 20, No. 1, January 1976, pp. 5–19.

Deyst, J.J., Jr., J.V. Harrison, E. Gai, and K. C. Daly. "Fault detection, identification, and reconfiguration for spacecraft systems," *Journal of Astronautical Science*, Vol. 29, No. 2, April-June 1981, pp. 113–126.

Goldberg, J. "New problems in fault-tolerant computing," *Proceedings of the 1975 International Symposium on Fault-Tolerant Computing*, Paris, 1975, pp. 29–34.

Harris, R.L., and E.E. Jones. "Fault tolerance applications to future military system avionics," *Proceedings of the IEEE 1980 National Aerospace and Electronics Conference*, Dayton, Oh., May 1980.

Hecht, H. "Fault tolerant computers for spacecraft," *Journal of Spacecraft*, Vol. 14, No. 10, October 1977.

Hopkins, A.L., Jr. "Design foundations for survivable, integrated, on-board computation and control," *Proceedings of the 1977 Joint Automatic Control Conference*, San Francisco, Calif., June 22–24, 1977, pp. 232–237.

Hopkins, A.L., Jr. "Fault tolerant system design: Broad brush and fine print," *Computer*, Vol. 13, No. 3, March 1980, pp. 39–46.

Jennings, R. "Fault secure avionic system development," *Proceedings of the 1981 International Aerospace and Electronics Conference*, Vol. 1, Dayton, Oh., May 9–11, 1981.

Kime, C.R. "Fault tolerant computing: An introduction and a perspective," *IEEE Transactions on Computers*, Vol. C-24, No. 5, May 1975, pp. 457–460.

Koczela, L.J., and G.J. Burnett. "Advanced space missions and computer systems," *IEEE Transactions on Aerospace and Electronic Systems*, Vol. AES-4, No. 3, May 1968, pp. 456–467.

Nelson, V.P., and B.D. Carroll. *Tutorial: Fault-Tolerant Computing*, IEEE Computer Society Press, Washington, D.C., 1986.

Ramamoorthy, C.V. "Fault tolerant computing: An introduction and an overview," *IEEE Transactions on Computers*, Vol. C-20, No. 11, November 1971, pp. 1241–1244.

Rennels, D.A. "Fault-tolerant computing—Concepts and examples," *IEEE Transactions on Computers*, Vol. C-33, No. 12, December 1984, pp. 1116–1129.

Rennels, D.A. "Reconfigurable modular computer networks for spacecraft onboard processing," *Computer*, Vol. 11, No. 7, July 1978, pp. 49–59.

Short, R., and J. Goldberg. "A summary of Soviet activities in the design of fault tolerant digital machines," *Computer*, Vol. 11, No. 1, January/February 1971, pp. 28–33.

Short, R., and J. Goldberg. "A survey of Soviet activities in the design of fault tolerant digital machines," *IEEE Transactions on Computers*, Vol. C-20, No. 11, November 1971, pp. 1337–1352.

Siewiorek, D.P., D.E. Thomas, and D.L. Scharfetter. "The use of LSI modules in computer structures: Trends and limitations," *Computer*, Vol. 11, No. 7, July 1978, pp. 16–25.

Teschler, L. "Computers that won't fail," *Machine Design*, Vol. 50, No. 10, May 11, 1978, pp. 91–97.

2

Fundamental Definitions

2.1 Introduction
2.2 Faults, Errors, and Failures
2.3 Causes of Faults
2.4 Characteristics of Faults
2.5 Fault Models
2.6 Error Models
2.7 Design Philosophies to Combat Faults
Summary
References
Additional Reading
Problems

2.1 Introduction

Throughout the history of fault-tolerant computing, there has been substantial disagreement on the definitions of several key concepts. For example, the terms *fault*, *failure*, and *error* have often been used interchangeably in the literature. To many people, a failure has occurred when the time-shared computer they are using fails to respond to their requests or demands. To other people, a failure is a more specific physical defect within some electronic component. Some groups view the physical defects as faults instead of failures.

The purpose of this chapter is to introduce the basic terminology used in the fault-tolerant computing field. It is important to understand the causes of faults and the types of faults that can occur before considering the

techniques that are available to tolerate faults in a design. In addition, it is vital to have a clear understanding of how faults manifest themselves in digital systems. In other words, once a fault has occurred, how is it likely to propagate throughout the system and what are the probable impacts of that fault? Finally, we discuss the role of fault tolerance and fault avoidance in combating faults in digital systems. Perhaps the best available references on the fault tolerance terminology are [Avizienis 1982], [Laprie 1985], and [Johnson 1984].

2.2 Faults, Errors, and Failures

Three fundamental terms in fault-tolerant design are fault, error, and failure. There is a cause-and-effect relationship between faults, errors, and failures. Specifically, faults are the cause of errors, and errors are the cause of failures.

A **fault** is a physical defect, imperfection, or flaw that occurs within some hardware or software component. Essentially, the definition of a fault, as used in the fault tolerance community, agrees with the definition found in the dictionary. A fault is a blemish, weakness, or shortcoming of a particular hardware or software component. Examples of faults include shorts between electrical conductors, opens or breaks in conductors, or physical flaws or imperfections in semiconductor devices. Similarly, in software, an example of a fault is a program loop that when entered can never be exited.

An **error** is the manifestation of a fault. Specifically, an error is a deviation from accuracy or correctness. For example, suppose that a physical short results in a line within a circuit being permanently stuck at a logic 1. The physical short is a fault within the circuit. If some condition occurs that requires the line to transition to a logic 0, the value on the line will be in error. In other words, the correct value for the line will be logic 0, but the existence of the fault has caused the line to have an erroneous value. In other words, an error is the result of a fault.

Finally, if the error results in the system performing one of its functions incorrectly, a system failure has occurred. Essentially, a **failure** is the nonperformance of some action that is due or expected. Although it is often used interchangeably with the term **malfunction,** the term *failure* is rapidly becoming more commonly accepted. A failure is also the performance of some function in a subnormal quantity or quality. As an example, suppose that a line in a circuit is responsible for turning a valve on or off: a logic 1 turns the valve on and a logic 0 turns the valve off. If the line is stuck at logic 1, the valve is stuck on. As long as the user of the system wants the valve on, the system will be functioning correctly. However, when the user wants to turn the valve off, the system will experience a failure.

Figure 2.1 illustrates the cause-and-effect relationship between faults, errors, and failures. Faults result in errors, and errors can lead to system failures. One way to think of Fig. 2.1 is as a hierarchy. At the bottom of the hierarchy are faults. Errors are the effect of faults, and, finally, failures are the effect of errors.

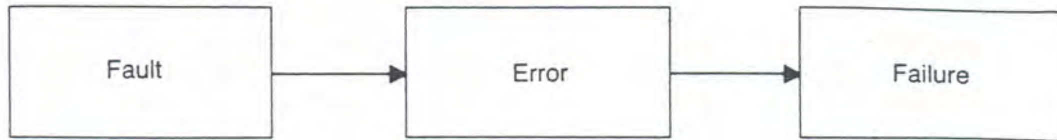


Fig. 2.1 Relationship between faults, errors, and failures. Failures are caused by errors, which are caused by faults.

The circuit shown in Fig. 2.2 further illustrates the distinction between faults and errors. The circuit of Fig. 2.2 is the logic diagram for a full-adder. The inputs A_1, B_1, and C_1 are the two bits of the operands and the carry bit, respectively. The truth table that shows the correct performance for this circuit is presented in Fig. 2.3. If a short occurs between line L and the power supply line resulting in line L becoming permanently fixed at a logic 1 value, a fault will have occurred. The fault is the actual short within the circuit.

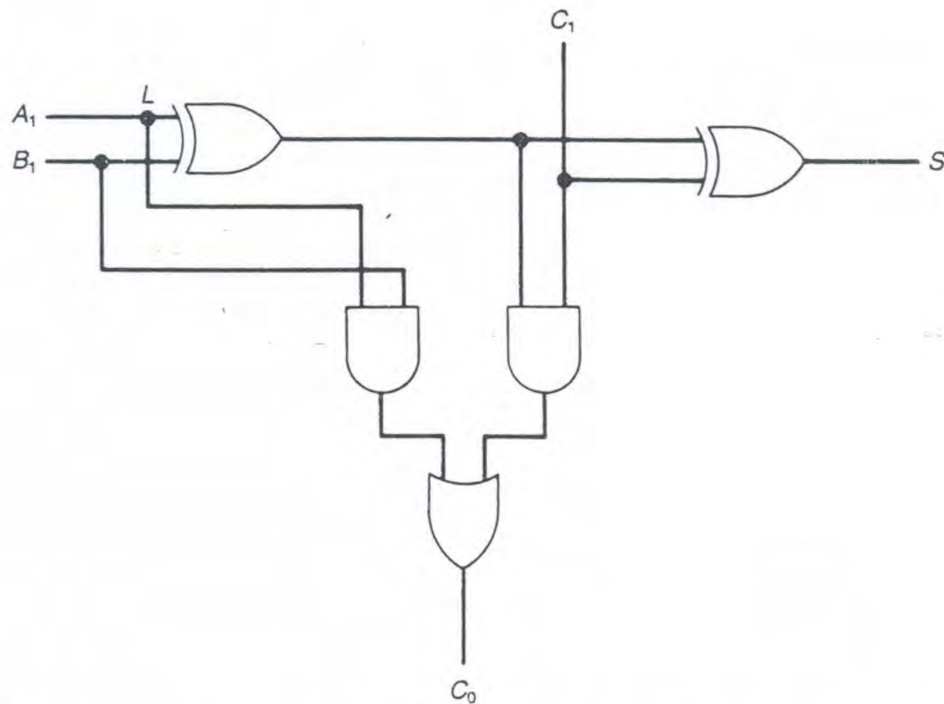
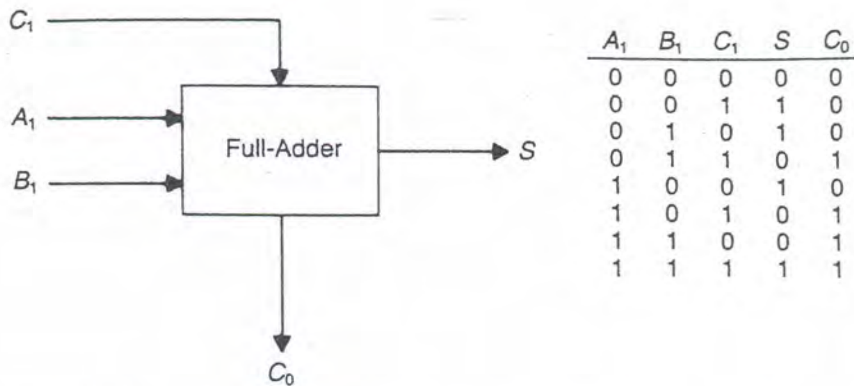


Fig. 2.2 Full-adder circuit to illustrate the difference between faults and errors.

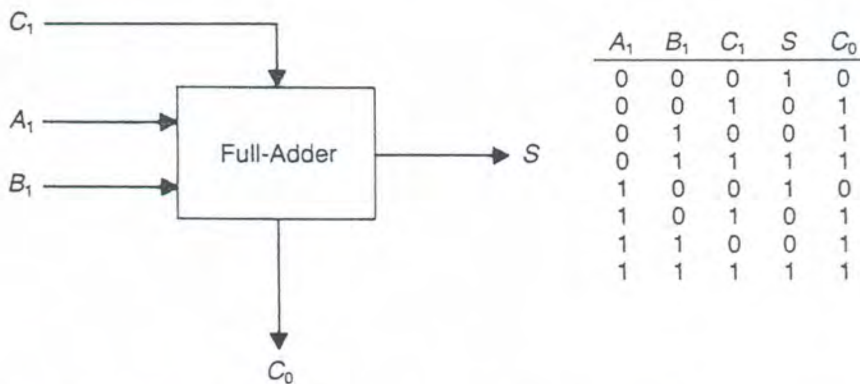


A_1	B_1	C_1	S	C_0
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

Fig. 2.3 Truth table for the fault-free full-adder circuit.

Figure 2.4 shows the truth table of the circuit that contains the physical short. By comparing the truth tables in Figs. 2.3 and 2.4, we can see that the circuit performs correctly for the input combinations 100, 101, 110, and 111, but not for 000, 001, 010, and 011. The physical short within the circuit is a fault. Whenever an input is applied that results in the circuit producing an incorrect output, an error has occurred. If the output of the circuit is controlling a relay and the relay is opened when it should be closed, a failure has occurred.

The concepts of faults, errors, and failures can be best presented by using a three-universe model, which is an adaptation of the four-universe model originally developed in [Avizienis 1982]. The first universe is the **physical universe** in which faults occur. The physical universe contains the semiconductor devices, mechanical elements, displays, printers, power supplies, and other physical entities that make up a system. A fault is a physical defect or alteration of some component within the physical universe.



A_1	B_1	C_1	S	C_0
0	0	0	1	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

Fig. 2.4 Truth table for the full-adder circuit with an example fault.

The second universe is the **informational universe.** The informational universe is where the error occurs. Errors affect units of information such as data words within a computer or digital voice or image information. Terms that are applicable to the informational universe include parity errors, message errors, typographical errors, and bit errors. An error has occurred when some unit of information becomes incorrect.

The third universe is the **external universe** (or **user's universe**). The external universe is where the user of a system ultimately sees the effect of faults and errors. The external universe is where failures occur. The failure is any deviation that occurs from the desired or expected behavior of a system. For example, the user may expect a system to correctly print payroll checks. If, for some reason, a check is printed incorrectly, the user has witnessed a failure of the system, and the failure has been witnessed in the external universe.

Figure 2.5 illustrates the relationships implied in the three-universe model. In summary, faults are physical events that occur in the physical universe. Faults can result in errors in the informational universe, and errors can ultimately lead to failures that are witnessed in the external universe of the system.

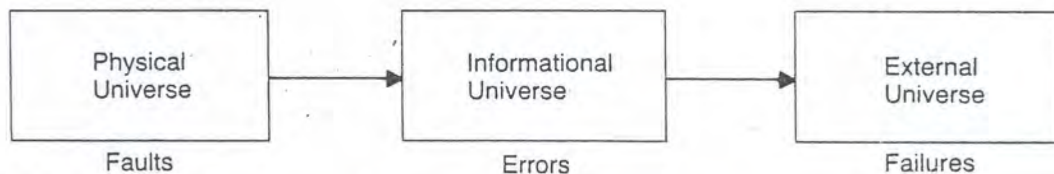


Fig. 2.5 Three-universe model representing the cause-and-effect relationship between faults, errors, and failures. Faults occur in the physical universe and cause errors to occur in the informational universe. Errors can result in failures that occur in the external universe.

The cause-and-effect relationship implied in the three-universe model leads to the definition of two important parameters: fault latency and error latency. **Fault latency** is the length of time between the occurrence of a fault and the appearance of an error due to that fault. A **latent fault** is one that is present in a system but has not yet produced an error. In other words, a latent fault has not yet produced any effect. **Error latency** is the length of time between the occurrence of an error and the appearance of the resulting failure. Based on the three-universe model, the total time between the occurrence of a physical fault and the appearance of a failure is the sum of the fault latency and the error latency.

2.3 Causes of Faults

Faults can be the result of a variety of things that occur within electronic components, external to the components, or during the component or system design process. It is very important to understand all the possible causes of faults. To understand the various causes of faults, we first examine a typical design process to identify areas where faults can occur.

A design process often begins with a somewhat vague statement of the problem. The designers generate the problem statement as their interpretation of the actual problem. From the problem statement, a high-level solution to the problem is outlined, and the development of specific algorithms and system architectures is begun. Hardware and software specifications are then developed from the initial algorithms and architectures. A complete hardware and software design and test procedure is then invoked.

Once the hardware and software are completed, the designs must pass through a complete integration and test procedure with the end result being an operational system that, designers hope, solves the original problem. The various parts of the design process are typically performed several times before the design is completed. For example, the preliminary design process can modify the problem statement, either by necessity or as the result of some design tradeoff.

Problems at any of several points within the design process can result in faults within the system. Possible **fault causes** can be associated with problems in four basic areas: specifications, implementation, components, and external factors.

The first cause of faults is the possibility of **specification mistakes.** These include incorrect algorithms, architectures, or hardware and software design specifications. For example, suppose that the designer of a digital circuit incorrectly specified the timing characteristics of some of the circuit's components. The design might perform correctly at times, but there could also be instances of incorrect performance.

The next cause of faults is **implementation mistakes.** Implementation, as defined here, is the process of transforming hardware and software specifications into the physical hardware and the actual software. The implementation can introduce faults due to poor design, poor component selection, poor construction, or software coding mistakes. For example, suppose that a printed circuit board is constructed such that adjacent lines of a circuit are shorted together. The components on the board can be performing correctly, but the board produces incorrect results because of a wiring mistake. A second crucial example of an implementation mistake is a software coding error. If the software is written incorrectly, a fault exists, and an error can result if the faulty software is executed at some point during the operation of the system.

The next cause of faults is **component defects.** Manufacturing imperfections, random device defects, and component wear-out are typical examples of component defects. Electronic components simply become defective sometimes. The defect can be the result of bonds breaking within the circuit or corrosion of the metal. Component defects are the most commonly considered cause of faults, but, as is evident from our discussions, component defects are only one of several causes of faults.

The final cause of faults is the **external disturbance,** for example, radiation, electromagnetic interference, battle damage, operator mistakes, and environmental extremes. If an electronic system is subjected to extreme temperature variations, the system can produce incorrect results. If the electronics in a military system are damaged during a battlefield encounter, a fault has been injected into the system by some external source. Also, electronic systems are usually very sensitive to electrostatic sources such as lightning or other weather-related effects. Finally, operator mistakes are considered to be external disturbances since the operator is external to the system's physical hardware and software. Clearly, an operator can provide incorrect commands to a system that can ultimately lead to system failures.

Figure 2.6 illustrates the general effects of faults in a system and follows the three-universe concepts discussed in the previous section. The four distinct causes—specification mistakes, implementation mistakes, component defects and external disturbances—result in hardware or software faults in the physical universe. The effect of a fault is to produce some error that is an unintentional deviation from correctness in the information of a system. The result of an error is a failure that occurs in the external universe.

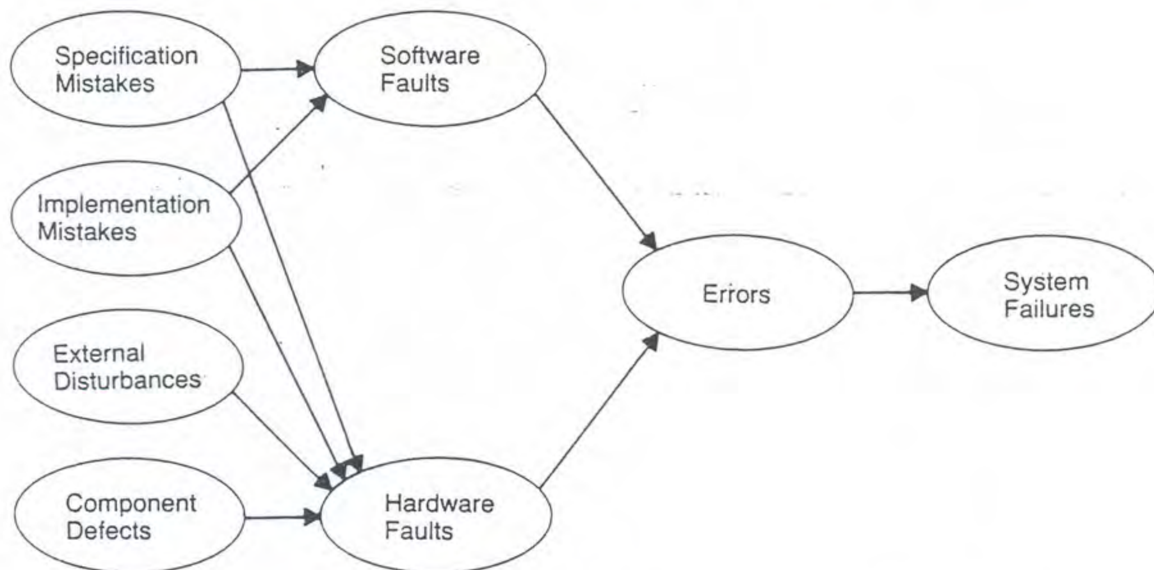


Fig. 2.6 Cause-and-effect relationship of faults, errors, and failures in a system.

2.4 Characteristics of Faults

We have spent quite some time developing the causes of faults and how faults, errors, and failures differ. To adequately describe faults, however, characteristics other than the cause are required. In addition to the cause, four major attributes are critical to the description of faults: nature, duration, extent, and value [Nelson and Carroll 1982]. Figure 2.7 illustrates each of the basic characteristics of faults.

The **fault nature** specifies the type of fault. For example, is the fault a hardware fault or a software fault? Also, is the fault in the analog or the digital circuitry? A power supply fault is an example of an analog hardware fault. A short circuit within a microprocessor is an example of a digital hardware fault. A loop within a subroutine that, when entered, can never be exited is an example of a software fault. In summary, the nature of a fault can be specified using the terms hardware, software, analog, and digital.

The **fault duration** specifies the length of time that a fault is active. First, there is the **permanent fault,** which remains in existence indefinitely if no corrective action is taken. Second, there is the **transient fault,** which can appear and disappear within a very short period of time. Third, there is the **intermittent fault,** which appears, disappears, and then reappears repeatedly. An example of a permanent fault is a logic line that is physically stuck at a logic 1. An example of a transient fault is one resulting from some external disturbance such as lightning. The lightning can temporarily dis-

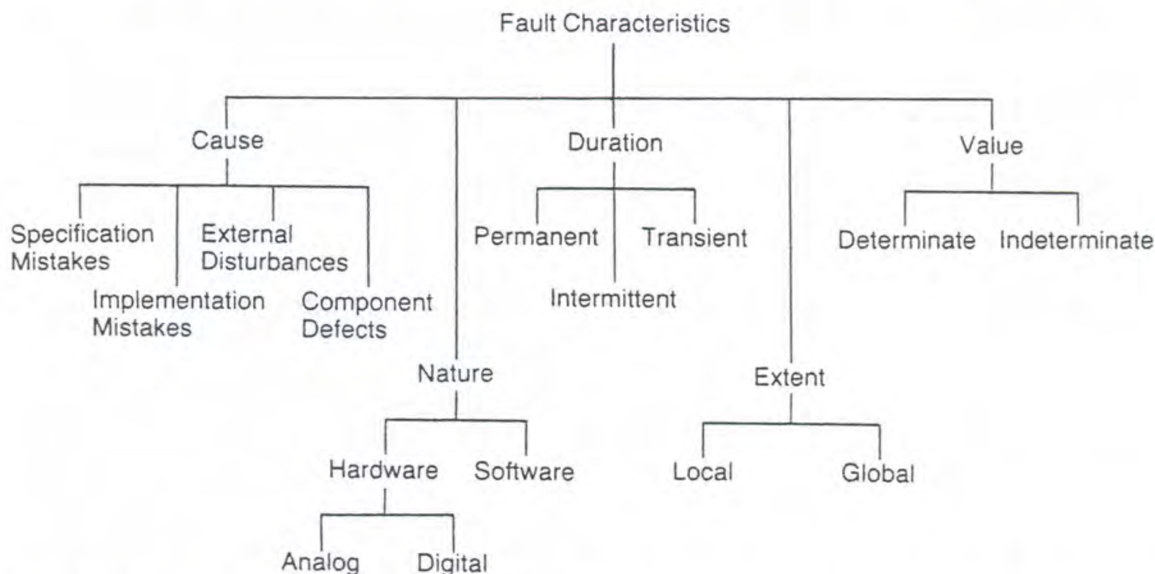


Fig. 2.7 Faults can be characterized by five attributes: cause, nature, duration, extent, and value.

rupt a system but do no permanent damage; as soon as the lightning disappears, the fault can also disappear. The effects of the external disturbance, that is, the errors, can take some time to disappear, but the actual fault may be present for only a very short time. An example of an intermittent fault is one resulting from a weak solder joint in a circuit. The solder joint can provide proper contact at certain times and improper contact at others.

The **fault extent** specifies whether the fault is localized to a given hardware or software module or whether it globally affects the hardware, the software, or both. A power supply fault is a good example of a fault that might globally affect a system. On the other hand, a fault in a memory that is used by only one processor might have a very localized impact on a system.

The **fault value** can be either determinate or indeterminate. A **determinate fault** is one whose status remains unchanged throughout time unless externally acted upon. For example, a fault that always results in a line being a logical 1 is a determinate fault. An **indeterminate fault** is one whose status at some time T may be different from its status at some increment of time greater than or less than T. For example, a number of hardware faults can produce state oscillations between a logical 1 and a logical 0. A good example is a fault that is sensitive to either the data or time.

2.5 Fault Models

In much of our work, it is necessary to assume that faults behave according to some fault model [Hayes 1985]. Although in practice faults can be transient in duration and indeterminate in value, it can be extremely difficult to analyze digital systems if we assume that faults have these characteristics. This is particularly true when we attempt to design test procedures for digital systems or to simulate faults within such systems. To make our problems more manageable, we need some way to restrict our attention to a subset of all faults that can occur.

Fault models allow us to specifically define the types of faults that will be considered and the behavior these faults will have. In addition, fault models allow us to represent the behavior of physical occurrences. We use models in all disciplines of engineering, and although we understand that models are not 100% accurate in all cases, we use them to make problems tractable and because their usage often introduces little error. The same is true of fault modeling.

Fault models attempt to represent the types of faults that can occur. In this section, we consider two primary fault models: the logical stuck-fault model that is used at the logic circuit level and the transistor stuck-fault model that is used at the transistor circuit level.

2.5.1 The Logical Stuck-Fault Model

The most common fault model is the **logical stuck-fault model** [Kohavi 1978], which has gained its popularity because of its effectiveness and simplicity. The logical stuck-fault model is sometimes referred to as the stuck-at-0 (s-a-0), stuck-at-1 (s-a-1) fault model or simply the stuck-fault model.

There are three basic assumptions of the stuck-fault model:

- a fault results in a module responding as if one of its inputs or outputs is physically stuck at a logic 1 or 0
- the basic functionality of the circuit is not altered by the fault
- the fault is permanent

The logic module can be a single gate or a collection of gates that implements some logic function.

The first assumption of the stuck-fault model is further illustrated in Fig. 2.8 by an AND gate having inputs *A* and *B* and output *F*. The input line *A* is assumed to remain free to take on either logic value (1 or 0), but the gate responds as if the line is physically stuck at either a logic 1 or logic 0. For example, a stuck-at-1 fault on line *A* results in the output of the AND gate in Fig. 2.8 being 1 whenever input *B* is 1, regardless of the actual value applied to line *A*. Line *A* is free, however, to assume any logic value. Likewise, a stuck-at-0 fault on line *A* causes the AND gate to always have an output of 0, regardless of the actual value applied to input *A*.

The second assumption of the stuck-fault model is that the basic functionality of the circuit is not affected by the fault. This assumption is often confusing, but it is very important in the stuck-fault model. It does not mean that the circuit continues to produce the correct results, but instead simply implies that the circuit produces the results expected of it, given the existence of the fault. For example, an AND gate that has a stuck-at-0 fault on one of its inputs should always produce a 0 at the output if the gate continues to behave as an AND gate. If, however, the fault transforms the AND

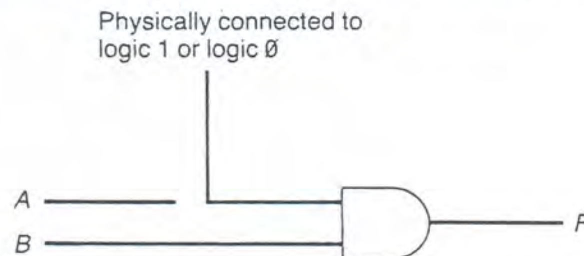


Fig. 2.8 Illustration of the basic concept of the logical stuck-fault model. The gate responds as if the input is permanently connected to a 1 or 0. The applied input, however, is free to assume any value.

gate into an EXCLUSIVE-OR gate, the assumptions of the stuck-fault model are violated. Likewise, if a fault transforms a combinational circuit into a sequential circuit, the assumptions of the stuck-fault model are violated. This latter example is a significant problem in many of today's integrated circuit technologies.

The third assumption of the stuck-fault model is that the fault is permanent. Simply stated, the faulty module *always* performs as if a line is stuck at a specific logic value. This assumption simplifies the fault model by avoiding the difficulty of modeling intermittent or transient faults.

Most applications of the stuck-fault model restrict the number of faults that can occur at any one time. Typically, we assume that a circuit will never have more than one stuck fault. Although the single-fault assumption is not a specific property of the stuck-fault model, it is a very common assumption that is used in conjunction with the remaining stuck-fault assumptions to simplify the process of analyzing a circuit or generating test patterns. In a circuit that contains n lines, at most $2n$ unique, single, stuck faults can occur. Consequently, an upper bound is placed on the number of faults that must be examined. Even with the bound, it is easy to see that large circuits can have a prohibitively large number of potential faults.

The effectiveness of a fault model can often be quantified by a coverage parameter. A fault model is said to cover a fault if and only if the actual physical fault is accurately represented by the chosen fault model. Ideally, one hopes that a fault model covers 100% of all physical faults, but this is seldom the case.

The classic example of a case where the stuck-fault model does not cover a very specific and practical physical fault is found in the Complementary Metal Oxide Semiconductor (CMOS) implementation of a two-input NOR gate [Wadsack 1978]. The logic diagram and the transistor realization of the CMOS NOR gate are shown in Fig. 2.9. The circuit is a combination of two p-channel transistors in series and two n-channel transistors in parallel. Based on the values of the inputs, A and B, a path for current flow is established from either V_{DD} or V_{SS} to the output of the circuit. For example, if both A and B are at the logic 0 level, both p-channel transistors are conducting while both n-channel transistors are turned off. A path is established between V_{DD} and the output, thereby forcing the output to a logic 1 value. In a similar manner, if either or both inputs are 1, the corresponding p-channel transistors are turned off while one or both n-channel transistors are turned on; the result is that a path is established from the output to V_{SS}, thus forcing the output to a logic 0 value.

Several faults that behave as stuck faults can be easily recognized in the NOR circuit. For example, if input line A becomes physically shorted to

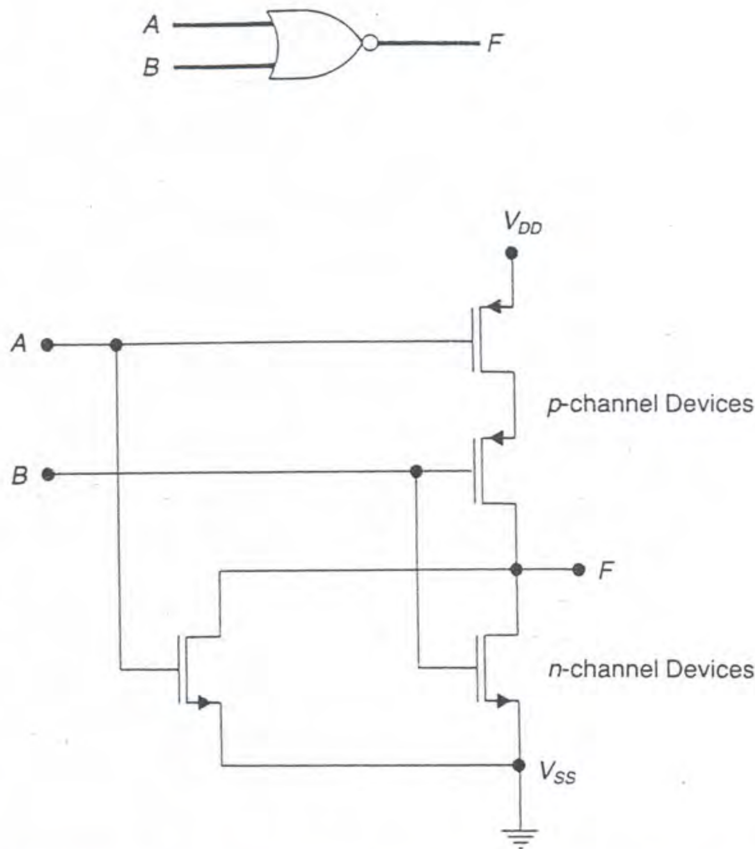


Fig. 2.9 Logic diagram and transistor implementation of the CMOS NOR gate. Inputs A and B determine if the output is pulled high (V_{DD}) or low (V_{SS}).

V_{DD}, the circuit behaves as if line A is stuck at 1; specifically, the output of the circuit will always be 0. Another example is when the drain and source of one of the n-channel devices become shorted together, causing the device to always have a logic 0 on the output. Consequently, the fault can be modeled as the output stuck at 0.

Several faults do not adhere to the stuck-fault model. One example is when a line within the circuit breaks. If a break in a line occurs, as shown in Fig. 2.10, and the input combination $AB=10$ is presented to the circuit, a path does not exist from either V_{DD} or V_{SS} to the circuit output. In other words, neither the series network of p-channel transistors nor the parallel network of n-channel transistors is conducting. The output under such conditions will be pulled neither to logic 0 nor to logic 1, but, due to load capacitances present on the output, the output retains the value defined by the previous input. The length of time that the output remains at the previous value depends on the length of time required to discharge the load capacitance. This type of fault is often referred to as a **stuck-open fault.**

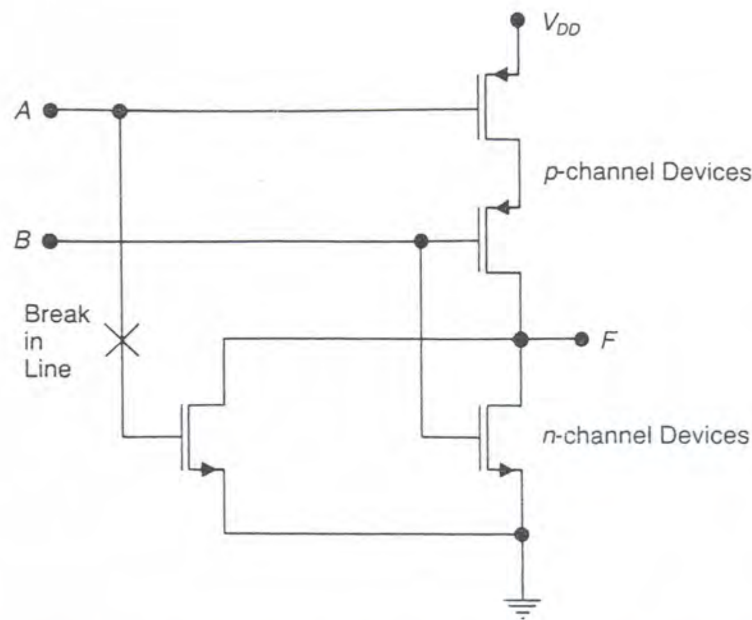


Fig. 2.10 A stuck-open fault exists when a break in a line occurs. Certain input patterns result in the output not being pulled either high or low. Instead the load capacitances on the output result in a previously defined output being retained.

When a fault results in the present output value depending on the previous output, the circuit has obtained a form of memory and is, therefore, no longer a combinational circuit. Instead, the circuit is a sequential circuit, and one of the basic assumptions of the stuck-fault model is violated. Therefore, the stuck-fault model cannot adequately model the stuck-open fault that can occur in the CMOS NOR gate.

Recognizing the limitations of the stuck-fault model, researchers have attempted to develop new and better fault models. For example, in [Wadsack 1978] logic circuit models of the various gates, such as the NOR gate, are developed to allow the effect of the stuck-open fault in CMOS circuits to be simulated. However, the fault model requires that additional gates be added to the description of the circuit, thereby increasing the complexity.

As an example, Fig. 2.11 shows the modified representation of a CMOS NOR gate. In Fig. 2.11, the D-type flip flop is used to model the memory that is introduced when the stuck-open fault occurs. The flip flop is a level-triggered device such that when the clock line is 1, the output of the flip flop simply follows the D input. When the clock transitions from a 1 to a 0, however, the value on the D input is latched, and the flip flop output retains that value until the clock returns to 1. Note that under all fault-free operating conditions, the clock remains at a logic 1, and the output of the flip flop is just the output of the NOR gate G_1. Traditional stuck-at-1 and stuck-at-0

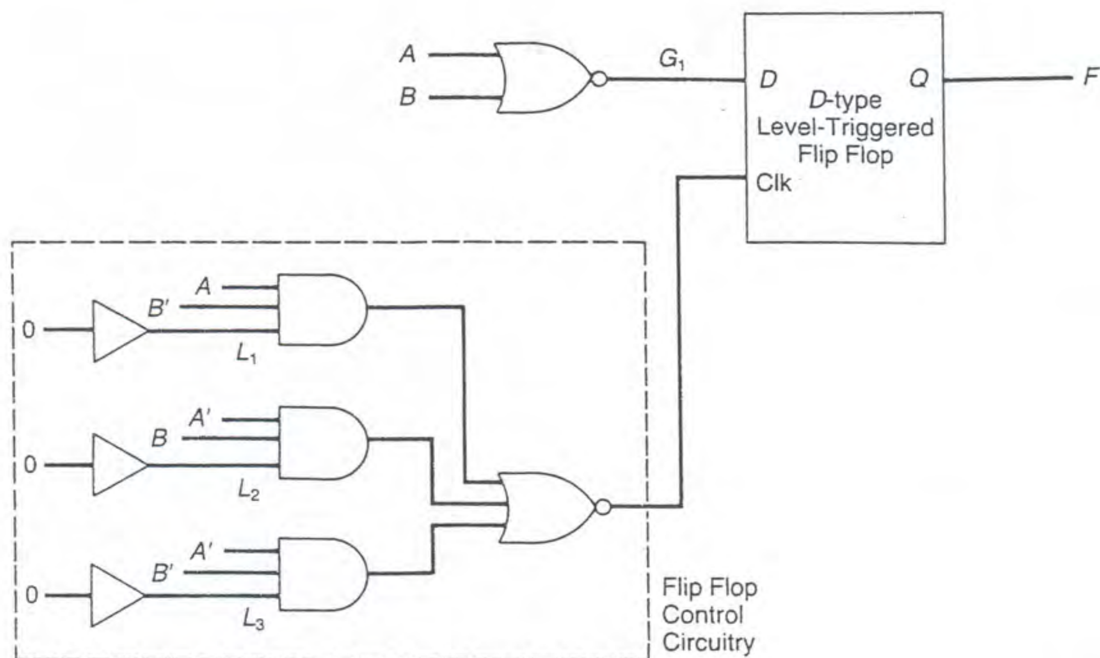


Fig. 2.11 Logical model of a NOR gate to account for the stuck-open fault. The D-type flip flop is used to simulate the memory effect produced by the stuck-open fault.

faults can be injected by simply placing them on the inputs A and B, and on the output G_1 of the NOR gate.

Stuck-open faults can be modeled by placing a stuck-at-1 fault on either line L_1, L_2, or L_3. L_1 stuck-at-1 represents a stuck-open fault affecting the n channel device driven by input A. L_2 stuck-at-1 represents a stuck-open fault affecting the n-channel device driven by input B. Finally, L_3 stuck-at-1 represents a stuck-open fault affecting either p-channel device. For example, if L_1 is stuck-at-1, the clock line on the flip flop will be logic 0 whenever the input combination $AB=10$ occurs. In other words, the n-channel device driven by input A should be on, but it is not because of the stuck-open fault. Consequently, the output of the gate is simply the output specified by the previous input combination. Any other input combination results in the gate performing as expected.

The difficulty with the modeling scheme developed in [Wadsack 1978] is the additional complexity that must be added to a circuit to allow for the modeling of stuck-open faults. For each gate, we must add four additional gates and a flip flop to allow the effects of stuck-open faults to be adequately modeled. Consequently, the complexity of the circuit, as far as simulation is concerned, is substantially increased.

2.5.2 Transistor Stuck-Fault Models

In some cases, such as the stuck-open fault, the logical stuck-fault model is inaccurate. One way to make the fault model more accurate is to construct the model at the transistor level as opposed to the gate level. Such models are often referred to as switch-level models because the transistor is essentially performing a switching function. Here, however, we use the terminology **transistor stuck-fault model** to describe the techniques employed.

In the transistor stuck-fault model, faults are represented as either a transistor stuck on or a transistor stuck off. For example, in the CMOS NOR gate of Fig. 2.9, the faults are assumed to result in transistors being permanently on or permanently off. The advantage of the transistor stuck-fault model is that the stuck-open fault described previously can be represented as a transistor permanently stuck off. For example, in Fig. 2.9, if the n-channel transistor driven by input A is permanently off, the response of the circuit to the input combination $AB = 10$ will be that the output is forced neither to V_{DD} nor to V_{SS}. Consequently, the output retains its previous value due to the load capacitances present on the output.

The clear disadvantage of the transistor stuck-fault model is the additional complexity. For example, a circuit containing 100 NOR gates contains 400 transistors, so the number of elements that must be represented and manipulated has quadrupled. Other gates such as EXCLUSIVE-ORs require even more transistors. In circuits containing many gates, the impact on complexity of using the transistor stuck-fault model can be overwhelming.

2.6 Error Models

Another technique that has been proposed for the modeling of the effects of faults is to model the effect in the informational universe. In other words, the *error* is modeled rather than the fault [Patel and Fung 1982]. The underlying assumption of this approach is that regardless of the fault, the effect of that fault in the informational universe is to change a logic value at some time and in some result produced by the system. For combinational circuits, this can be viewed as a modification of the circuit's truth table.

The truth table modification that results can vary as a function of time, but the truth table will be either correct or modified in some way. In other words, we may not know what form the fault takes, but we can look at the response of the circuit and determine whether or not the results are correct. We use this type of model in our daily lives when we compare pieces of information. For example, when we balance our checkbooks, we detect the existence of an error by the deviation between the bank's balance and our balance. Once we have detected the error, we must then examine the details

of the checkbook and the statement to locate the problem. However, if the bank's balance agrees with our own, we can be reasonably confident that both balances are correct.

Error models are useful in the design and verification of many self-testing schemes, but they are not normally used as a means of test pattern generation, as the stuck-fault model is. Also, the error model implies that some time can elapse before we recognize that a fault exists.

Various other forms of fault models have been developed in the past for modeling faults in digital systems [Hayes 1985]. For the purposes of this text, however, the logical stuck-fault model is used during our discussions of test pattern generation. The interested reader is referred to the additional reading for descriptions of other available fault models.

2.7 Design Philosophies to Combat Faults

There are three primary techniques for attempting to improve or maintain a system's normal performance in an environment where faults are of concern: fault avoidance, fault masking, and fault tolerance. Figure 2.12 illustrates the barriers that are constructed by each of the available techniques.

Fault avoidance is any technique that is used to prevent faults in the first place. Fault avoidance can include such things as design reviews, component screening, testing, and other quality control methods. If a design review, for example, is conducted appropriately, many of the specification mistakes that might otherwise result in faults can be eliminated. Also, a system can often be shielded to prevent external disturbances such as lightning or radiation from causing faults in the system. Shielding is a form of fault avoidance. Finally, if a design is effectively tested, many of the faults that might be in a system after manufacture can be detected and eliminated before the system is placed into operation.

Fault masking is any process that prevents faults in a system from introducing errors into the informational structure of that system. Error correcting memories, for example, correct a memory's data before a system uses the data. Thus the system never experiences the impact of the fault within the memory. Error correction in a memory is a form of fault masking. Another example of fault masking is majority voting. If a committee of three people makes a decision by voting yes or no on a proposition, any two votes that agree determine the decision of the committee. The ultimate decision of the committee represents the wishes of a majority of the committee members and masks the desires of a member that happens to disagree with the majority. Similar techniques can be applied to digital systems such that two modules can mask the effect of a faulty module.

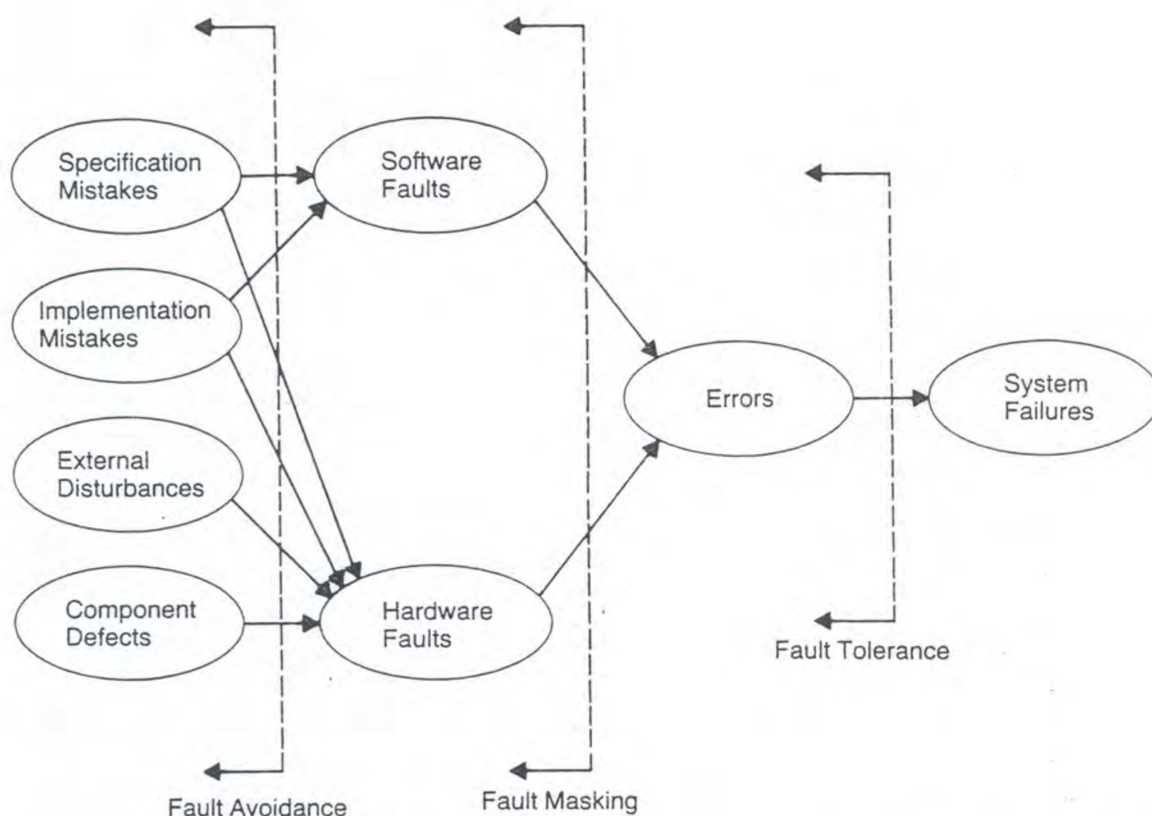


Fig. 2.12 Barriers constructed by design techniques of fault avoidance, fault tolerance, and fault masking.

Fault tolerance is the ability of a system to continue to perform its tasks after the occurrence of faults. The ultimate goal of fault tolerance is to prevent system failures from ever occurring. Since failures are directly caused by errors, the terms fault tolerance and error tolerance are often used interchangeably. For our purposes, however, we use the term fault tolerance.

Fault tolerance can be achieved by many techniques. Certainly, fault masking is one approach to tolerating faults that have occurred. Another approach is to detect and locate the fault that has occurred and reconfigure the system to remove the faulty component. **Reconfiguration** is the process of eliminating a faulty entity from a system and restoring the system to some operational condition or state. If the reconfiguration technique is used, the designer must be concerned with the following processes:

1. **Fault detection** is the process of recognizing that a fault has occurred. Fault detection is often required before any recovery procedure can be implemented.
2. **Fault location** is the process of determining where a fault has occurred so that an appropriate recovery can be implemented.

3. **Fault containment** is the process of isolating a fault and preventing the effects of that fault from propagating throughout a system. Fault containment is required in all fault-tolerant designs.
4. **Fault recovery** is the process of remaining operational or regaining operational status via reconfiguration even in the presence of faults.

Equivalent definitions can be provided in the informational universe:

1. **Error detection** is the process of recognizing that an error has occurred.
2. **Error location** is the process of determining which specific module produced the error.
3. **Error containment** is the process of preventing the error from propagating throughout a system.
4. **Error recovery** is the process of regaining operational status or restoring the system's integrity after the occurrence of an error.

A typical fault-tolerant system that employs reconfiguration performs as follows. Suppose a fault occurs in the system. Fault detection techniques can identify that a fault exists, and fault location procedures specifically identify the source of the fault. Fault recovery and reconfiguration techniques then remove the faulty module and either replace it with a fault-free module or degrade the functionality of the system to keep it operational.

Summary

This chapter has presented the definitions of several terms that are crucial to an understanding of fault-tolerant system design. In addition, we have discussed the basic causes and characteristics of faults in digital systems. Each fundamental term and concept is summarized in the following list.

Component Defects — physical imperfections or flaws in an electronic component.

Determinate Fault — a fault whose status remains unchanged throughout time.

Error — the occurrence of an incorrect value in some unit of information within a system.

Error Containment — the process of preventing an error from propagating throughout a system.

Error Detection — the process of recognizing that an error has occurred.

Error Latency — the length of time between the occurrence of an error and the appearance of a system failure.

Error Location — the process of determining the source of an error.

Error Recovery—the process of regaining operational status and restoring a system's integrity after the occurrence of an error.

External Disturbances—an action external to the system that produces a hardware or software fault.

External Universe—the domain where the user of a system ultimately sees the effects of faults and errors. The external universe is where failures occur.

Failure—a deviation in the expected performance of a system.

Fault—a physical defect, imperfection, or flaw that occurs in hardware or software.

Fault Avoidance—a technique that attempts to prevent the occurrence of faults.

Fault Cause—one of four basic items that can result in faults: specification mistakes, implementation mistakes, component defects, and external disturbances.

Fault Containment—the process of confining the effects of a fault to a limited locality.

Fault Detection—the process of recognizing that a fault has occurred.

Fault Duration—the length of time that a fault is active in a system.

Fault Extent—the characteristic that specifies whether a fault is localized to a given module or globally affects the system.

Fault Latency—the length of time between the occurrence of a fault and the appearance of an error.

Fault Location—the process of determining where a fault has occurred.

Fault Masking—the process of preventing faults from introducing errors.

Fault Nature—a characteristic of a fault that describes its type. The fault type can be hardware, software, digital, or analog.

Fault Recovery—the process of maintaining or regaining operational status after a fault has occurred.

Fault Tolerance—the ability to continue the correct performance of functions in the presence of faults.

Fault Value—the characteristic that specifies whether a fault is determinate or indeterminate.

Implementation Mistakes—incorrect actions occurring during the transformation of hardware and software specifications into physical hardware and actual software.

Indeterminate Fault—a fault whose status changes from time to time.

Informational Universe—the domain that contains units of information and is the location of errors.

Intermittent Fault—a fault that appears, disappears, and then reappears within a system.

Latent Fault—a fault that is present within a system but that has not yet produced an error.

Logical Stuck-Fault Model—a representation that assumes all faults will appear as lines in the logic diagram being physically stuck at a logic 1 or logic 0 value.

Malfunction—the incorrect performance of some system function. A malfunction is equivalent to a failure.

Permanent Fault—a fault that remains in a system indefinitely.

Physical Universe—a domain that contains physical entities and is the location of faults.

Reconfiguration—the process of eliminating a faulty entity from a system and restoring the system to some operational state.

Specification Mistakes—incorrect algorithms, architectures, or hardware/software design specifications that lead to either hardware or software faults.

Stuck-open Fault—a physical fault resulting in the output of a gate depending on the present input and the previous output.

Transient Fault—a fault that appears and then disappears a short time later.

Transistor Stuck-Fault Model—a representation that assumes all faults will appear as transistors in the circuit diagram being physically stuck on or stuck off.

User's Universe—equivalent to the external universe. The domain where the user sees the effects of faults and errors.

References

1. Avizienis, A. "The four-universe information system model for the study of fault tolerance," *Proceedings of the 12th Annual International Symposium on Fault-Tolerant Computing*, Santa Monica, California, June 22-24, 1982, pp. 6–13.
2. Hayes, J.P. "Fault modeling," *IEEE Design and Test*, Vol. 2, No. 2, April 1985, pp. 88–95.
3. Johnson, B.W. "Fault-tolerant microprocessor-based systems," *IEEE Micro*, Vol. 4, No. 6, December 1984, pp. 6–21.
4. Kohavi, Z. *Switching and Finite Automata Theory*, McGraw-Hill, New York, 1978.
5. Laprie, J-C. "Dependable computing and fault tolerance: Concepts and terminology," *Proceedings of the 15th Annual International Symposium on Fault-Tolerant Computing*, Ann Arbor, Mich., June 19-21, 1985, pp. 2–11.

6. Nelson, V.P., and B.D. Carroll. "Fault-tolerant computing (A tutorial)," presented at the AIAA Fault Tolerant Computing Workshop, November 8-10, 1982, Fort Worth, Tex.
7. Patel, J.H., and L.Y. Fung. "Concurrent error detection in ALUs by recomputing with shifted operands," *IEEE Transactions on Computers*, Vol. C-31, No. 7, July 1982, pp. 589–595.
8. Wadsack, R.L. "Fault modeling and logic simulation of CMOS and MOS integrated circuits," *The Bell System Technical Journal*, Vol. 57, No. 5, May-June 1978, pp. 1449–1475.

Additional Reading

The following list of references is provided for the reader who is interested in pursuing the topics of this chapter in more detail.

Anderson, T., and P.A. Lee. *Fault Tolerance Principles and Practices*, Prentice-Hall International, London, 1981.

Anderson, T., and P.A. Lee. "Fault tolerance terminology proposals," *Proceedings of the 12th Annual International Symposium on Fault-Tolerant Computing*, Santa Monica, Calif., June 22-24, 1982, pp. 29–33.

Avizienis, A. "Fault tolerance: The survival attribute of digital systems," *Proceedings of the IEEE*, Vol. 66, No. 10, October 1978, pp. 1109–1125.

Banerjee, P., and J.A. Abraham. "Characterization and testing of physical failures in MOS logic circuits," *IEEE Design and Test*, Vol. 1, No. 4, August 1984, pp. 76–86.

Beh, C.C., K.H. Arya, C.E. Radke, and K.E. Torku. "Do stuck fault models reflect manufacturing defects?" *Proceedings of the 1982 International Test Conference*, October 1982, pp. 35–42.

Bryant, R.E. "A switch-level model and simulator for MOS digital systems," *IEEE Transactions on Computers*, Vol. C-33, No. 2, February 1984, pp. 160–177.

Bryant, R.E., and M.D. Schuster. "Fault simulation of MOS digital circuits," *VLSI Design*, Vol. 4, No. 10, October 1983, pp. 24–30.

Carter, W.C. "A time for reflection," *Proceedings of the 12th Annual International Symposium on Fault-Tolerant Computing*, Santa Monica, Calif., June 22–24, 1982, p. 41.

Case, G.R. "Analysis of actual fault mechanisms in CMOS logic gates," *Proceedings of the 13th Design Automation Conference*, June 1976, pp. 265–270.

Chandramouli, R. "On testing stuck-open faults," *Proceedings of the Fault Tolerant Computing Symposium*, 1983, pp. 258–265.

Courtois, B. "Failure mechanisms, fault hypothesis, and analytical testing on LSI NMOS (HMOS) circuits," *VLSI 81*, University of Edinburgh, August 1981, Academic Press, 1981.

El-ziq, Y.M. "Classifying, testing, and eliminating VLSI MOS failures," *VLSI Design*, Vol. 4, No. 9, September 1983, pp. 30–35.

El-ziq, Y.M. "Automatic test generation for stuck-open faults in CMOS VLSI," *Proceedings of the 18th Design Automation Conference*, June 1981, pp. 347–352.

Galiay, J., Y. Crouzet, and M. Verniault. "Physical versus logical fault models in MOS LSI circuits: Impact on their testability," *IEEE Transactions on Computers*, Vol. C-29, No. 6, June 1980, pp. 527–531.

Goldberg, J. "A time for integration," *Proceedings of the 12th Annual International Symposium on Fault-Tolerant Computing*, Santa Monica, Calif., June 22–24, 1982, p. 42.

Gupta, A.K., and J.R. Armstrong. "Functional fault modeling and simulation for VLSI devices" , Department of Electrical Engineering, Virginia Polytechnic Institute and State University, Blacksburg, Va., 1985.

Hayes, J.P. "Modeling faults in digital logic circuits," in *Rational Fault Analysis*, R. Saeks and S.R. Liberty, ed., Marcel-Dekker, New York, 1977, pp. 78–95.

Hayes, J.P. "Fault modeling for digital MOS integrated circuits," *IEEE Transactions on Computer-Aided Design*, Vol. CAD-3, No. 3, July 1984, pp. 200–207.

Hayes, J.P. "An introduction to switch-level modeling," *IEEE Design and Test of Computers*, Vol. 4, No. 4, August 1987, pp. 18–25.

Kopetz, H. "The failure-fault (FF) model," *Proceedings of the 12th Annual International Symposium on Fault-Tolerant Computing*, Santa Monica, California, June 22-24, 1982, pp. 14–17.

Lala, P.K. *Fault Tolerant and Fault Testable Hardware Design*, Prentice-Hall International, London, England, 1985.

Lee, P.A., and D.E. Morgan. "Fundamental concepts of fault-tolerant computing—progress report," *Proceedings of the 12th Annual International Symposium on Fault-Tolerant Computing*, Santa Monica, California, June 22-24, 1982, pp. 34–38.

Mangir, T.E. "Sources of failure and yield improvement for VLSI," *Proceedings of the IEEE*, Vol. 72, No. 6, June 1984, pp. 690–708.

Pradhan, D.K. *Fault-Tolerant Computing—Theory and Techniques*, Volumes I and II, Prentice-Hall, Englewood Cliffs, N.J., 1986.

Reddy, S.M., M.K. Reddy, and V.D. Agrawal. "Robust tests for stuck-open faults in CMOS combinational logic circuits," *Proceedings of the 14th International Fault Tolerant Computing Symposium*, June 1984, pp. 44–49.

Rennels, D.A. "Fault-tolerant computing—concepts and examples," *IEEE Transactions on Computers*, Vol. C-33, No. 12, December 1984, pp. 1116–1129.

Robinson, A.S. "A user oriented perspective of fault-tolerant systems models and terminologies," *Proceedings of the 12th Annual International Symposium on Fault-Tolerant Computing*, Santa Monica, Calif., June 22-24, 1982, pp. 22–28.

Siewiorek, D.P., and R.S. Swarz. *The Theory and Practice of Reliable System Design*, Digital Press, Bedford, Mass., 1982.

Timoc, C., M. Buehler, T. Griswold, C. Pina, F. Stott, and L. Hess. "Logical models of physical failures," *Proceedings of the International Test Conference*, 1983, pp. 546–553.

Problems

2.1. Devise an original example to illustrate the differences between faults, errors, and failures. As you illustrate these concepts, relate them to the three-universe model.

2.2. Some systems are designed for reliability, whereas others are designed to be highly available. Explain the difference between these two concepts and give an original example of an application requiring high reliability and one that needs high availability. How would you expect the four different causes of faults to complicate the designs of high reliability applications as compared to high availability applications?

2.3. Faults can be characterized by five major attributes. Give original examples of faults that illustrate each of these attributes.

2.4. Develop a design methodology for digital systems that incorporates fault avoidance at each of the major steps. In other words, examine the design process and determine where fault avoidance techniques such as designs reviews should be incorporated. Pretend you are making recommendations to your employer on how fault avoidance can be used to improve the quality of your company's products.

2.5. Fault masking is an attractive technique for use in systems that cannot allow even momentary erroneous results to be generated. However, fault masking does have several serious limitations. In your opinion, what are the disadvantages of using a fault masking approach?

2.6. Develop a glossary of the key terms used in the fault-tolerant computing field. Keep the glossary available to use as a reference and study guide.

2.7. Explain the difference between the stuck-at-1, stuck-at-0 fault model and the transistor stuck-fault model. Use the transistor diagram of a CMOS NAND gate to illustrate a physical fault that is covered by the stuck-at-1, stuck-at-0 fault model but is not covered by the transistor stuck-fault model. Also, determine a physical fault that is covered by the transistor stuck-fault model but is not covered by the stuck-at-1, stuck-at-0 fault model.

2.8. The stuck-at-0, stuck-at-1 fault model was developed when transistor transistor logic (TTL) was extremely popular. Contrast the differences between TTL and CMOS, and develop an explanation of whether the stuck-at-0, stuck-at-1 fault model is more appropriate for TTL than CMOS. If the model is more appropriate, provide a detailed discussion of why.

3

Design Techniques to Achieve Fault Tolerance

3.1 Introduction
3.2 Primary Design Issues
3.3 The Concept of Redundancy
3.4 Hardware Redundancy
3.5 Information Redundancy
3.6 Time Redundancy
3.7 Software Redundancy
Summary
References
Additional Reading
Problems

3.1 Introduction

The two preceding chapters covered the fundamental definitions and concepts in fault-tolerant computing. In this chapter, we examine the techniques that are available to achieve fault tolerance. In particular, we begin by identifying the fundamental design issues, defining the concept of redundancy, and examining the forms that redundancy can take. The overall purpose of this chapter is to provide the reader with the fundamental techniques that are used in the design of fault-tolerant systems. Chapter 4 then examines evaluation techniques that allow us to compare two or more

approaches and select the best approach for a particular application. Chapter 5 provides examples of systems that use the various techniques presented in this chapter.

3.2 Primary Design Issues

The development of a fault-tolerant system requires the consideration of many design issues. Among these are fault detection, fault containment, fault location, fault recovery, and fault masking. Each of these concepts was defined in Chapter 2, but here we consider the role of each of these functions in the design of fault-tolerant systems.

A system that employs fault masking achieves fault tolerance by "hiding" faults that occur. Such systems do not require that the fault be detected before it can be tolerated, but it is required that the fault be contained. In other words, we want to make the *effect* attribute of all faults local; we do not want one fault to globally affect the performance of a system. Fault containment techniques prevent the effects of faults from spreading throughout a system. Fault masking is often an effective method of achieving fault containment.

Systems that do not use fault masking require fault detection, fault location, and fault recovery to achieve fault tolerance. Fault detection is the cornerstone of many fault-tolerant systems. The ability to detect faults is essential to the fault location and fault recovery processes. Fault location is required after fault detection to identify exactly which component is faulty such that a fault recovery procedure can be implemented. Typically, fault recovery involves some form of reconfiguration that is usually accomplished by disabling, either physically or logically, a faulty component and enabling, again either physically or logically, a replacement component.

This chapter introduces techniques for achieving fault detection, fault location, fault containment, fault masking, fault recovery, and fault tolerance, each of which requires the use of some form of redundancy. Before discussing the specific forms of redundancy, a fundamental definition of redundancy is presented and illustrated.

3.3 The Concept of Redundancy

In the early days of fault-tolerant designs, the concept of redundancy was almost always considered as physical hardware redundancy. The most common technique used to achieve some form of fault tolerance was the physical replication of boxes or hardware components within a system. We now

have a better understanding of redundancy and the various forms of redundancy that can occur [Johnson 1984].

Redundancy is simply the addition of information, resources, or time beyond what is needed for normal system operation. The redundancy can take one of several forms:

1. **Hardware redundancy** is the addition of extra hardware, usually for the purpose of either detecting or tolerating faults.
2. **Software redundancy** is the addition of extra software, beyond what is needed to perform a given function, to detect and possibly tolerate faults.
3. **Information redundancy** is the addition of extra information beyond that required to implement a given function; for example, error detecting codes use a form of information redundancy.
4. **Time redundancy** uses additional time to perform the functions of a system such that fault detection and often fault tolerance can be achieved.

To illustrate the concept of redundancy, consider a simple digital filtering application. The hardware that might be used in the digital filter is shown in Fig. 3.1 and includes an analog-to-digital converter, a small microprocessor, and a digital-to-analog converter. All the hardware shown in Fig. 3.1 is required to implement the function of the digital filter, so there is no hardware redundancy in this system. Likewise, suppose that the software that executes in the microprocessor performs no functions other than the minimum required to implement the filter. Consequently, the digital filter contains no redundancy in the software. In a similar manner, we can see that the simple digital filter contains no redundancies of any form as long as the system is designed strictly to implement the basic requirements of the digital filter.

Suppose now that we want to enhance the design of the digital filter to allow some primitive forms of fault detection. One simple and familiar approach is to add several lines of software to perform a validity check on the results that are being generated. For example, we may know that the values generated by the digital filter should never overflow the range that can be

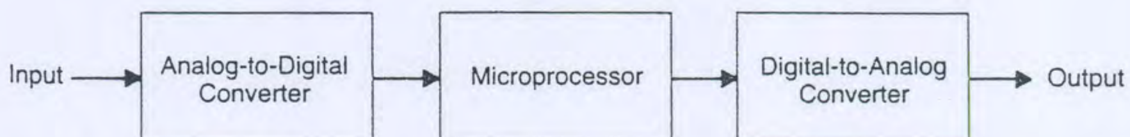


Fig. 3.1 A simple digital system used to implement a digital filter. The system is designed such that no redundancy is present.

represented by an 8-bit digital word. Consequently, we can add several lines of software to verify that we never have an overflow from the 8-bit range of representation. The extra software that has been added is redundant in that it is not needed to perform the basic functions of the digital filter, but it was added to give an additional capability. If adding the extra software requires adding more memory to store the program, hardware redundancy has been introduced to the system because the extra memory was not mandatory for the basic digital filtering operation.

Now suppose that we are concerned about transient faults upsetting the digital filtering system. In an attempt to detect transients, we can perform each calculation of the filter twice at separate times. If the two results differ, some transient fault may have upset one of the two calculations. The required computations of the filtering application now take longer, so we have introduced time redundancy into the system.

Finally, suppose that we modify the output of the digital filter such that each word is appended with a single bit. The extra bit assumes a value of 1 if the unappended output of the filter has an even number of 1s and a value of 0 if the unappended output has an odd number of 1s; this is the concept of odd parity. The added bit has additional information that can be used for the purpose of detecting faults; specifically, the extra bit indicates whether the total number of bits in the information is even or odd. For example, if one bit in the resulting appended output becomes corrupted, the number of 1s in the information changes, and the erroneous condition can be detected. Simple parity schemes such as this are one form of information redundancy. The extra information is not needed to perform the basic task of digital filtering, but it can provide some additional capability such as error detection.

The addition of redundancy never comes cheaply. The extra software in the preceding example would require extra time to develop as well as extra expense. Likewise, the extra memory necessary to store the extra lines of code would increase the filter's cost, weight, power consumption, and size. If the time redundancy approach is employed, we might be forced to use a faster, more expensive processor to allow the performance of all computations twice and still meet the sampling rate requirements of the filtering application. Finally, the information redundancy used in the filtering application requires either additional software or hardware to generate and store the extra bit of information and to interpret what that information means.

The use of redundancy can provide additional capabilities within a system. In fact, if fault tolerance or fault detection is required, some form of redundancy is also required. But, redundancy can have a very important impact on a system's performance, size, weight, power consumption, and reliability. We must understand the types of redundancy techniques available and the methods that can be used to evaluate the impact of the redundancy.

3.4 Hardware Redundancy

The physical replication of hardware is perhaps the most common form of redundancy used in digital systems today. As semiconductor components have become smaller and less expensive, the concept of hardware redundancy has become more common and more practical. The costs of replicating hardware within a system are decreasing simply because hardware costs are decreasing.

There are three basic forms of hardware redundancy [Johnson 1984]: passive, active, and hybrid. **Passive** techniques use the concept of fault masking to hide the occurrence of faults and prevent the faults from resulting in errors. Passive approaches are designed to achieve fault tolerance without requiring any action on the part of the system or an operator. Passive techniques, in their most basic form, mask faults rather than detect them.

The **active** approach, which is sometimes called the dynamic method, achieves fault tolerance by detecting the existence of faults and performing some action to remove the faulty hardware from the system. In other words, active techniques require that the system be reconfigured to tolerate faults. Active hardware redundancy uses fault detection, fault location, and fault recovery in an attempt to achieve fault tolerance.

Hybrid techniques combine the attractive features of both the passive and active approaches. Fault masking is used in hybrid systems to prevent erroneous results from being generated. Fault detection, fault location, and fault recovery are also used in the hybrid approaches to improve fault tolerance by removing faulty hardware and replacing it with spares. A spare element within a system is one that is extra; the spare element is not needed until another element becomes faulty, at which time the spare is used to replace the faulty element. Providing spares is one form of providing redundancy in a system. Hybrid methods are most often used in the critical-computation applications where fault masking is required to prevent momentary errors, and high reliability must be achieved. Hybrid hardware redundancy is usually a very expensive form of redundancy to implement.

3.4.1 Passive Hardware Redundancy

Passive hardware redundancy relies on voting mechanisms to mask the occurrence of faults. Most passive approaches are developed around the concept of majority voting. As previously mentioned, the passive approaches achieve fault tolerance without the need for fault detection or system reconfiguration; the passive designs inherently tolerate the faults.

Triple Modular Redundancy

The most common form of passive hardware redundancy is called **triple modular redundancy** (TMR). The basic concept of TMR (illustrated in Fig. 3.2) is to triplicate the hardware and perform a majority vote to determine the output of the system. If one of the modules becomes faulty, the two remaining fault-free modules mask the results of the faulty module when the majority vote is performed. In typical applications, the replicated modules are processors, memories, or any hardware entity. In addition, TMR can be applied to software where three different versions of programs that perform the same function are used to protect against software faults in any one of the three. For the moment, however, we will focus on hardware techniques.

The primary difficulty with TMR is the voter; if the voter fails, the complete system fails. In other words, the reliability of the simplest form of TMR can be no better than the reliability of the voter. Any single component within a system whose failure leads to a failure of the system is called a **single point of failure.** Several techniques can be used to overcome the effects of voter failure. One approach is to triplicate the voters and provide three independent outputs, as shown in Fig. 3.3. In Fig. 3.3, the three functional modules each receive identical inputs and perform identical functions using those inputs. The results generated by the three modules are voted on to produce three results. Each result is correct as long as no more than one module, or input, is faulty. Several stages of TMR can be interconnected using this approach, as shown in Fig. 3.4. If a voter fails in one stage

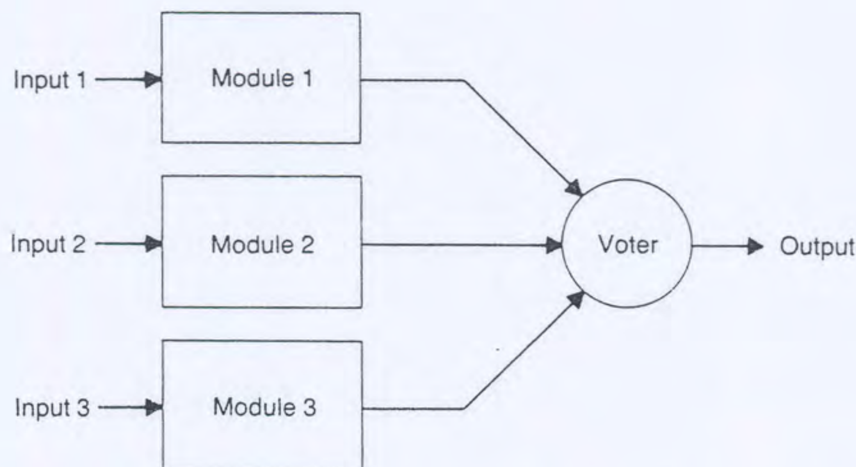


Fig. 3.2 Triple modular redundancy (TMR) uses three identical modules, performing identical operations, with a majority voter determining the output.

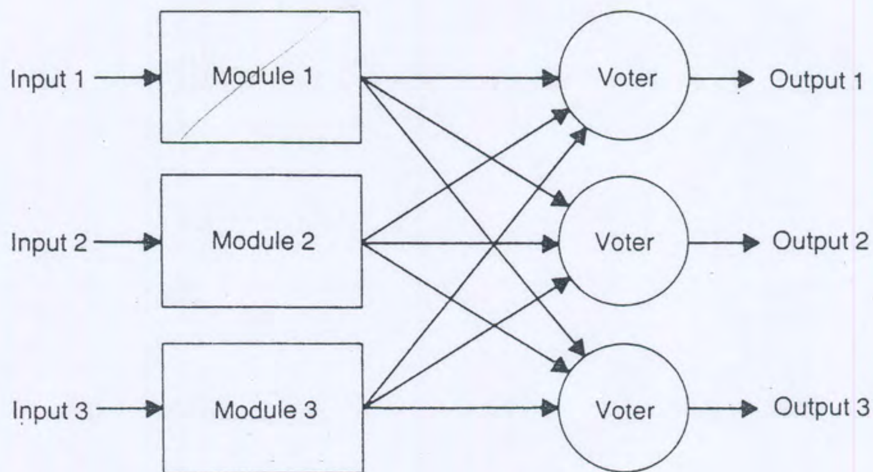


Fig. 3.3 Triple modular redundancy with triplicated voters can be used to overcome susceptibility to voter failure. The voter is no longer a single point of failure in the system.

of the system, the subsequent stage sees the failure as one input becoming corrupted. Voting at the output of the stage that gets the erroneous input corrects the erroneous result. A TMR system with triplicated voters is commonly called a **restoring organ** because the configuration produces three correct outputs even if one input is faulty. In essence, the TMR with triplicated voters restores the error-free signal.

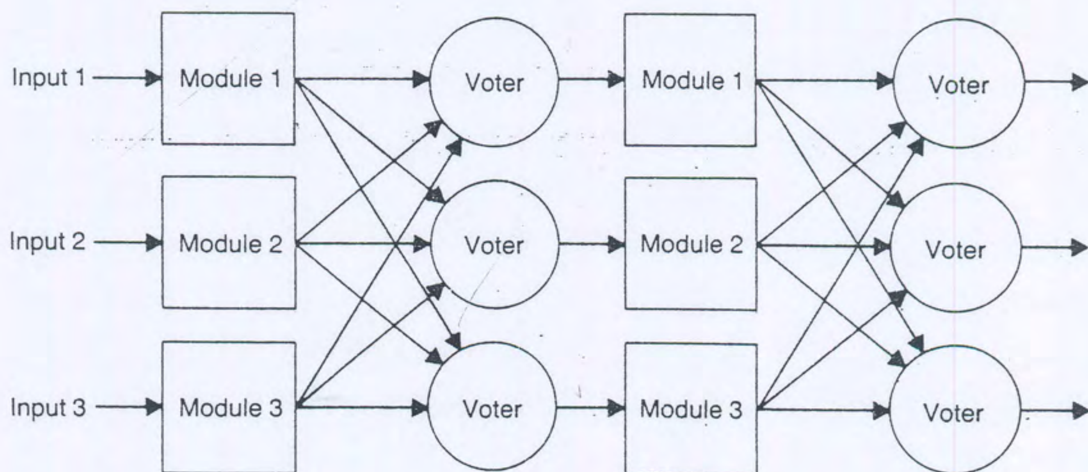


Fig. 3.4 In multiple-stage TMR systems, voting occurs between each stage so that errors are corrected before being passed to a subsequent module.

N-Modular Redundancy

A generalization of the TMR approach is the **N-modular redundancy (NMR)** technique. NMR applies the same principle as TMR but uses N of a given module as opposed to only three. In most cases, N is selected as an odd number so that a majority voting arrangement can be used. The concept of NMR is shown in Fig. 3.5. The advantage of using N modules rather than three is that more module faults can often be tolerated. For example, a 5MR system contains five replicated modules and a voter as shown in Fig. 3.6. A majority voting arrangement allows the 5MR system to produce correct results in the face of as many as two module faults. In many critical-computation applications, two faults must be tolerated to allow the required reliability and fault tolerance capabilities to be achieved.

The primary tradeoff in NMR is the fault tolerance achieved versus the hardware required. Clearly, practical applications must limit the amount of redundancy that can be employed. Power, weight, cost, and size limitations very often determine the value of N in an NMR system.

Voting Techniques

Voting within NMR systems can occur at several points. For example, an industrial controller can sample the temperature of a chemical process

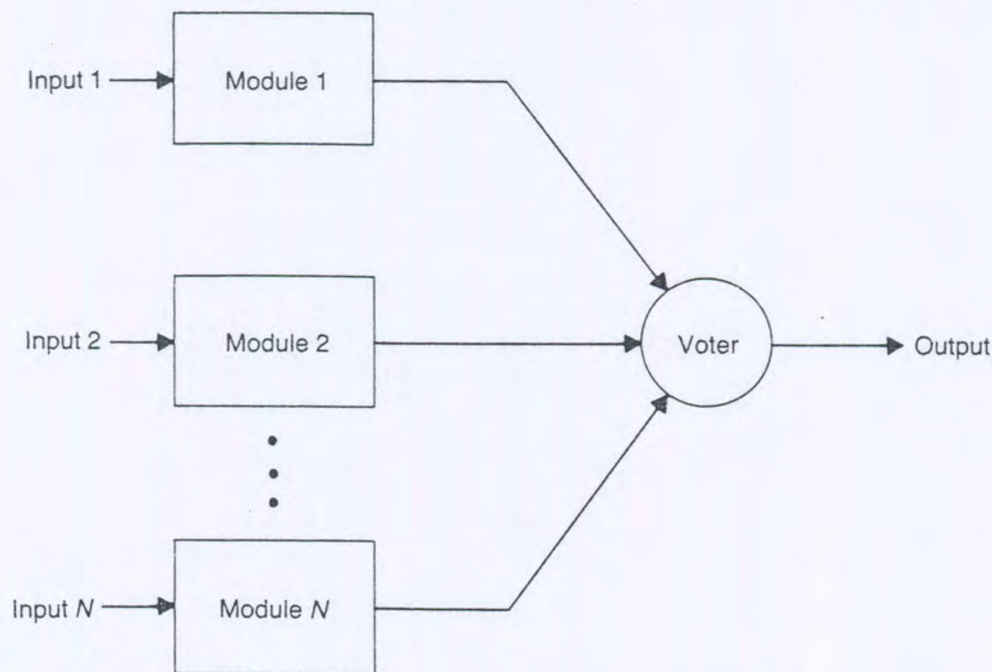


Fig. 3.5 N-modular redundancy (NMR) is a generalization of TMR with N identical modules performing identical operations. If N is odd, majority voting can be used to produce an output.

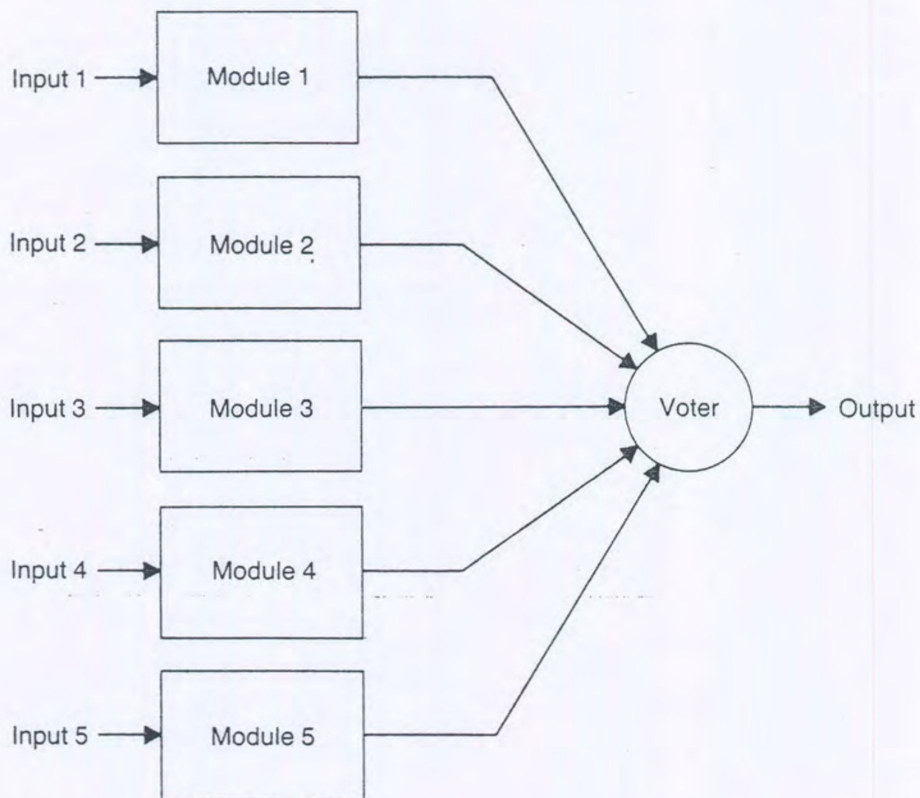


Fig. 3.6 5MR is an example of NMR with five identical modules. Majority voting allows the failure of two modules to be tolerated.

from three independent sensors, perform a vote to determine which of the three sensor values to use, calculate the amount of heat or cooling to provide to the process (the calculations being performed by three or more separate modules), and then vote on the calculations to determine a result. The voting can be performed on both analog and digital data. The alternative, in this example, might be to sample the temperature from three independent sensors, perform the calculations, and then provide a single vote on the final result. The primary difference between the two approaches is fault containment. If voting is not performed on the temperature values from the sensors, the effect of a sensor fault is allowed to propagate beyond the sensors and into the primary calculations. Voting at the sensors, however, masks, and contains, the effects of a sensor fault. Providing several levels of voting, however, does require additional redundancy, and the benefits of fault containment must be weighed against the cost of the extra redundancy.

Voting not only involves a number of design tradeoffs, it also poses several procedural problems. The first is deciding whether a hardware voter will be used or whether the voting process will be implemented in software.

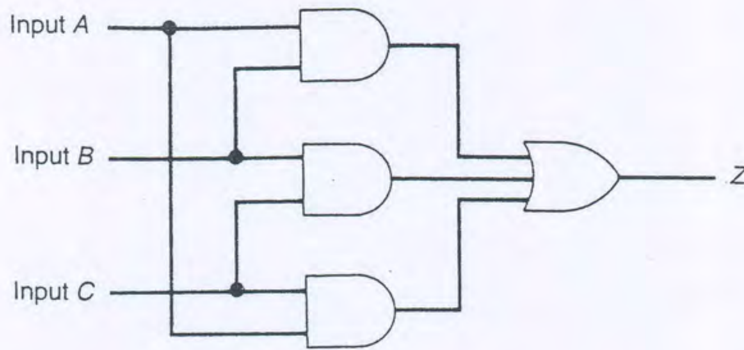


Fig. 3.7 A 1-bit majority voter produces an output of 1 when two out of three inputs are 1 and an output of 0 when two out of three inputs are 0.

Hardware voters for digital data are relatively simple and easy to design. For example, the combinational circuit shown in Fig. 3.7 produces an output bit *Z* that is 1 if the majority of the input bits are 1, and 0 if the majority of the input bits are 0. An 8-bit or 16-bit voter can be constructed using eight or sixteen, respectively, of the circuits shown in Fig. 3.7. Each circuit operates independently on the appropriate bits of each word of digital data. The time required to perform the vote using hardware is simply the propagation delay through the digital logic circuit.

In most practical applications, timing is critical in the voting procedure. If values arrive at the voter at slightly different times, incorrect results can be generated temporarily. In many applications, an incorrect result cannot be allowed for even a very small period of time. To overcome timing problems, flip flops can be used at the inputs of the voter to synchronize the voting process. An example is shown in Fig. 3.8 where master-slave flip flops are used to solve timing problems at the inputs of a 1-bit voter. Master-slave flip flops are shown here because of their extensive use in traditional digital systems design. Each *D* flip flop shown in Fig. 3.8 is positive-edge triggered. In other words, each *D* flip flop stores the value present on the *D* input at the time when the clock goes from 0 to 1.

The timing diagram in Fig. 3.8 shows a two-phase clock that drives the master-slave flip flops. The inputs to the voter are stored in the master flip flop on the positive edge of the Phase 1 clock pulse. On the positive edge of the Phase 2 clock, the data is stored in the slave flip flop, which then applies the data to the combinational circuit that actually performs the voting. The output of the combinational circuit is stored in the output master flip flop on the positive edge of the Phase 1 clock, and the voter output *Z* becomes valid after the rising edge of the Phase 2 clock. An 8-, 16-, 32-, or, in general,

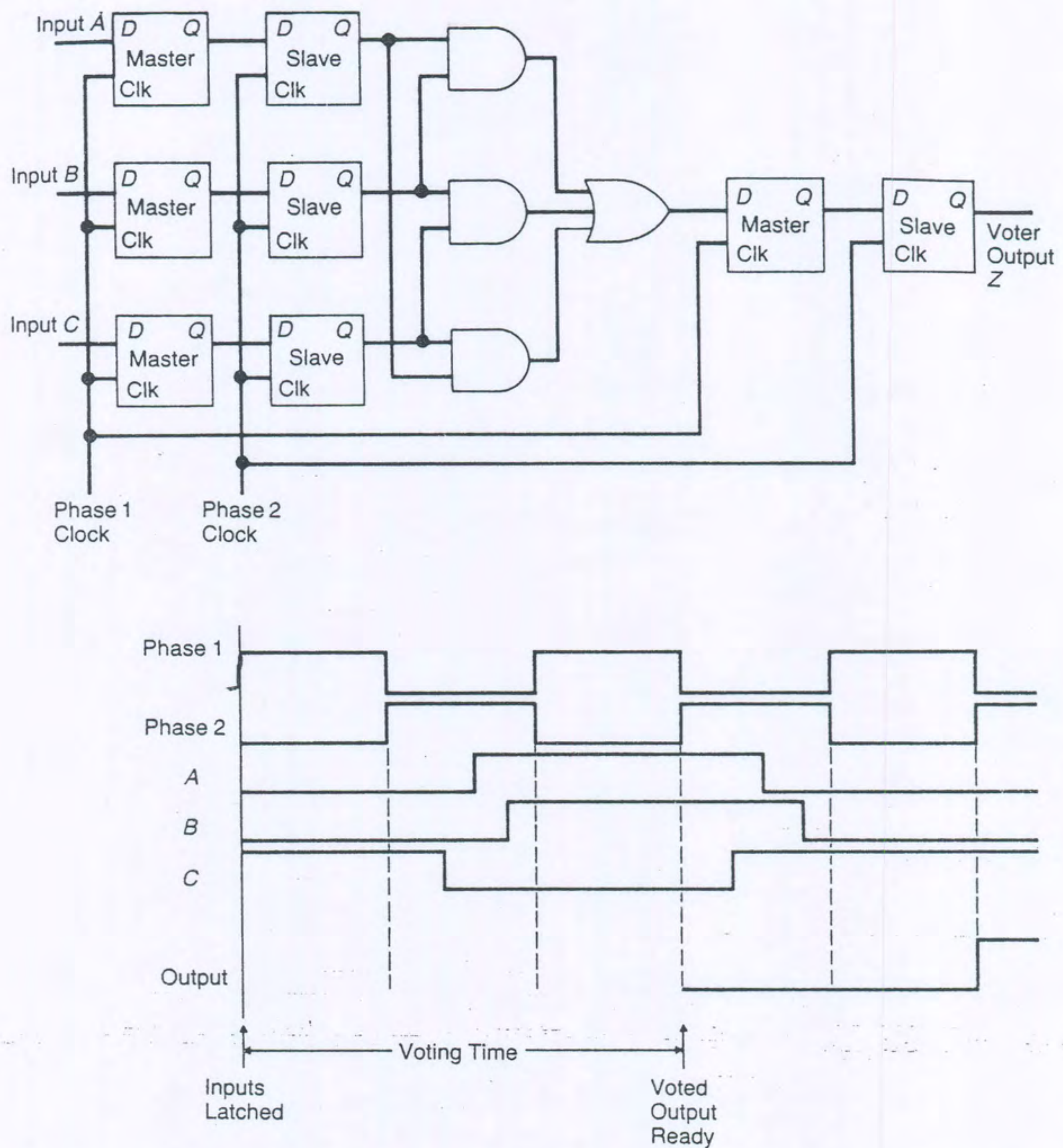


Fig. 3.8 A synchronized majority voter uses latches on the inputs and outputs to synchronize the arrival of inputs and availability of the output.

an n-bit voter can be constructed using an appropriate number of the circuits of Fig. 3.8 operating in parallel.

If voting is performed using software, a mechanism must be available to provide the software routine with the data on which to vote. An example of a microprocessor system that uses software voting is shown in Fig. 3.9. The

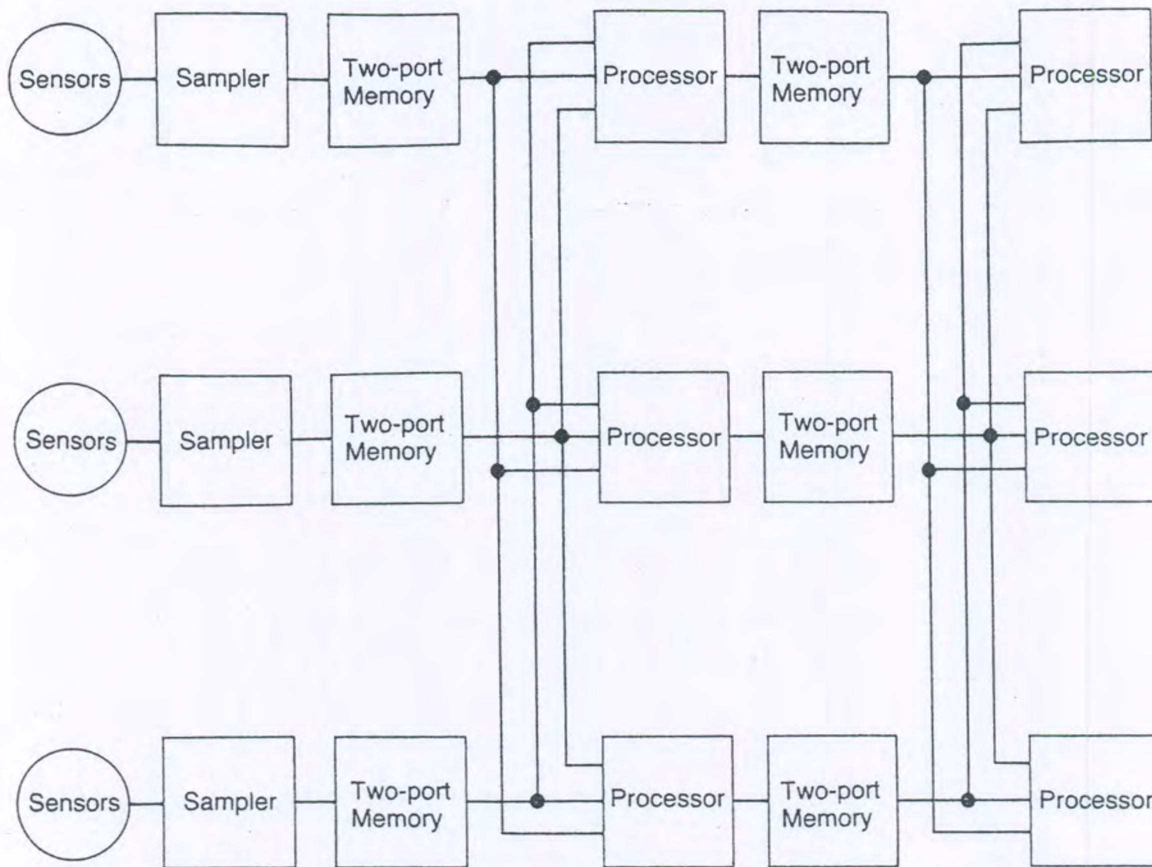


Fig. 3.9 A microprocessor system using software voting. Each processor performs a majority vote on three inputs to determine the appropriate value to use in calculations.

sensor values are sampled and stored in each of three two-port memories. A two-port memory is typically implemented as a standard memory with the address and data lines multiplexed between two users. The memory and the multiplexing scheme are fast enough to give each user the appearance of being the only user of the memory. Each processor can read each memory to obtain all the sensor values. Once each sensor value is read from memory, each processor votes independently on the data to determine which of the three results to use in its calculations. The voting program can be as simple as a sequence of three comparisons, with the outcome of the vote being the value that agrees with at least one of the other two. After completing a set of computations, each processor stores the results in a second set of two-port memories, as shown in Fig. 3.9, so that other processors can use the results. This concept is a software implementation of the triplicated voting scheme presented in Fig. 3.3. Each voter is software implemented and dis-

tributed among three processors such that the failure of any one processor does not disable the voting mechanism.

The primary tradeoff between hardware voting and software voting is speed versus hardware. In a hardware voter, the actual delay between the application of the inputs and the availability of the output can be made very small, depending on the propagation delays of the various electronic components used to construct the circuit. The disadvantage of the hardware voter is the number of logic elements that must be used; for example, a 16-bit voter using the circuit of Fig. 3.8 requires 64 logic gates and 64 master-slave flip flops. Each master-slave flip flop is actually two *D* flip flops, so a total of 128 *D* flip flops must be provided. Also, one hardware voter is required for each "voted" output that must be provided. For example, if TMR with triplicated voters is employed, three distinct hardware voters are required. The impact of the hardware required for the voter is to increase the system's power consumption, weight, and size.

By taking advantage of a processor's computational capabilities, a software voter performs the voting process with a minimum amount of additional hardware. Also, by simply modifying the software, the software voter can modify the manner in which the voting is performed. The disadvantage of the software voter is that the voting can require more time to perform simply because the processor cannot execute instructions and process data as rapidly as a dedicated hardware voter.

The decision to use hardware or software voting typically depends on:

1. The availability of a processor to perform the voting
2. The speed at which voting must be performed
3. The criticality of space, power, and weight limitations
4. The number of different voters that must be provided
5. The flexibility required of the voter with respect to future changes in the system.

A second major problem with the practical application of voting is that the three results in a TMR system, for example, may not completely agree, even in a fault-free environment. The sensors that are used in many control systems can seldom be manufactured such that their values agree exactly. Also, an analog-to-digital converter can produce quantities that disagree in the least significant bits, even if the exact signal is passed through the same converter multiple times. When values that disagree slightly are processed, the disagreement can propagate into larger discrepancies. For example, suppose that two signals x_1 and x_2 are supposed to have the same value, which is A; however, x_1 has a value of A and x_2 has a value of $A + \Delta$. Comparing x_1 and x_2 clearly produces a difference of Δ. However, if a multiplication of each signal by some constant C is performed, the difference is ampli-

fied if C is greater than 1. Specifically, $Cx_1 = CA$ and $Cx_2 = C(A + \Delta)$. The difference between the resulting products is now $C\Delta$ In other words, small differences in inputs can produce large differences in outputs that can significantly affect the voting process. Consequently, a majority voter may find that no two results agree exactly in a TMR system, even though the system may be functioning perfectly.

One approach that alleviates the problem of disagreeing results is called the **mid-value select** technique. Basically, the mid-value select approach chooses a value from the three available in a TMR system by selecting the value that lies between the remaining two. As an example, consider Fig. 3.10. Of three available signals, if two signals are the result of fault-free computations and the third is the result of a faulty calculation, one of the fault-free results should lie between the other fault-free result and the faulty result. The mid-value select technique can be applied to any system that uses an odd number of modules such that one signal must lie in the middle of the others.

Another approach that is often used when quantities never exactly agree is to ignore the least significant bits of the information. In other words, a majority vote is performed but only on the k most significant bits of the data. The assumption is that acceptable disagreements will occur only in the least-significant bits of the data. Disagreements that affect the most sig-

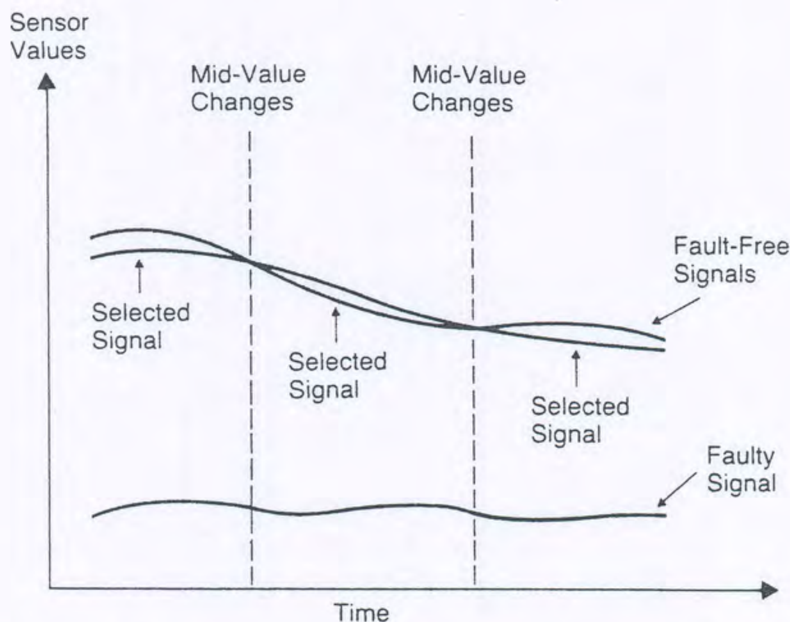


Fig. 3.10 The mid-value select technique chooses the signal that has a value in between the remaining signals. The total number of signals must be odd.

nificant bits of the information are not acceptable and must be corrected. The number of bits that are ignored depends on the application and is a function of the accuracy of the components being used.

The major difficulty with most techniques that use some form of voting is that a single result must ultimately be produced, thus creating a potential point where one failure can cause a system failure. Clearly, single points of failure are to be avoided if a system is to be truly fault-tolerant. The need for a single result is apparent in many applications. For example, banking systems must display one balance for each checking account, not three. Even though the bank's computers may vote internally on some of the results, one result must ultimately be created. The same is true in critical-computation applications such as aircraft flight control. Most aircraft, even military aircraft, do not have redundancy of the actuators, or motors, that physically move the control surfaces. Consequently, a single control signal must be provided.

Several approaches have been used successfully to create single results from redundant computations. One approach is the **flux-summing** technique, which is illustrated in Fig. 3.11. In Fig. 3.11, a TMR system is employed to control the armature current of a small motor. The flux-summing approach uses the inherent properties of closed-loop control systems to compensate for faults. The flux-summer, in this example, is a transformer that has three primary windings and a single secondary winding. The current produced in the secondary winding is proportional to the sum of the individual currents in the three primary windings. Under fault-free circum-

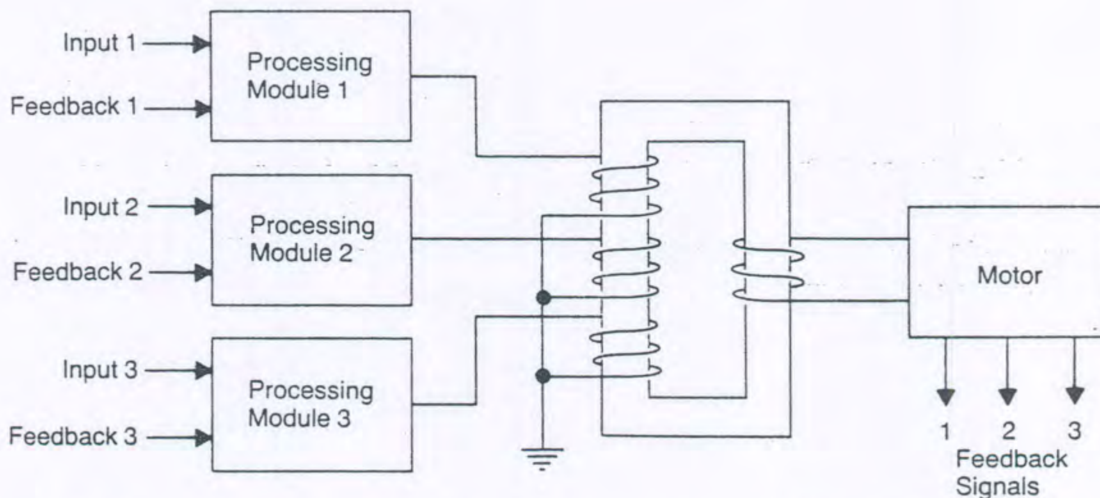


Fig. 3.11 Flux-summing uses the inherent properties of closed-loop control systems to tolerate faults. If one module becomes faulty, the remaining modules automatically compensate.

stances, each module provides approximately one-third of the total current produced in the secondary winding.

If a module fails, several scenarios can result. First, the faulty module may stop providing current to the transformer. In this case, the motor loses approximately one-third of the current necessary to maintain the present shaft position, or shaft velocity, depending on what quantity is being controlled. The remaining two modules sense, via the feedback path, that the motor is deviating from the desired position or velocity. In other words, the error signal produced by each module as part of the closed-loop control process increases. The result is that the two fault-free modules will change the current they are providing to offset the loss of current from the faulty module.

In a second failure scenario, the faulty module may provide a maximum current to the flux-summer, regardless of the input signal values. The inherent feedback of the system once again compensates for the condition by modifying the currents produced by the remaining fault-free modules. To understand the response of the flux-summer, visualize one fault-free module as providing a current of equal magnitude but of opposite polarity to the faulty module such that the effect of the faulty module is canceled. The remaining fault-free module is then capable of controlling the system.

Note that the flux-summing approach is not a voting process, but it has the same effect of masking faults. The flux-summer can be used in the basic TMR approach or in the more general NMR technique. The primary limitation is the number of coils that can be physically mounted on an iron core. The flux-summers can be designed in a very reliable manner and are extremely insensitive to external disturbances of various types.

3.4.2 Active Hardware Redundancy

Active hardware redundancy techniques attempt to achieve fault tolerance by fault detection, fault location, and fault recovery. Because the faults of many designs are detected on the basis of the *errors* they produce, it is often appropriate to use the terms error detection, error location, and error recovery. The property of fault masking, however, is not achieved by using the active redundancy approach. In other words, this approach does not attempt to prevent faults from producing errors within the system. Consequently, active approaches are most common in applications that can tolerate temporary, erroneous results as long as the system reconfigures and regains its operational status in a satisfactory length of time. Satellite systems are good examples of applications of active redundancy. Typically, it is not catastrophic if satellites have infrequent, temporary failures. In fact, it is usually preferable to have temporary failures than to accommodate the high degree of redundancy necessary to achieve fault masking.

Duplication with Comparison

The first example of active redundancy is the simple **duplication with comparison** scheme shown in Fig. 3.12. The basic concept of duplication with comparison is to develop two identical pieces of hardware, have them perform the same computations in parallel, and compare the results of those computations. In the event of a disagreement, an error message is generated. In its most basic form, the duplication concept can only detect the existence of faults, not tolerate them, because there is no method for determining which of the two modules is faulty. However, duplication with comparison can be used as the fundamental fault detection technique in an active redundancy approach.

The duplication with comparison method of active redundancy poses several potential problems. First, if the modules both receive the same input, a failure of the input device or in the lines over which the input signals must be transmitted will cause both modules to produce the same, erroneous results. Second, the comparator may not be able to perform an exact comparison, depending on the application area. Although duplicated telephone switching processors may always exactly agree if they are fault free, the processors in a digital control application may never exactly agree. Finally, faults in the comparator can cause an error indication when no error exists, or, worse yet, the comparator can fail such that eventual faults in the duplicated modules are never detected.

A technique that can be used in a duplicated microprocessor system to overcome some of the problems just mentioned is illustrated in Fig. 3.13. The basic concept is to implement the comparison process in software

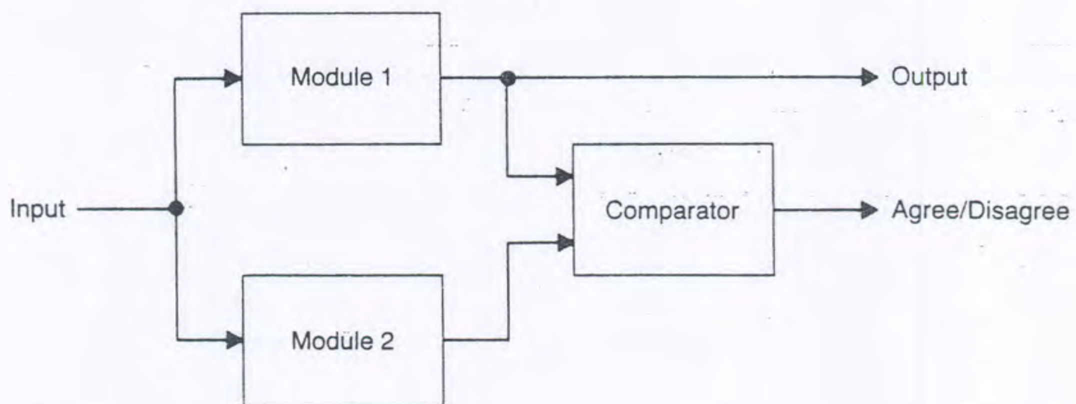


Fig. 3.12 Duplication with comparison uses two identical modules performing the same operations and compares their results. Fault detection is provided but not fault tolerance.

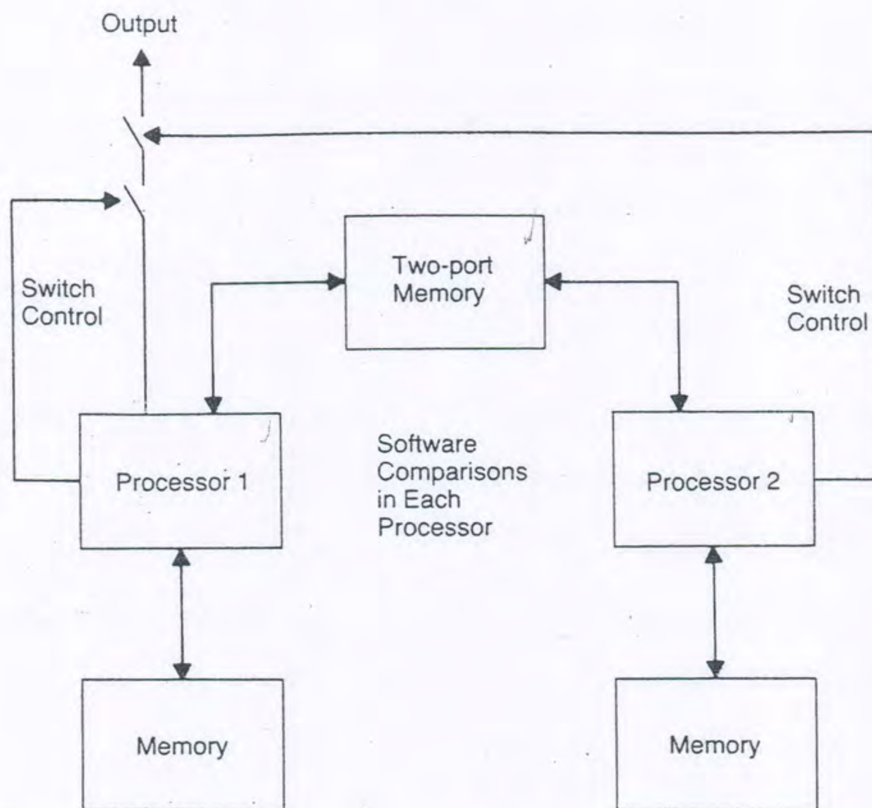


Fig. 3.13 The necessary comparisons in duplication with comparison can be implemented in software. Both processors must agree that results match before an output is generated.

that executes in each of the microprocessors. Each processor has its own memory to store programs and data. In addition, a two-port memory can transfer results from one processor to the other for comparison purposes. The processors perform identical calculations and each place one copy of the result in their own memory and a second copy in the two-port memory. Each processor then reads the other processor's results from the two-port memory and compares the result with its own. If the two-port memory fails, both processors detect a disagreement between the data stored in their own memories and that contained in the two-port memory. If one of the processors fails, the condition can also be detected using this approach. To provide for the capability to disable the outputs of the system in the event that a disagreement occurs, two sets of switches are provided. The switches can be implemented in either the digital or the analog domain, depending on the application. If either processor detects a disagreement with the other, the processor detecting the problem opens its switch and disables the

outputs. Both processors must agree that their computations are identical before the system is allowed to produce an output.

The comparisons between the two processors can be performed in one of several ways. The first, and most straightforward technique, is to simply compare each digital word bit by bit. In hardware, a bit-by-bit comparison can be performed using two-input EXCLUSIVE-OR gates. The two bits to be compared are provided as inputs to the EXCLUSIVE-OR gate, and the output of the gate is 1 if the two bits disagree and 0 if they agree. n EXCLUSIVE-OR gates can be operated in parallel to achieve the desired comparison of two n-bit words. If the output of any one EXCLUSIVE-OR gate is 1, the two words do not exactly agree. In software, the comparisons are easily implemented using a COMPARE instruction that is commonly found in the instruction sets of almost all microprocessors.

Bit-by-bit comparisons have the same problems as found in voting circuits. Specifically, many applications require comparing digital words that do not agree exactly, even though the system is fault free. In such cases, it is common to compare only the most significant bits of words. For example, in the comparison of two 16-bit words, the two least significant bits might be ignored because their impact on the system is negligible. The feasibility of using such a comparison technique clearly depends on the application. In banking systems, for example, bank balances calculated on duplicate processors must agree exactly. In many real-time control applications, however, minor differences are expected and have little, if any, impact. For example, an analog signal ranging in voltage from 0 to +5 volts and generated via an 8-bit digital-to-analog converter changes by no more than approximately 20 millivolts when the least significant bit changes. In fact, the analog signal changes by no more than approximately 80 millivolts when the two least significant bits are changed.

Standby Sparing

A second form of active hardware redundancy is called the **standby sparing** (or standby replacement) technique and is illustrated in Fig. 3.14. In standby sparing, one module is operational and one or more modules serve as standbys, or spares. Various fault detection or error detection schemes are used to determine when a module has become faulty, and fault location is used to determine exactly which module, if any, is faulty. If a fault is detected and located, the faulty module is removed from operation and replaced with a spare. The reconfiguration operation in standby sparing can be viewed conceptually as a switch whose output is selected from one, and only one, of the modules providing inputs to the switch. The switch examines error reports from the error detection circuitry associated with each module to decide which module's output to use. If all modules are provid-

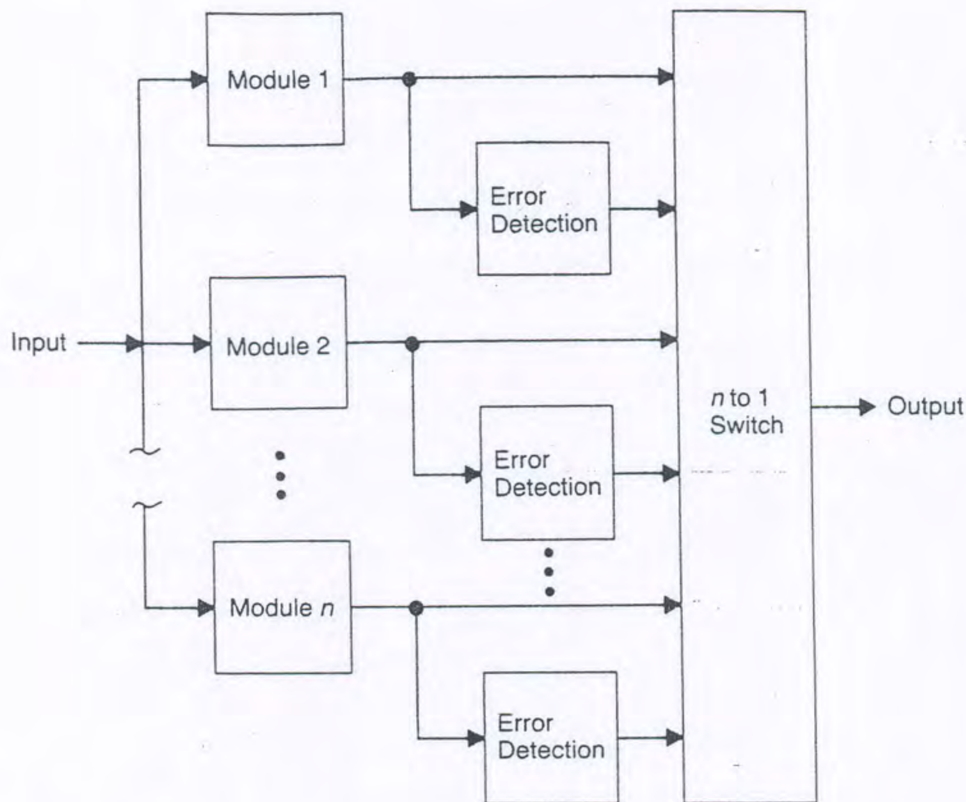


Fig. 3.14 In standby sparing, one of n modules is used to provide the system's output, and the remaining $n - 1$ modules serve as spares. Error detection techniques identify faulty modules so that a fault-free module is always selected to provide the system's output.

ing error-free results, the selection can be made using a fixed priority. Any module that provides erroneous results is eliminated from consideration.

Standby sparing can bring a system back to full operational capability after the occurrence of a fault, but it requires that a momentary disruption in performance occur while the reconfiguration is performed. If the disruption in processing must be minimized, **hot standby sparing** can be used. In the hot standby sparing technique, the spares operate in synchrony with the on line modules and are prepared to take over at any time. In contrast to hot standby sparing is **cold standby sparing** where the spares are unpowered until needed to replace a faulty module. The disadvantage of the cold standby sparing approach is the time required to apply power to a module and perform initialization prior to bringing the module into active service. The advantage of cold standby sparing is that spares do not consume power until needed to replace a faulty module. A satellite application where power consumption is extremely critical is an example where cold standby sparing

may be desirable, or required. A process control system that controls a chemical reaction is an example where the reconfiguration time needs to be minimized, and cold standby sparing is undesirable, or unusable.

A key advantage of standby sparing is that a system containing n identical modules can often provide fault tolerance capabilities with significantly fewer than n redundant modules. For example, consider a multiprocessor containing n processing modules. If each processor is identical and any processor can perform the functions of the other, a single spare processor can protect the system against the failure of any one of the original n processors. In general, k spare processors can protect the system against the failure of any k of the original n processors. The percentage of redundant processors in this case is simply $(k/n) \times 100$.

A key component of the standby sparing approach is the fault detection or error detection scheme used to identify the faulty module. Throughout Chapter 3 we identify several approaches that can be employed, the first of which is the pair-and-a-spare technique.

Pair-and-a-Spare Technique

The **pair-and-a-spare** technique, shown in Fig. 3.15, combines the features present in both standby sparing and duplication with comparison. In essence, the pair-and-a-spare approach uses standby sparing; however, two modules are operated in parallel at all times and their results are compared to provide the error detection capability required in the standby sparing approach. The error signal from the comparison is used to initiate the reconfiguration process that removes faulty modules and replaces them with spares.

The reconfiguration process can be viewed conceptually as a switch that accepts the modules' outputs and error reports and provides the comparator with the outputs of two modules, one of which forms the output of the system. As long as the two selected outputs agree, the spares are not used. When a miscompare occurs, however, the switch uses the error reports from the modules to first identify the faulty module and then select a replacement module. In other words, the switch uses the error information from the comparator and the individual modules to maintain two fault-free modules operating in a duplication with comparison arrangement.

A variation on the pair-and-a-spare technique is to *always* operate modules in pairs. During the design, modules are permanently paired together, and when one module fails, neither module in the pair is used. In other words, modules are always operated and discarded in pairs so that the specific identification of which module is faulty is never required, only the identification of a faulty pair is necessary. Faulty pairs are easily identified based on the outcome of the comparison process.

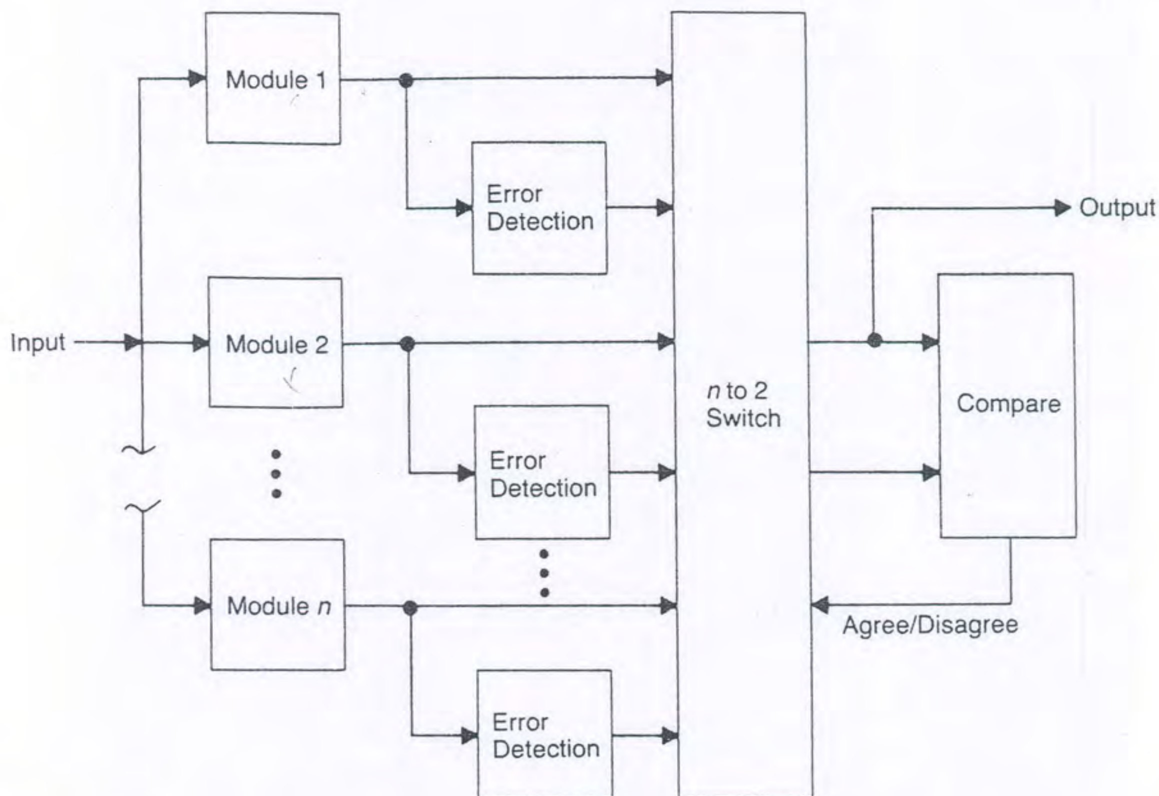


Fig. 3.15 The pair-and-a-spare technique combines duplication with comparison and standby sparing. Two modules are always online and compared, and any spare can replace either of the online modules.

Watchdog Timers

One form of active hardware redundancy that is extremely useful for detecting faults in a system is the **watchdog timer**. A watchdog timer is an active form of hardware redundancy because some action is required on the part of the system to indicate a fault-free status. The concept of a watchdog timer is that the lack of an action is indicative of a fault.

A watchdog timer is a timer that must be reset on a repetitive basis. The failure of the system to perform the reset function results in the system being reset or turned off to prevent a system failure from occurring. The fundamental assumption is that the system is fault free if it possesses the capability to repetitively perform a function such as setting a timer. For example, in standby sparing, each module that successfully and repetitively resets a timer would be interpreted as having fully functional capabilities. A module that cannot reset the timer is removed from consideration as a viable module for use within the system.

The frequency at which the timer must be reset is a function of the system. In an aircraft control system, for example, it may be imperative to detect faults within 100 milliseconds, or less, of their occurrence. Consequently, the watchdog timer must be reset at intervals of less than 100 milliseconds to allow the timer to detect a fault before any catastrophic effects of the fault occur. In banking systems, however, the faults may only need to be detected within one second, for example. Therefore, the timer is set to expire at 1-second intervals.

The watchdog timer provides good fault detection capability for certain types of faults. For example, if a processor simply ceases its functions, the watchdog timer can detect the problem. Also, if a processor becomes overloaded and requires unusual amounts of time to perform its functions, the watchdog timer detects the existence of such conditions. Watchdog timers are particularly useful for detecting a lack of response. As an analogy, if you fail to receive any mail for three or four days, you do not know whether there has not been any mail for you or if the mail has simply not run for some reason. If, however, your postman always left a blank piece of paper in your box, regardless of whether or not you had mail, and you removed the paper each day, you could determine that the postman had indeed come. The concept of the watchdog timer is identical to this simple example.

A watchdog timer can be used to detect faults in both the hardware and the software of a system. In many applications, software routines must execute in prespecified lengths of time. In digital control systems, for example, the routines execute repetitively at specific intervals. If a routine suddenly begins requiring more than the expected time to execute, a fault may have appeared in the software; for example, the fault may be an infinite loop. A timer can be used to detect the condition of a software routine requiring more than the specified length of time. The timer is set to a value that corresponds to the expected execution time at the beginning of the execution period. If the timer expires before the routine completes its tasks, the timer generates an interrupt for the processor. If the routine completes its functions before the timer expires, the timer is reset.

3.4.3 Hybrid Hardware Redundancy

The fundamental concept of hybrid hardware redundancy is to combine the attractive features of both the active and the passive approaches. Fault masking is used to prevent the system from producing erroneous results; and fault detection, fault location, and fault recovery are used to reconfigure the system in the event of a fault. Hybrid redundancy is usually very expensive in terms of the amount of hardware required to implement a system. Consequently, hybrid redundancy is most often used in applications that require extremely high integrity of the computations.

N-Modular Redundancy with Spares

Although there are several approaches to hybrid redundancy, most are based on the concept of **N-modular redundancy (NMR) with spares**. The idea of NMR with spares is to provide a basic core of N modules arranged in a voting, or a form of voting, configuration. In addition, spares are provided to replace failed units in the NMR core. The benefit of NMR with spares is that a voting configuration can be restored after a fault has occurred. For example, a design that uses TMR with one spare will mask the first module fault that occurs. If the faulty module is then replaced with the spare unit, the second module fault can also be masked, thus providing tolerance of two module faults. For a passive approach to tolerate two module faults, five modules must be configured in a fault masking arrangement. The hybrid approach can accomplish the same results using only four modules and some fault detection, location, and recovery techniques.

The NMR with spares technique is illustrated in Fig. 3.16. The system remains in the basic NMR configuration until the disagreement detector determines that a faulty unit exists. One approach to fault detection is to compare the output of the voter with the individual outputs of the modules. A

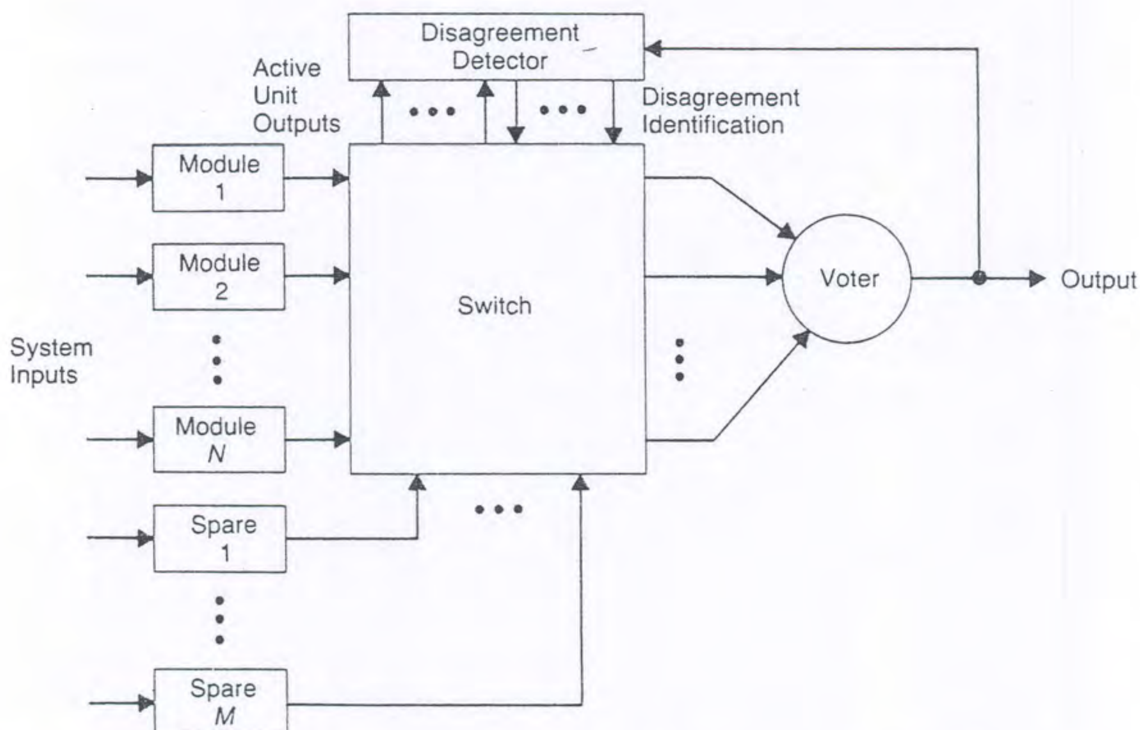


Fig. 3.16 N-modular redundancy with spares combines NMR and standby sparing. The voted output is used to identify faulty modules, which are then replaced with spares.

module that disagrees with the majority is labeled as faulty and removed from the NMR core. A spare unit is then switched in to replace the faulty module. The reliability of the basic NMR system is maintained as long as the pool of spares is not exhausted. Voting always occurs among the active participants in the NMR core, masking faults and ensuring continuous, error-free computations.

Self-Purging Redundancy

A second approach to hybrid redundancy is called **self-purging redundancy** [Losq 1976]. The basic concept of self-purging redundancy is similar to that of the NMR with spares approach. The major difference is that all units are actively participating in the system in the self-purging technique, whereas some units function as spares in the NMR approach and may not be an active part of the system until a fault occurs. The overall concept of self-purging redundancy is illustrated in Fig. 3.17. Each of the N identical modules is designed with the capability to remove itself from the system in the event that its output disagrees with the voted output of the system.

There are three basic features of the self-purging redundancy concept. First, N identical modules are obtained. Each module is capable of performing the functions required of the system. Second, a set of N switches is developed, and one switch is associated with each of the N modules. The function of the switch is to remove, or purge, its associated module from the system in the event that the module fails. Third, a voter is developed to produce the system output and provide masking of any faults that occur. The voter used in the self-purging technique is a threshold gate. Before proceeding to discuss the self-purging approach, it should be beneficial to review the basic concept of a threshold gate [Kohavi 1978].

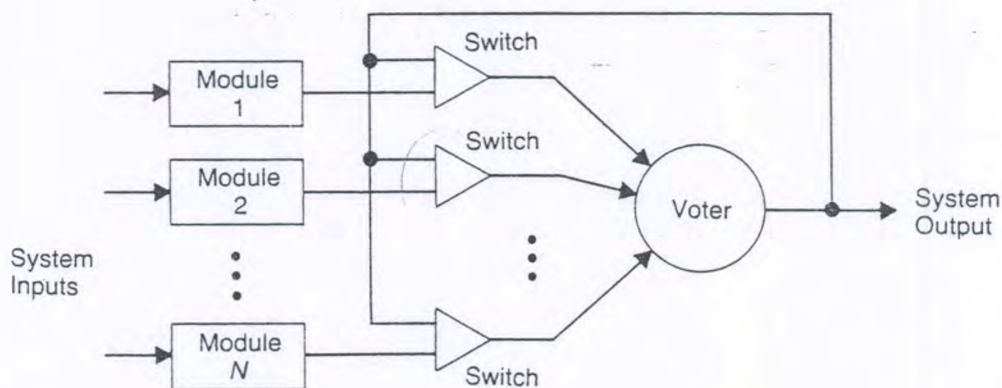


Fig. 3.17 Self-purging redundancy uses the system output to remove modules whose output disagrees with the system output. (From [Losq, 1976] © 1976 IEEE)

A binary threshold gate has n binary inputs, $x_1, x_2, \ldots, x_n$, and one binary output z. Each input is weighted according to the parameters $w_1, w_2, \ldots, w_n$. The output z, of the threshold gate is determined by comparing the weighted sum of the inputs to some prespecified threshold T. In other words, the output z, is given by

$$z = \begin{cases} 1, & \text{if } \sum_{i=1}^{n} w_i x_i \geq T \\ 0, & \text{if } \sum_{i=1}^{n} w_i x_i < T \end{cases}$$

The addition and multiplication functions are the conventional decimal operations. The summation $\Sigma_{i=1}^{n} w_i x_i$ is typically called the weighted sum, and T is the threshold of the gate.

The use of a threshold gate is illustrated in Table 3.1 where three input variables are used, and the threshold is set as 2. Each input is weighted by a factor of one. Any time the weighted sum of the inputs is 2 or greater, meaning at least two of the inputs have a value of 1, the output assumes the value of 1. If the weighted sum is less than 2, meaning at least two of the inputs have a value of 0, the output assumes a value of 0. In this special case, the threshold gate has the same characteristics as a majority voter.

One approach to using a threshold gate in a fault-tolerant design is to force to zero the weight of all modules that have been identified as faulty. Consequently, faulty modules do not contribute to the output of the system. For example, four modules using a threshold gate with a threshold of 2 can produce the correct output after two modules have failed as long as the contribution of the failed modules to the threshold gate is made zero. In a similar manner, the threshold can be varied to allow fewer modules to control the system. For example, if the threshold gate has three inputs, two

5 → 3
4 → 3
3 → 2
2 → 2

TABLE 3.1 Operation of a three-input threshold gate

Inputs X_1	X_2	X_3	Weighted sum	Output
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	2	1
1	0	0	1	0
1	0	1	2	1
1	1	0	2	1
1	1	1	3	1

Threshold = 2

of which have been assigned zero weights because they are faulty, the threshold can be reduced to 1 to allow the remaining fault-free module to control the system.

Perhaps the most difficult problem with threshold gates is the fact that they are analog elements; usually constructed from operational amplifiers, resistor and transistor circuits, or magnetic cores that perform the summation and multiplication operators. Consequently, threshold elements are often not practical for use in fault-tolerant digital systems. This is one reason that the self-purging technique is not commonly used.

In addition to the threshold gate, the switch is the next most basic element of the self-purging approach. The purpose of the switch is to force a module's contribution to the threshold gate to be 0 if that module produces an output that disagrees with the output of the threshold gate. In other words, the switch sets the weight associated with a given module to zero if the output of that module disagrees with the output of the system. This is accomplished through the use of a flip flop with inputs J and K and outputs Q and $\overline{Q}$ that disables the output of a module if that module produces an erroneous output. The basic structure of the switch is shown in Fig. 3.18. Typically, the flip flop has an initialization mechanism that allows a disabled module to once again contribute to the summation process, in the event that the module is restored to a fault-free state. While excluded from the

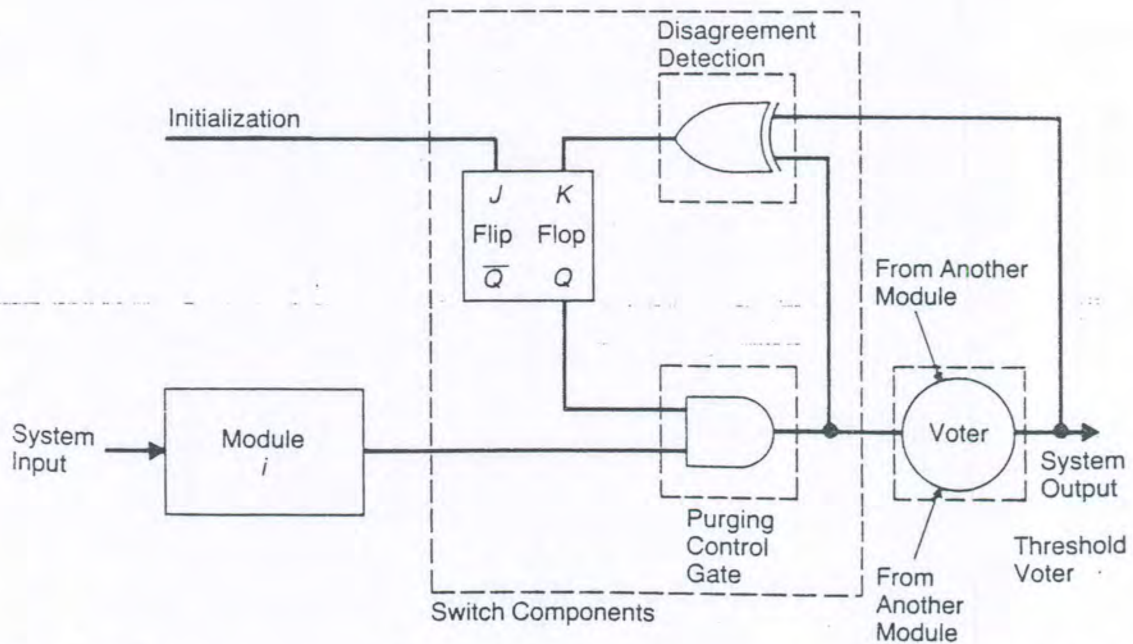


Fig. 3.18 Basic structure of the switch in a self-purging system. The switch removes the module from the voting process if the system output disagrees with the module output. (From [Losq, 1976] © 1976 IEEE)

summation, modules can be removed from the system without affecting the results generated by the system. This is an extremely attractive feature of the self-purging approach that facilitates maintenance and repair; maintenance personnel can disable individual modules and replace them without interrupting the service provided by the system.

As an example of the use of self-purging redundancy, consider the design of a 1-bit full-adder. Three full-adders are used to provide triple redundancy. Each 1-bit full-adder must produce both a sum bit and a carry bit. The logic diagram for the full-adder is shown in Fig. 3.19. Two threshold voters are required in this application; one voter for the sum bit and one for the carry bit. The switching mechanism must be modified to accommodate the multiple outputs. Two approaches can be taken. The first approach is to completely remove a module from both threshold voters in the event that either the carry bit or the sum bit is found to be in error. The problem with this concept is that the module may be perfectly capable of producing the correct carry bit even though the sum bit circuitry is faulty. If the complete module is removed, the redundancy available in both the sum circuitry and the carry circuitry has been reduced when only one may have been faulty. The second approach is to consider the sum and carry circuitry as completely separate. The major problem with this approach is the additional hardware required; a flip flop must be provided for each module's sum output and a separate flip flop provided for each module's carry output. This doubles the number of flip flops required compared to the first approach.

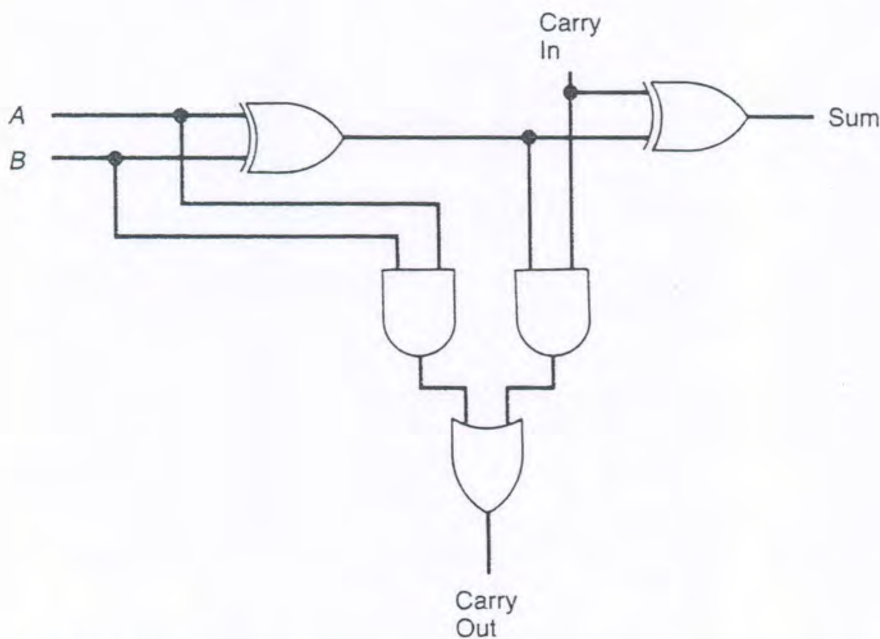


Fig. 3.19 Logic diagram of the full-adder circuit.

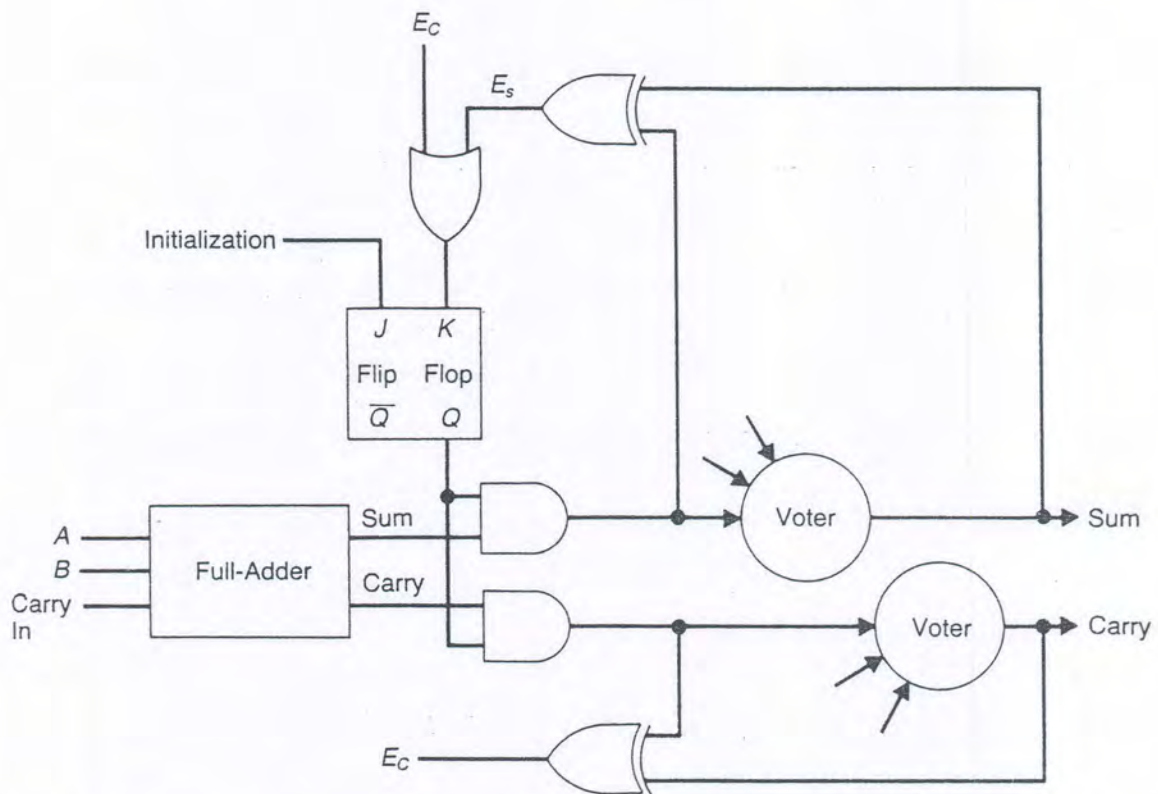


Fig. 3.20 Switching mechanism for implementation of self-purging full adder—both the sum and carry bits are considered. Only one of three switching mechanisms is shown.

The switching mechanism using the first approach is shown in Fig. 3.20. Only one of the three full-adder modules is shown. The remaining two full-adders have identical switching circuitry associated with them. Note that a single flip flop can purge both the contribution of the sum bit and the carry bit by a module. Also note that each module's carry and sum bits are compared to those produced by the voter to create the carry error signal E_c and the sum error signal E_s. The voters in this case are threshold gates.

Sift-Out Modular Redundancy

Another hybrid redundancy method is called **sift-out modular redundancy** [deSousa and Mathur 1978]. Sift-out modular redundancy also uses N identical modules that are configured into a system using special circuits called comparators, detectors, and collectors. The basic structure of the sift-out modular redundancy technique is illustrated in Fig. 3.21.

The function of the comparator is to compare each module's output with the remaining modules' outputs. Thus, the comparator produces one signal for each comparison that can be performed. For example, if five mod-

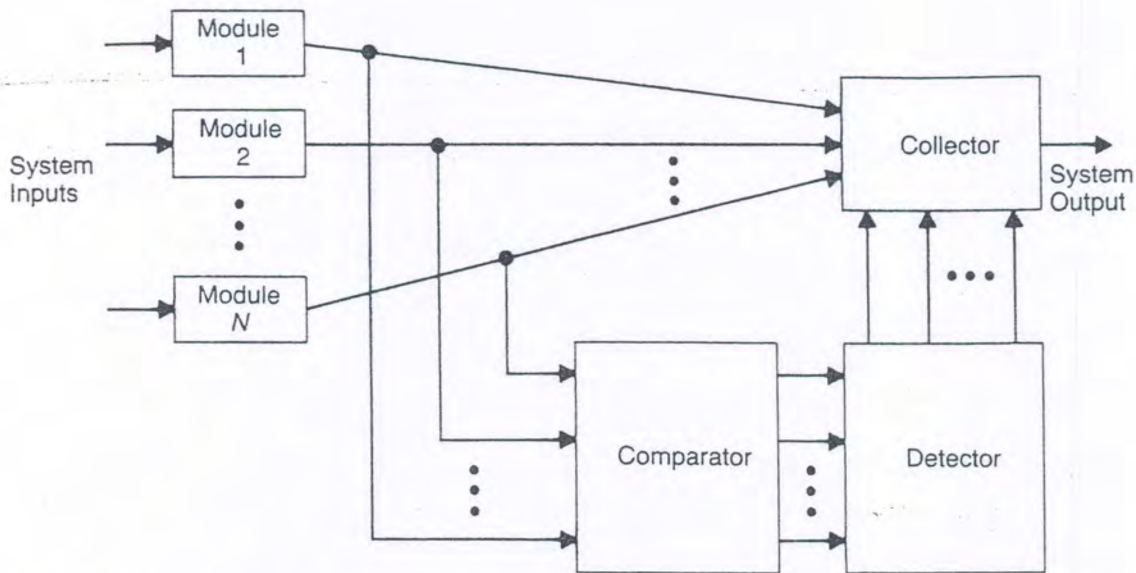


Fig. 3.21 Sift-out modular redundancy uses a centralized collector to create the system output. All modules are compared to detect faulty modules. (From [De Sousa and Mather, 1978] © 1978 IEEE)

ules are used, ten comparisons are performed; if four modules are used, six comparisons are made, and if three modules exist, three comparisons are made. Each signal generated by the comparator is 1 if the two units being compared disagree; and 0, otherwise. The logic diagram for a comparator for three modules is shown in Fig. 3.22.

The function of the detector is to determine which disagreements are reported by the comparator and to disable a unit that disagrees with a majority of the remaining modules. The detector produces one signal for each

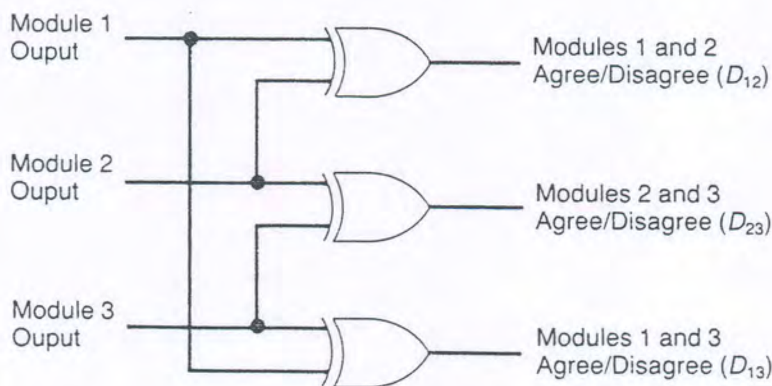


Fig. 3.22 The comparator compares the output of each module with all others to produce disagreement signals (D_{ij}).

module. The value of that signal is 1 if the module disagrees with a majority of the remaining modules, and it is 0 if the module agrees with a majority of the remaining modules. In a manner similar to that of self-purging redundancy, the detector uses flip flops to force modules that are identified as faulty to remain identified as faulty until an initialization is generated. An initialization signal allows modules to be placed back into the system, in a functional sense, once it is decided proper to do so. This feature allows the system to recover from transient faults and facilitates maintenance and repair. The logic diagram of a detector for a three-module system is shown in Fig. 3.23.

The last major component of the sift-out modular redundancy approach is the collector. The function of the collector is to produce the system's output, given the outputs of the individual modules and the signals from the detector that indicate which modules are faulty. A module that is properly identified as faulty is not allowed to influence the output of the system. The logical structure of the collector is simple, as is illustrated in the three-channel collector of Fig. 3.24.

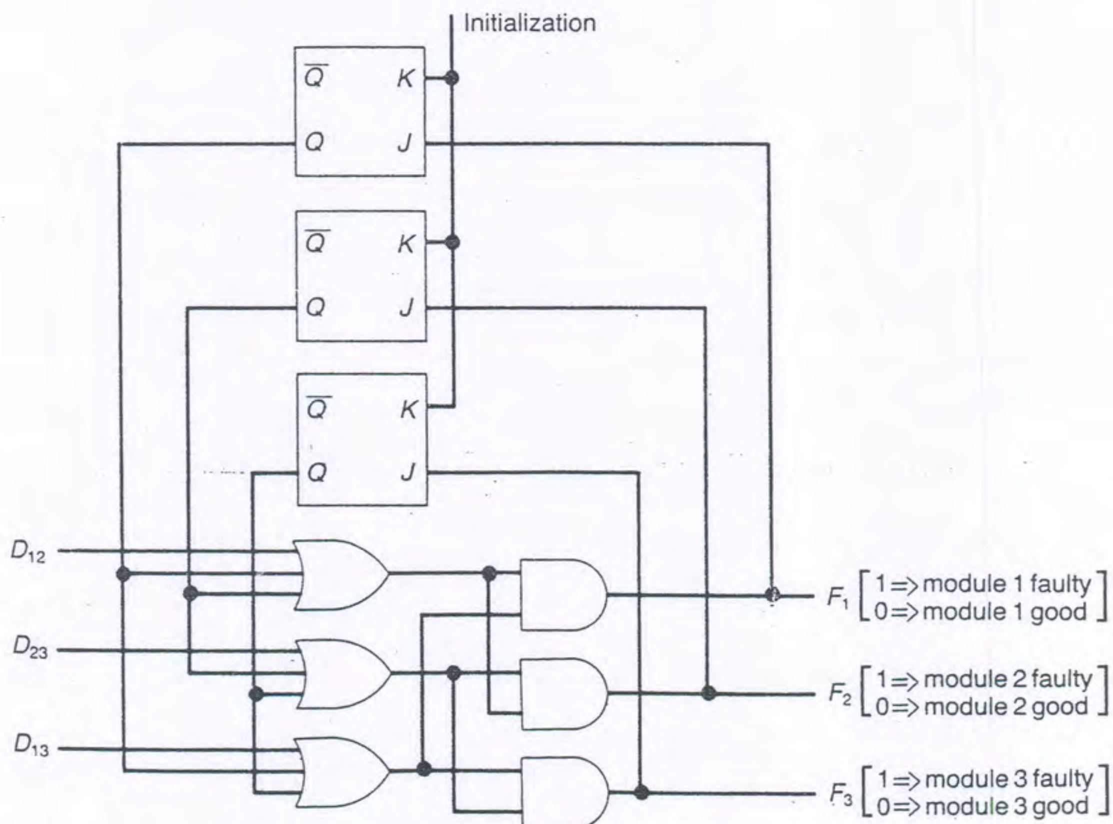


Fig. 3.23 The detector uses the disagreement signals to identify modules as faulty or fault-free.

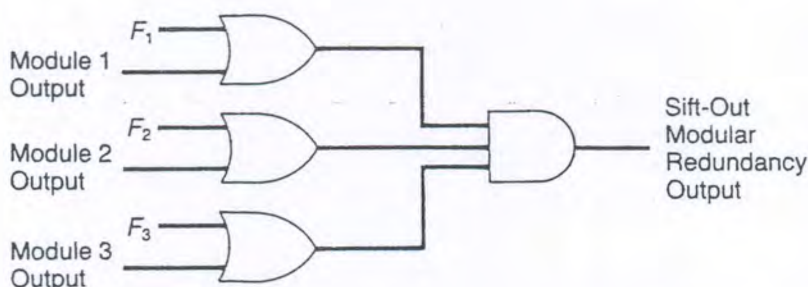


Fig. 3.24 The collector combines module outputs to produce the system output. Contributions from modules identified as faulty are ignored.

The self-purging redundancy concept and the sift-out modular redundancy technique are similar in many respects. Both approaches depend on removing modules from the functional operation of the system to achieve high reliability. The major difference in the two methods is the technique used to identify a module as faulty. The self-purging method compares each module's output with the output of the system, in a way that allows modules to determine independently if they are good or faulty; thus, the name self-purging. Sift-out modular redundancy, on the other hand, uses a more centralized approach to fault detection. A central comparator checks each module against the others to identify faulty modules.

It is interesting to compare the hardware configurations of the self-purging and sift-out modular redundancy approaches. We have already seen the configuration of the triply redundant full-adder using the self-purging approach. The sift-out modular redundancy circuit for the same application is shown in Fig. 3.25. Table 3.2 compares the number of gates and flip flops required in each implementation. In the self-purging circuitry each voter is assumed to be a simple majority voter. As you can see, the number of required elements is identical for both implementations of the full-adder circuit. In general, this result may or may not be true. However, the hardware required for the self-purging and the sift-out approaches is typically very similar.

Triple-Duplex Architecture

The final hybrid redundancy technique is called the **triple-duplex architecture** because it combines duplication with comparison and triple modular redundancy. The use of TMR allows faults to be masked and continuous, error-free performance to be provided for up to one faulty module. The use of the duplication with comparison allows faults to be detected and faulty modules removed from the TMR voting process. The triple-duplex architecture is typically used in control applications where the flux-summing arrangement can be employed as the voting mechanism.

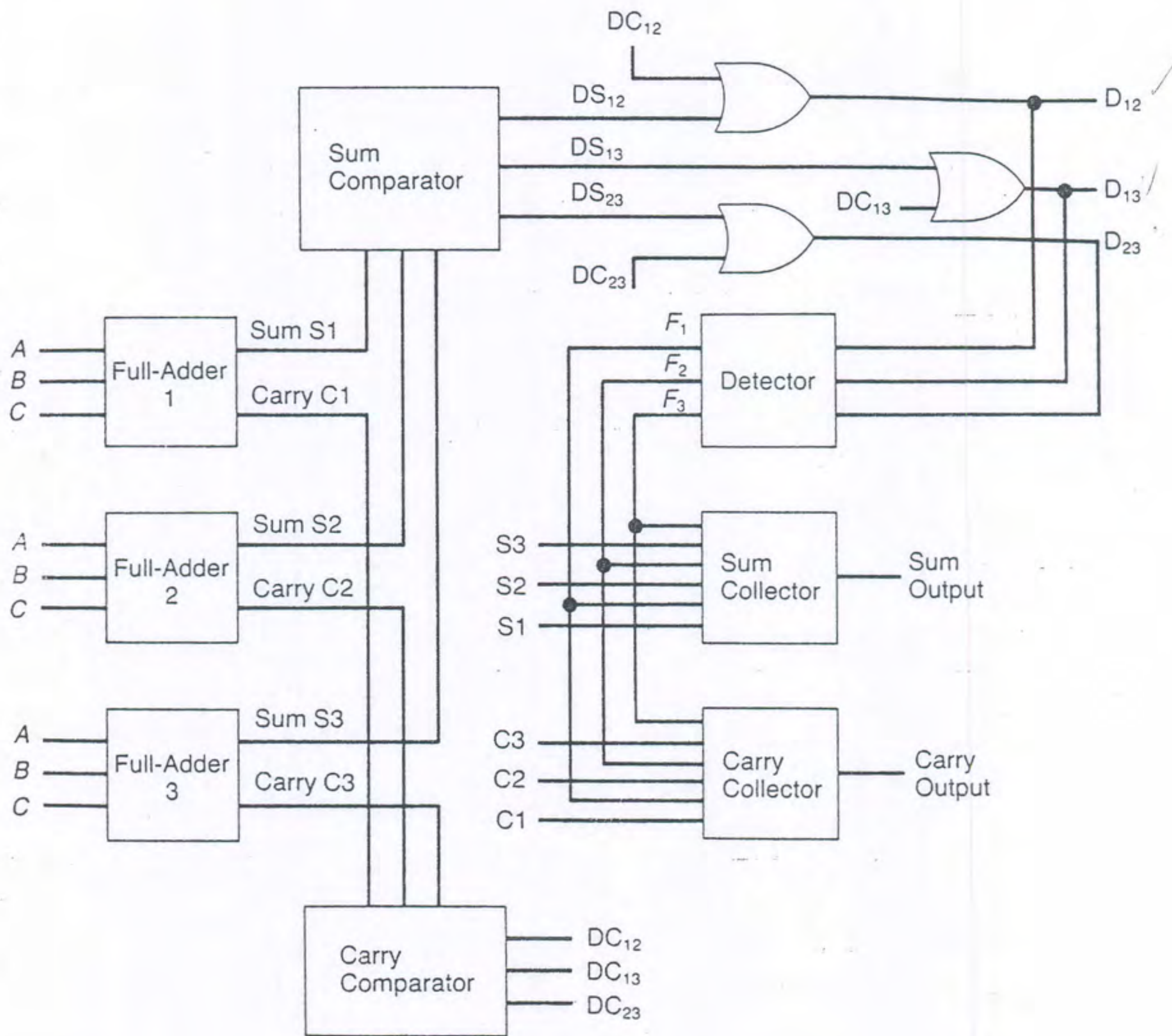


Fig. 3.25 Implementation of a full-adder using the sift-out modular redundancy technique.

The basic structure of the triple-duplex architecture using the flux-summing technique is shown in Fig. 3.26. The flux-summing process is capable of tolerating any single fault that occurs in the system. To allow faults

TABLE 3.2 Comparison of self-purging and sift-out full-adder implementations

Redundancy technique	Number of gates	Number of flip flops
Self-purging	38	3
Sift-out	38	3

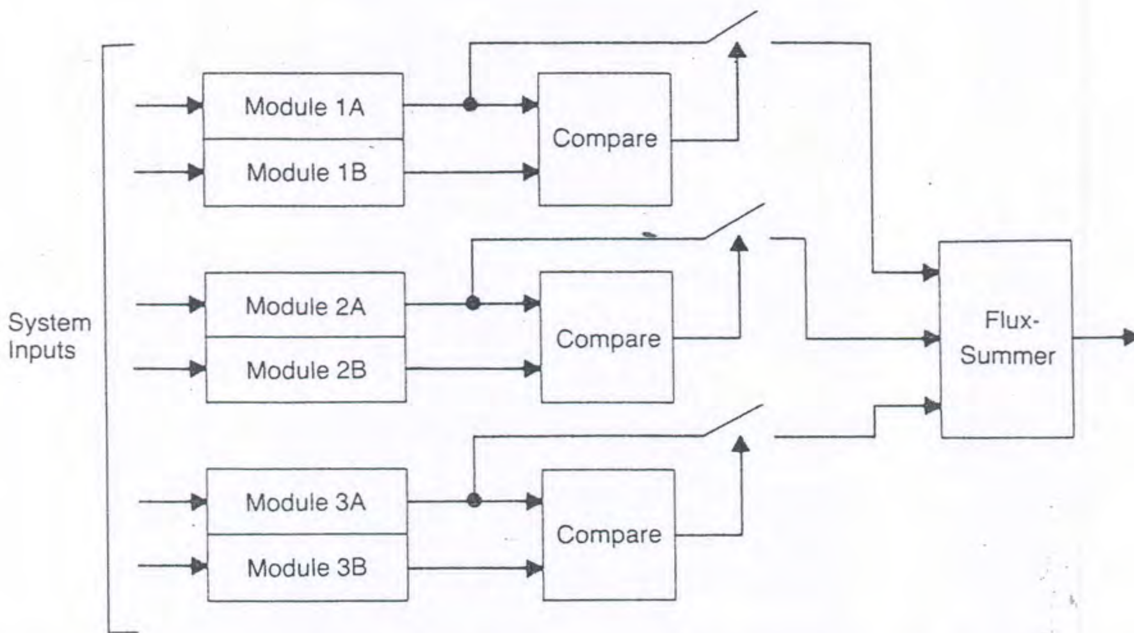


Fig. 3.26 The triple-duplex architecture uses duplication with comparison to detect faulty modules, and triplication is used to provide fault masking.

to be detected, each module is constructed using the duplication with comparison method. If the comparison process detects a fault, the faulty module is removed from the flux-summing arrangement. The removal of faulty modules allows future faults to be tolerated. For example, a single module is capable of providing the necessary control for the system provided the remaining modules are removed from the flux-summer. If the duplication with comparison can detect the faults, the triple-duplex architecture with flux summing can tolerate up to two module faults.

The output of the comparison process can be used to disconnect a module from the flux-summer, as shown in Fig. 3.26. If the comparison detects a fault, the appropriate switch can be opened to remove the affected module from the flux-summer. The remaining modules are then capable of providing the necessary current to the flux-summer. In practical applications, when a module is removed from the flux-summing arrangement, that particular input to the flux-summer is automatically connected to a known, fixed value such as a ground reference.

3.4.4 Summary of Hardware Redundancy

The three primary hardware redundancy techniques—passive, active, and hybrid—each have advantages and disadvantages that are important in different applications. The key differences are as follows:

1. Passive techniques rely strictly on fault masking.
2. Active techniques do not use fault masking but instead employ detection, location, and recovery techniques (reconfiguration).
3. Hybrid approaches employ both fault masking and reconfiguration.

The choice of a hardware approach depends heavily on the application. Critical-computation applications usually mandate some form of either passive or hybrid redundancy because momentary, erroneous results are not acceptable in such systems. The highest reliability is usually achieved using the hybrid techniques. In long-life and high-availability applications, active approaches are often used because it is typically acceptable to have temporary, erroneous outputs; the important thing is that the system can be restored quickly to an operational state using reconfiguration techniques.

The cost, in terms of hardware, of the redundancy techniques increases as we go from active to passive and finally to hybrid. Active techniques typically use less hardware, but they have the disadvantage of potentially producing momentary, erroneous outputs. Passive techniques provide fault masking, but they use substantial investments in hardware. Finally, hybrid approaches provide the advantage of fault masking, but they require enough hardware to use voting *and* they require hardware for spares. Hybrid approaches are typically the most costly in terms of hardware and are used when the highest levels of reliability are required.

3.5 Information Redundancy

Information redundancy is the addition of redundant information to data to allow fault detection, fault masking, or possibly fault tolerance. Good examples of information redundancy are error detecting and error correcting codes, formed by the addition of redundant information to data words, or by the mapping of data words into new representations containing redundant information. Before beginning the discussions of various codes, we will define several basic terms that will appear throughout the textbook [Tang and Chien 1969].

In general, a **code** is a means of representing information, or data, using a well-defined set of rules. For example, a telephone in an office complex can be designed to produce one long ring when the incoming call originated within the complex and two short rings when the incoming call originated outside the complex. The number of rings is a code that allows the recipient of the call to identify, at least partially, the origin of the call.

A **code word** is a collection of symbols, often called digits if the symbols are numbers, used to represent a particular piece of data based on a specified code. A **binary code** is one in which the symbols forming each code word consist of only the digits 0 and 1. For example, a binary coded decimal

(BCD) code defines a 4-bit code word for each decimal digit. The BCD code, for example, is clearly a binary code. A code word is said to be *valid* if the code word adheres to all the rules that define the code; otherwise, the code word is said to be *invalid*.

The **encoding process** is the process of determining the corresponding code word for a particular data item. In other words, the encoding process takes an original data item and represents it as a code word using the rules of the code. For example, given the decimal digit 9, the encoding process determines the BCD code word to be 1001. The **decoding process** is the process of recovering the original data from the code word. In other words, the decoding process takes a code word and determines the data that it represents. For example, the decoding process transforms the BCD code word 0011 into the decimal digit 3.

Of primary interest in this chapter are binary codes. In many binary code words, a single error in one of the binary digits causes the resulting code word to no longer be correct, but, at the same time, the code word is valid. Consequently, the user of the information has no means of determining the correctness of the information. For example, if the BCD code word 0011 is corrupted by complementing the least significant bit to yield 0010, the user of the code word has no way of knowing that the corresponding decimal digit should be 3 instead of 2.

It is possible, however, to create a binary code for which the valid code words are a subset of the total number of possible combinations of 1s and 0s. If the code words are formed correctly, errors introduced into a code word force it to lie in the range of illegal, or invalid, code words, and the error can be detected. This is the basic concept of the error *detecting* codes. The basic concept of the error *correcting* code is that the code word is structured such that it is possible to determine the correct code word from the corrupted, or erroneous, code word.

An **error detecting code** is a specific representation allowing errors introduced into a code word to be detected. For example, suppose that a code word is created and then transferred from one point to another within a system. If the code word is constructed according to an error detecting code, errors that are introduced during the transfer process can be detected at the receiving point.

The basic concept of the error detecting codes is very simple. If a piece of data contains n bits, there are 2^n possible combinations of 1s and 0s for that data. If the code words are structured such that only some of those 2^n combinations are considered to be valid, the occurrence of one of the invalid combinations can be used to signal the existence of an error. For example, the binary coded decimal (BCD) representation encodes each of the ten decimal digits as a 4-bit binary number. Because 4 bits are sufficient to represent 16 unique combinations, several of the combinations are not used

in the BCD code. If one of the unused combinations occurs, an error has been generated, and a relatively simple circuit could be designed to detect that error. The BCD code cannot detect all errors, however. If the BCD representation 0000 were to have the least significant bit corrupted yielding the quantity 0001, the error could never be detected because 0001 is a valid BCD representation. The key to an error detecting code, therefore, is to structure the code words such that some maximum number of errors is guaranteed to result in invalid representations.

In many applications, it is not sufficient to simply detect errors. Often one would like to correct the error before its effects can impact the system. Real-time correction enables the operation of the system to continue in an uninterrupted manner—a mandatory attribute of systems that perform critical computations. A number of techniques are available to allow coding schemes to correct errors. Such codes are called **error correcting codes**. Typically, the code is described by the number of bit errors that can be corrected. For example, a code that can correct single-bit errors is called a *single-error* correcting code. A code that can correct 2-bit errors is called a *double-error* correcting code, and so on.

In general, an error correcting code is a specific representation allowing errors introduced into a code word to be corrected. Once again, suppose that a code word is created and then transferred from one point in a system to another. If the code word is constructed according to an error correcting code, errors that are introduced during the transfer process can be corrected at the receiving point. For example, error *detecting* codes can be used to initiate reconfiguration in an active redundancy scheme, whereas error *correcting* codes can be used to provide fault masking in passive redundancy.

A fundamental concept in the characterization of both error detecting and error correcting codes is the **Hamming distance**. The Hamming distance between any two binary words is the number of bit positions in which the two words differ. For example, the binary words 0000 and 0001 differ in only one position, and therefore have a Hamming distance of 1. The binary words 0000 and 0101, however, differ in two positions; consequently, their Hamming distance is 2. Clearly, if two words have a Hamming distance of 1, it is possible to change one word into the other simply by modifying one bit in one of the words. If, however, two words differ in two bit positions, it is impossible to transform one word into the other by changing one bit in one of the words.

The Hamming distance gives insight into the requirements of error detecting codes and error correcting codes. We define the **code distance** as the minimum Hamming distance between any two valid code words. If a code has a distance of two, any single-bit error introduced into a code word results in the erroneous word being an invalid code word because all valid code words differ in at least two bit positions. If a code has a distance of

three, any single-bit error or any double-bit error results in the erroneous word being an invalid code word because all valid code words differ in at least three positions. However, a code distance of three allows any single-bit error to be corrected, if it is desired to do so, because the erroneous word with a single-bit error is a Hamming distance of 1 from the correct code word and a Hamming distance of 2 from all others. Consequently, the correct code word can be identified from the corrupted code word.

In general, a code can correct up to c bit errors and detect up to d additional bit errors if and only if

$$2c + d + 1 \leq H_d$$

where H_d is the Hamming distance of the code [Nelson and Carroll 1986]. For example, a code with a Hamming distance of 2 cannot provide any error correction, but it can detect single-bit errors. Similarly, a code with a Hamming distance of 3 can correct single-bit errors or detect double-bit errors.

A second fundamental concept of codes is separability. A **separable code** is one in which the original information is appended with new information to form the code word, thus allowing the decoding process to consist of simply removing the unwanted information and keeping the original data. In other words, the original data is obtained from the code word by stripping away extra bits, called the code bits or check bits, and retaining only those associated with the original information. A **nonseparable code** does not possess the property of separability, and, consequently, requires more complicated decoding procedures.

3.5.1 Parity Codes

Perhaps the simplest form of a code is the **parity code.** The basic concept of parity is very straightforward, but there are variations on the fundamental idea. Single-bit parity codes require the addition of an extra bit to a binary word such that the resulting code word has either an even number of 1s or an odd number of 1s. If the extra bit results in the total number of 1s in the code word being odd, the code is referred to as *odd* parity. If the resulting number of 1s in the code word is even, the code is called *even* parity. If a code word with odd parity experiences an error in one of its bits, the parity becomes even. Likewise, if a code word with even parity encounters a single-bit error, the parity becomes odd. Consequently, a single-bit error can be detected by checking the number of 1s in the code words. The single-bit parity code (either odd or even) has a Hamming distance of 2, therefore allowing any single-bit error to be detected but not corrected. The single-bit parity codes (both odd and even) for BCD words are listed in Table 3.3. It is important to note that the parity code is a separable code.

The most common application of parity is in the memories of computer systems. Before being written to memory, a data word is encoded to achieve the correct parity. The encoding is the appending of a bit to force the result-

TABLE 3.3 Odd and even parity codes for BCD data

Decimal digit	BCD	BCD odd parity	BCD even parity
0	0000	0000 1	0000 0
1	0001	0001 0	0001 1
2	0010	0010 0	0010 1
3	0011	0011 1	0011 0
4	0100	0100 0	0100 1
5	0101	0101 1	0101 0
6	0110	0110 1	0110 0
7	0111	0111 0	0111 1
8	1000	1000 0	1000 1
9	1001	1001 1	1001 0
		↑ Parity bit	↑ Parity bit

ing word to have the appropriate number of 1s. When the data word is subsequently read from memory, the parity must be checked to verify that it has not changed as a result of a fault within the memory. If an error is detected, the user of the memory is notified via an error signal that a potential problem exists. Additional hardware is required to handle the extra information (the extra bit appended to each word). For example, the memory must contain one extra bit per word to store the extra information, and the hardware must be designed to create and check the parity bit. As can be seen, the information redundancy concept often requires hardware redundancy as well.

The organization of a memory with parity coding is shown in Fig. 3.27. The data entering the memory also passes into a parity generator that per-

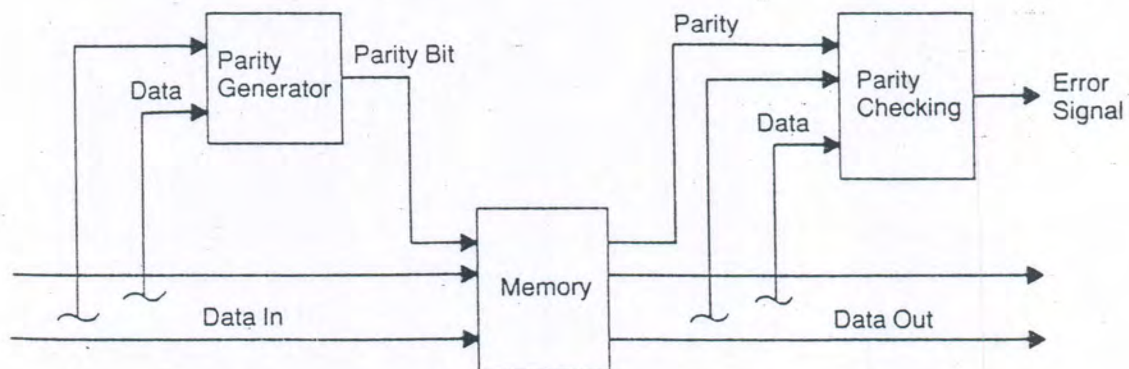


Fig. 3.27 Organization of a memory that uses single-bit parity. The parity bit is generated when data is written to memory and checked when data is read.

forms the encoding process. The generated parity bit and the original data are then stored in memory as a complete code word. When the code word is read from memory, the data portion of the code word passes through a parity checker that regenerates the parity bit and compares it to the original parity bit stored in memory as part of the code word. A disagreement between the original parity bit and the regenerated parity bit causes an error signal to be generated.

Circuits to generate and check parity bits are relatively simple. Each data bit must be examined to determine the total number of 1s present, such that the value of the parity bit can be specified. Recall that the EXCLUSIVE-OR operation, when performed on several bits, produces a 1 if the group of bits contains an odd number of 1s, and a 0 if the group contains an even number of 1s. This is exactly the function required to generate and check parity. Note that the parity generation process and the parity checking process are the same function. Therefore, a circuit that can check parity can also generate parity. If we limit ourselves to use two-input, EXCLUSIVE-OR gates, a circuit that generates and checks even parity for 4-bit data words is shown in Fig. 3.28. The generated parity line is 1 if the data bits d_0 through d_3, contain an odd number of 1s; otherwise, the generated parity bit is 0. When the parity is checked (for example, when the data is read from memory), parity is first regenerated for the data bits, and the regenerated parity bit is compared to the parity bit P that was stored in memory. If a disagreement between the two bits occurs, the error signal becomes 1.

The single-bit parity code has a minimum Hamming distance of 2. This is easily seen by examining the combinations of 1s and 0s available when the code is constructed. Suppose that the original data word consists of n

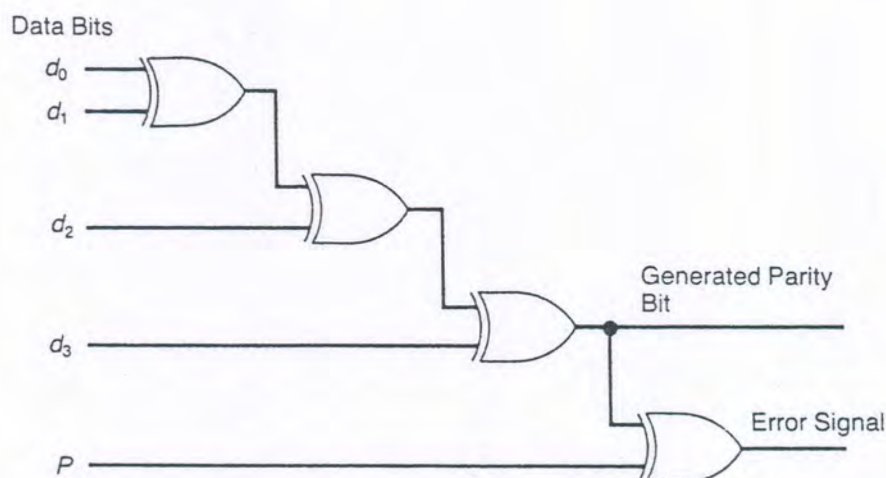


Fig. 3.28 A 4-bit parity generation and checking circuit for even parity.

bits, such that 2^n possible combinations are available. The addition of a parity bit yields a code word with $n + 1$ bits, or 2^{n+1} different combinations. In other words, the addition of the parity bit results in twice as many combinations being available. Of the 2^{n+1} combinations, $2^{n+1}/2$ have an odd number of 1s and $2^{n+1}/2$ have an even number of 1s. For an odd parity code, only those combinations that have an odd number of 1s are valid code words; the others are invalid. Likewise, for an even parity code, only those combinations that have an even number of 1s are valid code words. Changing any single bit in a code word changes the parity of the code word. For example, changing a 1 to a 0 decreases the number of 1s by one, and changing a 0 to a 1 increases the number of 1s by one. In either case, a word with odd parity is mapped into a word with even parity by a 1-bit error. Likewise, a word with even parity is mapped into a word with odd parity by a 1-bit error. To map an odd parity code word into another odd parity code word requires changing a minimum of two bits. Similarly, changing an even parity code word into another even parity code word requires changing a minimum of two bits. Therefore, the distance of the single-bit parity code is two. Any single-bit error can be detected by the single-bit parity code.

One of the biggest problems with single-bit parity codes is their inability to guarantee the detection of some very common multiple-bit errors. For example, a memory can be constructed from individual chips that each contain several bits; 4 bits is a very common number. If a chip in the memory becomes faulty, the simple parity code may be unable to detect the resulting error because multiple bits are affected. The basic parity scheme can be modified to provide additional error detection capability. There are five basic parity approaches:

1. Bit-per-word parity
2. Bit-per-byte parity
3. Bit-per-chip parity
4. Bit-per-multiple-chips parity
5. Interlaced parity

The basic concept of each approach is illustrated in Fig. 3.29.

The basic idea of **bit-per-word parity** is to append one parity bit to each word. The primary disadvantage of the bit-per-word approach is that certain errors can go undetected. For example, if a word, including the parity bit, becomes all 1s because of a complete failure of a bus or a set of data buffers, the odd parity method only detects the condition if the total number of bits, including the parity bit, is even. Likewise, even parity only detects this problem if the total number of bits is odd. In a similar manner, the condition of all bits becoming 0 can never be detected by the even bit-per-word parity method because 0 is considered to be an even number of 1s. Odd bit-per-word parity always detects the condition of all bits being 0.

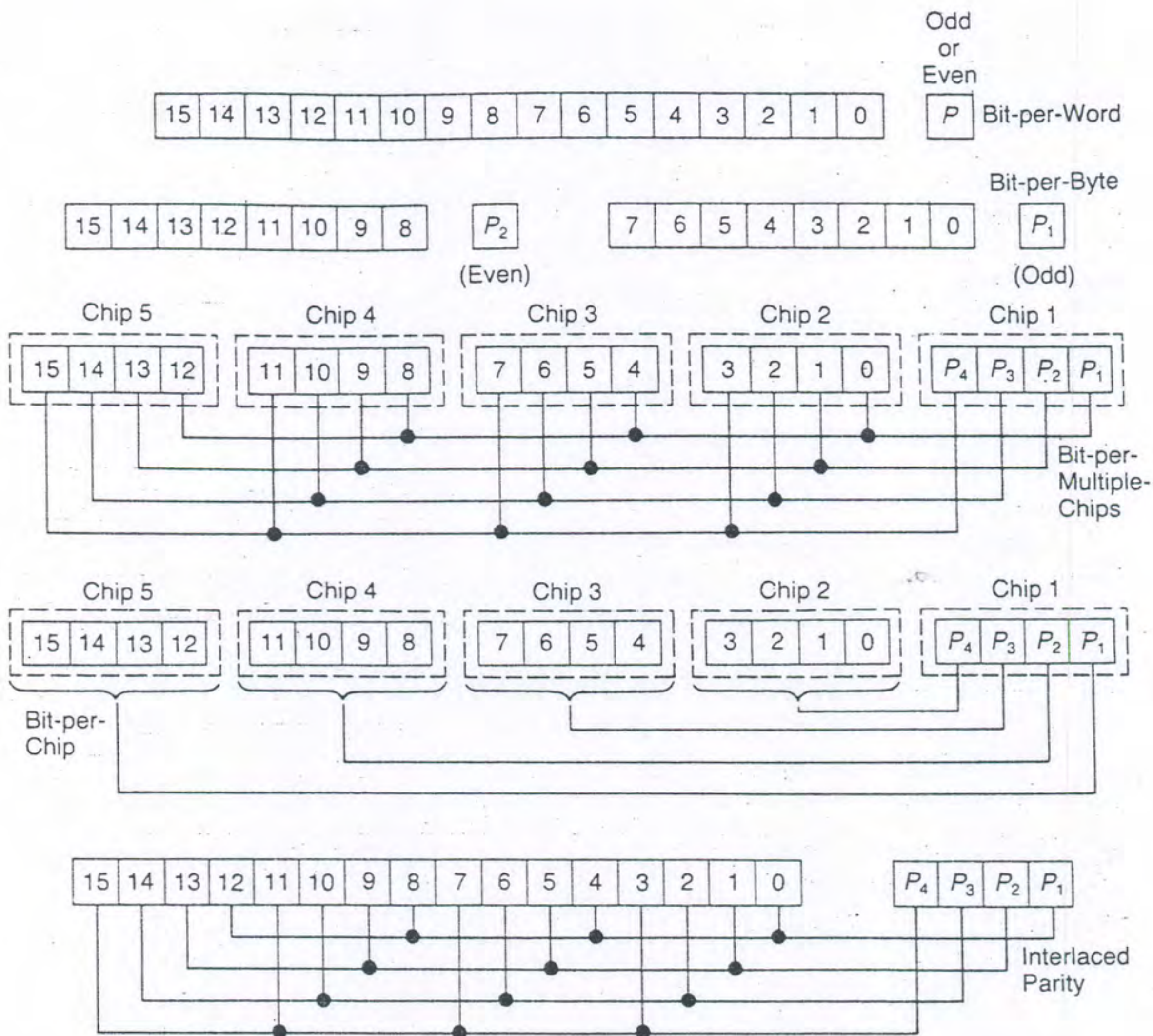


Fig. 3.29 The five basic forms of the parity code include: bit-per-word, bit-per-byte, bit-per-multiple-chips, bit-per-chip, and interlaced parity.

An alternate approach is the **bit-per-byte parity** technique. Here, two parity bits are used on two separate portions of the original data. The technique is called bit-per-byte, but the parity groups can be any number of bits; not just the 8 bits normally associated with the term byte. To gain the full advantages of the approach, however, the number of information bits associated with each parity bit should be even. Also, the parity of one group should be even and the parity of the other group should be odd. The primary advantage of this approach is the ability to detect both the *all-1s* and *all 0s* conditions. If the complete code word becomes all 1s, the even parity bit is erroneous. If the complete code word becomes all 0s, the odd parity bit is erroneous. In both cases, the erroneous conditions are detected.

The bit-per-byte technique also provides additional protection against multiple-bit errors; for example, 2-bit errors will always be detected as long as one bit is in the even parity group and the other is in the odd parity group.

The fundamental disadvantage of both the bit-per-word and bit-per-byte parity approaches is the ineffective detection capability for multiple-bit errors. Many memories are organized using memory chips that contain either 4 bits, 8 bits, or more, of memory. Several of these chips are then used in parallel to form the complete number of bits of each word in the memory. If one chip fails (this is called the whole-chip failure mode), several bits of each word of memory can be affected. Therefore, the single-bit error assumption is often ineffective.

One approach that is useful for detecting the failure of a complete chip is the **bit-per-multiple-chips parity** method, as illustrated in Fig. 3.29. The basic concept is to have one bit from each chip of the memory associated with a single parity bit. Sufficient parity bits are provided to allow each data bit within a chip to be associated with a distinct parity bit. In Fig. 3.29, for example, parity bit P_1 establishes the parity of a group of bits including bits 0, 4, 8, and 12. Note that each parity group includes one, and only one, bit from each chip. If one chip fails, all the parity groups are affected, but no more than one bit in each parity group is corrupted, so the parity code detects the error.

At first it might appear that the bit-per-multiple-chips approach is relatively expensive because at least one chip of the memory is necessary to store the parity bits. If each chip is 4 bits wide, a memory capable of storing 16-bit words requires four of these chips. If bit-per-word parity is used, an extra chip must be added to store just that one parity bit. Typically, the same type of chip is used to store the parity bit and to store the data, so a 4-bit chip is used to store a single bit in the bit-per-word approach. The bit-per-multiple-chips approach simply takes advantage of the extra bits that probably are present in the design anyway. Consequently, the cost of using the bit-per-multiple-chips approach in this simple example is minimal.

One disadvantage of the bit-per-multiple-chip parity approach is that the failure of a complete chip is detected, but it is not located. The failure of any one chip causes all parity groups to be in error, so the cause of the problem cannot be identified. One procedure that overcomes this problem is the **bit-per-chip parity** organization. Here, each parity bit is associated with one chip of the memory, as illustrated in Fig. 3.29. Specifically, parity bit P_1 establishes correct parity for the group containing data bits 12, 13, 14, and 15. If a single bit becomes erroneous, the existence of the error is detected, and the chip that contains the erroneous bit is identified. This is extremely valuable from a maintenance standpoint; not only does the system have the capability to warn of the occurrence of a problem, but the system can direct maintenance personnel to the source of the problem. The primary disadvan-

tage of the bit-per-chip parity method is the susceptibility to the whole-chip failure mode. Because the basic parity code can detect only single errors, the multiple error condition associated with the failure of a complete chip can go undetected.

An alternate organization of the parity code is called **interlaced parity.** Interlaced parity is very similar in form to the bit-per-multiple-chips approach with one key difference. In interlaced parity, the parity groups are formed without regard to the memory's physical organization. This is in contrast to the bit-per-multiple-chip organization, which is intimately tied to the physical structure of the memory. In interlaced parity, the information bits are divided into equal-sized groups, and one parity bit is associated with each group. The bits of each group are then positioned such that no two adjacent bits are from the same parity group. This is accomplished by placing the first bit from group 1 in the least significant bit position, the first bit from group 2 in the next most significant position, the first bit from group 3 in the next position, and so on. Once the first bits of each group are placed, the remaining bits are added to the word in a similar manner. The basic idea is illustrated in Fig. 3.29.

The interlaced parity method is most often used when errors in adjacent bits are of major concern. Because no two adjacent bits are in the same parity group, errors in any two adjacent bits are detected. A good example is a parallel bus; in many buses, two adjacent bits can become shorted together. The interlaced organization of parity detects errors due to this type of fault.

In the approaches we have considered so far, each bit was contained in one and only one parity group. In the **overlapping parity** approach, however, parity groups are formed with each bit appearing in more than one parity group. The primary advantage of overlapping parity is that errors can be located in addition to being detected. Once the erroneous bit is located, it can be corrected by a simple complementation, if desired. Overlapping parity is the basic concept of some of the Hamming error correcting codes.

Figure 3.30 illustrates the basic idea of overlapping parity when applied to 4 bits of information. Three parity groups are required to uniquely identify each erroneous bit in the 4 bits of information. The concept of overlapping parity is to place each bit in a unique combination of the parity groups. For example, referring to Fig. 3.30, bit 3 appears in each parity group, whereas bits 0, 1, and 2 appear in different combinations of two groups. If any one bit becomes erroneous, the impact is unique, as is illustrated in Fig. 3.30. For example, if bit 3 is in error, all the parity groups are affected, but if bit 1 is erroneous, the parity groups associated with P_0 and P_2 are affected and P_1 is unaffected. As shown in Fig. 3.30, each possible single-bit error produces a unique impact on the parity of the three groups.

The overlapping parity approach can be transformed into an error correction scheme by using several comparators (EXCLUSIVE-OR gates) and a

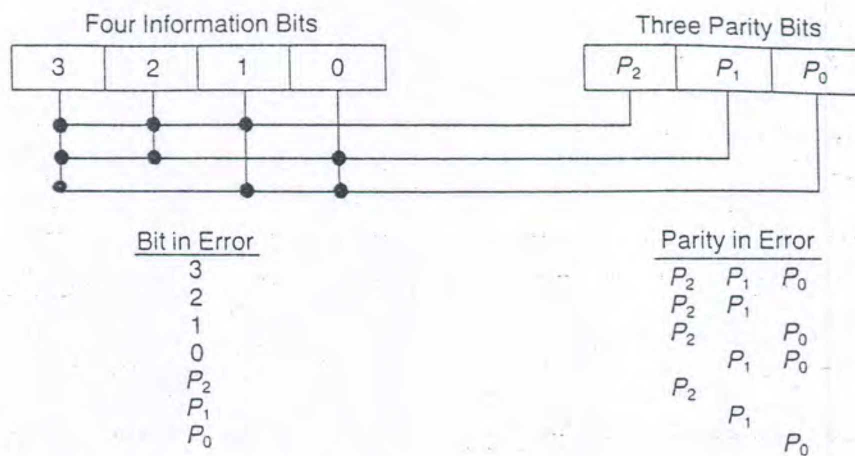


Fig. 3.30 Overlapping parity assigns each bit to multiple parity groups.

decoder in conjunction with the parity checking circuits. In addition, the correction process is performed by complementing the appropriate bit using an EXCLUSIVE-OR gate as a programmable inverter. The concept is illustrated in Fig. 3.31 for 4 bits of information and three parity groups. When the 4 bits of information are written to memory, the three parity bits are generated and stored with the original four bits of information as a single code word. When the code word is subsequently read from memory, the parity bits are regenerated using the parity generation circuits. The regenerated parity bits P_{r0}, P_{r1}, and P_{r2}, are compared to the parity bits that were stored with the original information in memory. The results of the comparisons are fed to a 3-8 decoder that produces a logic 1 on one of its outputs and a logic 0 on all the others, based on the values of its three inputs. For example, the decoder is wired such that all 0s on its inputs forces a 1 to occur on the "no error" line. If the input to the decoder is all 1s, the output labeled "correct bit 3" is set to 1 indicating that bit 3 is the erroneous bit that needs correcting. The correction is performed using a simple collection of EXCLUSIVE-OR gates. If the correction line on one of the EXCLUSIVE-OR gates is 1, the bit is complemented, and therefore corrected. If the correction line is 0, the associated bit is passed through the EXCLUSIVE-OR gate uncomplemented, and therefore unchanged.

The penalty for using overlapping parity on 4 bits of information is high; 3 parity bits are required to detect and locate errors for the 4 bits of information, a redundancy of 75%. As the number of information bits increases, however, the number of parity bits required becomes a smaller percentage of the number of actual information bits. The required relationship between the number of information bits and the number of required parity bits can be determined in a fairly simple manner. Let m be the number of

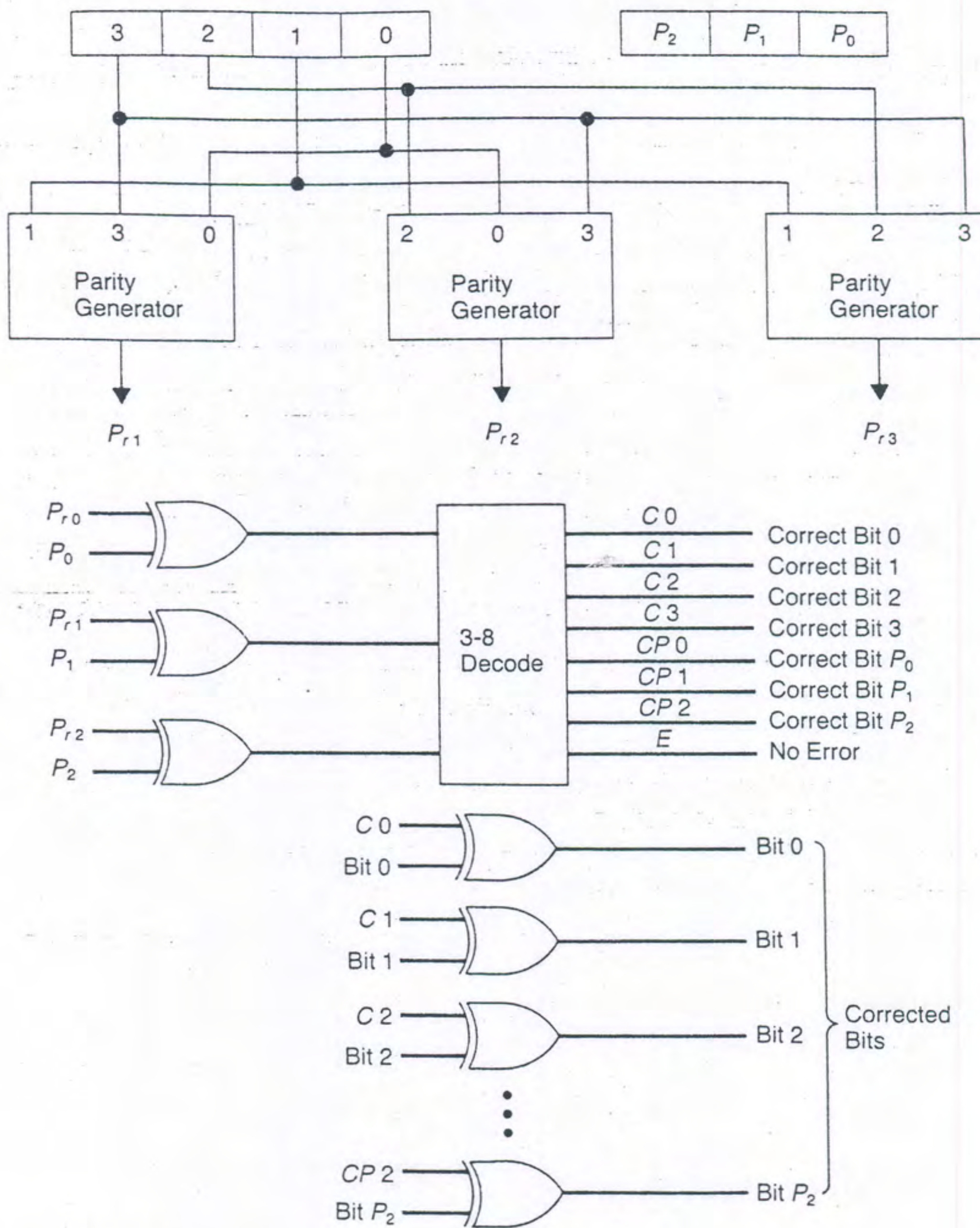


Fig. 3.31 Error correction using overlapped parity.

information bits to be protected using an overlapping parity approach, and let k be the number of parity bits required to protect those m information bits. Each bit error that can occur must produce a unique result when the parity is checked. With k parity bits, there are 2^k unique outcomes of the parity checking process. With k parity bits and m information bits, there are $m + k$ different, single errors that can occur. So, we know that 2^k must be at

least as large as $m + k$. Also, we must have a unique result of the parity checks when there is no error, so the total number of unique combinations must be at least as large as $m + k + 1$. Therefore, the relationship between k and m must be

$$2^k \geq m + k + 1$$

Table 3.4 shows the minimum number of parity bits required, as a function of the number of information bits, for the overlapping parity approach. Note that the percentage of redundancy decreases substantially as the number of information bits increases.

3.5.2 *m*-of-*n* Codes

The basic concept of the ***m*-of-*n* code** is to define code words that are n bits in length and contain exactly m 1s. As a result, any single-bit error forces the resulting erroneous word to have either $m + 1$ or $m - 1$ 1s, and the error can be detected. The primary advantage of the m-of-n code is the conceptual simplicity; it is very easy to visualize the error detection process. The major disadvantage, however, is that the encoding, decoding, and the detection processes are often extremely difficult to perform, despite their conceptual simplicity.

The easiest way to construct an m-of-n code is to take the original i bits of information and append i bits. The appended bits are chosen such that the resulting $2i$-bit code words each have exactly i 1s, therefore producing an i-of-$2i$ code. For example, the 3-of-6 code words constructed from 3 bits of information are shown in Table 3.5. The obvious disadvantage of using the i-of-$2i$ code is that twice as many bits are required to represent the in-

TABLE 3.4 Minimum number of parity bits for overlapping parity code

Number of information bits	Number of parity bits	Redundancy percentage
2	3	150.0%
4	3	75.0
6	4	66.7
8	4	50.0
10	4	40.0
12	5	41.7
16	5	31.25
24	5	20.8
32	6	18.75
64	7	10.9

TABLE 3.5 3-of-6 code for representing three bits of information

Original information	3-of-6 code	
000	000	111
001	001	110
010	010	101
011	011	100
100	100	011
101	101	010
110	110	001
111	111	000
	Original information	Appended information

formation; consequently, the redundancy of the code is 100%. The advantage of creating an i-of-$2i$ code is that both the encoding and the decoding processes are simple because the code is separable. The encoding procedure can be performed by examining the pattern of 1s in the information to be encoded and looking up in a table the desired bits to append, based on the pattern of 1s in the original information. The decoding can be performed by simply removing the appended bits from the code word and retaining the original information.

It is easy to see that m-of-n codes have a distance of two. Any single-bit error in an m-of-n code word changes the number of 1s to either $m + 1$ or $m - 1$, depending on whether the error changed a 0 to a 1 or a 1 to a 0. A second bit-error, however, can change the number of 1s back to m. For example, if one bit is changed from 0 to 1 and a second bit is changed from 1 to 0, the number of 1s in the code word remains unchanged. Consequently, the error goes undetected. If the errors are all unidirectional, meaning that all errors are either a change of a 1 to a 0 or a change of a 0 to a 1, but not combinations of the two changes, the m-of-n code detects the multiple errors. Consequently, the m-of-n code provides detection of all single errors and all multiple, unidirectional errors.

The m-of-n codes can often be constructed more efficiently, but the separable nature of the code is usually lost. For example, the 2-of-5 code for BCD data is shown in Table 3.6. The code words each have 5 bits, and the code is capable of detecting any single-bit error that occurs. The 2-of-5 code of Table 3.6 possesses the same error detection capability as the simple parity code and uses the same number of extra bits. The difficulty is that the information is not readily available from the code word. The code words must be converted back to the original information words by performing a

TABLE 3.6 Nonseparable 2-of-5 code for BCD data

Decimal digit	BCD data	2-of-5 code
0	0000	00011
1	0001	11000
2	0010	10100
3	0011	01100
4	0100	10010
5	0101	01010
6	0110	00110
7	0111	10001
8	1000	01001
9	1001	00101

decoding process. The decoding can be easily performed by look-up tables, but the process is much more complicated than needed for the separable *m*-of-*n* codes.

3.5.3 Duplication Codes

Duplication codes are based on the concept of completely duplicating the original information to form the code word. The primary advantage of the duplication code is simplicity. The major disadvantage is clearly the number of extra bits that must be provided to allow the code to be constructed.

Duplication codes are found in many applications, including memory systems and some communication systems. The encoding process for the duplication code consists of simply appending the original i bits of information to itself to form a code word of length $2i$ bits. If a single-bit error occurs, the two halves of the code word disagree, and the error can be detected. In communication systems, the duplication concept is often applied by transmitting all information twice; if both copies agree, the information is assumed to be correct. The penalty paid in the communications application is a decrease in the information rate of the system because $2i$ bits must be transmitted to obtain i bits of information.

A variation on the basic duplication code is to complement the duplicated portion of the code word. The use of **complemented duplication** is particularly advantageous when the original information and its duplicate must be processed by the same hardware. For example, consider the memory system shown in Fig. 3.32. Each word of the memory is i bits. As part of the encoding process, each i-bit word of information is followed sequentially in memory by its complemented duplicate. Suppose that one bit slice of the memory becomes faulty such that every bit stored in that slice assumes a value of 1, regardless of the desired value. Each word of informa-

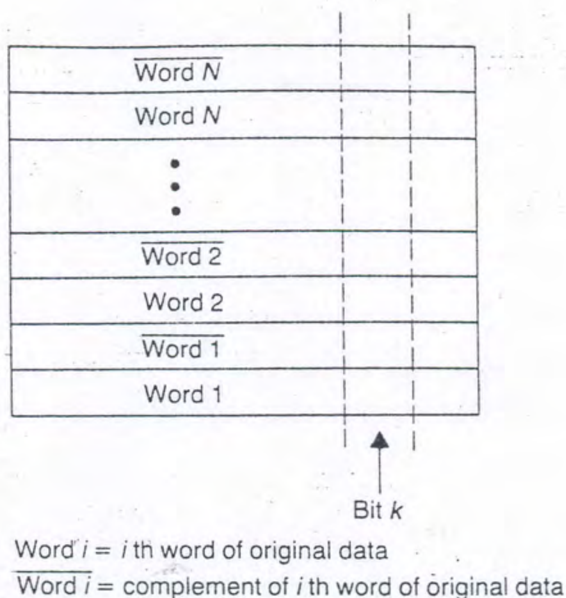


Fig. 3.32 Example of complemented duplication in a memory containing N words of data.

tion and its complement now has one bit position in which the bits are not complements but are exactly the same. Because the complemented duplication is used, the error can be detected. If the simple duplication method had been employed, the error would not have been detected.

The duplicated complement approach is also effective in communication systems where the same physical media is used by both the original information and its duplicate. As illustrated in Fig. 3.33, a faulty line in the

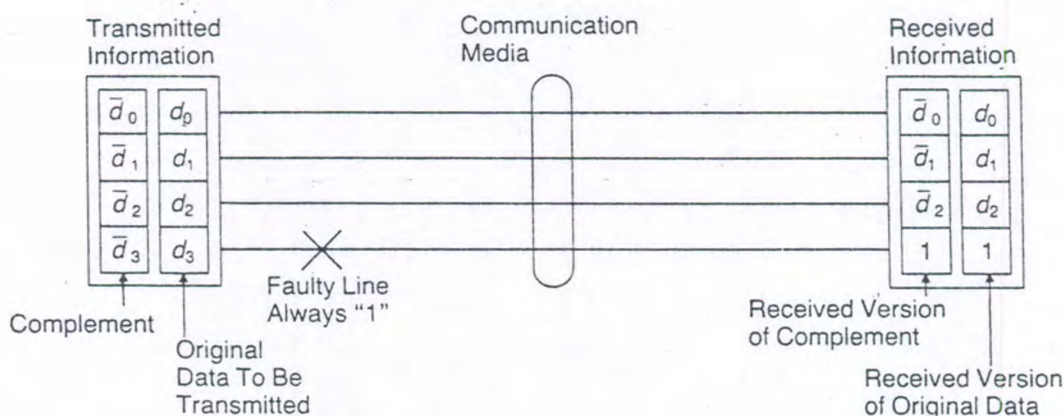


Fig. 3.33 Example of complemented duplication for error detection in a communication system.

communication media causes one bit to be the same in both the original information and the duplicate. If the original information and its duplicate are supposed to be complements, the error is detected; otherwise, the error goes undetected.

A final variation on the duplication code is the **swap and compare** method. The basic concept of swap and compare is to maintain two copies of the original information, but to swap the upper and lower halves of the second copy. Figure 3.34 illustrates the idea when applied to a memory system. A single bit slice that is faulty affects the upper half of one copy of the information and the lower half of the other copy. By comparing the appropriate halves, the error can be detected.

The primary advantage of all variations of the duplication codes is the simplicity associated with generating the code words and the ease of obtaining the original information from the code word. The advantage is usually offset, however, by the cost of completely duplicating the original information. Also, it is usually very time consuming to implement the duplication codes. In memory applications, each word must be written and read twice. In communication applications, each word must be transmitted twice. In both cases, the time required to perform the operation is doubled. So, the duplication code often requires not only the 100% redundancy in information, but typically a 100% redundancy in hardware and time as well.

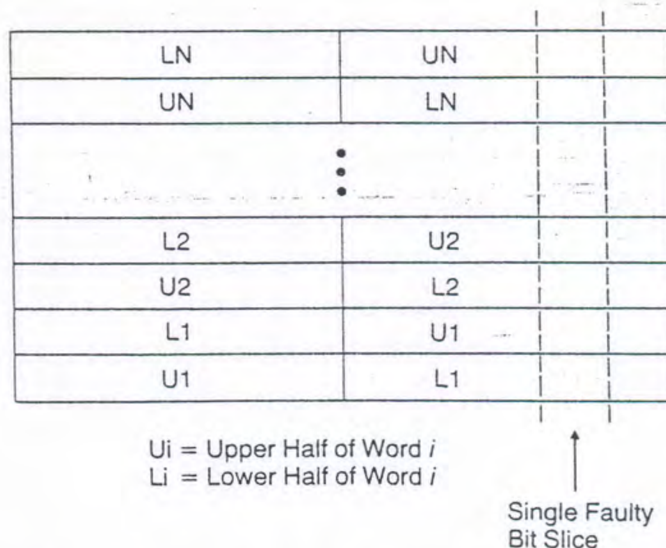


Fig. 3.34 Swap and compare technique applied to a memory system containing N words of data.

3.5.4 Checksums

The checksum is another form of separable code that is most applicable when blocks of data are to be transferred from one point to another. For example, checksums are used frequently in data transfers between mass storage devices—such as disks—and a computer, and packet-switched networks. The **checksum** is a quantity of information that is added to the block of data to help detect errors. Four primary types of checksums can be used: single-precision, double-precision, Honeywell, and the residue checksum.

The basic concept of the checksum is illustrated in Fig. 3.35. When the original data is created, an additional piece of information, called the checksum, is appended to the block of data. The checksum is then regenerated when the data is received at the destination or, in some applications, when the data is read from memory. The regenerated checksum and the original checksum are compared to determine if an error has occurred in the data, the checksum generation, checksum regeneration, or the checksum comparison.

The checksum is basically the sum of the original data. The difference between the various forms of the checksum is the way in which the summa-

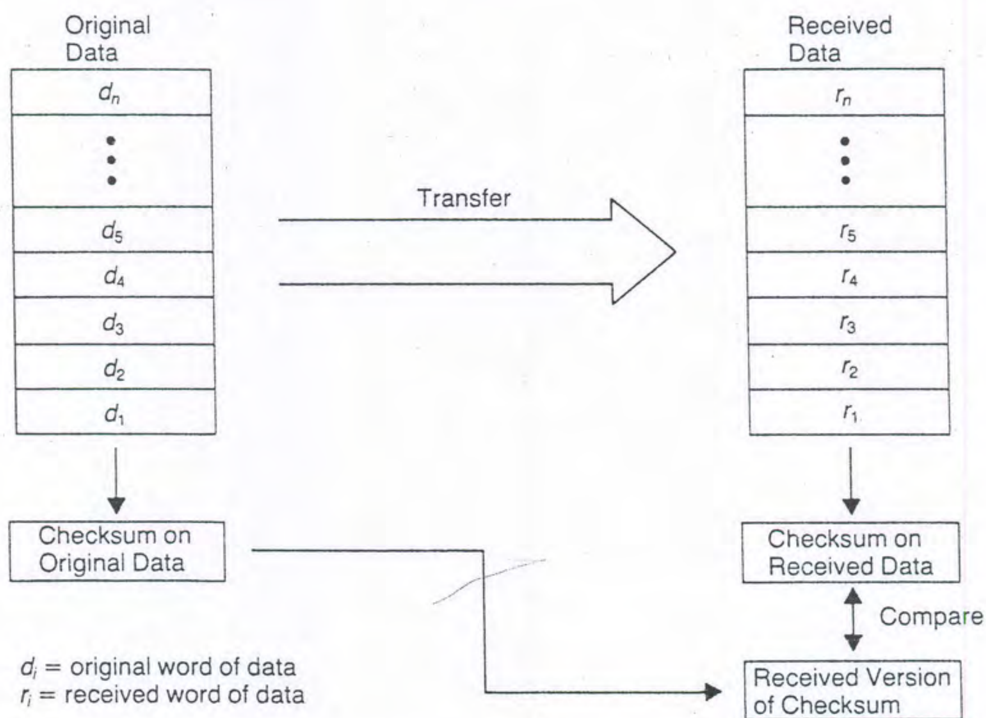


Fig. 3.35 In checksum coding, the sum of the original data words is appended to the block of data.

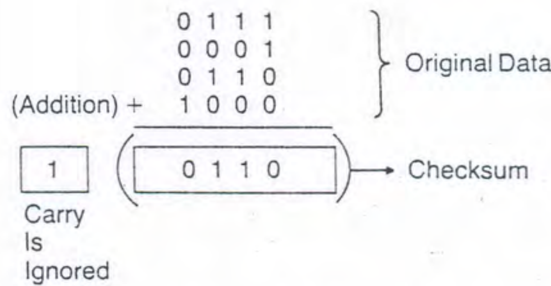


Fig. 3.36 A single-precision checksum is formed by adding the data words and ignoring any overflow.

tion is generated. The simplest form of the checksum is the **single-precision checksum**. The single-precision checksum is formed by performing the binary addition of the data that is to be protected by the checksum and ignoring any overflow that occurs. For example, if each data word is n bits, the checksum is n bits as well. If the true binary sum of the data exceeds $2^n - 1$, an overflow has occurred; in the single-precision checksum the overflow is ignored. In other words, the single-precision checksum is formed by adding the n-bit data in a modulo-2^n fashion. Figure 3.36 shows an example of the single-precision checksum for a block of four words, each of which is four bits.

The primary difficulty with the single-precision checksum is that information, and, as a result, the ability to detect errors are lost in the ignored overflow. As an example, consider the use of the single-precision checksum in the communications system shown in Fig. 3.37. The single-precision

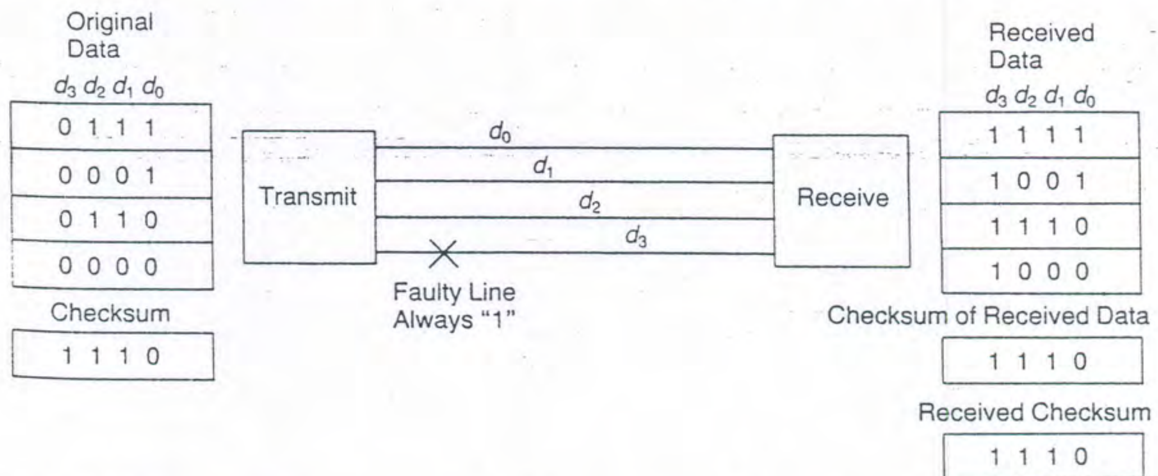


Fig. 3.37 The single-precision checksum is unable to detect certain types of errors. The received checksum and the checksum of the received data are equal, so no error is detected.

checksum is formed on the data at the transmitting point and is sent to the receiver along with the four words of data. If the most significant data line becomes logically stuck at a value of 1, the most significant bit of both the data and the checksum also become stuck at 1. When the checksum is generated on the received data, the regenerated and the received checksums agree because the overflow was ignored. But, the received information is certainly not correct.

One technique that is often used to overcome the limitations of the single-precision checksum is to compute the checksum in double precision, thus the name **double-precision checksum.** The basic concept of the double-precision checksum is to compute a $2n$-bit checksum for a block of n-bit words using modulo-2^{2n} arithmetic. Overflow is still a concern, however, but it is now overflow from a $2n$-bit sum as opposed to an n-bit sum. As an example, consider the problem illustrated in Fig. 3.37. We saw that the single-precision checksum failed to detect the error that occurred in the example. The double-precision checksum, as shown in Fig. 3.38, detects the stuck-at-1 fault. The checksum that is generated at the transmission point is sent as two 4-bit quantities. At the receiving point, each word has its most significant bit stuck at the logic 1 value. Also, the checksum has the most significant bit of each of its 4-bit halves stuck at 1. The regenerated checksum now disagrees with the received checksum, and the error is detected.

A third form of the checksum is called the **Honeywell checksum.** The basic idea of the Honeywell checksum is to concatenate consecutive words to form a collection of double-length words. For example, if there are k n-bit words, a set of $k/2$ $2n$-bit words is formed. The double-length words are

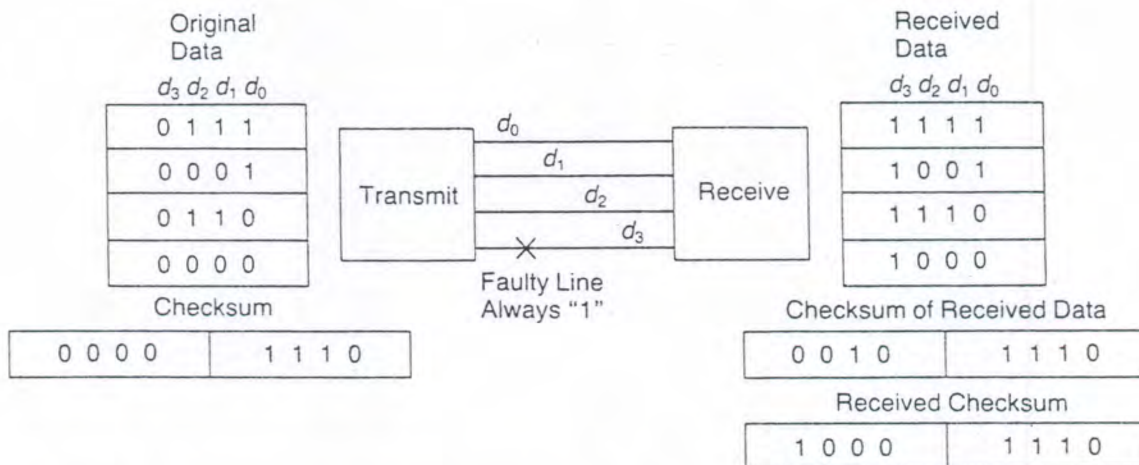


Fig. 3.38 A double-precision checksum is formed by adding the data using double-precision arithmetic. The received checksum and the checksum of the received data are not equal, so the error is detected.

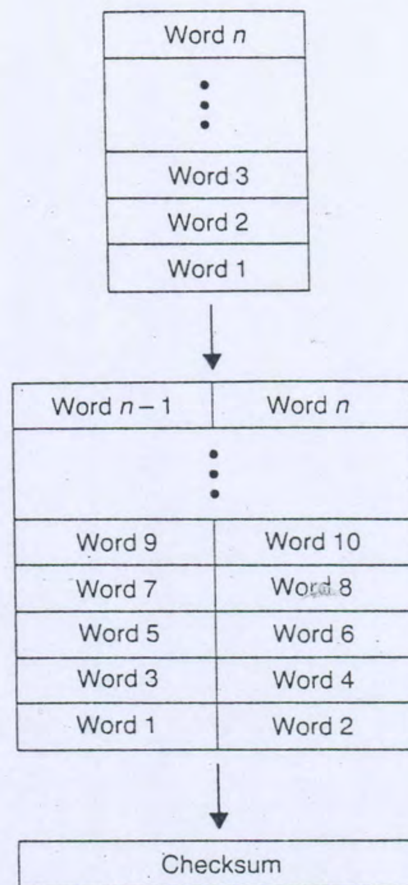


Fig. 3.39 In the Honeywell checksum, adjacent data words are concatenated prior to forming the checksum.

structured as shown in Fig. 3.39, and a checksum is formed over the newly structured data. The primary advantage of the Honeywell checksum is that a bit error that appears in the same bit position of all words will affect at least two bit positions of the checksum. For example, if a complete column of the original data is erroneous, the modified data structure has two erroneous columns.

An example of the Honeywell checksum is shown in Fig. 3.40. At the transmitting point, the new data structure is formed and the checksum is computed. If line d_3 is faulty, the regenerated checksum at the receiving point will differ from the checksum that is transmitted.

The final form of the checksum is the **residue checksum**. The concept of the residue checksum is the same as the single-precision checksum except that the carry bit out of the most significant bit position is not ignored but is added back to the checksum in an end-around carry fashion, as illus-

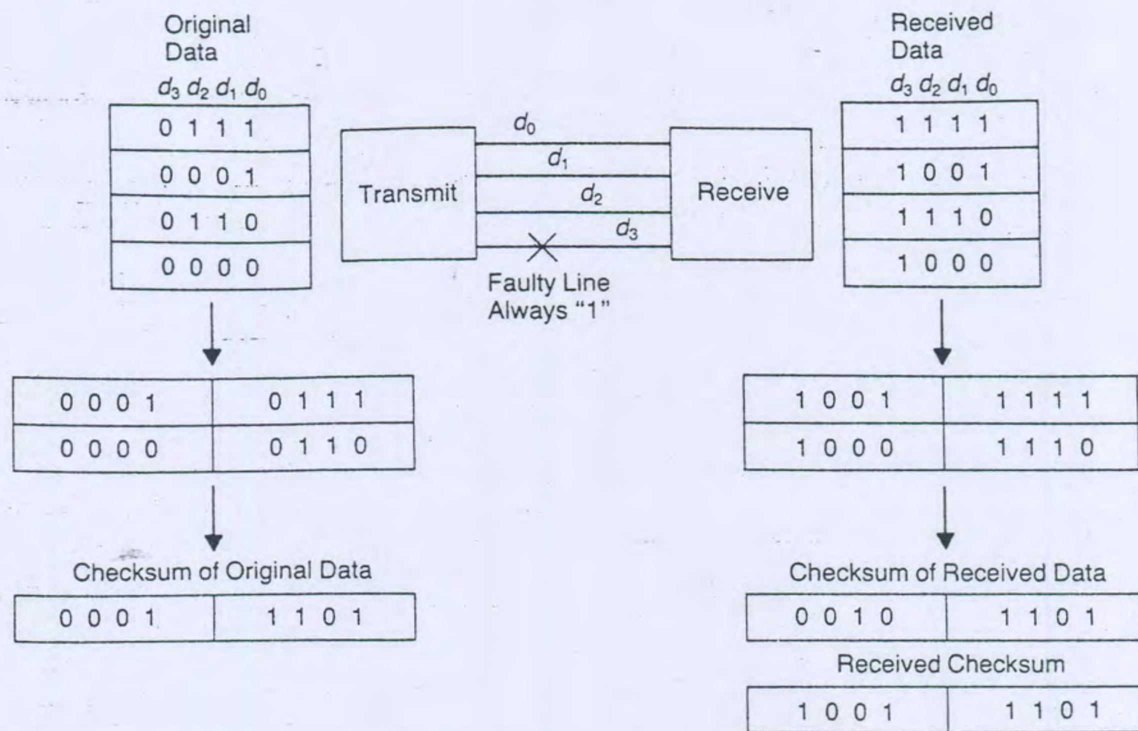


Fig. 3.40 Illustration of the error detection capability of the Honeywell checksum. The received checksum and the checksum of the received data are not equal, so the error is detected.

trated in Fig. 3.41. Using the same example as before (where the single-precision checksum was unable to detect the error), Fig. 3.42 shows that the residue checksum does indeed allow the error to be detected. The checksum regenerated at the receiving point differs from the checksum generated at the transmission point and transmitted to the receiver.

Note that checksums can detect errors but not locate them. If the checksum generated at the receiving point differs from the checksum generated at the transmission point, an error is indicated, but there is not enough information available to determine where the error has occurred. The complete block of data over which the checksum was formed must be corrected.

3.5.5 Cyclic Codes

The fundamental feature of **cyclic codes** is that any end-around shift of a code word will produce another code word [Lin 1983]. In other words, the cyclic code is invariant to the end-around shift operation. Cyclic codes are frequently applied to sequential-access devices such as tapes, bubble memories, and disks. In addition, cyclic codes are extremely popular for use in data links. One reason that cyclic codes are attractive is because the en-

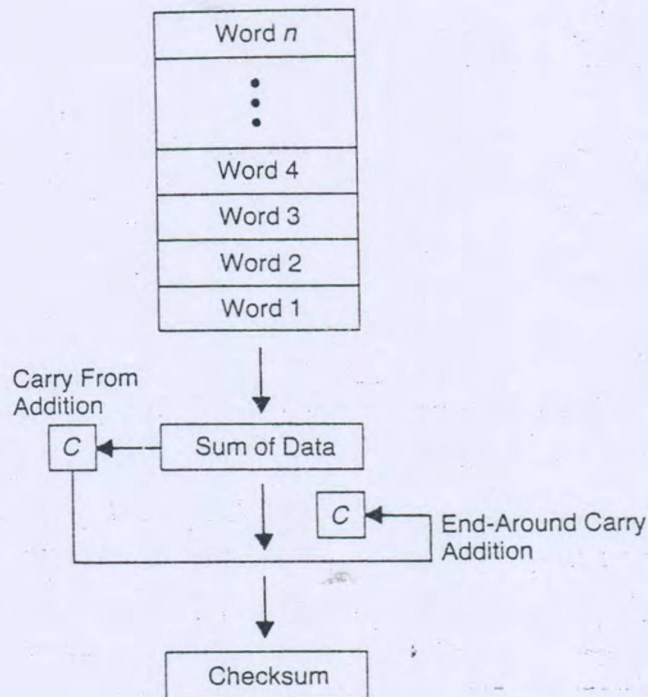


Fig. 3.41 The residue checksum is formed using end-around carry addition so that information in the carry bit is not lost.

coding operation can be implemented using simple shift registers with feedback connections.

A cyclic code is characterized by its generator polynomial $G(X)$, which is a polynomial of degree $n - k$ or greater, where n is the number of bits contained in the complete code word produced by $G(X)$, and k is the number of bits in the original information to be encoded. For binary cyclic codes, the coefficients of the generator polynomial are all either 0 or 1. The integers n and k specify the characteristics of the cyclic code. A cyclic code with a generator polynomial of degree $(n - k)$ is called an (n, k) cyclic code. Such codes possess the property of being able to detect all single errors and all multiple, adjacent errors affecting fewer than $(n - k)$ bits [Lin and Costello 1983]. The error detection property of cyclic codes is particularly important in communications applications where burst errors can occur. A **burst error** is the result of a transient fault and usually introduces a number of adjacent errors into a given data item. For example, a word that is transmitted serially can have several adjacent bits corrupted by a single disturbance; one would hope that the coding scheme could detect such errors. (n, k) cyclic codes can detect adjacent errors as long as the number of adjacent bits affected does not exceed $(n - k)$.

Cyclic codes depend on the representation of data by polynomials. We are very much accustomed to these types of representations because of the

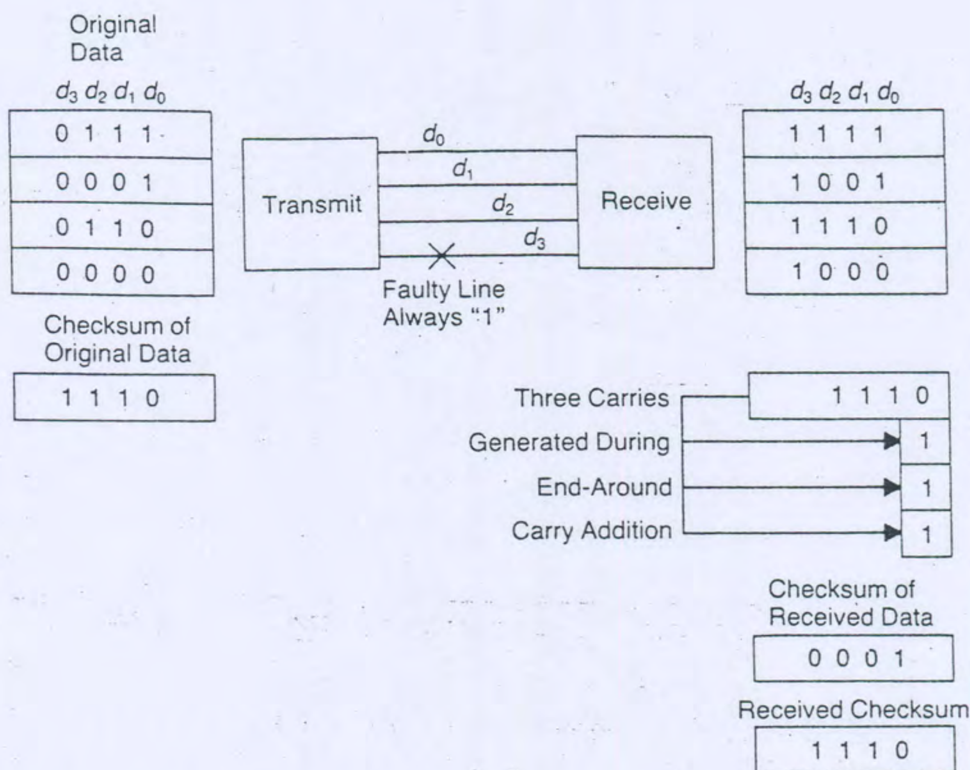


Fig. 3.42 Illustration of the error detection capability of the residue checksum. The checksum of the received data and the received checksum are not equal, so the error is detected.

number systems in which we work. For example, the decimal equivalent of the binary number 1011 can be represented as the polynomial $1 * r^3 + 0 * r^2 + 1 * r^1 + 1 * r^0$, where r is the base of the number system, in this case, $r = 2$, and $*$ is decimal multiplication. The coefficients of the polynomial represent the digits of the number.

The properties of cyclic codes are generated by representing the code words as coefficients of a polynomial. For example, suppose we have the code word $v = (v_0, v_1, \ldots, v_{n-1})$. This code word corresponds to the polynomial $V(X)$, where

$$V(X) = v_0 + v_1 X + v_2 X^2 + \cdots + v_{n-1} X^{n-1}$$

Each n-bit code word is represented by a polynomial of degree $(n - 1)$ or less. If the coefficient v_{n-1} is 0, the degree of the polynomial will be less than $n - 1$; but if the coefficient v_{n-1} is 1, the polynomial will have a degree of $(n - 1)$. The polynomial $V(X)$ is called the **code polynomial** of the code word v.

The code polynomials for a nonseparable cyclic code are generated by multiplying a polynomial, representing the data to be encoded, by another

polynomial known as the **generator polynomial.** The generator polynomial determines the characteristics of the cyclic code. Any additions required during the multiplication of the two polynomials are performed using modulo-2 addition. For example, suppose that we have a generator polynomial, $G(X) = 1 + X + X^3$ and we wish to encode the binary data (1101). The data (1101) can be represented by the data polynomial $D(X) = 1 + X + X^3$. The code polynomial is generated by multiplying the data polynomial and the generator polynomial. Specifically, the code polynomial is generated as $V(X) = D(X) * G(X) = (1 + X + X^3)(1 + X + X^3) = 1 + X^2 + X^6$. In more exact terms, the code polynomial is given by $V(X) = 1 + 0*X + 1*X^2 + 0*X^3 + 0*X^4 + 0*X^5 + 1*X^6$ and the code word v consists of the coefficients of that code polynomial. In other words, $v = (1010001)$.

Suppose we look at another example to clarify the basic concepts. Once again, consider the generator polynomial to be $G(X) = 1 + X + X^3$. Now, suppose that we wish to encode the data (1111), which is represented by the data polynomial $D(X) = 1 + X + X^2 + X^3$. To determine the code polynomial $V(X)$ we multiply the data polynomial $D(X)$ by the generator polynomial $G(X)$. The resulting code polynomial is $1 + 0*X + 0*X^2 + 1*X^3 + 0*X^4 + 1*X^5 + 1*X^6$. Note that the coefficients are added in a modulo-2 fashion, which allows for some very simple implementations of the cyclic codes. The resulting code word in this example is $v = (1001011)$.

The nonseparable cyclic codes generated for four bits of information data using the generator polynomial, $G(X)= 1 + X + X^3$, are shown in Table 3.7. Note that the cyclic code shown in Table 3.7 is a code of distance 3; thus, any 2-bit errors can be detected. The distance can be easily determined by comparing all code words and seeing that all possible pairs of code words differ in at least three bit positions. The "cost" of the error detection capability is reflected in the three extra bits required to represent the information.

Perhaps the most interesting aspect of the nonseparable cyclic codes is the manner in which they can be generated. Recall that the code polynomial is generated by multiplying the data polynomial by the generator polynomial and adding the coefficients in a modulo-2 fashion. If we consider the blocks labeled X as multipliers by the factor X, and the addition elements as modulo-2 adders, the circuit shown in Fig. 3.43 performs the multiplication of two polynomials. For example, if $D(X) = 1$, the output $V(X)$ of the circuit will be $1 + X^2 + X^3$. Likewise, if $D(X) = 1 + X + X^3$, the output will be given by

$$V(X) = 1 + X + X^3 + [X(1 + X + X^3) + (1 + X + X^3)]X^2$$

or

$$V(X) = 1 + X + X^2 + X^3 + X^4 + X^5 + X^6$$

TABLE 3.7 Cyclic code words for 4-bit information words.

Information (d_0, d_1, d_2, d_3)	Code $(v_0, v_1, v_2, v_3, v_4, v_5, v_6)$
0000	0000000
0001	0001101*
0010	0011010
0011	0010111
0100	0110100
0101	0111001
0110	0101110
0111	0100011
1000	1101000
1001	1100101
1010	1110010
1011	1111111
1100	1011100
1101	1010001*
1110	1000110
1111	1001011

Data polynomial = $d_0 + d_1x + d_2x^2 + d_3x^3$
Generator polynomial = $1 + x + x^3$
Code polynomial = $v_0 + v_1x + v_2x^2 + v_3x^3 + v_4x^4 + v_5x^5 + v_6x^6$

Therefore, if the generator polynomial is $G(X) = 1 + X^2+X^3$, the circuit shown in Fig. 3.43 will generate the code polynomial by multiplying the data polynomial by the generator polynomial. For example, if the data to be

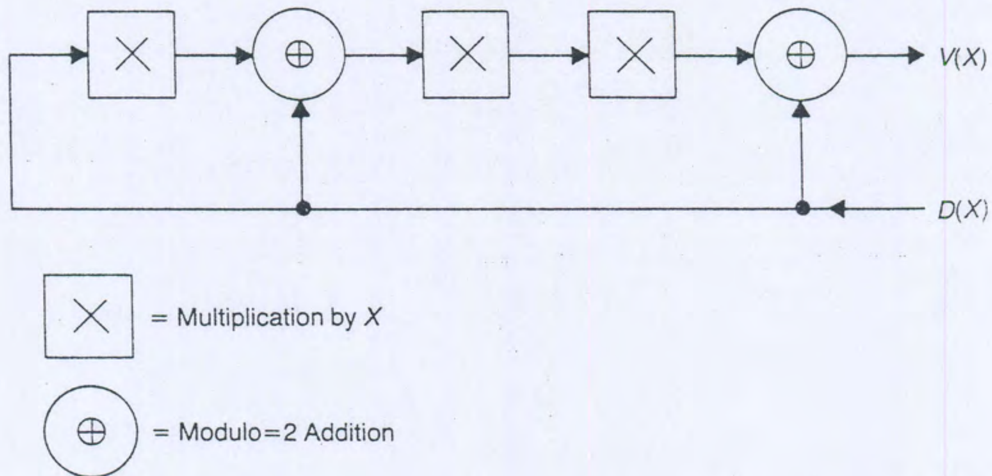


Fig. 3.43 Example circuit for generating a cyclic code word by multiplying an incoming data polynomial $D(X)$ by the generator polynomial.

encoded is (1101), the data polynomial will be $D(X) = 1 + X + X^3$. The resulting code polynomial yields the code word (1111111).

The generation circuits for cyclic codes can be implemented in digital hardware using storage elements and EXCLUSIVE-OR gates. The function of the storage elements is to implement a time delay. The circuit shown in Fig. 3.44 creates the code associated with the generator polynomial $G(X) = 1 + X + X^3$. Initially, the storage elements, or registers, are loaded with all 0s. The data to be encoded appears on the line $D(X)$ serially. The bits of the code word will appear on the line $V(X)$ serially. The occurrence of a clock pulse causes the registers to be loaded with the values on their inputs. Table 3.8 illustrates the operation of the circuit during the encoding of the data (1101).

Having examined the process of generating cyclic codes, we now consider the decoding procedure. The version of the cyclic code that has been presented thus far is not a separable code, so the decoding process involves more than simply picking certain bits from the code word. The structure of the cyclic code, however, makes the decoding process relatively easy.

Suppose that we wish to determine if the code word $(r_0, r_1, r_2, \ldots, r_{n-1})$ is valid. We know that this code word can be represented by the code polynomial $R(X) = r_0 + r_1X + r_2X^2 + \cdots + r_{n-1}X^{n-1}$. We also know that the correct code polynomial was generated by multiplying the original data polynomial by the generator polynomial. In other words, if $R(X)$ is a valid code polynomial, it was generated as $R(X) = D(X)G(X)$, where $G(X)$ is the generator polynomial and $D(X)$ is the original data polynomial. If we write

$$R(X) = D(X)G(X) + S(X)$$

then the quantity $S(X)$ should be zero if the polynomial $R(X)$ is a valid code polynomial. In other words $R(X)$ should be an exact multiple of the generator polynomial. One way to determine if $R(X)$ is indeed an exact multiple of the generator polynomial is to divide the polynomial $R(X)$ by the generator

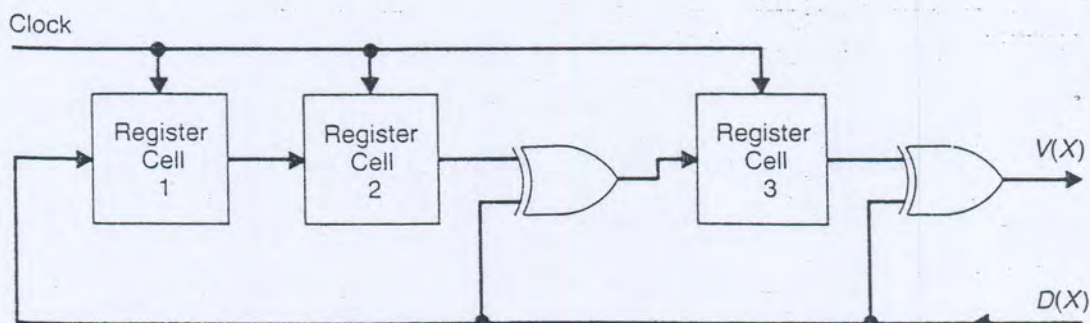


Fig. 3.44 Circuit for generating cyclic code words for the generator polynomial $G(X) = 1 + X + X^3$.

TABLE 3.8 The encoding process for the circuit of Fig. 3.44

	Register values				
Clock period	**1**	**2**	**3**	***D*(*x*)**	***V*(*x*)**
0	0	0	0		
				1	1
1	1	0	1		
				1	0
2	1	1	1		
				0	1
3	0	1	1		
				1	0
4	1	0	0		
				0	0
5	0	1	0		
				0	0
6	0	0	1		
				0	1
7	0	0	0		

polynomial $G(X)$ and see if the remainder of the division is zero. If the remainder is zero, the polynomial is an exact multiple of the generator polynomial and is a valid code polynomial. The quantity $S(X)$ is called the **syndrome polynomial.**

The process of division may at first seem complicated and difficult to implement. It turns out to be quite simple, however, when feedback circuits similar to the cyclic code generators are used. The circuit shown in Fig. 3.45, for example, is capable of dividing a polynomial by the polynomial $1 + X + X^3$. Once again, the blocks labeled as X perform multiplication by the factor X. The adders in the circuit of Fig. 3.45 are modulo-2 adders. The polynomial that appears on line $B(X)$ of the circuit is given by

$$B(X) = (X^3 + X)D(X)$$

But the values present on line $D(X)$ are determined by both line $B(X)$ and line $V(X)$. Specifically,

$$V(X) + B(X) = D(X)$$

$$V(X) = D(X) - B(X) = D(X) - (X^3 + X)D(X)$$

Because the functions of addition and subtraction are the same in the modulo-2 system, we obtain

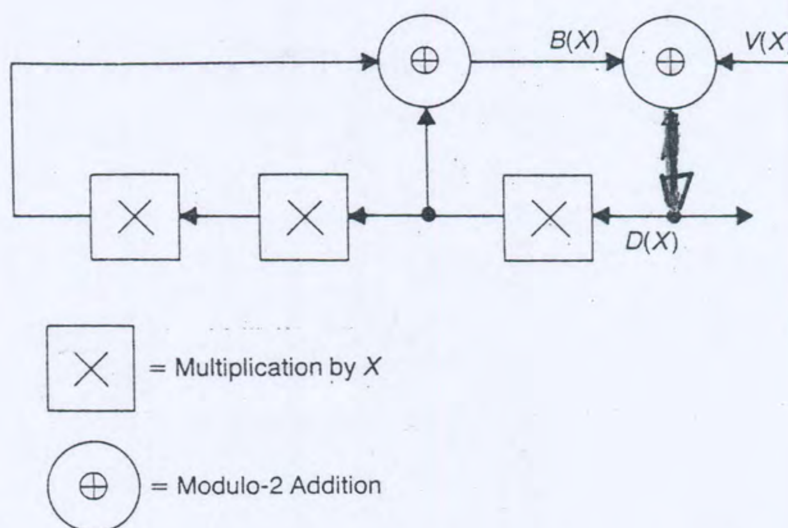


Fig. 3.45 A division circuit for use in decoding cyclic code words.

$$V(X) = (X^3 + X + 1)D(X)$$

$$D(X) = \frac{V(X)}{(X^3 + X + 1)}$$

The circuit of Fig. 3.45 divides the polynomial $V(X)$ by the polynomial $(X^3 + X + 1)$.

If the multiplicative elements of the circuit of Fig. 3.45 are replaced with storage elements and the modulo-2 adders are constructed using EXCLUSIVE-OR gates, the circuit of Fig. 3.46 results. If the storage elements, or registers, are initialized to 0, the polynomial represented by the data stream appearing on line $V(X)$ is divided by $(X^3 + X + 1)$, and the result appears as a data stream on line $D(X)$. Once the division has been completed, the registers contain the value of the syndrome, or remainder, of the division process. If the syndrome is zero, a valid code word has been received. Otherwise, the word received was an invalid code word. Table 3.9 illustrates the decoding process for the code word (1010001).

As an example, consider the results generated by the circuit of Fig. 3.46 when the code word is erroneous. Suppose that the code word (1010001) was the intended code word, but an error resulted in the received code word being (1011001). In other words, a single error has resulted in the 0 in the center bit position becoming a 1. Table 3.10 details the functions of the circuit when the received data stream on line V is (1011001). As can be seen, the syndrome that results is nonzero, clearly indicating an invalid code word; the received code word is not evenly divisible by the generator polynomial.

TABLE 3.9 The decoding process for the circuit of Fig. 3.46

Clock period	Register values 1	2	3	$V(x)$	$B(x)$	$D(x)$
0	0	0	0			
				1	0	1
1	0	0	1			
				0	1	1
2	0	1	1			
				1	1	0
3	1	1	0			
				0	1	1
4	1	0	1			
				0	0	0
5	0	1	0			
				0	0	0
6	1	0	0			
				1	1	0
7	0	0	0	↑		↑
	Syndrome			Code word		Original information

The primary disadvantage of the cyclic codes discussed thus far is that they are not separable. It is possible, however, to generate a separable, cyclic code [Nelson and Carroll 1986]. To generate an (n, k) code, the original data polynomial $D(X)$ is first multiplied by X^{n-k}, and the result is divided by the generator polynomial $G(X)$ to obtain a remainder of $R(X)$. The code polynomial is then computed as $V(X) = R(X) + X^{n-k}D(X)$, and $V(X)$ is an exact multiple of the generator polynomial $G(X)$. Note that the multipli-

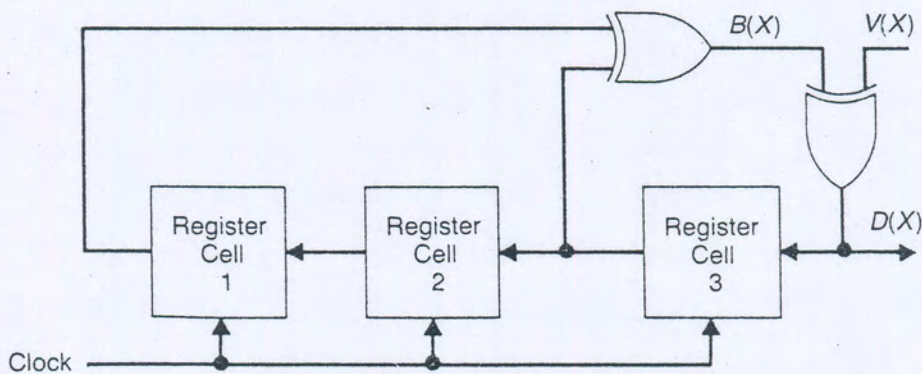


Fig. 3.46 Decoding circuit for the cyclic code with generator polynomial, $G(X) = 1 + X + X^3$.

TABLE 3.10 The decoding process with erroneous information

Clock period	Register values 1	2	3	$V(x)$	$B(x)$	$D(x)$
0	0	0	0			
				1	0	1
1	0	0	1			
				0	1	1
2	0	1	1			
				1	1	0
3	1	1	0			
				1	1	0
4	1	0	0			
				0	1	1
5	0	0	1			
				0	1	1
6	0	1	1			
				1	1	0
7	1	1	0	↑ Received word		
	Nonzero syndrome					

cation by X^{n-k} can be performed by simply shifting the coefficients of the data polynomial. Also note that the addition of the remainder polynomial is equivalent to simply appending the remainder to the polynomial $X^{n-k}D(X)$.

The validity of the encoding process just described is relatively obvious. Suppose, however, that we have an arbitrary data polynomial $D(X)$ and a generator polynomial $G(X)$. The code polynomial $V(X)$, is given by

$$V(X) = X^{n-k}D(X) + R(X)$$

where $R(X)$ is the remainder obtained when $X^{n-k}D(X)$ is divided by $G(X)$. In other words,

$$\frac{X^{n-k}D(X)}{G(X)} = Q(X) + \frac{R(X)}{G(X)}$$

where $Q(X)$ is the quotient computed in the division process.

Multiplying both sides of the previous equation by $G(X)$ yields

$$X^{n-k}D(X) = G(X)Q(X) + R(X)$$

$$X^{n-k}D(X) - R(X) = G(X)Q(X)$$

Recall, however, that all addition and subtraction operations are performed using modulo-2 arithmetic, so addition and subtraction are identical. Consequently,

$$X^{n-k}D(X) - R(X) = X^{n-k}D(X) + R(X) = G(X)Q(X) = V(X)$$

Therefore, the code polynomial $V(X)$, formed as

$$V(X) = X^{n-k}D(X) + R(X)$$

is an exact multiple of the generator polynomial $G(X)$.

As an example of the construction of the separable (7,4) cyclic code consider the data (1001) and the generator polynomial $G(X) = 1 + X + X^3$. The data polynomial corresponding to (1001) is $D(X) = 1 + X^3$. Therefore, $X^{n-k}D(X) = X^3D(X) = X^3+X^6$. Dividing $X^3 + X^6$ by $G(X)$ yields a remainder of $R(X) = X^2 + X$. Adding the remainder polynomial to $X^3 + X^6$ results in $V(X) = X + X^2 + X^3 + X^6$. So, the code word is given by (011|1001). Note that the last four bits of the code word are the coefficients of the original data polynomial, and the first three bits are the coefficients of the remainder polynomial.

3.5.6 Arithmetic Codes

Arithmetic codes are very useful when it is desired to check arithmetic operations such as addition, multiplication, and division [Avizienis 1971]. The basic concept is the same as all coding techniques. The data presented to the arithmetic operation is encoded before the operations are performed. After completing the arithmetic operations, the resulting code words are checked to make sure that they are valid. If they are not, an error condition exists.

An arithmetic code must be invariant to a set of arithmetic operations. An arithmetic code A has the property that $A(b * c) = A(b) * A(c)$, where b and c are operands, $*$ is some arithmetic operation, and $A(b)$ and $A(c)$ are the arithmetic code words for the operands b and c, respectively. Stated in words, the performance of the arithmetic operation on two arithmetic code words will produce the arithmetic code word of the result of the arithmetic operation. To completely define an arithmetic code, the method of encoding and the arithmetic operations for which the code is invariant must be specified. Examples of arithmetic codes are the *AN* codes, residue codes, inverse-residue codes, and the residue number system.

AN Codes

The simplest arithmetic code is the *AN* code which is formed by multiplying each data word N by some constant A. The *AN* codes are invariant to addition and subtraction but not multiplication and division. If N_1 and N_2 are

two operands to be encoded, the resulting code words are AN_1 and AN_2, respectively. If the two code words are added, the sum is $A(N_1 + N_2)$, which is the code word of the correct sum. The operations performed under an AN code can be checked by determining if the results are evenly divisible by the constant A. If they are not, an error condition exists.

The magnitude of the constant A determines both the number of extra bits required to represent the code words and the error detection capability provided. The selection of the constant A is critical to the effectiveness and efficiency of the resulting code. First, for binary codes, the constant must not be a power of two. To see the reason for this limitation, suppose that we encode the binary number $(a_{n-1}a_{n-2} \cdots a_2a_1a_0)$ by multiplying by the constant $A = 2^a$. Multiplication by 2^a is equivalent to a left arithmetic shift of the original binary word, so the resulting code word is $(a_{n-1}a_{n-2} \cdots a_2a_1a_0 00 \cdots 0)$, where a 0s have been appended to the original binary number. The decimal representation of the code word is given by

$$a_{n-1}2^{a+n-1} + \cdots + a_2 2^{a+2} + a_1 2^{a+1} + a_0 2^a + 02^{a-1} + \cdots + 02^1 + 02^0$$

which is clearly, evenly divisible by 2^a. It is also easy to see, however, that changing just one coefficient still yields a result that is evenly divisible by 2^a. For example, if the coefficient of the 2^a term changes from 0 to 1, the result remains evenly divisible by 2^a. Thus, an AN code that has $A = 2^a$ is not capable of detecting single-bit errors.

An example of a valid AN code is the $3N$ code, where all words are encoded by multiplying them by 3. If the original data words are n bits long, the code words for the $3N$ code require $n + 2$ bits. Table 3.11 shows the $3N$ code for 4-bit information. An example of the effectiveness of the $3N$ code is shown in Fig. 3.47 where a binary adder is protected by the $3N$ code. Under fault-free circumstances, the result of adding the two code words (010010) and (000011) results in the valid code word (010101), which is evenly divisible by 3. If, however, line S_1 is stuck at the logic 1 value, the result of the addition of (010010) and (000011) is (010111), which is not evenly divisible by 3. Therefore, the error can be detected by checking each resulting code word to determine if it is evenly divisible by 3.

The encoding of operands in the $3N$ code can be performed by a simple addition if we recognize that we can multiply any number by 3 by adding the original number to a value that is twice that number. In other words, we form $3N$ by adding N and $2N$. The quantity $2N$ is easily created by shifting the binary number left by one place. The numbers N and $2N$ can then be added. Figure 3.48 illustrates the use of an $(n+1)$-bit adder to produce the $(n + 2)$-bit code word for an n-bit operand. An $(n + 1)$-bit adder can be used if the carry-bit out of the adder is used as the most significant bit of the resulting sum.

TABLE 3.11 Resulting 3N code words for 4-bit information words

Original information	3N code word
0000	000000
0001	000011
0010	000110
0011	001001
0100	001100
0101	001111
0110	010010
0111	010101
1000	011000
1001	011011
1010	011110
1011	100001
1100	100100
1101	100111
1110	101010
1111	101101

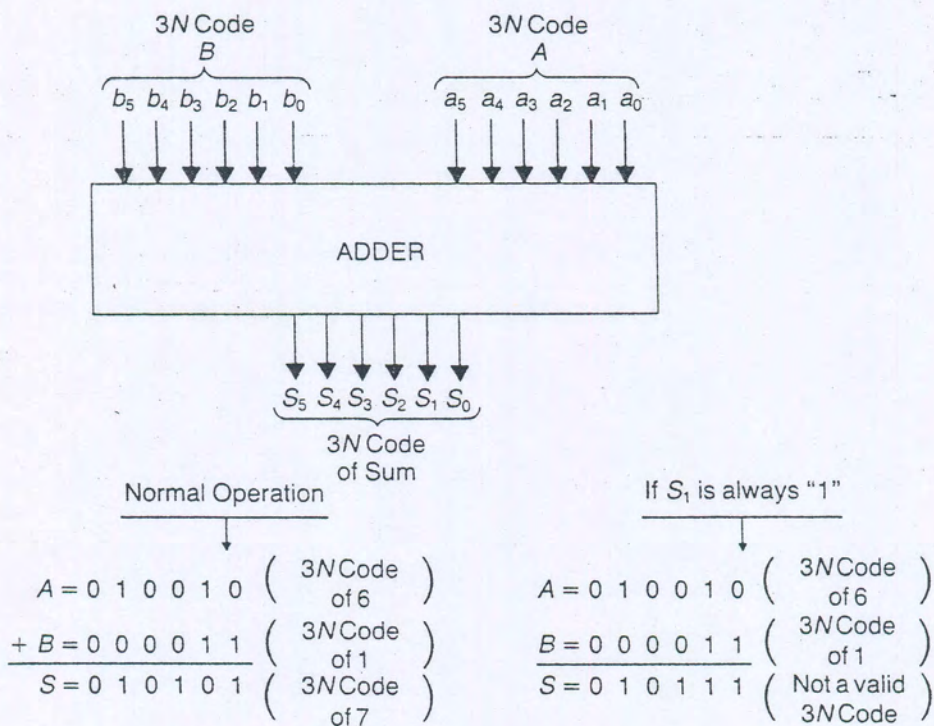


Fig. 3.47 Illustration of the error detection capabilities of the 3N arithmetic code. The presence of the fault results in the sum being an invalid 3N code.

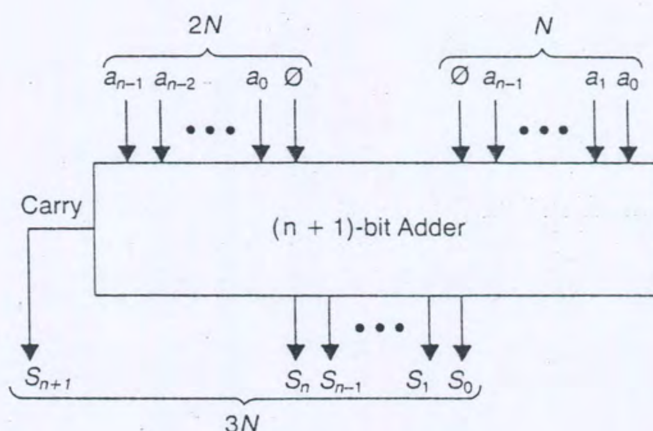


Fig. 3.48 Illustration of the use of an $(n + 1)$-bit adder to create $3N$ code words.

The $3N$ code can be checked by using a simple combinational circuit. For example, suppose that we wish to design an error checker for the $3N$ code of 2-bit data; in this case, the code words are 4 bits long. The Karnaugh map of the combinational circuit that performs the checking operation is shown in Fig. 3.49. The circuit produces a value of 1 when the inputs to the circuit represent a valid $3N$ code word; otherwise, the output of the circuit is 0. The combinational circuit can be implemented using a multiplexer, as shown in Fig. 3.49.

Residue Codes

The next class of arithmetic codes to be studied are the residue codes. A residue code is a separable arithmetic code created by appending the residue of a number to that number. In other words, the code word is constructed as $D|R$, where D is the original data and R is the residue of that data. The encoding operation consists of determining the residue and appending it to the original data. The decoding process involves simply removing the residue, thus leaving the original data word.

The residue of a number is simply the remainder generated when the number is divided by an integer. For example, suppose we have an integer N and we divide N by another integer m. N may be written as an integer multiple of m as

$$N = Im + r$$

$$\frac{N}{m} = I + \frac{r}{m}$$

where r is the remainder, sometimes called the residue, and I is the quotient. The quantity m is called the check base, or the modulus. For example,

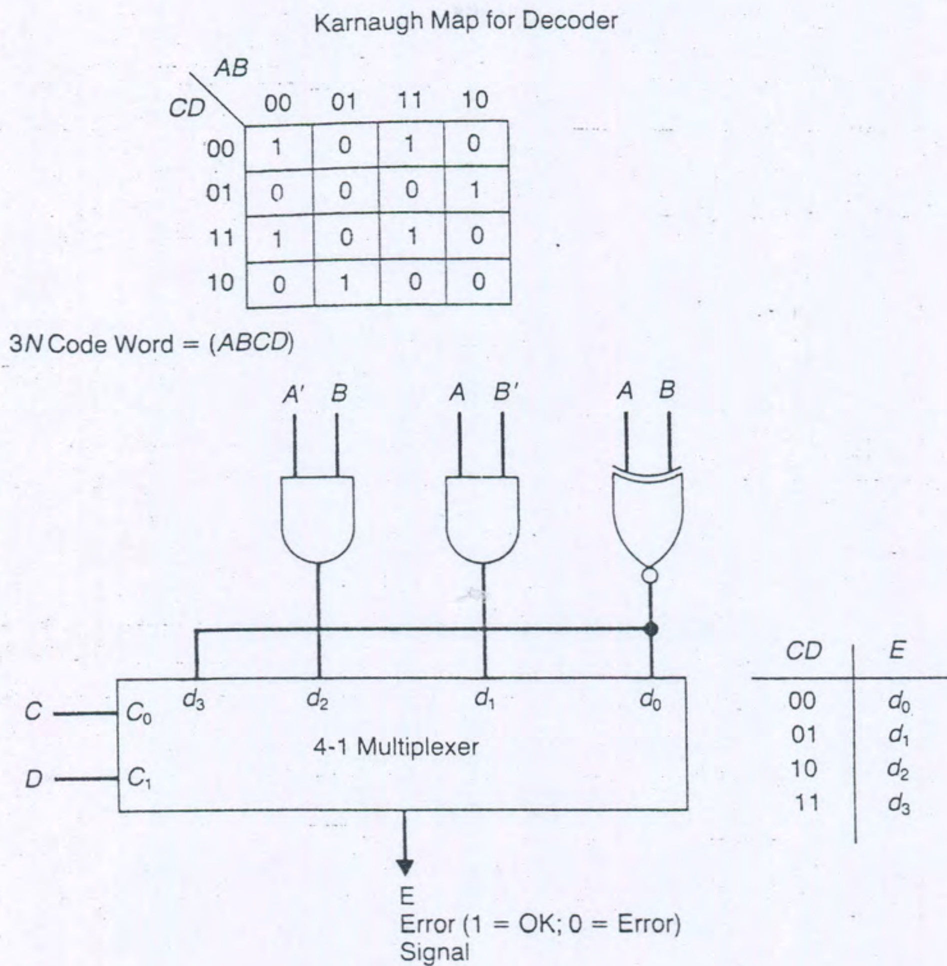


Fig. 3.49 A simple error detection circuit for the $3N$ code can be constructed using combinational logic.

if $N = 14$ and $m = 3$, the quotient I will be 4 and the residue will be 2. We often write this as

$$14 = 2 \text{ modulo}(3)$$

Separable residue codes, as mentioned previously, are formed by appending the residue of a data word to that data word. The number of extra bits required to represent the code word depends on the particular modulus selected. The residue will never be larger than the modulus; in fact, $0 \leq r < m$. For example, if the original data is n bits and the modulus is 3, the code word will require $n + 2$ bits. Table 3.12 illustrates the residue code that results when 4-bit data is encoded using a modulus of 3.

The primary advantages of the residue codes are that they are invariant to the operation of addition, and the residues can be handled separately

TABLE 3.12 Residue code words for 4-bit information words using a modulus of three

Information	Residue	Code word
0000	0	0000 00
0001	1	0001 01
0010	2	0010 10
0011	0	0011 00
0100	1	0100 01
0101	2	0101 10
0110	0	0110 00
0111	1	0111 01
1000	2	1000 10
1001	0	1001 00
1010	1	1010 01
1011	2	1011 10
1100	0	1100 00
1101	1	1101 01
1110	2	1110 10
1111	0	1111 00

from the data during the addition process. The structure of an adder that uses the separable residue code for error detection is shown in Fig. 3.50. The two data words D_1 and D_2 are added to form a sum word S. The residues r_1 and r_2 of D_1 and D_2, respectively, are also added using a modulo-m adder, where m is the modulus used to encode D_1 and D_2. If the operations are performed correctly, the modulo-m addition of r_1 and r_2 yields the residue r_s, of the sum S. A separate circuit is then used to calculate the residue of S. If the calculated residue r_c, differs from r_s, an error has occurred in one part of the process. For example, errors can be detected that occur in the generation of S, r_s, or r_c.

If the modulus for the residue code is selected in a special manner, a low-cost residue code results. Specifically, low-cost residue codes have a modulus of $m = 2^b - 1$, where b is some integer greater than or equal to 2. The number of extra bits required in a low-cost residue code is equal to b. For example, the residue code shown in Table 3.12 for a modulus of 3 is a low-cost residue code.

The main advantage of the low-cost residue code is the ease with which the encoding process can be performed. Recall that we must determine a remainder to encode information using a residue code; therefore, a division is necessary. The low-cost residue codes, however, allow the division to be recast as an addition process. The information bits to be encoded, are first di-

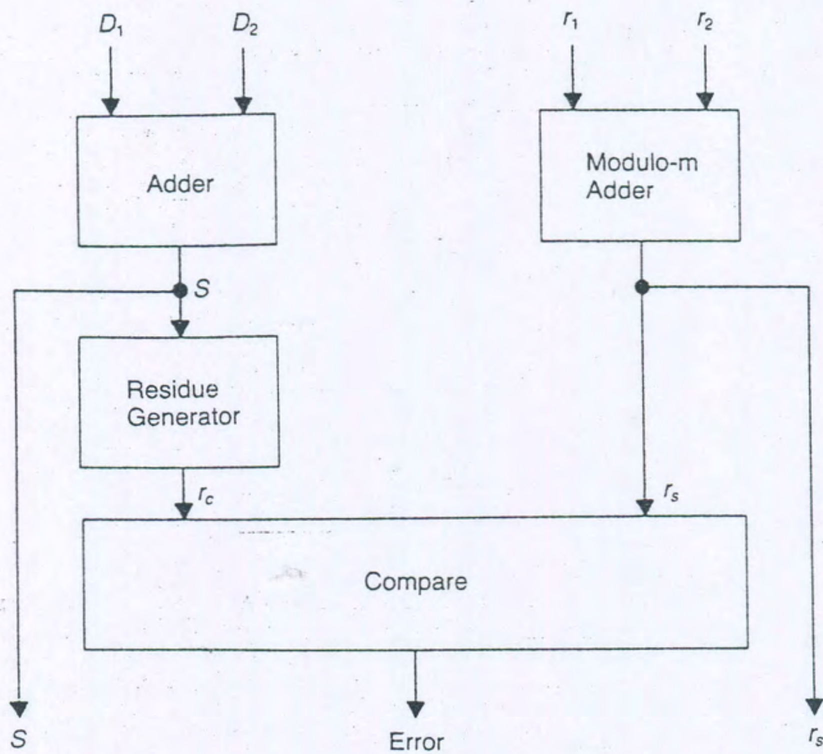


Fig. 3.50 The structure of an adder designed using the separable residue code.

vided into groups, each group containing b bits. The groups are then added in a modulo-$(2^b - 1)$ fashion to form the residue of the information bits. For example, Fig. 3.51 shows the procedure for determining the residue for the information bits (10100111) using a modulus of 3. Eight-bit information

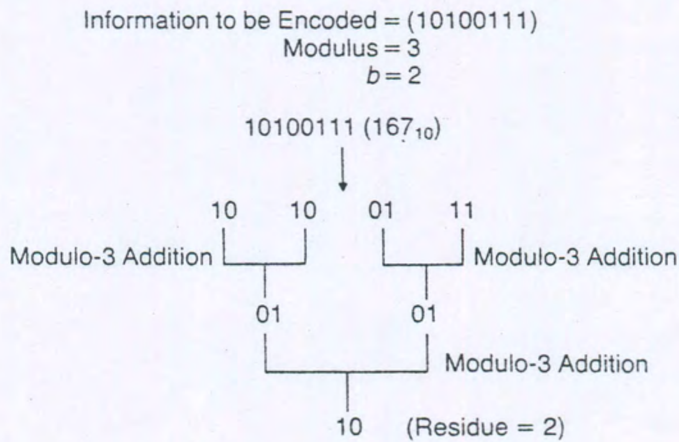


Fig. 3.51 The residue calculation for a low-cost residue code can be performed using successive additions.

words can be encoded using three, 2-bit, modulo-3 adders, as shown in Fig. 3.52, when the modulus is chosen as three.

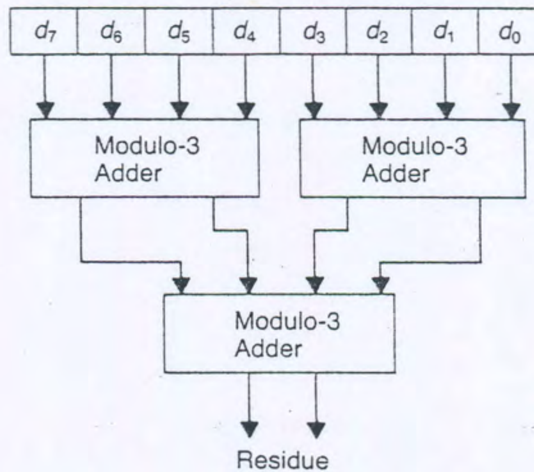


Fig. 3.52 The residue generation for 8 bits of data can be performed using three modulo-3 adders.

Inverse-Residue Codes

A modification of the separable residue code is the separable **inverse-residue code.** The inverse-residue code is formed in a manner similar to that of the residue code by appending information to the original data. Rather than append the residue, the inverse residue is calculated and appended. The inverse residue Q is calculated for a data word N as $m - r$, where m is the modulus and r is the residue of N. The code word for N then becomes $N \mid Q$. Table 3.13, for example, shows the inverse-residue code for 4-bit data using a modulus of 3.

The inverse-residue codes have been found to have better fault detection capability for repeated-use faults [Avizienis 1971]. A **repeated-use fault** is one that is encountered multiple times before the code is checked because the hardware is used multiple times before the code is checked. For example, if repeated addition is used to perform multiplication and the adder has a fault of some type, a repeated-use fault occurs. Repeated-use faults are particularly difficult to detect because subsequent effects of the fault can cancel the previous effects of the fault, thus rendering the fault undetectable.

Residue Number System

The final arithmetic code to consider is the **residue number system** (RNS) [Taylor 1984]. The residue number system has certain advantages in the speed at which arithmetic operations can be performed. In the RNS, num-

TABLE 3.13 Inverse-residue code words for 4-bit information words using a modulus of three

Information	Residue	Inverse residue	Code
0000	0	3	0000 11
0001	1	2	0001 10
0010	2	1	0010 01
0011	0	3	0011 11
0100	1	2	0100 10
0101	2	1	0101 01
0110	0	3	0110 11
0111	1	2	0111 10
1000	2	1	1000 01
1001	0	3	1001 11
1010	1	2	1010 10
1011	2	1	1011 01
1100	0	3	1100 11
1101	1	2	1101 10
1110	2	1	1110 01
1111	0	3	1111 11

bers are represented by a set of residues. The RNS does not produce a separable code, as did the residue and the inverse-residue codes.

To represent a number in the RNS, we first define a set of relatively prime moduli. The concept of relatively prime implies that the largest number that divides evenly into any two moduli is 1. The moduli set is given as

$$P = \text{moduli set} = [p_1, p_2, \ldots, p_L]$$

The range of numbers that can be represented using this moduli set is $0 \le N \le M$, where N is the number to be represented and M is the product of all the moduli. The representation of a number N in the RNS is determined by computing the residue of N for each of the moduli p_i. In other words N is represented as the collection of residues

$$N = (x_1, x_2, \ldots, x_L)$$

where x_i is the residue of N calculated using the modulus p_i. For example, suppose we wish to represent the integer 32 using the moduli set [3,4,5]. If we divide 32 by 3, we obtain a remainder of 2. If we divide 32 by 4, we obtain a remainder of 0. If we divide 32 by 5, the remainder is 2. Consequently, the representation of 32, using the moduli set [3,4,5] is given by (2,0,2).

One of the biggest advantages of the RNS is the fact that it is a carry-free number system. Therefore, arithmetic operations such as addition can

be performed on the individual digits of numbers independent of the remaining digits of the number. In contrast to the decimal and the binary number systems where carry information from the previous digits must be known before the result for the present digit can be calculated, the RNS allows the calculation of all digits of the result of an operation in parallel. As an example, suppose we wish to add the two numbers 32 and 14. If the moduli set is [3,4,5], the RNS representation of 32 is (2,0,2) and the representation of 14 is (2,2,4). The sum of 32 and 14 is 46, and the RNS representation of 46 is (1,2,1), also using the moduli set [3,4,5]. The RNS representation of 46 can be obtained by adding the RNS representations of 32 and 14 using the appropriate modulus to perform the addition of each digit. This process is illustrated in Fig. 3.53. Stated mathematically, the sum of two RNS numbers $(x_1, x_2, \cdots, x_L)$ and $(y_1, y_2, \cdots, y_L)$ is given by $(z_1, z_2, \cdots, z_L)$, where each z_i is computed as $z_i = (x_i + y_i)$ modulo-p_i.

To better see the possibility of speed improvements in an RNS addition compared to a binary addition, consider Fig. 3.54 where the ripple-carry addition of two 6-bit binary numbers is compared to the addition of the same two numbers when RNS representations and arithmetic are used. Note that modulo adders can be used to add the corresponding residues of the RNS representations. The RNS addition allows the residues to be added in parallel, and each residue has a smaller number of bits than the 6 bits obtained in the binary representations. Also, RNS adders can be designed

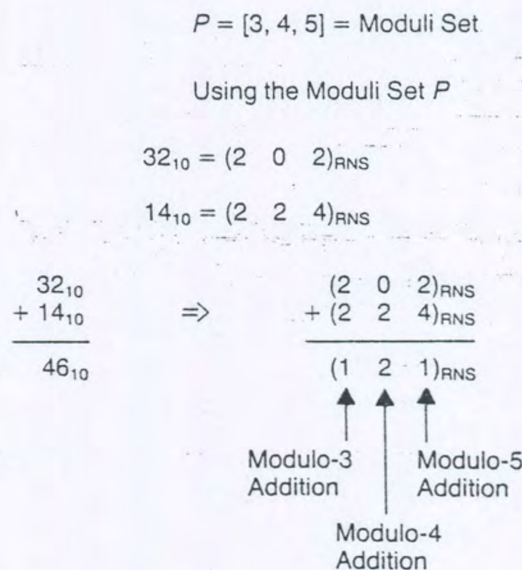


Fig. 3.53 In residue arithmetic, each modulus is added independently to produce the result.

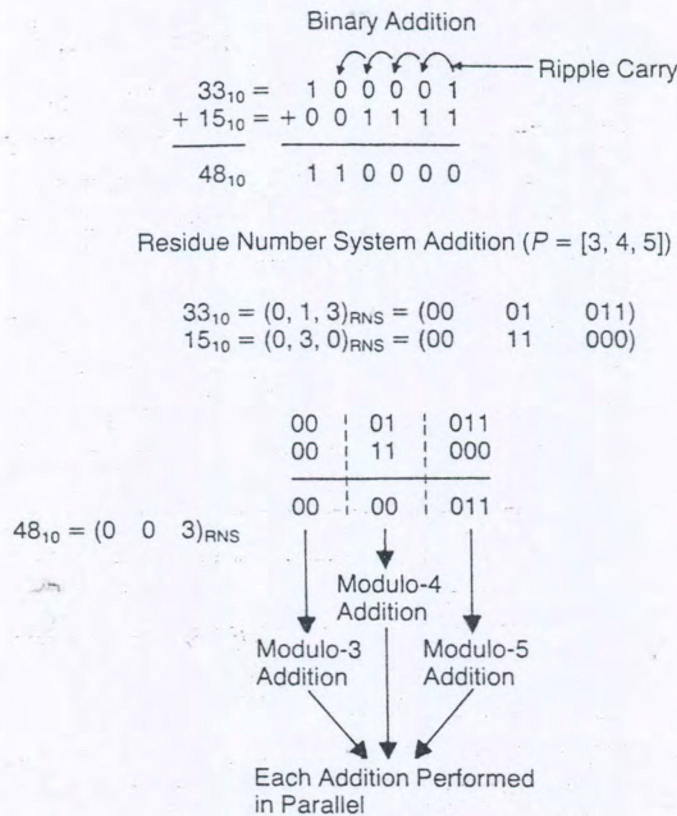


Fig. 3.54 The residue number system offers the advantage of speed over binary addition because the residues are added in parallel.

using techniques such as look-up tables. Since the residues are typically small, look-up tables are practical and allow for very-high-speed arithmetic operations.

Although speed is certainly the primary reason for using the RNS, the error detection capabilities of the code add to its attractiveness in many applications. To obtain error detection capability in the residue number system, redundant moduli are added to the moduli set. Suppose we selected the moduli set $[p_1, p_2, \cdots, p_L, p_{L+1} \cdots, p_M]$, where p_1 through p_L are sufficient to represent the desired range of numbers and p_{L+1} through p_M are redundant. Define M and T as

$$M = p_1 p_2 \cdots p_L$$

$$T = p_{L+1} p_{L+2} \cdots p_M$$

The nonredundant moduli represent the range of numbers from 0 through M. The addition of the redundant moduli allows the representable range of numbers to be expanded to 0 through MT. A code can now be structured such that valid code words lie in the range from 0 through M, and invalid

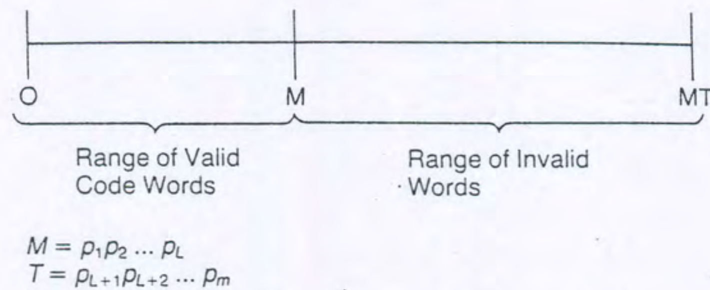


Fig. 3.55 Error detection capability is created in a residue representation because of the use of redundant moduli.

code words lie in the range from $M + 1$ through MT. This concept is illustrated in the simple number line shown in Fig. 3.55. As an example, consider Table 3.14. Table 3.14a shows the RNS representations of the numbers from 0 to 7 using the nonredundant moduli set [3,4]. This representation clearly has no error detection capability. A single-bit error can map a valid code word, for example, (1010), into another valid code word, in this example, (0010). If redundant moduli are used, however, error detection capability can be provided. Table 3.14b shows the RNS representations obtained when the moduli set [3,4,5] is used. The modulus 5 is redundant in this example. As can be seen, the RNS representation using the redundant moduli of this example forms a distance-2 code that allows the detection of any single-bit errors.

The residue number system offers advantages of speed and error detection in many applications, however, its use is not widespread. The primary difficulty with using the RNS is the conversion between a representation such as binary or decimal and the RNS. Human beings have difficulty interpreting numbers in the RNS, consequently applications involving human/computer interfaces must provide conversions between the RNS and a representation such as decimal. Because such conversions are extremely difficult, the RNS has not found widespread applicability.

3.5.7 Berger Codes

A very simple form of coding is the **Berger code** [Lala 1985]. Berger codes are formed by appending a special set of bits, called the check bits, to each word of information. Therefore, the Berger code is a separable code. The check bits are created based on the number of 1s in the original information. A Berger code of length n has I information bits and k check bits, where $k = \lceil \log_2(I + 1) \rceil$ and $n = I + k$. A code word is formed by first creating a binary number that corresponds to the number of 1s in the original I

TABLE 3.14 Residue number system representations
(a) Representation using the nonredundant moduli set
$P = [3,4]$

Number	RNS representation	RNS (binary form)	
0	0 0	00 00	
1	1 1	01 01	
2	2 2	10 10 ←	Single-bit
3	0 3	00 11	error can
4	1 0	01 00	map one
5	2 1	10 01	into the
6	0 2	00 10 ←	other.
7	1 3	01 11	

(b) Representation using the redundant moduli set
$P = [3,4,5]$

Number	RNS representation	RNS (binary form)
0	0 0 0	00 00 000
1	1 1 1	01 01 001
2	2 2 2	10 10 010
3	0 3 3	00 11 011
4	1 0 4	01 00 100
5	2 1 0	10 01 000
6	0 2 1	00 10 001
7	1 3 2	01 11 010

bits of information. The resulting binary number is then complemented and appended to the I information bits to form the $(I + k)$-bit code word. For example, suppose that the information to be encoded is (0111010), such that $I = 7$. The value of k is then $k = \lceil \log_2(7 + 1) \rceil = 3$. The number of 1s in this word of information is four, and the 3-bit binary representation of four is (100). The complement of (100) is (011), so the resulting code word is (0111010011), which is simply the original information with 011 appended.

If the number of information bits is small, the redundancy of a Berger code is high. Table 3.15, for example, shows the number of check bits required as a function of the number of information bits. As can be seen, the redundancy is 50% or greater if the number of information bits is eight or less. However, as the number of information bits increases, the efficiency of the code improves substantially.

If the number of information bits is related to the number of check bits by the relationship

$$I = 2^k - 1$$

TABLE 3.15 Number of required check bits in a Berger code

Number of information bits	Number of check bits	Percentage redundancy
4	3	75.00%
8	4	50.00
16	5	31.25
32	6	18.75
64	7	10.94

the resulting code is called a maximal length Berger code. For example, the code constructed for $I = 7$ and $k = 3$ is a maximal length Berger code.

The primary advantages of the Berger codes are that they are separable and they detect all multiple, unidirectional errors. For the error detection capability it provides, the Berger codes use the fewest number of check bits of the available separable codes [Lala 1985]. The resulting Berger codes for $I = 4$ and $k = 3$ are illustrated in Table 3.16.

3.5.8 Horizontal and Vertical Parity

The use of both horizontal and vertical parity is a very simple extension of the basic parity scheme. Suppose that we have a memory consisting of

TABLE 3.16 Berger code words for 4-bit information words

Original information	Berger code	
0000	0000	111
0001	0001	110
0010	0010	110
0011	0011	101
0100	0100	110
0101	0101	101
0110	0110	101
0111	0111	100
1000	1000	110
1001	1001	101
1010	1010	101
1011	1011	100
1100	1100	101
1101	1101	100
1110	1110	100
1111	1111	011

three 4-bit words, as shown in Fig. 3.56. By forming an odd parity bit, for example, for both the columns and the rows, we can detect and locate an error because the error affects the parity in both a column and a row. The erroneous bit is at the intersection of the row and column that has erroneous parity. Once the location of the erroneous bit is known, the error can be corrected by simply complementing the affected bit. If the bit is a 0 and it is incorrect, the correct value is a 1. Likewise, if the bit is a 1 and it is incorrect, the correct value is a 0.

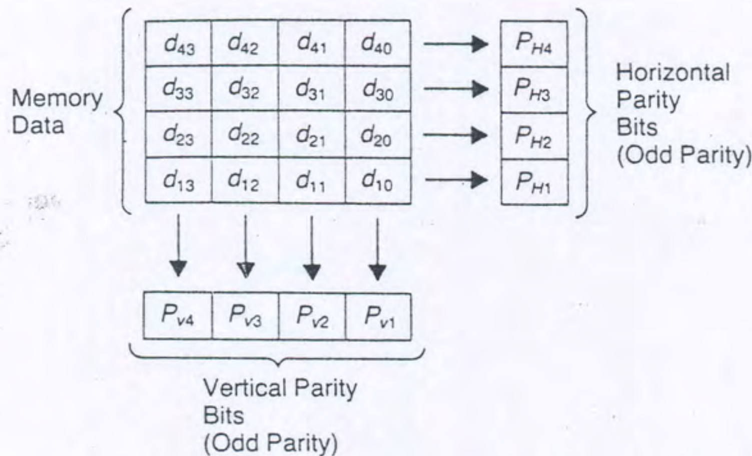


Fig. 3.56 Vertical and horizontal parity uses a parity bit for each row and each column. If bit d_{31}, for example, becomes in error, both P_{H3} and P_{V2} will be erroneous. All other parity bits will be correct.

Horizontal and vertical parity is a very useful technique for correcting errors in groups of data words. Suppose, for example, that a block of n k-bit words is to be transmitted from one point within a system to another. Each word can be extended to include a single parity bit, so that each word is $k + 1$ bits. Also, the n words can be EXCLUSIVE-OR'ed together on a bit-by-bit basis to create one $(k + 1)$-bit word that is the parity word for the block of data. At the receiving point, the parity bit for each word and the parity word for the group are recomputed, and the location of any single error is determined and corrected.

Horizontal and vertical parity, however, cannot help correct *multiple* errors within the block of data words. The existence of multiple columns and rows with erroneous parity prevents successful identification of the erroneous bit. For example, if rows i and j and columns k and m have erroneous parity, the errors can be in bit positions (i,k), (i,m), (j,k), or (j,m). Consequently, correction is not possible, but the detection of a problem is still accomplished.

3.5.9 Hamming Error-Correcting Codes

Possibly the most common extension of the fundamental parity approach is the **Hamming error-correcting code** [Hamming 1950]. Many memory designs incorporate error correction for several reasons. First, Hamming error correction is relatively inexpensive; typically, the Hamming codes require anywhere from 10% to 40% redundancy. Second, the Hamming codes are efficient in terms of the time required to perform the correction process; the encoding and the decoding processes inject relatively small time delays. Third, the error correction circuit is readily available on inexpensive chips. Finally, the memory can contribute as much as 60% to 70% of the faults in a system. In addition, transient faults are becoming much more prevalent as memory chips become denser. The combination of permanent and transient faults in memories makes the use of error correction very attractive.

The Hamming codes are best thought of as overlapping parity. The Hamming single error-correcting code uses c parity check bits to protect k bits of information. The relationship between the values of c and k is

$$2^c \geq c + k + 1$$

The total length of the code word is $n = c + k$. As we saw in the overlapping parity approach, the c check bits provide one unique combination for each possible information bit that can be erroneous, one combination for each parity check bit that can be erroneous, and one combination for the error-free case.

The Hamming code is formed by partitioning the information bits into parity groups and specifying a parity bit for each group. The ability to locate which bit is erroneous is obtained by overlapping the groups of bits. In other words, a given information bit will appear in more than one group in such a way that if the bit is erroneous, the parity bits that are in error will identify the erroneous bit. For example, suppose that there are four information bits (d_3, d_2, d_1, d_0) and, as a result, three parity check bits (c_1, c_2, c_3). The bits are partitioned into groups as (d_3, d_1, d_0, c_1), (d_3, d_2, d_0, c_2), and (d_3, d_2, d_1, c_3). Each check bit is specified to set the parity, either even or odd, of its respective group. Now, if bit d_0, for example, is erroneous, both c_1 and c_2 are incorrect. However, c_3 is correct because the value of d_0 has no impact on the value of c_3. Table 3.17 shows the check bits that are affected for each possible erroneous bit when four bits of information and three check bits are used. Note that the effect of each erroneous bit is unique; a unique combination of parity check bits is produced in each case, so the erroneous bit can be located.

The basic process involved in the Hamming codes is no different from that of other codes. First, the original data is encoded by generating a set, call it C_g, of parity check bits. When the information is to be checked for

TABLE 3.17 Check bits affected by single data bit errors

Erroneous bit	Check bits affected
d_0	c_1, c_2
d_1	c_1, c_3
d_2	c_2, c_3
d_3	c_1, c_2, c_3
c_1	c_1
c_2	c_2
c_3	c_3

correctness, the encoding process is repeated and a set, call it C_c, of parity check bits is regenerated. If C_g and C_c agree, the information is correct. If, however, C_g and C_c disagree, the information is incorrect and must be corrected. To aid in the correction, we define the syndrome S as the result obtained by forming the EXCLUSIVE-OR of C_g and C_c. The syndrome is a binary word that is 1 in each bit position in which C_g and C_c disagree. A syndrome that is all 0s indicates correct information. Using the example of four information bits and three check bits, Table 3.18 illustrates the syndromes that result for each possible erroneous bit when the group partitions of Table 3.17 are used.

The syndrome can be used to point directly to the erroneous bit if the bits are arranged appropriately. Suppose that we number the bit positions from right to left with 1 representing the rightmost position and n representing the leftmost position in an n-bit word. Each bit is then placed in the bit position corresponding to the syndrome that will result when that bit becomes erroneous. For example, the syndromes shown in Table 3.18 suggest the bit ordering of Fig. 3.57. The value of the syndrome now locates specifically the bit that is in error.

TABLE 3.18 Resulting syndromes for each possible single bit error

Erroneous bit	Syndromes
d_0	110
d_1	101
d_2	011
d_3	111
c_1	100
c_2	010
c_3	001

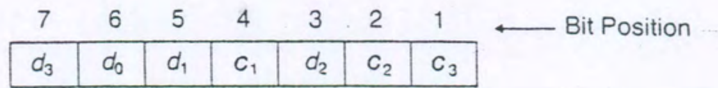


Fig. 3.57 Required ordering of bits to allow syndrome to identify specific erroneous bit.

The structure of a Hamming single-error–correction unit for four bits of information is shown in Fig. 3.58. When the information and the check bits are received, the parity check bits are regenerated and the syndrome bits

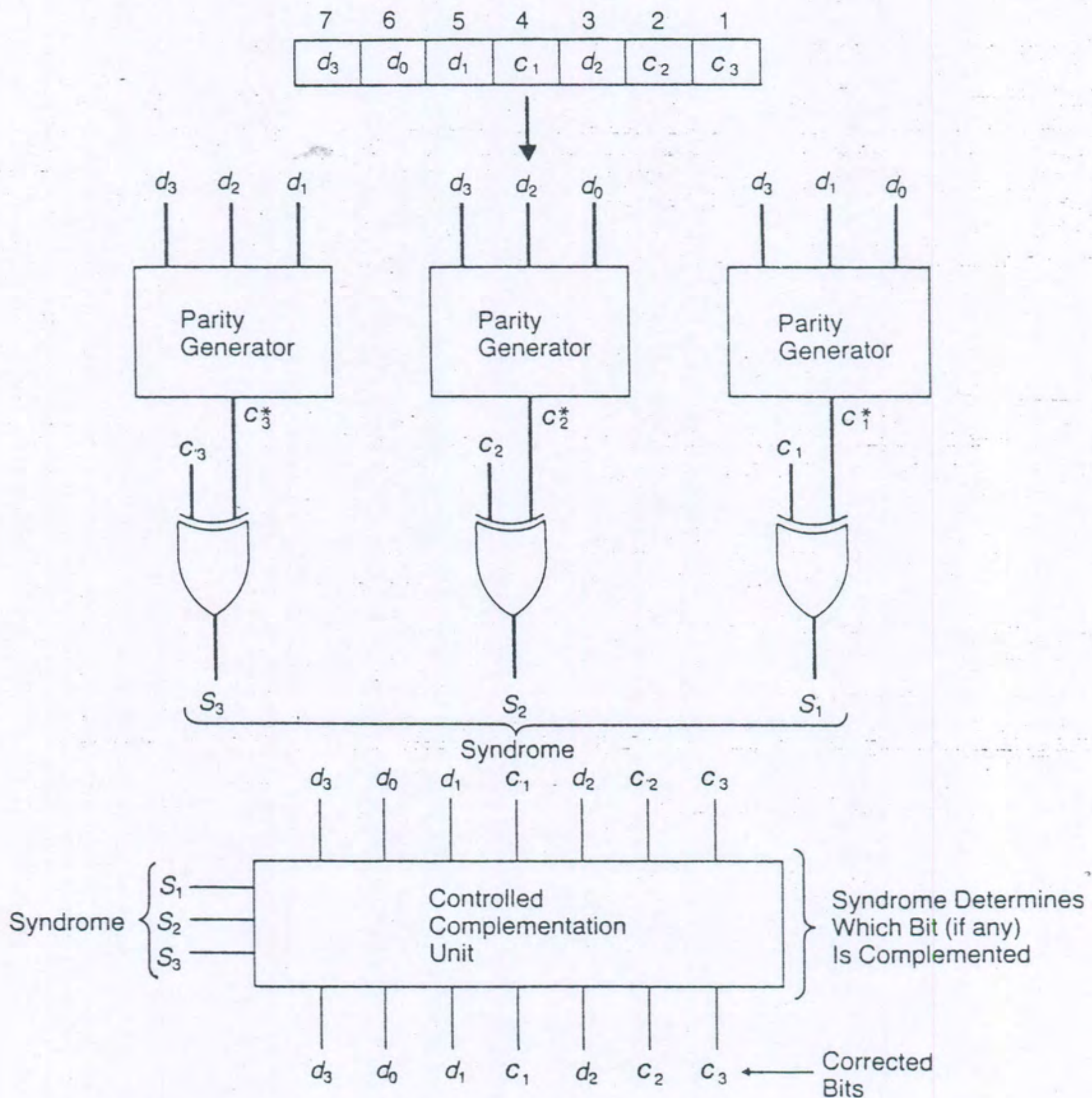


Fig. 3.58 Hamming single error correction unit for four information bits and three check bits.

are created. The syndrome bits then point to the erroneous bit, if one exists. By using a controlled complementation unit, the error can be corrected. The syndrome can specify which output, if any, of a multiplexer is set to the logic 1 state. If none of the multiplexer outputs is set to 1, the original information is passed through the EXCLUSIVE-OR gates in an uncomplemented fashion. If a bit is erroneous, the syndrome value results in the appropriate bit being complemented, thus the correction process has been implemented.

The structure of a memory that uses the Hamming single-error–correcting code is shown in Fig. 3.59. When data is written to memory, the check bits are generated and stored in memory along with the original information. Upon reading the information and the check bits from memory, the check bits are regenerated and compared to the stored check bits to generate the syndrome. The syndrome is then decoded to determine if a bit is erroneous, and if so, the erroneous bit is corrected by complementation. The corrected data is then passed to the user of the memory. Memories that use this type of correction are usually designed such that data is corrected without interrupting the normal operation of the system. The user of the memory might be informed, however, that a correction has occurred such that maintenance can be performed if corrections are continually required.

The basic Hamming code just described provides for the correction of single-bit errors. Unfortunately, double-bit errors are erroneously corrected using the basic Hamming code. Consider the preceding example that used three check bits to encode four information bits. Suppose that both bit d_0 and c_2 are in error. The resulting syndrome is 100, which implies that the bit located in position four is in error and should be corrected. Consequently, bit c_1 has been erroneously corrected.

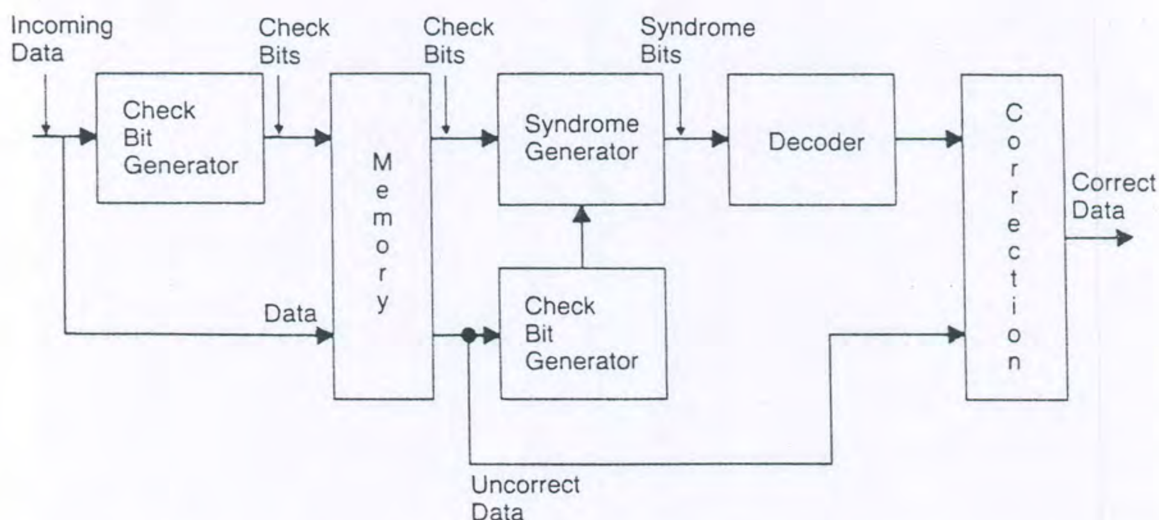


Fig. 3.59 Basic structure of a memory using Hamming single error correcting code.

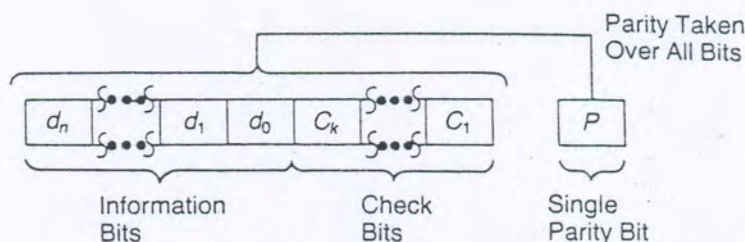


Fig. 3.60 Hamming code modification to achieve double error detection and single error correction.

To overcome the problem of erroneous correction and provide a code that can correct single-bit errors and identify double-bit errors, the basic Hamming code is modified. The resulting code is called the *modified* Hamming code. The modification consists of simply adding one additional parity check bit that checks parity over the entire Hamming code word, as shown in Fig. 3.60. If a single bit is in error, the additional parity bit indicates that the overall parity is incorrect. The syndrome then points to the correct bit that is erroneous. If a double-bit error occurs, the additional parity bit indicates that the overall parity is correct because a single parity check cannot detect a double-bit error. But, the syndrome is nonzero because the remaining parity checks indicate an error. Therefore, the double error can be detected, and an erroneous correction prevented.

To summarize, if the overall parity is incorrect and the syndrome is not 0, a single-bit error is corrected. If the overall parity is correct and the syndrome is not 0, a double-bit error is identified and no correction occurs. If the overall parity is correct and the syndrome is 0, the data is assumed to be correct.

3.5.10 Error-Correcting Integrated Circuits

Several commercial integrated circuits (ICs) are available to detect, locate, and correct errors. Examples include the Intel 8206, Motorola MC68540, Advanced Micro Devices AM2960 and AMZ8160, National Semiconductor DP8400, and Fujitsu MB1412A. The ICs are designed to generate the check bits, generate the syndrome, decode the syndrome, and perform the data correction. Typical units are organized to support 16-bit data but can usually be expanded to provide error detection and correction for larger word sizes. For example, the Intel 8206 can support up to 80-bit words in a memory [Intel 1985]. The ICs almost always use the modified Hamming code to detect double errors and correct single errors.

A block diagram of the fundamental structure of an error-correcting IC is shown in Fig. 3.61. A word coming from the system bus goes directly into the memory and into a write check bit generator. Both the original data and the generated check bits are stored in memory. Upon reading a word from memory, the following events occur:

1. The data is used to regenerate the check bits using a read check bit generator.
2. A syndrome is created by comparing the original check bits with the regenerated check bits.

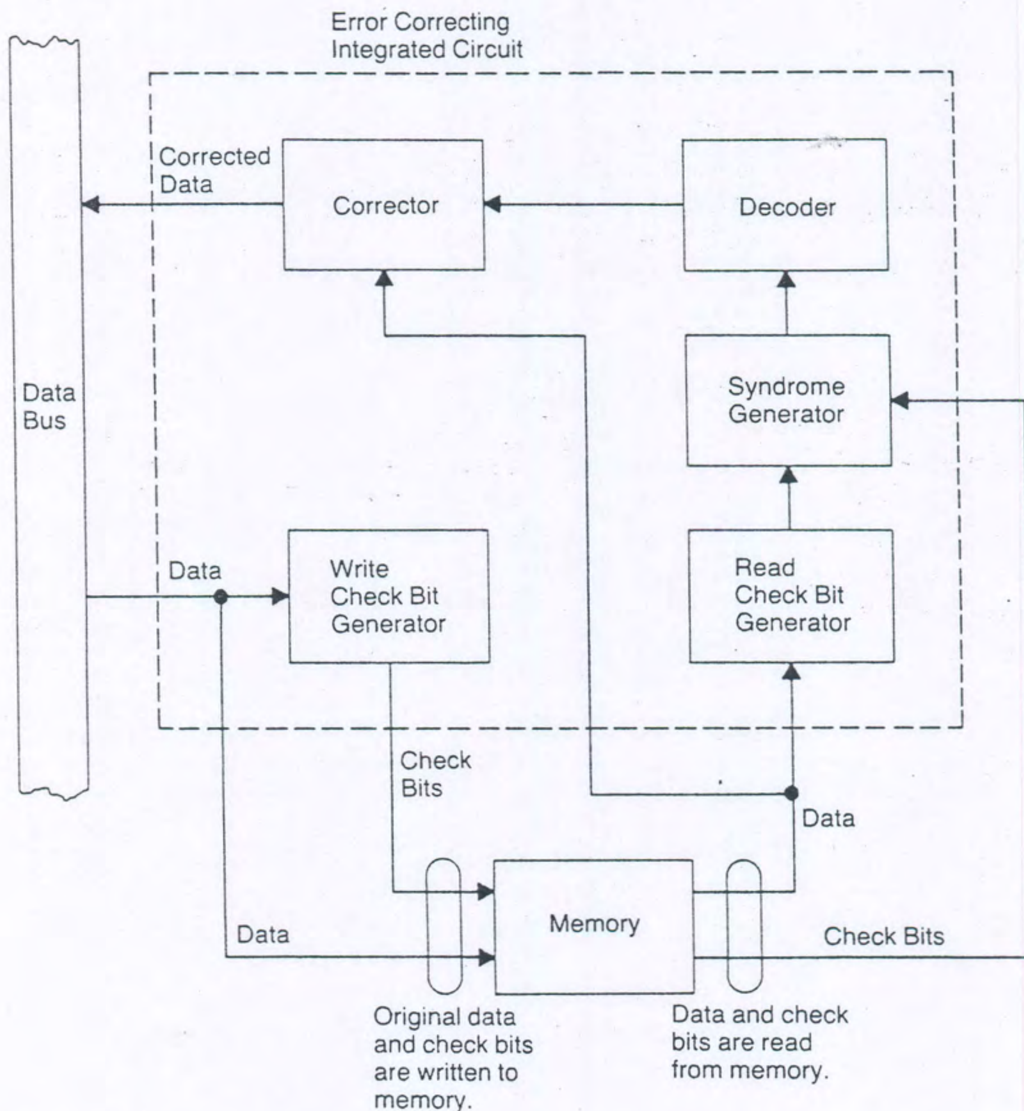


Fig. 3.61 Typical organization of a commercially available error correction integrated circuit.

3. The syndrome is used to identify any erroneous bits.
4. The data is passed through a correction unit.
5. The corrected data is placed on the system bus.

The device typically provides an error indication so that the user can record the occurrence of the error for maintenance purposes. Often, the number of error occurrences is used to indicate a permanent fault in the memory.

The additional time required during a memory write operation is simply the time necessary to generate the check bits. When data is read from memory, however, the check bits must be regenerated, the syndrome calculated and decoded, and the correction completed. Once the syndrome is generated and decoded the existence or nonexistence of an error is known. The final step of correction is necessary, however, before the data is used.

The Intel 8206, for example, can perform the detection process in a 16-bit system in a maximum of 52 nanoseconds [Intel 1985]. The combination of the detection and the correction requires no more than 67 nanoseconds. Designed using Intel's HMOS technology, the 8206 is available in 68-pin leadless packages and operates from a single 5-volt power supply.

3.5.11 Code Selection Issues

Several tradeoffs must be performed prior to the selection of a particular coding technique. As previously mentioned, codes are considered a form of information redundancy because the fundamental concept is to modify the original data in such a manner that sufficient information is provided to allow error detection and/or correction to be performed. The information redundancy requires, however, that other forms of redundancy exist as well. For example, the encoding and decoding processes require time, so redundancy of time is produced. Likewise, the additional bits found in the code word require additional storage and processing circuitry, so redundancy of hardware is produced. The amount of redundancy required in a code is a good measure of the cost of the code. On the other hand, the effectiveness of the code is usually measured in terms of the number of bit errors that can be detected or corrected. The key to the design process is to select a code that fulfills the desired error detection and/or correction capability while maintaining costs at an acceptable level.

A major decision in the selection of a code is whether or not the code needs to be separable. Separable codes are easier to use because the decoding process consists of simply ignoring the check bits that have been added. As a result, the information is available immediately, and, in many cases, a system can begin to use the information in parallel with the checking of the information. For example, a memory that uses single-bit parity can provide

the data to a processor concurrent with checking the parity of that data. Nonseparable codes, however, require that the decoding process be performed before the data is available for use. In some applications, encoding and decoding can be performed in parallel with transmission and reception to minimize the impact of the encoding and decoding times. For example, encoding and decoding of cyclic codes can be performed bit by bit as the information is transmitted and received in a serial fashion. Consequently, cyclic codes find wide applicability in systems in which data is transmitted serially.

A second major decision in the selection of a code is whether error detection, error correction, or both are required. In applications in which temporary, erroneous results are acceptable—for example, in active redundancy systems that require reconfiguration—error detection is normally sufficient. However, if fault masking is mandatory to prevent erroneous data from propagating throughout a system, error correction is needed. If single-error correction, as opposed to single-error detection, is needed, a code with a larger distance is mandatory, and the redundancy required increases.

A third major decision in the selection of a code is the number of bit errors that need to be detected or corrected. As we have seen, error detection and correction capability is directly related to the distance of the code and, as a result, the necessary redundancy. The application dictates, to a large degree, the necessary distance of the code. For example, in serial communication systems, a single fault can easily affect several consecutive bits in a code word; so the ability to handle multiple-bit errors is important. In many memory designs, however, single-bit errors are common, and single-error detection or single-error correction codes are typically employed.

3.6 Time Redundancy

The fundamental problem with the forms of redundancy discussed thus far is the penalty paid in extra hardware for the implementation of the various techniques. Both hardware redundancy and information redundancy can require large amounts of extra hardware for their implementation. In an effort to decrease the hardware required to achieve fault detection or fault tolerance, time redundancy has recently received much attention. Time redundancy methods attempt to reduce the amount of extra hardware at the expense of using additional time. In many applications, the time is of much less importance than the hardware because hardware is a physical entity that impacts weight, size, power consumption, and cost. Time, on the other hand, may be readily available in some applications.

The selection of a particular type of redundancy is very dependent upon the application. For example, some systems can better stand additional

hardware than additional time; others can tolerate additional time much more easily than additional hardware. The selection in each case must be made by examining the requirements of the application and the available techniques that can meet such requirements.

3.6.1 Transient Fault Detection

The basic concept of time redundancy is the repetition of computations in ways that allow faults to be detected. Time redundancy can function in a system in several ways, but the most basic form of time redundancy is illustrated in Fig. 3.62. The fundamental concept is to perform the same computation two or more times and compare the results to determine if a discrepancy exists. If an error is detected, the computations can be performed again to see if the disagreement remains or disappears. Such approaches are often good for detecting errors resulting from transient faults, but they cannot protect against errors resulting from permanent faults.

A second time redundancy approach makes use of other detection techniques that can be built into a system. Suppose, for example, that an error-detecting code provides the primary means of error detection within a

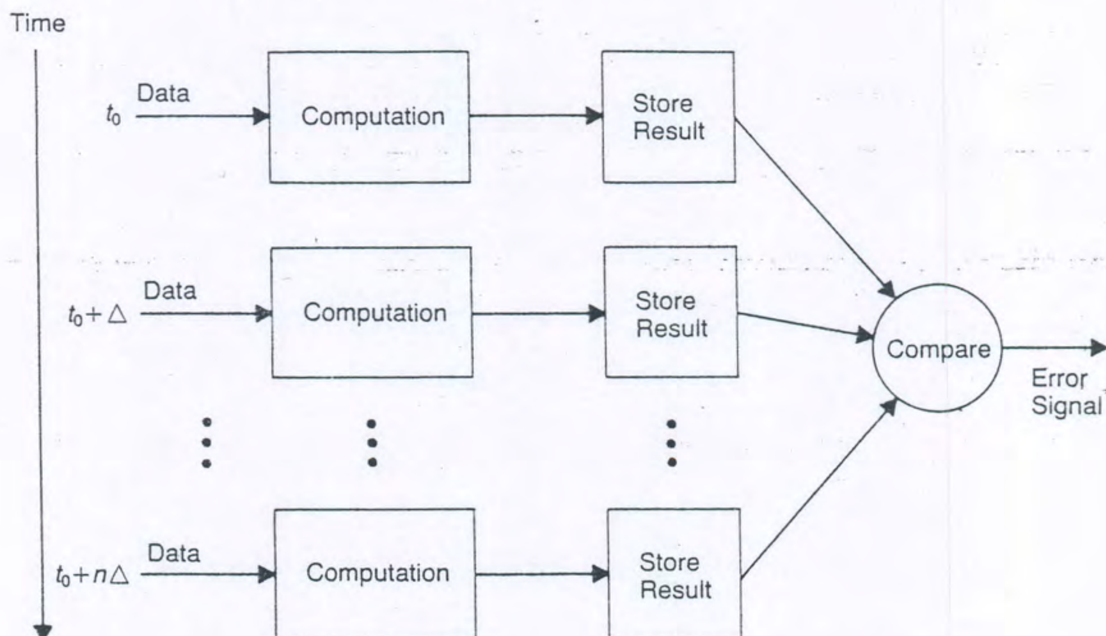


Fig. 3.62 In time redundancy, computations are repeated at different points in time and then compared.

system. When the coding scheme detects an error, either of two conditions exists:

1. A permanent fault has produced the error, and the correct course of action is to shut down the faulty portions of the system.
2. A transient fault has produced the error, in which case the hardware is still usable, and it would be a waste of resources to shut down the system.

Time redundancy can often be employed to distinguish between the permanent and the transient faults. The system can perform the computations one or more times after the detection of the first error; if the error condition clears, the fault that caused the error can be assumed to have been transient. If, however, the problem continues to be detected, the fault is most likely permanent, and the faulty parts of the system must be removed from operation.

The main problem with many time redundancy techniques is assuring that the system has the same data to manipulate each time it redundantly performs a computation. If a transient fault has occurred, a system's data may be completely corrupted, making it difficult to repeat a given computation.

Another example of time redundancy actually combines both information and time redundancy. The concept requires that the same computations be performed multiple times using different coding schemes in each case. For example, suppose that a processor uses an *AN* arithmetic code to detect errors in the arithmetic operations performed by that processor. The processor can perform the functions first using a $3N$ code and second using a $5N$ code. The assumption is that a fault should affect the two computations in different ways because the data that is being manipulated by the hardware differs in the two cases.

3.6.2 Permanent Fault Detection

In the past, time redundancy has been used primarily to detect transients in systems. One of the biggest potentials of time redundancy, however, now appears to be the ability to detect permanent faults while using a minimum of extra hardware. Four approaches are considered; alternating logic [Reynolds and Metze 1978], recomputing with shifted operands (RESO) [Patel and Fung 1982], recomputing with swapped operands (RESWO) [Johnson 1984], and recomputing with duplication with comparison (REDWC) [Johnson, Aylor, and Hana 1988].

The fundamental concept of each of the approaches considered is illustrated in Fig. 3.63. During the first computation or transmission, the operands are used as presented and the results are stored in a register. Prior

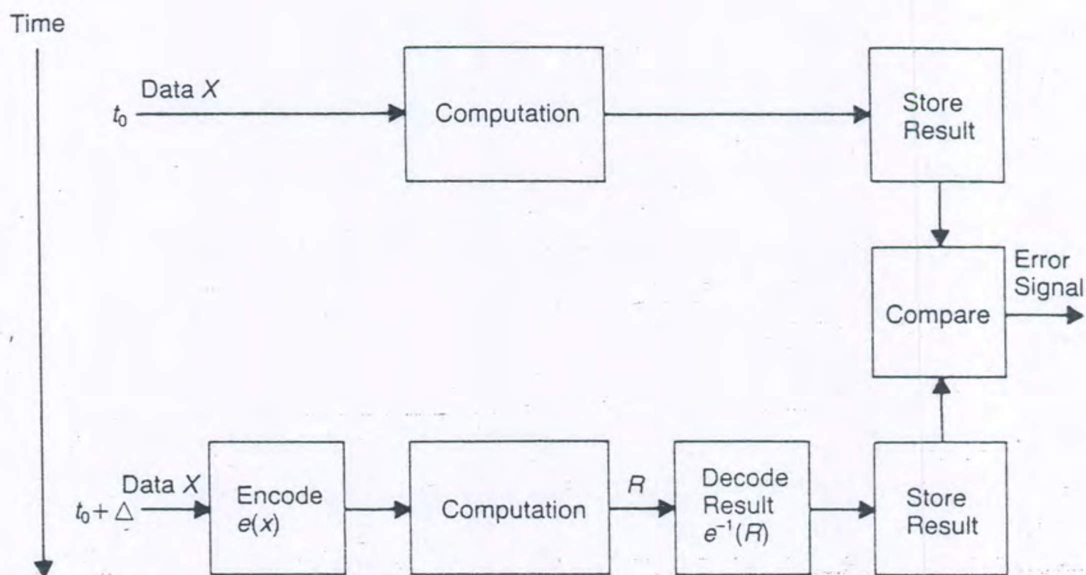


Fig. 3.63 Permanent faults can be detected using time redundancy by modifying the way in which computations are performed the second time.

to the second computation or transmission, the operands are encoded in some fashion using the encoding function e. After the operations have been performed on the encoded data, the results are then decoded and compared to those obtained during the first operation. The selection of the encoding function is made so as to allow faults in the hardware to be detected. The alternating logic approach uses the complementation operator as the encoding function. RESO uses an arithmetic shift as the encoding function, RESWO uses a swapping function to encode the operands and REDWC is a variation of RESWO. Each approach has both advantages and disadvantages.

Alternating Logic

The **alternating logic** concept has been applied to the transmission of digital data over wire media and the detection of faults in digital circuits. Suppose that we wish to detect errors in data that is transmitted over a parallel bus, as shown in Fig. 3.64, using the time redundancy approach. At time t_0 we transmit the original data, and at time $t_0 + \delta$ we transmit the complement of the data. As shown in Fig. 3.65, if a line of the bus is stuck at either a 1 or a 0, the two versions of the information that are received will not be complements of each other. Therefore, the fault can be detected. In general, if a sequence of information is transmitted using this approach, each bit line should alternate between a logic 1 and a logic 0; provided the transmission is error free. Thus, the reason for the name *alternating logic*.

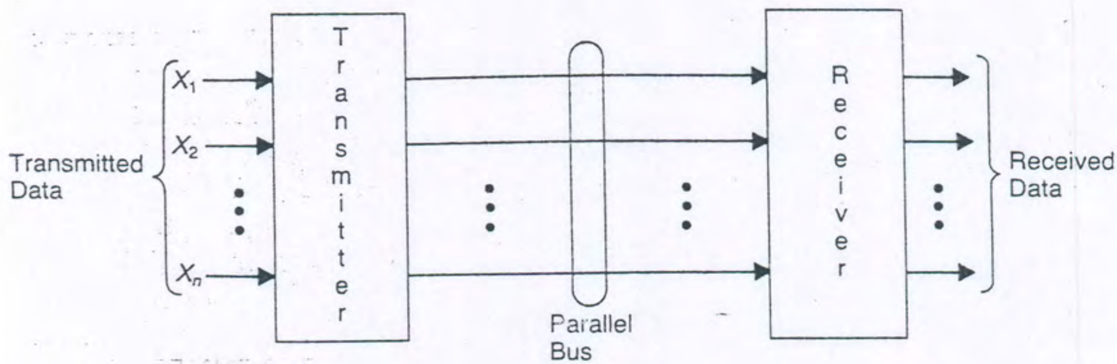


Fig. 3.64 Transmission system to be protected using time redundancy.

The concept of alternating logic can be applied to general, combinational logic circuits if the circuit possesses the property of self-duality. A combinational circuit is said to be self dual if and only if $f(X) = \overline{f}(\overline{X})$, where f is the output of the circuit and X is the input vector for the circuit [Kohavi 1978]. Stated verbally, a combinational circuit is self-dual if the output for the input vector X is the complement of the output when the input vector $\overline{X}$ is applied. For example, the full-adder circuit, shown in Fig. 3.66 with its truth table, is a self-dual circuit. For a self-dual circuit, the application of an input X followed by the input $\overline{X}$, produces outputs that alternate between 1 and 0. The key to the detection of faults using the alternating logic ap-

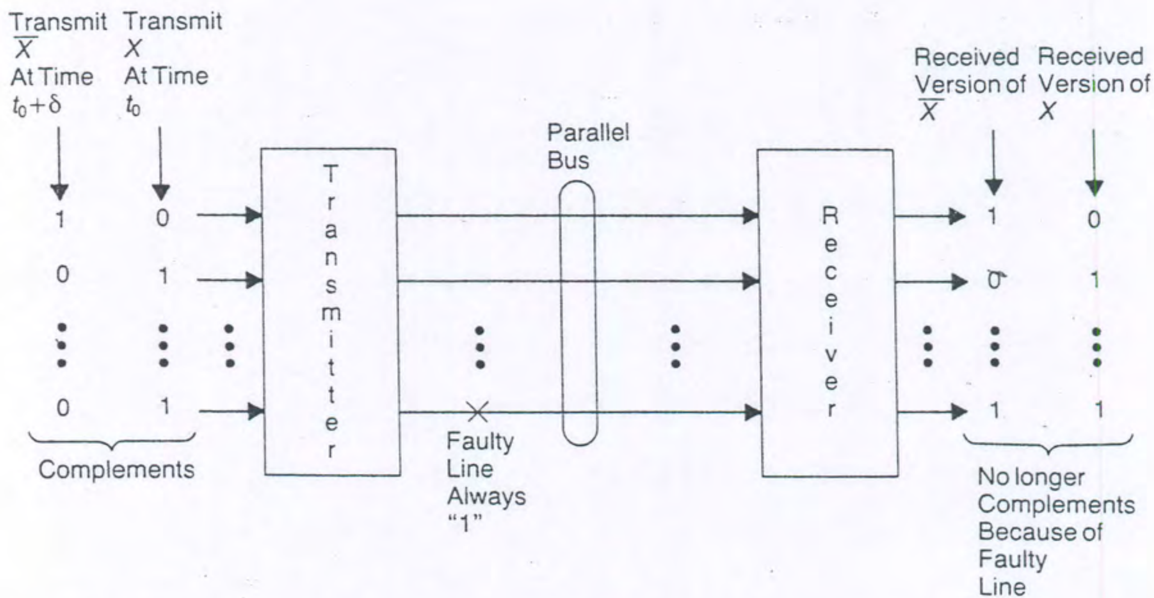


Fig. 3.65 Illustration of alternating logic time redundancy—the second transmission is the complement of the first.

proach is determining that at least one input combination exists for which the fault does not result in alternating outputs.

A key advantage of the alternating logic approach is that any combinational circuit with n input variables can be transformed into a self-dual circuit with no more than $n + 1$ input variables. To see this, first define the dual of a function. The dual f_d of an n-variable function f is given by [Kohavi 1978]

$$f_d = \overline{f}(\overline{x}_1, \overline{x}_2, \cdots, \overline{x}_n)$$

In other words, the dual of the function f is obtained by first complementing f and then replacing each variable with the complement of the variable. The function f_{sd} given by

$$f_{sd} = x_{n+1} f + \overline{x}_{n+1} f_d$$

is then a self-dual function because when x_{n+1} is 1, the value of f_{sd} is f, and when x_{n+1} is 0, the value of f_{sd} is f_d. Thus, x_{n+1} is a control line that determines which of two functions appears on the output line. So, defining $f_{sd}(x_1, x_2, \cdots, x_n, x_{n+1})$ to be a function of $n + 1$ variables, complementary inputs produce complementary outputs.

The use of alternating logic detects a set of faults if for every fault within the set there is at least one input combination that produces nonalternating outputs. For example, Table 3.19 shows the truth table that results when one stuck-at-1 or stuck-at-0 fault is present in the full adder circuit of Fig. 3.66. The notation A_0 and A_1 is used to represent the case where line A is stuck at logic 0 and logic 1, respectively. Similar notation is used for all other lines. As can be seen, each stuck-type fault results in at least one set of nonalternating outputs being produced for complementary inputs at either the carry or the sum output.

Note, however, that faults may not be immediately detected using alternating logic. For example, suppose that the full-adder contains a stuck-at-0 fault on line D. The sum output is not affected by this fault, so the carry output is depended upon for the detection of the fault. The carry output, however, will have alternating outputs for the complementary input combinations (000) and (111) as well as (001) and (110). In other words, the fault is not detected until one of the remaining four input combinations is applied to the circuit. Depending on the application, the time elapsed before the detection of the fault can be significant.

Recomputing with Shifted Operands

Another form of time redundancy is called **recomputing with shifted operands (RESO)** [Patel and Fung 1982]. RESO was developed as a method to provide concurrent error detection in arithmetic logic units (ALUs). RESO uses the basic time redundancy method that was shown in Fig. 3.63,

TABLE 3.19 Truth table for single faults in the full-adder circuit of Fig. 3.66

(a) Sum output truth table

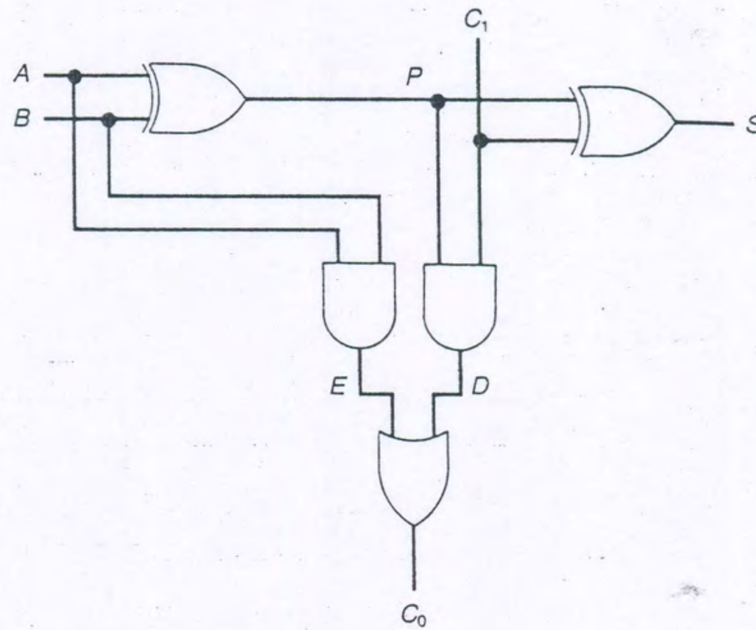
			Fault-free sum	Sum when specified fault is present															
A	B	C_1	S	A_0	B_0	P_0	C_{10}	D_0	E_0	C_{00}	S_0	A_1	B_1	P_1	C_{11}	D_1	E_1	C_{01}	S_1
0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1	0	0	0	1
0	0	1	1	1	1	1	0	1	1	1	0	0	0	0	1	1	1	1	1
0	1	0	1	1	0	0	1	1	1	1	0	0	1	1	0	1	1	1	1
0	1	1	0	0	1	1	1	0	0	0	0	1	0	0	0	0	0	0	1
1	0	0	1	0	1	0	1	1	1	1	0	1	0	1	0	1	1	1	1
1	0	1	0	1	0	1	1	0	0	0	0	0	1	0	0	0	0	0	1
1	1	0	0	1	1	0	0	0	0	0	0	0	0	1	1	0	0	0	1
1	1	1	1	0	0	1	0	1	1	1	0	1	1	0	1	1	1	1	1

(b) Carry output truth table

			Fault-free carry	Carry when specified fault is present															
A	B	C_1	C_0	A_0	B_0	P_0	C_{10}	D_0	E_0	C_{00}	S_0	A_1	B_1	P_1	C_{11}	D_1	E_1	C_{01}	S_1
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	0
0	0	1	0	0	0	0	0	0	0	0	0	1	1	1	0	1	1	1	0
0	1	0	0	0	0	0	0	0	0	0	0	1	0	0	1	1	1	1	0
0	1	1	1	1	0	0	0	0	1	0	1	1	1	1	1	1	1	1	1
1	0	0	0	0	0	0	0	0	0	0	0	0	1	0	1	1	1	1	0
1	0	1	1	0	1	0	0	0	1	0	1	1	1	1	1	1	1	1	1
1	1	0	1	0	0	1	1	1	0	0	1	1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1	1	0	0	1	1	1	1	1	1	1	1	1

and the encoding function is selected as the left shift operation with the decoding function being the right shift operation. In many cases, the shift operations can be either arithmetic or logical shifts. The RESO technique was derived assuming bit-sliced organizations of the hardware.

In logical operations, it is relatively easy to understand the error detection capability of the RESO technique. Suppose that bit slice i of a circuit is faulty and produces an erroneous value for the function's output at that bit slice. During the first computation with the operands not shifted, the ith output of the circuit is erroneous. When the operands are shifted by one bit, the faulty bit slice then operates on, and corrupts, the $(i - 1)$th bit. When the result is shifted back to the right, the two results—the first with unshifted operands and the second with shifted operands—are either both



A	B	C_1	S	C_0
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

Self-Dual Function

Fig. 3.66 The full-adder is a self-dual circuit—complementary inputs produce complementary outputs.

correct or they disagree in either (or both) the ith or the $(i - 1)$th bits. As an example, consider the logical AND operation performed on two 8-bit operands, as illustrated in Fig. 3.67. If only one bit slice is faulty and that faulty slice has no impact on any other bit slices, a single left shift detects the errors that occur in logical operations.

For a bit-sliced, ripple-carry adder, a 2-bit arithmetic shift is required to guarantee the detection of errors that can occur [Patel and Fung 1982]. Once again suppose that bit slice i is faulty. In a ripple-carry adder, a faulty bit slice can have one of three effects: the sum bit out of the bit slice can be

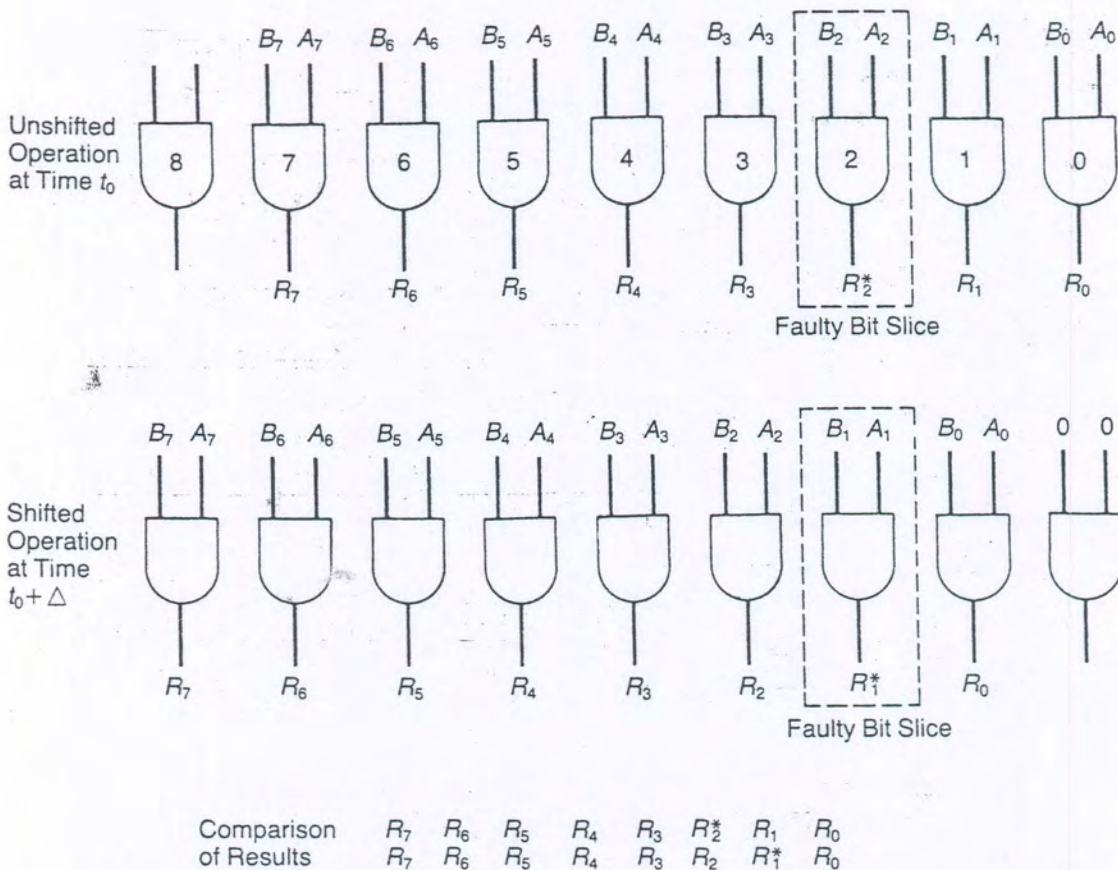


Fig. 3.67 Illustration of time redundancy for a logical operation. The fault affects different bits of the result each time and can therefore be detected.

erroneous, the carry bit out of the bit slice can be erroneous, or both may be in error. Table 3.20 shows the effect on the sum of each possible error.

In summary, the result generated for the unshifted operands if bit i is faulty is incorrect by one of $[0, \pm 2^i, \pm 2^{i+1}, \pm 3 \cdot 2^i]$.

TABLE 3.20 Errors and their effect on sum

Error	Effect on Sum
Sum is 0	-2^i
Sum is 1	$+2^i$
Carry is 0	-2^{i+1}
Carry is 1	$+2^{i+1}$
Sum is 0, carry is 0	$-(2^{i+1} + 2^i) = -3 \cdot 2^i$
Sum is 0, carry is 1	$2^{i+1} - 2^i = +2^i$
Sum is 1, carry is 0	$-(-2^i + 2^{i+1}) = -2^i$
Sum is 1, carry is 1	$2^{i+1} + 2^i = +3 \cdot 2^i$

When the operands are shifted to the left by two bits, a similar analysis can show that the result will be incorrect by one of $[0, \pm 2^{i-2}, \pm 2^{i-1}, \pm 3 \cdot 2^{i-2}]$. As can be seen, the results of the two computations cannot agree unless both are correct. Therefore, the error will be detected.

The structure of an ALU that uses the RESO technique is shown in Fig. 3.68. The additional hardware required for the technique is the three shifters, the storage register to hold the results of the first computation, and

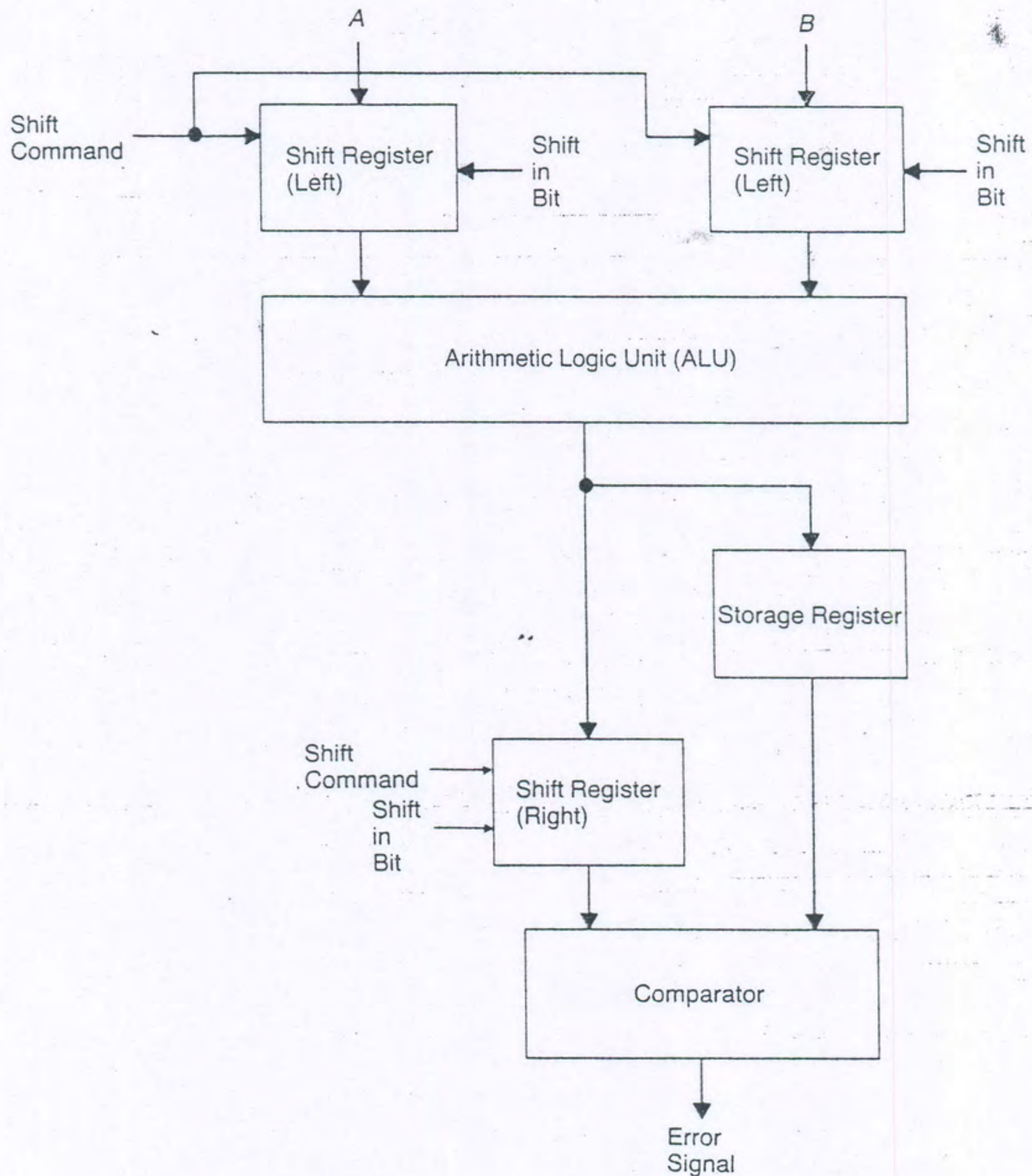


Fig. 3.68 ALU structure using RESO.

the comparator. Also, the ALU must be extended by 2 bits to allow a two-bit, arithmetic shift to be performed.

The primary problems with the RESO approach are the additional hardware required, the lack of coverage provided for faults in the shifters, and the requirement that the comparator be totally self-checking such that faults in the comparator do not render the approach ineffective.

The extra bits required for the ALU in the RESO approach can be eliminated if a cyclic shift is used instead of an arithmetic shift. But, a cyclic shift of one or two bits requires that the carry bits between bit slices be directed either to the neighboring bit slice when the operands are not shifted or another bit slice when the operands are shifted. The circuitry required to handle the carry bits can become more complex than the extra bit slices added to the ALU to accommodate the arithmetic shifts. A compromise is to swap the upper and lower halves of the operands as proposed in the technique called **recomputing with swapped operands (RESWO)** [Hana and Johnson 1986]. Several carry bits must still be appropriately handled, but the affected carries are the carry-in, the carry-out, and the auxiliary carry between the upper and lower halves of the ALU. Traditionally, ALUs are often designed such that these carry bits are easily accessible and controllable anyway.

Recomputing with Swapped Operands

The basic concept of the RESWO technique is shown in Fig. 3.69. During the first computation, the operands are manipulated in their standard form. During the second computation, the upper and lower halves of the operands are swapped such that a faulty bit slice operates on either the lower or upper half of the operands during the first computation and the opposite half of the operands during the second computation.

The RESWO technique can detect the existence of any single faulty bit slice in a ripple-carry adder. Using the same approach that we used to

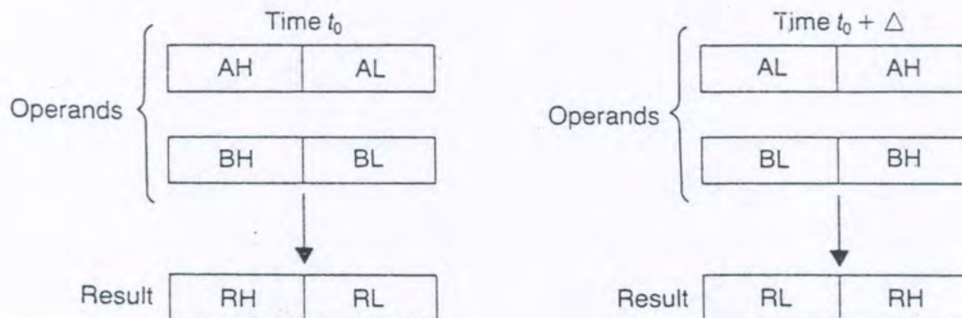


Fig. 3.69 In RESWO, operand halves are swapped prior to repeating the operation.

verify the fault detection capability of the RESO technique, consider the following two-step procedure. Suppose the words to be added are n bits each, with the lower half containing bits 0 through r and the upper half containing bits $r + 1$ through $n - 1$. If bit slice i is faulty, the sum and carry bits from slice i can impact the sum by $\pm 2^i$ and $\pm 2^{i+1}$, respectively. If both the sum and the carry of bit slice i are erroneous, the resulting sum may be off by $\pm 2^i \pm 2^{i+1}$. During the first computation, with the operand halves not swapped, the faulty bit slice i can cause the sum to be in error by one of $[0, \pm 2^i, \pm 2^{i+1}, \pm 2^i \pm 2^{i+1}]$. If $i \leq r$, the result of the recomputation with swapped operand halves is in error by one of $[0, \pm 2^{i+r+1}, \pm 2^{i+r+2}, \pm 2^{i+r+1} \pm 2^{i+r+2}]$. For $i > r$, the result of the recomputation with the operand halves swapped is in error by one of $[0, \pm 2^{i-r-1}, \pm 2^{i-r}, \pm 2^{i-r-1} \pm 2^{i-r}]$. In both cases, the results of the two computations either are not in error or they disagree, in which case the error can be detected.

For all logic operations, it is fairly easy to see the effectiveness of the swapped operand approach. During the computations with the operand halves not swapped, a faulty bit slice i affects bit i of the result. Once the operand halves are swapped, a faulty bit slice i affects bit $i \pm (r + 1)$, depending on whether the faulty bit slice is in the upper or lower half of the logical unit. In either case, the error is detected.

The structure of an ALU designed to accommodate the RESWO technique is shown in Fig. 3.70. C_{in} is the carry-in for the addition process and C_{out} is the carry-out of the addition process. Multiplexers are positioned appropriately to switch the carry bits when the operand halves are swapped. During the recomputation with swapped operand halves, the multiplexers direct C_{in} to the upper half of the adder and C_{out} is taken from the carry-out of the lower half of the adder. The carry-in for the lower half of the adder is then selected as the carry-out of the upper half of the adder. The multiplexers are very simple circuits that do not add significant circuitry to the overall ALU. For example, Fig. 3.71 shows the circuit required for the 2-to-1 multiplexer.

Recomputing with Duplication with Comparison

An alternative method that takes advantage of both time redundancy and hardware redundancy concepts is called **recomputing with duplication with comparison (REDWC)** [Johnson, Aylor, and Hana 1988]. The method with which error detection is accomplished resembles that of duplication with comparison. Time redundancy is then used to complete the calculation and obtain the final result. For example, a 32-bit addition operation can be realized by using each of two 16-bit adders twice. During each calculation, the error detection is accomplished by comparing the results from using the two adders. REDWC is similar in many respects to a method described in [Toy 1982].

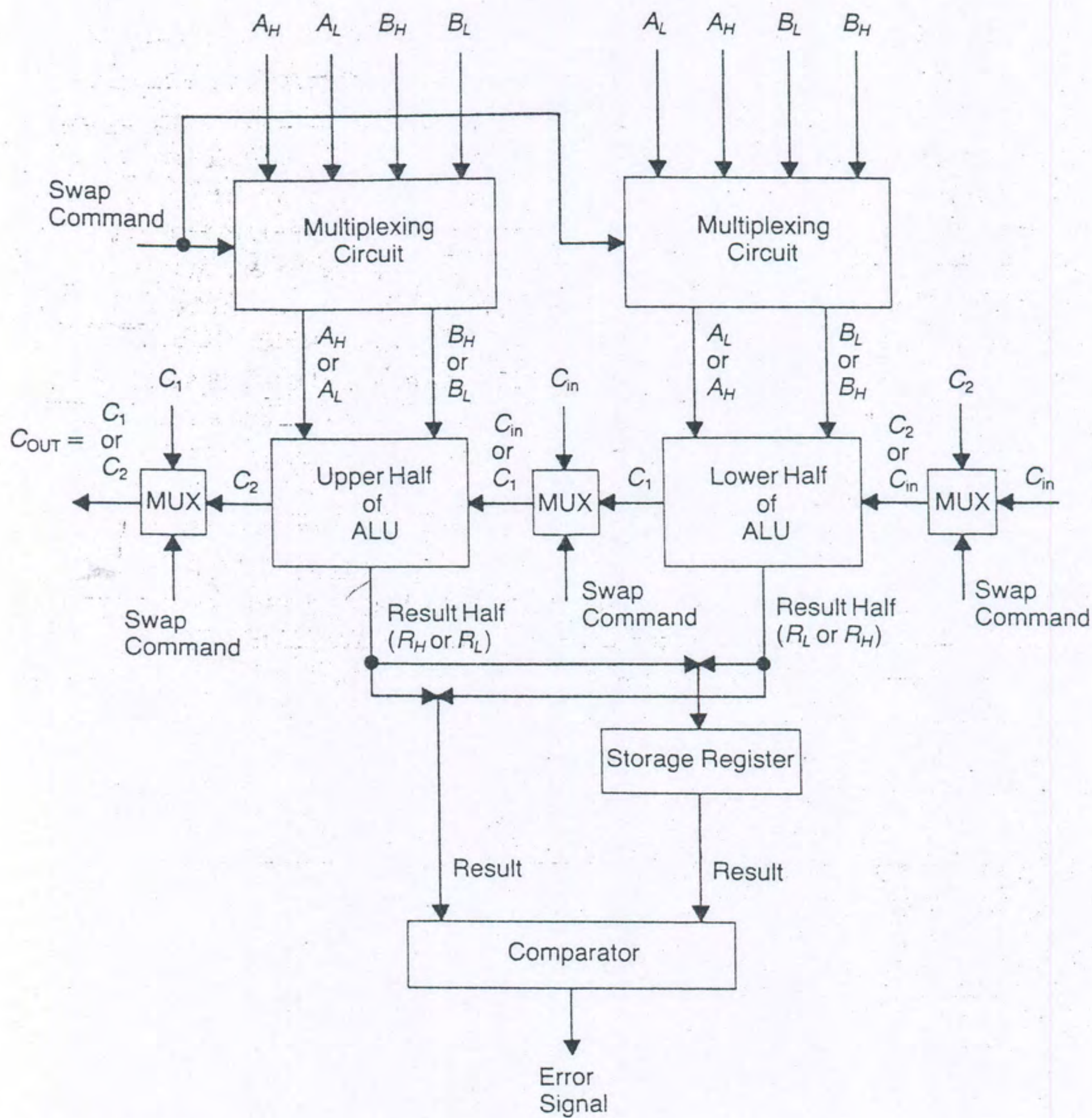


Fig. 3.70 ALU structure using RESWO.

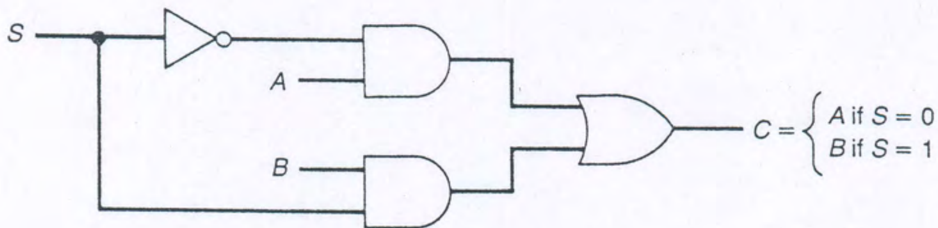


Fig. 3.71 2-to-1 multiplexer to perform single-bit swapping.

To best describe the basis of REDWC, the operation of an n-bit full-adder is shown as an example. The n-bit adder, as well as the operands, is divided into lower and upper halves with each half consisting of $n/2$ bits. To perform an addition, two calculations are executed. In the first calculation, the lower halves of the operands are added in both halves of the adder. In other words, the adder performs the addition of the lower halves of the operands twice but in parallel on the upper and lower halves of the adder, as seen in Fig. 3.72. The results are then compared and one result is stored to represent the lower half of the final output. The adder hardware is then used a second time to perform a second calculation. The second calculation starts by selecting the upper halves of the operands with the output carry of the first computation to be added in both halves of the adder. The results of the second calculation are then compared and are available as the upper half of the final output.

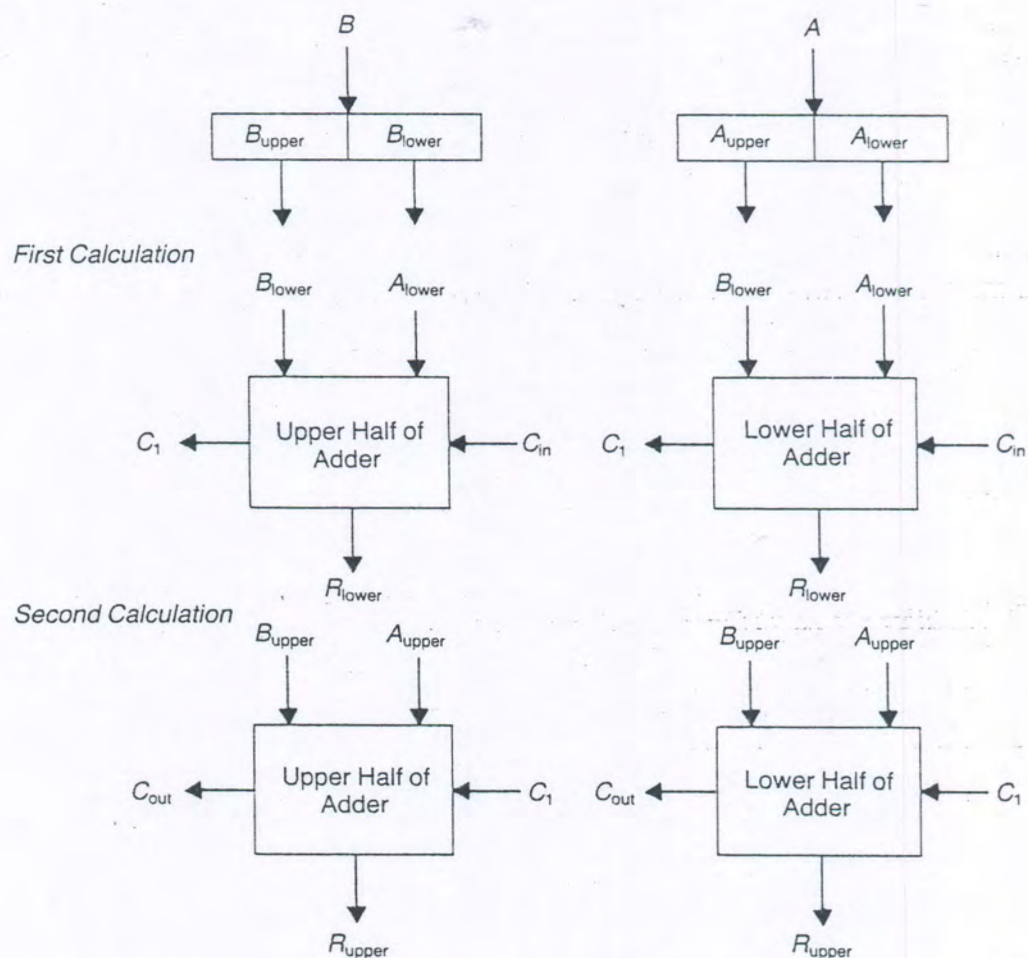


Fig. 3.72 REDWC uses both the upper and lower halves of the adder to perform identical calculations. The adder hardware is used twice to complete the final result.

Selection of the appropriate operand half is performed using multiplexers (MUXs). Also, the carries at the boundaries of the adder are handled using one MUX. Figure 3.73 shows a block diagram of a 32-bit adder that uses the REDWC technique for error detection.

The checking for errors is performed during each calculation by comparing the outputs of the upper and lower halves of the adder. Although error detection is accomplished by a simple comparison between the results

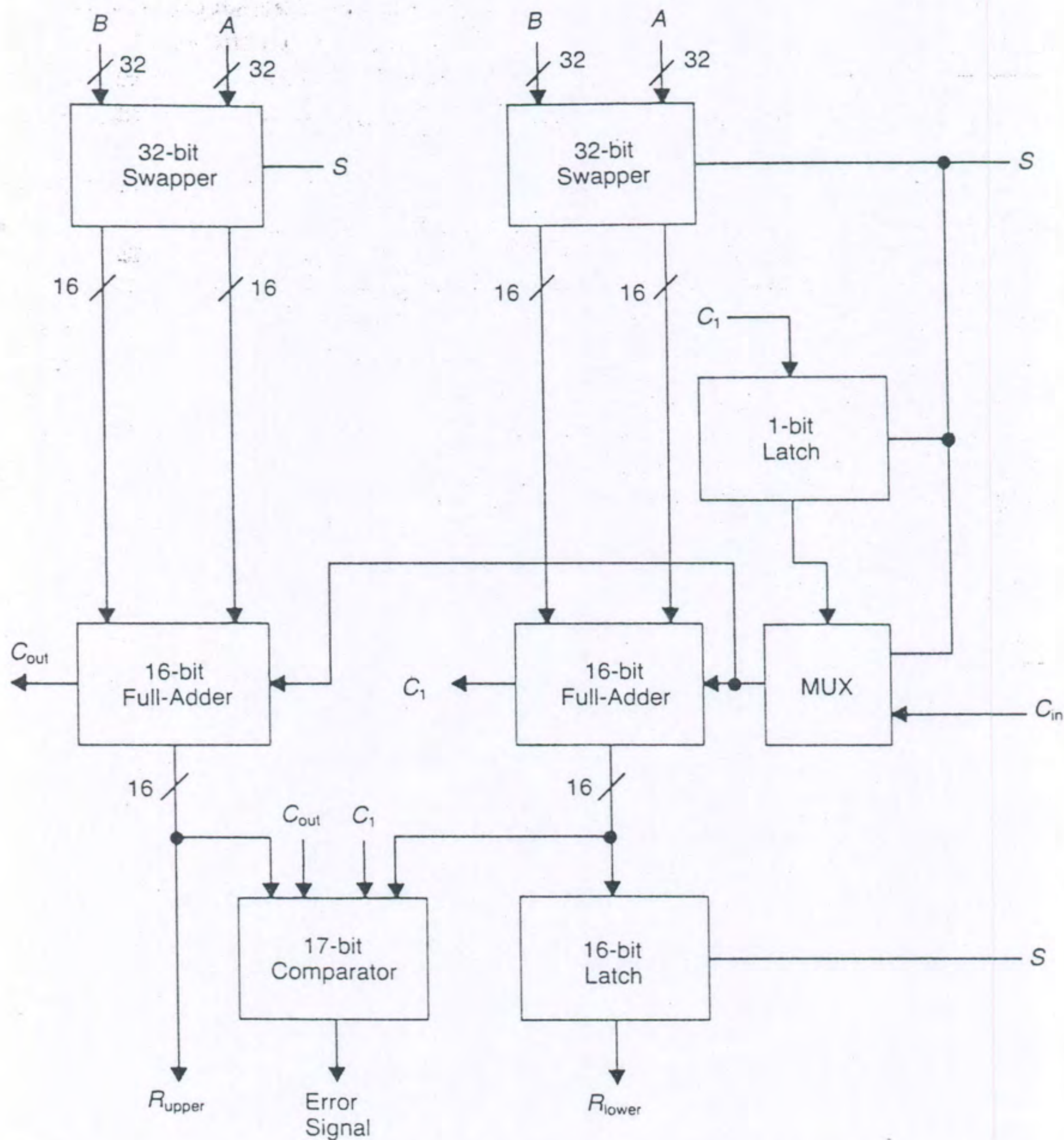


Fig. 3.73 A 32-bit adder designed using the REDWC technique.

of the two halves, the same hardware is used twice to obtain the final results. Therefore, the concept of duplication with comparison is employed as the error detection mechanism, and the concept of time redundancy is applied to complete the calculation and obtain the final output.

Since the error detection mechanism of REDWC is identical to that of duplication with comparison, both techniques possess the same ability to detect faults; that is, the ability to detect all single faults confined to one half of the adder in arithmetic and logic operations, as long as both halves do not become faulty in a similar manner and at the same time. REDWC is also capable of detecting errors in lookahead-carry adders provided that the lookahead-carry operation does not overlap the boundary between the two halves of the adder. In other words, the carry between the two halves must ripple from the lower to the upper half of the adder, but within each half of the adder, lookahead-carry operations can be performed.

The 32-bit adder, shown in Fig. 3.73, illustrates the structure required to implement REDWC. The adder consists of two "swappers," the adder itself, a multiplexer (MUX), a storage register, and a comparator. Each swapper consists of 32 2-1 MUXs that select the appropriate operand half. A control signal S is provided to manipulate the MUXs and, consequently, control the operation of the adder. When S is low, the lower halves are selected to be added in both halves of the adder; and when S is high, the upper halves are selected. Each half of the adder consists of four 4-bit lookahead-carry adder cells. Each cell possesses a lookahead-carry capability, but the carry is rippled between each cell, as shown in Fig. 3.74. The

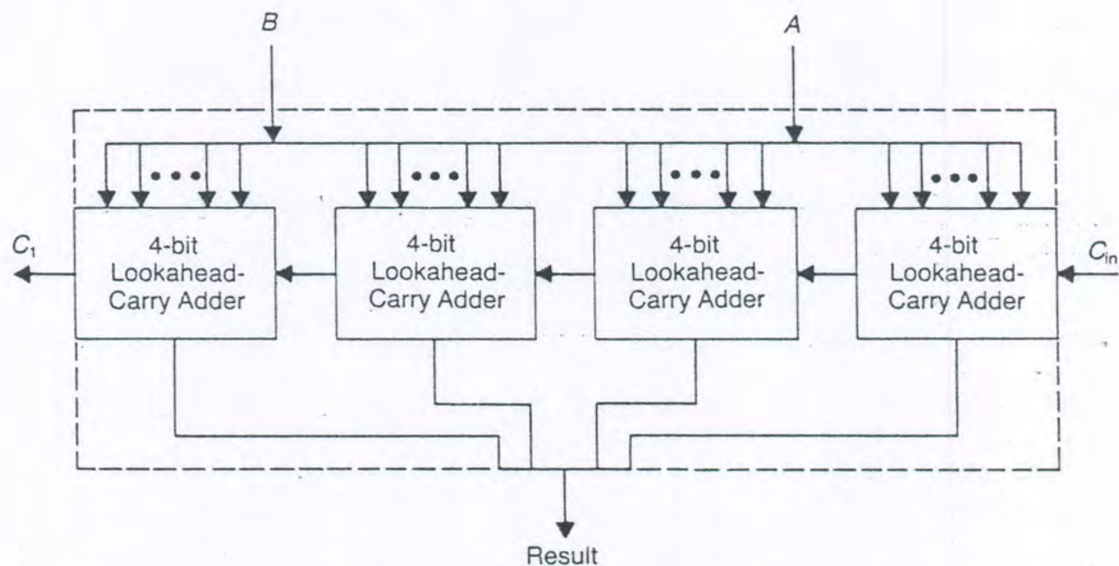


Fig. 3.74 A 16-bit adder constructed from four 4-bit lookahead carry units—the carry bit ripples between lookahead units.

TABLE 3.21 Best-case and worst-case cell counts for three error detection techniques compared to the nonredundant case

Technique	Number of cells		Average number of cells	Percentage increase in average number of cells
	Best case	Worst case		
Nonredundant	253.33	320	286.67	0%
Duplication with comparison	542.33	685	613.67	114
RESO	665.00	876	770.50	169
REDWC	446.67	582	514.34	79

output carry of the lower half (C1) is stored in a 1-bit latch to be used as the input carry for the second calculation. The 2-1 MUX is used to appropriately handle the carry at the adder boundaries.

Table 3.21 shows the best-case and the worst-case cell count, as well as the average cell count for a gate array implementation of adders designed using duplication with comparison (DWC), RESO, REDWC, and also an adder designed without redundancy [Johnson, Aylor, and Hana 1988]. The best-case cell count corresponds to optimal wire routing conditions in the VLSI implementation, whereas the worst-case cell count corresponds to the worst possible routing conditions. The average number of cells is used as a means of comparison to eliminate routing considerations. Note that the REDWC technique uses significantly fewer cells than either duplication with comparison or RESO. Also note that all of the time redundancy approaches require significant amounts of hardware redundancy.

TABLE 3.22 Calculation times for adders with concurrent error detection compared to the nonredundant case

Technique	Calculation time (nanoseconds)	Percentage increase
Nonredundant	37.5	0.00%
Duplication with comparison	48.3	28.80
RESO	83.5	122.67
REDWC	52.6	40.27

An equally important criterion for comparison is the resulting system throughput. Table 3.22 summarizes simulation results for the nonredundant, RESO, DWC, and REDWC adders [Johnson, Aylor, and Hana 1988]. The table shows the amount of time (in nanoseconds) required to complete a calculation (including error detection), as well as the percentage increase in calculation time over that of the nonredundant adder. The calculation time represents the maximum time between the application of the input carry to the adder and the availability of the error signal on the output of the comparator. The comparators used for the various adders have the same 10.8 nanoseconds of propagation delay. The result of the addition is actually available 10.8 nanoseconds earlier than the times shown in Table 3.22, however, the error signal is not available until approximately 10.8 nanoseconds after the addition is completed.

From Table 3.22, one can see that the penalty, in time, paid by using REDWC for error detection is considerably less than that of RESO and comparable to that of DWC. Although time redundancy is part of the REDWC technique, the extra time required to complete the calculation is only 40.27% over that of the nonredundant operation.

Time redundancy techniques form an important class of options for designing fault-tolerant systems. Just as all redundancy approaches, however, time redundancy cannot be used in all applications because of the additional time that must be employed. If time is available, however, time redundancy techniques do provide an opportunity to minimize the amount of additional hardware required.

3.6.3 Recomputation for Error Correction

The time redundancy approach can also provide for error correction if the computations are repeated three or more times. Consider the logical AND operation that was illustrated in Fig. 3.67. Suppose the operation is performed three times: first, without shifting the operands; second, with a 1-bit arithmetic shift of the operands; and third, with a 2-bit arithmetic shift of the operands. The results generated using the shifted operands are then shifted the appropriate number of bits to the right to properly position the bits of the results. Because each of the three operations used operands that were displaced from each other by at least one bit position, a different bit in each result will be affected by the faulty bit slice. If the bits in each position are then compared, the results due to the faulty bit slice can be corrected by performing a majority vote on the three results obtained for each bit position. This process is illustrated in Fig. 3.75.

Unfortunately, the approach just described does not work for arithmetic operations because the adjacent bits are not independent. A single, faulty bit slice can affect more than one bit of the result.

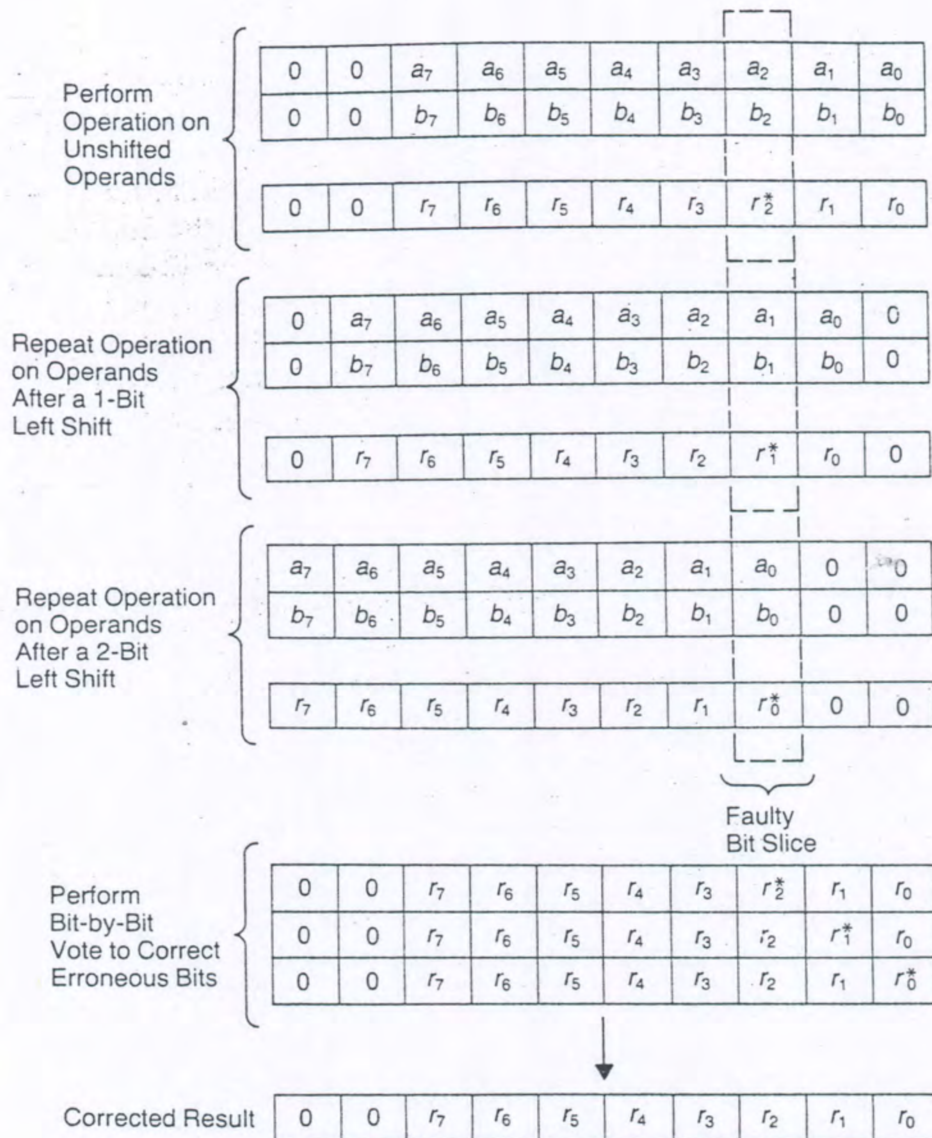


Fig. 3.75 Example of error correction using time redundancy.

3.7 Software Redundancy

In applications that use computers, many fault detection and fault tolerance techniques can be implemented in software. The redundant hardware necessary to implement the capabilities can be minimal, whereas the redundant software can be substantial. Redundant software can occur in many forms; you do not have to replicate complete programs to have redundant software. Software redundancy can appear as several extra lines of code used to check the magnitude of a signal or as a small routine used to periodically test a memory by writing and reading specific locations. In this

section, we consider three major software redundancy techniques: consistency checks, capability checks, and software replication methods [Chen and Avizienis 1978].

3.7.1 Consistency Checks

A **consistency check** uses *a priori* knowledge about the characteristics of information to verify the correctness of that information. For example, in some applications, it is known in advance that a digital quantity should never exceed a certain magnitude. If the signal exceeds that magnitude, an error of some sort is present. A consistency check can often be implemented easily in hardware, but it is most likely to appear in a system's software. For example, a processing system can sample and store many sensor readings in a typical control application. Each sensor reading can be checked to verify that it lies within an acceptable range of values. As another example, the amount of cash requested by a patron at a bank's teller machine should never exceed the maximum withdrawal allowed. Likewise, the address generated by a computer should never lie outside the address range of the available memory.

An example of consistency checking that can be performed in hardware is the detection of invalid instruction codes in computers. Many computers use n-bit quantities to represent 2^k possible instruction codes where $2^k < 2^n$. In other words, there are $2^n - 2^k$ instruction codes that are illegal. Each instruction code can be checked to verify that it is not one of the illegal codes. If an illegal code occurs, the processor can be halted to prevent an erroneous operation from occurring. This technique is particularly useful in detecting a "run-away" processor that is erroneously interpreting data as instructions.

Another form of consistency checking that can prove valuable in many control applications is to compare the measured performance of the system with some predicted performance. This technique is particularly useful in control applications where some dynamic system is under control. The dynamic system can be modeled and the predicted performance obtained from a software implementation of the model. The actual performance of the system can then be measured and compared with the model-predicted performance. Any significant deviations of the measured performance from the predicted performance can indicate a fault. The difficulty with this approach is twofold. First, the model must be accurate if good results are to be obtained. Second, it is difficult to establish the level of deviation that will be allowed before an error is signaled. In some applications, the nonlinearity of a system can result in a linearized model deviating substantially from the actual performance under certain input conditions.

Another form of consistency check that is very useful in systems where data is transferred over buses is the word count overflow check. The data is

often transferred in packets, with each packet possessing a specific number of data words. At the beginning of each packet can be a header that states the number of words in that packet. If the receiving device detects a disagreement between the actual number of words received and the number specified in the header, an error can be indicated. The process of counting the words and comparing that count with the received count can be implemented in software in many applications; thus, the redundancy occurs in software.

3.7.2 Capability Checks

Capability checks are performed to verify that a system possesses the capability expected. For example, you would like to know whether or not you have your complete memory available or if all the processors in your multiprocessor system are working properly. As another example, you might want to know if the ALU in your processor is working properly.

Several forms of capability checks exist. The first is a simple memory test. A processor can simply write specific patterns to certain memory locations and read those locations to verify that the data was stored and retrieved properly. In many cases, it is not necessary to write and read a large number of locations to achieve reasonably good fault coverage. The memory test can be a supplement to parity as protection against faults in the memory.

Another form of capability check is a set of ALU tests. Periodically, a processor can execute specific instructions on specific data and compare the results to known results stored in a read-only memory (ROM). This form of capability check can verify both the ALU and the memory in which the known results are stored. The instructions that are executed can consist of adds, multiplies, logical operations, and data transfers.

Another form of capability check consists of verifying that all processors in a multiple processor system are capable of communicating with each other. This can consist of periodically passing specific information from one processor to another. For example, each processor may be required to set a specific bit in a shared memory to indicate their capability to communicate with that memory, and, as a result, communicate with other processors through that memory.

3.7.3 *N*-Version Programming

The software redundancy techniques that we have considered thus far have been those that use extra, or redundant, software to detect faults that can occur in hardware. We have not yet considered approaches for detecting, or possibly tolerating, faults that can occur in the software of a system. Software faults are unusual entities. Software does not break as hardware does,

but instead software faults are the result of incorrect software designs or coding mistakes. Therefore, any technique that detects faults in software must detect design flaws. A simple duplication and comparison procedure will not detect software faults if the duplicated software modules are identical, because the design mistakes will appear in both modules.

The concept of **N-version programming** was developed to allow certain design flaws in software modules to be detected [Chen and Avizienis 1978]. The basic concept of N-version programming is to design and code the software module N times and to compare the N results produced by these modules. Each of the N modules is designed and coded by a separate group of programmers. Each group designs the software from the same set of specifications such that each of the N modules performs the same function. However, it is hoped that by performing the N designs independently, the same mistakes will not be made by the different groups. Therefore, when a fault occurs, the fault either does not occur in all modules or it occurs differently in each module, so that the results generated by the modules will differ.

Certainly, the importance of software fault tolerance is easy to see. If we design a microprocessor-based system to be fault tolerant using a TMR configuration, the hardware redundancy is of little use if a single software fault can disable each of the redundant processors. The N-version programming technique states that each of the three processor's software should be designed and coded independently such that a common fault is less likely to occur.

The primary difficulties with the N-version approach are twofold. First, software designers and coders can tend to make similar mistakes. Therefore, we are not guaranteed that two completely independent versions of a program will not have identical faults. Second, the N versions of a program are still developed from a common specification, so the N-version approach will not allow the detection of specification mistakes.

To overcome many of the problems associated with N-version programming, software designers employ rigid design rules and methods to attempt to prevent faults from occurring. This approach we know as fault avoidance, and it is very important in the design of reliable software. If the software is designed correctly in the first place, fault tolerance techniques for the software will not be necessary.

Summary

This chapter has presented the basic techniques for achieving fault tolerance; hardware redundancy, information redundancy, time redundancy, and software redundancy. In most cases, the various techniques have been illustrated with examples of their actual use. In many applications, a com-

bination of the techniques must be used to meet the requirements of reliability and fault tolerance. Subsequent chapters illustrate the design and evaluation of fault-tolerant systems using the techniques presented in this chapter. The following list summarizes the important concepts introduced in this chapter.

Active Hardware Redundancy—techniques that achieve fault tolerance through the use of fault detection and reconfiguration.

Alternating Logic—a time redundancy technique in which a calculation is repeated after complementing the original data.

AN Code—an arithmetic code in which code words are created by multiplying the original data N by a constant A.

Arithmetic Code—a code in which code words are invariant to arithmetic operations.

Berger Code—a separable code formed by concatenating the original data and the complement of the binary word representing the number of 1s in the original data.

Binary Code—a code in which code words contain only symbols that are either 0 or 1.

Bit-per-Byte Parity—a parity code in which one parity bit is assigned to each of two groups of $n/2$ bits in an n-bit word.

Bit-per-Chip Parity—a parity code in which one parity bit is assigned to each chip in a memory design.

Bit-per-Multiple-Chips Parity—a parity code in which each parity group has one, and only one, bit from each chip.

Bit-per-Word Parity—a parity code in which one parity bit is assigned to each n-bit word.

Burst Error—a condition in which consecutive bits of a code word are corrupted by a single fault.

Capability Check—a verification of an expected capability or feature.

Checksum—the sum of a list of data items.

Code—a specific set of rules for representing information.

Code Distance—the minimum Hamming distance between any two valid code words within a given code.

Code Polynomial—a polynomial of degree $n - 1$ whose coefficients are the n bits of a code word.

Code Word—a collection of symbols, or digits, use to represent information according to the rules of a given code.

Cold Standby Sparing—the use of unpowered standby spares.

Complemented Duplication—a code in which data is duplicated but in a complemented form.

Consistency Check—a check for consistency between actual results and expected results.

Cyclic Code—a code in which code words are invariant to end-around shifts.

Decoding Process—the process of determining a data word from a code word.

Double-Precision Checksum—a code in which the checksum is formed using double-precision arithmetic.

Duplication Codes—a code in which data is completely and exactly duplicated.

Duplication with Comparison—the use of two identical modules performing identical computations and the comparison of their results.

Encoding Process—the process of determining a code word from an original data word.

Error Detecting Code—a code that allows errors to be detected.

Error Correcting Code—a code that allows errors to be corrected.

Flux-Summing—an alternative to majority voting that uses the inherent properties of feedback systems to compensate for faults in replicated modules.

Generator Polynomial—a polynomial $G(X)$ used to create a cyclic code word $V(X)$ by forming $V(X) = D(X)G(X)$, where $D(X)$ is the original data polynomial.

Hamming Distance—the number of bit positions in which two binary code words differ.

Hamming Error-Correcting Codes—a class of parity codes that allows error correction through the use of overlapping parity groups.

Hardware Redundancy—the addition of redundant hardware to a system.

Honeywell Checksum—a code in which the checksum is calculated after concatenating consecutive words to form double words.

Hot Standby Sparing—the use of powered standby spares.

Hybrid Hardware Redundancy—techniques that use a combination of fault masking and reconfiguration.

Information Redundancy—the addition of redundant information to a data item.

Interlaced Parity—a parity code in which adjacent bits are assigned to different parity groups.

Inverse Residue Code—a code in which code words are created by appending the inverse residue to the original data.

***m*-of-*n* Code**—a code in which each *n*-bit code word has exactly *m* 1s.

Mid-Value Select—a form of voting that selects the middle value from amongst an odd number of signals.

N Modular Redundancy—a form of passive redundancy that uses *N* modules and majority voting.

N Modular Redundancy with Spares—a hybrid technique that combines *N* modular redundancy with standby sparing.

Nonseparable Code—a code in which the original data cannot be separated from the check bits.

***N*-version Progamming**—a technique that compares the results from *N* separate programs, each of which performs the same operations.

Overlapped Parity—a parity code in which each data bit belongs to more than one parity group.

Pair-and-a-Spare Technique—the combination of duplication with comparison and standby sparing.

Parity Code—a code in which all code words have an even (odd) number of 1s. If the number of 1s is even, the code is called even parity. If the number of 1s is odd, the code is called odd parity.

Passive Hardware Redundancy—techniques that achieve fault tolerance through the use of fault masking.

Recomputing with Duplication with Comparison (REDWC)—a redundancy technique that combines time redundancy and duplication with comparison.

Recomputing with Shifted Operands (RESO)—a time redundancy technique in which a calculation is repeated after shifting the operands.

Recomputing with Swapped Operands (RESWO)—a time redundancy technique in which a calculation is repeated after swapping the upper and lower halves of each operand.

Redundancy—the addition of information, resources, or time beyond what is needed for normal system operation.

Repeated Use Faults—a fault within a hardware or software unit used multiple times to create a single result.

Residue Checksum—a code in which the checksum is calculated using end-around carry addition.

Residue Code—a code in which code words are created by appending the residue to the original data.

Residue Number System—a system in which numbers are represented using multiple residues.

Restoring Organ—a unit that creates a correct output even though one of its inputs is incorrect.

Self-Purging Redundancy—a hybrid technique where faulty modules remove themselves from the voting arrangement.

Separable Code—a code formed by appending check bits to the original data.

Sift-Out Modular Redundancy—a hybrid technique that uses a centralized process to remove faulty units from a voting arrangement.

Single point of failure—a single component or element whose failure results in the failure of the system.

Single-Precision Checksum—a code in which the checksum is formed using single-precision arithmetic.

Software Redundancy—the addition of redundant software to a system.

Standby Sparing—the use of spare units that replace faulty ones during a reconfiguration process.

Swap-and-Compare Code—a code formed by duplicating the data and swapping the upper and lower halves of the duplicated data.

Syndrome Polynomial—the remainder polynomial found by dividing a cyclic code polynomial by the generator polynomial.

Time Redundancy—the use of redundant time in a system's operations.

Triple-Duplex Architecture—triple modular redundancy where each triplicated unit uses duplication with comparison.

Triple Modular Redundancy (TMR)—a form of passive redundancy that triplicates modules and uses majority voting.

Watchdog Timer—a timer that if not set at a specific frequency is used to indicate faulty behavior.

References

1. Avizienis, A. "Arithmetic error codes: Cost and effectiveness studies for application in digital system design," *IEEE Transactions on Computers*, Vol. C-20, No. 11, November 1971, pp. 1322–1331.
2. Chen, L., and A. Avizienis. "N-version programming: A fault tolerant approach to reliability of software operation," *Proceedings of the International Symposium on Fault Tolerant Computing*, 1978, pp. 3–9.
3. de Sousa, P.T., and F.P. Mathur. "Sift-out modular redundancy," *IEEE Transac-*

tions on Computers, Vol. C-27, No. 7, July 1978, pp. 624–627.

4. Hamming, R. W. "Error detecting and error correcting codes," *Bell System Technical Journal*, Vol. 26, No. 2, April 1950, pp. 147–160.
5. Hana, H. H., and B. W. Johnson. "Concurrent error detection in VLSI circuits using time redundancy," *Proceedings of SOUTHEASTCON '86*, Richmond, Va., March 23–25, 1986, pp. 208–212.
6. *Intel Memory Components Handbook*, Intel Corporation, Santa Clara, Calif., 1985.
7. Johnson, B. W. "Fault-tolerant microprocessor-based systems," *IEEE Micro*, Vol. 4, No. 6, December 1984, pp. 6–21.
8. Johnson, B. W., J. H. Aylor, and H. H. Hana. "Efficient use of time and hardware redundancy for concurrent error detection in a 32-bit VLSI adder," *IEEE Journal of Solid-State Circuits*, Vol. 23, No. 1, February 1988, pp. 208–215.
9. Kohavi, Z. *Switching and Finite Automata Theory*, McGraw-Hill, New York, 1978.
10. Lala, P. K. *Fault Tolerant and Fault Testable Hardware Design*, Prentice-Hall, Englewood Cliffs, N.J., 1985.
11. Lin, S., and D. J. Costello, Jr. *Error Control Coding: Fundamentals and Applications*, Prentice-Hall, Englewood Cliffs, N.J., 1983.
12. Losq, J. "A highly efficient redundancy scheme: Self-purging redundancy," *IEEE Transactions on Computers*, Vol. C-25, No. 6, June 1976, pp. 569–578.
13. Nelson, V. P., and B. D. Carroll. *Tutorial: Fault-Tolerant Computing*, IEEE Computer Society Press, Washington, D.C., 1986.
14. Patel, J. H., and L. Y. Fung. "Concurrent error detection in ALUs by recomputing with shifted operands," *IEEE Transactions on Computers*, Vol. C-31, No. 7, July 1982, pp. 589–595.
15. Reynolds, D. A., and G. Metze. "Fault detection capabilities of alternating logic," *IEEE Transactions on Computers*, Vol. C-27, No. 12, December 1978, pp. 1093–1098.
16. Tang, D. T., and R. T. Chien. "Coding for error control," *IBM Systems Journal*, Vol. 8, No. 1, January 1969, pp. 48–86.
17. Taylor, F. J. "Residue arithmetic: A tutorial with examples," *Computer*, Vol. 17, No. 5, May 1984, pp. 50–62.
18. Toy, W., N., "Self-checking arithmetic unit," United States Patent Number 4, 314, 350, Bell Telephone Laboratories, Murray Hill, N.J., Feb. 2, 1982.

Additional Reading

While this chapter has attempted to cover many of the most important techniques in use in today's fault-tolerant designs, the literature contains other approaches that are beyond the intent of this book. For the reader

who is interested in studying further, the following list of articles is provided as suggested reading.

Abdelaziz, M., and A.D. Friedman. "Efficient design of self-checking checker for any m-out-of-n code," *IEEE Transactions on Computers*, Vol. C-27, No. 6, June 1978.

Armstrong, D.B. "A general method of applying error correction to synchronous digital machines," *Bell System Technical Journal*, Vol. 40, No. 3, March 1961.

Armstrong, J.R. and F.G. Gray, "Fault Diagnosis in a Boolean *n* Cube Array of Microprocessors" , *IEEE Transactions on Computers*, Vol. C-30, No. 8, August 1981.

Clark, E.M., and C.N. Nickolaous. "Distributed reconfiguration strategies for fault tolerant microprocessor systems," *IEEE Transactions on Computers*, Vol. C-31, No. 8, August 1982.

Coyne, R. "Dynamic reconfiguration by a local network's operating system," *Data Communications*, December 1981.

Dishon, Y., and C.J. Georgiou. "A highly available storage system using the checksum method," *Proceedings of the Seventeenth International Symposium on Fault-Tolerant Computing*, July 6–8, 1987, Pittsburgh, Pa.

Gannon, T.G., and S.D. Shapiro. "An optimal approach to fault tolerant software systems design," *IEEE Transactions on Software Engineering*, Vol. SE-4, No. 5, September 1978.

Goodenough, J.B., and C.L. McGowan. "Software quality assurance: Testing and validation," *Proceedings of the IEEE*, Vol. 68, No. 9, September 1980.

Hamill, T.G., and R. Phillips. "A fault tolerant reconfigurable multiprocessor system," *Proceedings of the International Conference on Distributed Computer Control*, Birmingham, England, September 1977.

Jarwala, N., and D.K. Pradhan. "Cost analysis of on chip error control coding for fault tolerant dynamic RAMs," *Proceedings of the Seventeenth International Symposium on Fault-Tolerant Computing*, July 6–8, 1987, Pittsburgh, Pa.

Jensen, P.A. "Quadded NOR logic," *IEEE Transactions on Reliability*, Vol. R-12, No. 3, September 1963.

Kameyama, M., and T. Higuchi. "Design of dependent failure tolerant microcomputer system using triple modular redundancy," *IEEE Transactions on Computers*, Vol. C-29, No. 2, February 1980.

Kinney, L.L., and W.Y. Yueh. "An architecture for a VHSIC computer," *Proceedings of the 8th Annual Symposium on Computer Architecture*, May 1981, Minneapolis, Minn.

Klaschka, T.F. "Reliability improvement by redundancy in electronic systems, II: An efficient new redundancy scheme—Radial logic," *Royal Aircraft Establishment*, Ministry of Technology, 69045, Farnborough, United Kingdom, 1969.

Koren, I. "A reconfigurable and fault tolerant VLSI multiprocessor array," *Proceedings of the 8th Annual Symposium on Computer Architecture*, May 1981, Minneapolis, Minn.

Lin, S. *An Introduction to Error Control Coding*, Prentice-Hall, Englewood Cliffs, N.J., 1970.

Petersen, W.W., and E.J. Weldon, Jr. *Error Correcting Codes*, Second Edition, MIT Press, Cambridge, Mass., 1972.

Pierce, W.H. *Fault Tolerant Design*, Academic Press, Orlando, Fla., 1965.

Rao, T.R. N. *Error Coding for Arithmetic Processors*, Academic Press, Orlando, Fla., 1974.

Stiffler, J.J. "Coding for random access memories," *IEEE Transactions on Computers*, Vol. C-27, No. 6, June 1978.

Taylor, D.J., D.E. Morgan, and J.P. Black. "Redundancy in data structures: Improving software fault tolerance," *IEEE Transactions on Software Engineering*, Vol. SE-6, No. 6, November 1980.

Tryon, J.G. "Quadded logic," in *Redundancy Techniques for Computers*, edited by Wilcox and Mann, Spartan Books, Washington, D. C., 1962.

Wakerly, J.F. *Error Detecting Codes, Self-Checking Circuits, and Applications*, Elsivier North-Holland, New York, 1978.

Wakerly, J.F. "Transient failures in triple modular redundancy," *IEEE Transactions on Computers*, Vol. C-24, No. 5, May 1975.

Weinstock, C.B., and M.W. Green. "Reconfigurable strategies for the SIFT fault tolerant computer," *Proceedings of Compsac 78*, Chicago, November 1978.

Wright, R.S., S.C. Crist, and M. Arozullah. "Fault tolerant techniques for a multiple microprocessor-based space borne packet switch," *Proceedings of the National Telecommunications Conference*, New Orleans, November 1981.

Problems

3.1. Most helicopters use a stabilizer to level the aircraft during flight so that the air drag is significantly reduced. Typically, the stabilizer is moved by opening a hydraulic valve, and movement is stopped when the valve is closed. The valve is controlled by a digital circuit that issues a 1-bit command to a relay that controls the valve. The command originates from triply redundant switches activated by the pilot. Design a TMR circuit to generate the correct command after any single fault. The voter in your circuit will clearly be the weak point. Make sure that your design can at least detect faults that occur within the voter.

3.2. The company that you work for is designing an industrial controller that maintains the temperature of a process during a chemical reaction. The non-

redundant controller is fairly simple and consists of an analog-to-digital (A/D) converter, processor, and a digital-to-analog (D/A) converter. You have been asked to develop at least two alternatives for making the controller tolerant of any two component failures. (The term *component* here means an A/D, processor, or D/A.) Show block diagrams of your approaches and compare them. Which approach would you recommend and why?

3.3. A communications link between two processors consists of transmitters, physical media, receivers, buffers, and a memory, as shown in Fig. 3.76. The buffers are first-in, first-out (FIFO) devices and are constructed using chips that are 4-bits wide. The communications link is 40 bits wide, so there are ten such chips in parallel to accommodate the 40 bits. Parity is to be used as a means of error detection with the parity being generated prior to transmission and checked as soon as the data exits the FIFOs. If only 8 of the 40 bits are available for parity bits, devise a parity organization that provides what you feel is the best protection. Justify your solution. One last constraint is that the data is transmitted over the bus at the rate of one word every 10 nanoseconds. Assuming that a two-input logic gate has a propagation delay of 0.5 nanoseconds, make sure that your approach is feasible from a time standpoint.

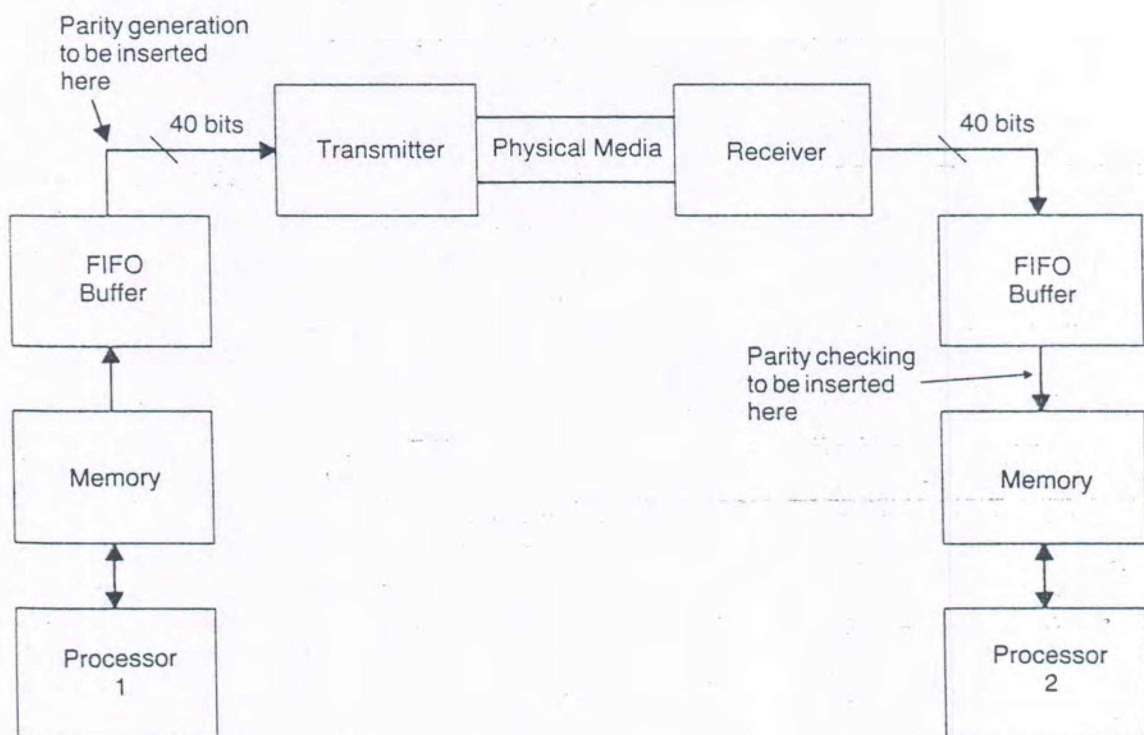


Fig. 3.76 Architecture of system for Problem 3.3.

3.4. As a new employee, you have been assigned to the Fault-Tolerant Systems Branch of your company. Your first assignment has been to design a fault-

tolerant network for interconnecting multiple processors. The processors form the core of a transactions processing system for a bank with several branches. Each branch is a node in the network. The basic structure of the nonredundant network is that of a unidirectional ring, as shown in Fig. 3.77. Data is transferred from one node to another around the ring in packets of 20 16-bit words. Develop an architecture for the ring that allows any node failure to be toler-

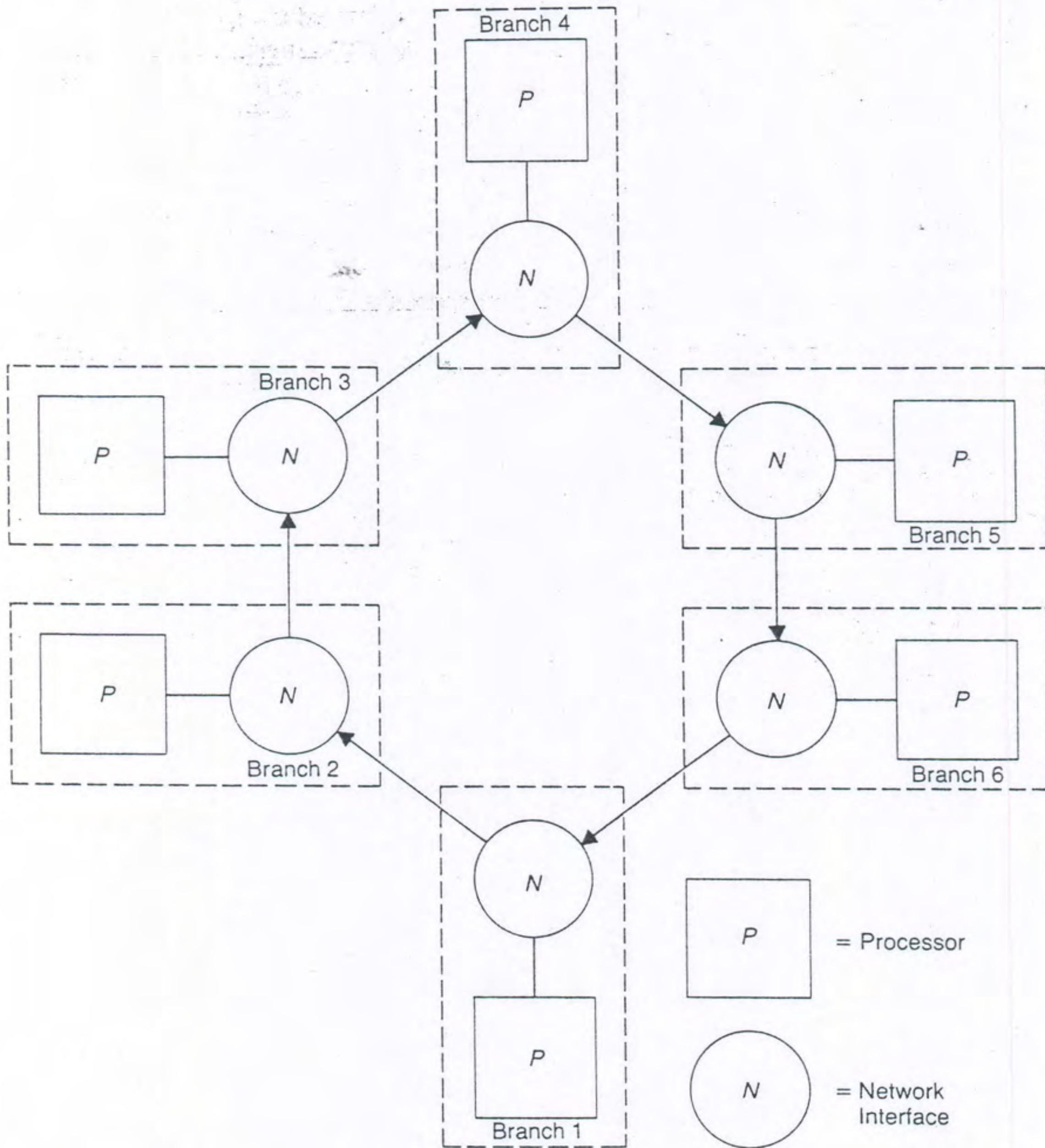


Fig. 3.77 Architecture of the bank transactions processing system for Problem 3.4. The nonredundant network is a unidirectional ring.

ated. Also, describe the techniques that you would use to detect errors in the data as it is transferred around the ring.

3.5. Show the organization of an 8-bit memory with Hamming single-error correction and double-error detection. Be explicit in your descriptions and show the parity groups that result. Associate the syndromes with the particular bit that they identify as erroneous.

3.6. Design a simple, nonredundant circuit to detect erroneous codes in a 4-bit binary coded decimal (BCD) representation. The circuit should produce an output of 1 when the four bits at the inputs do not represent a valid BCD code. Now show the implementation of a self-purging version of that circuit that is capable of tolerating any two faults that can occur.

3.7. Show the sift-out modular redundancy implementation of the combinational circuit of Problem 3.6. Once again, the circuit should tolerate any two faults that occur.

3.8. Show the separable and nonseparable cyclic code words that result for 4-bit information words when the generator polynomial is $G(X) = 1 + X + X^4$. Also, develop a circuit that is capable of encoding the original information and a second circuit that is capable of decoding the code words.

3.9. Show the separable residue and the separable inverse residue codes for the 4-bit, BCD representations of the numbers 0 through 9. Use a modulus of 5.

3.10. Using the Intel 8206 error detection and correction unit, design a 16-bit memory capable of correcting any single error and detecting any double error. Assume that you are using any standard memory chip that is four bits wide. Show the complete block diagram of your memory.

3.11. Investigate the error detection capability of the following time redundancy approach when used on a ripple-carry adder. During the first addition, the operands are encoded using a $3N$ arithmetic code. During the second addition, the operands are encoded using a $5N$ arithmetic code. Will this scheme detect any single error that can occur in the adder? Will the approach detect any double error that can occur in the adder?

3.12. Design an 8-bit adder using the RESWO technique. Make sure that you provide the ability to swap the operands and to compare the results that are computed at the two different points in time.

3.13. Prove or disprove that the RESWO technique is capable of detecting any single error that can occur in a lookahead-carry adder. Assume for simplicity that the adder is eight bits and the lookahead operation is performed over four bits.

3.14. A digital filtering application requires the implementation of the equation, $y((n+1)T) = Ay(nT) + Bu(nT)$, where A and B are constants, $u(nT)$ is the input at time nT, and $y(nT)$ is the output of the filter at time nT. The input is limited, because of the analog-to-digital converter, to an 8-bit representation. Your design implements the filter in software using a 16-bit processor. Design a simple processing system, including the structure of the software, to provide as much error detection capability as possible without using more than one of any of the hardware components. You may resort to time, information, or software redundancy but not hardware redundancy.

3.15. Figure 3.78 shows the architecture of a simple microprocessor, which has four fundamental components: the arithmetic logic unit (ALU), memory, control logic, and an internal bus. The processor is organized as an 8-bit machine, so each of the ALU, memory, and bus is an 8-bit unit. For the purpose of this design, assume that the control logic is simply a memory capable of storing 16 4-bit words. To simplify matters even further, assume that the ALU can only perform addition. The bus is an 8-bit, parallel bus. Finally, the memory can store up to 32 8-bit words. Develop a design that incorporates redundancy into the various units to achieve fault detection. Your design should be capable of detecting all single faults. You may use any type of redundancy, but you should show an analysis of the extra hardware, time, information, or software that your design requires. Explain, in detail, why you selected your specific approach.

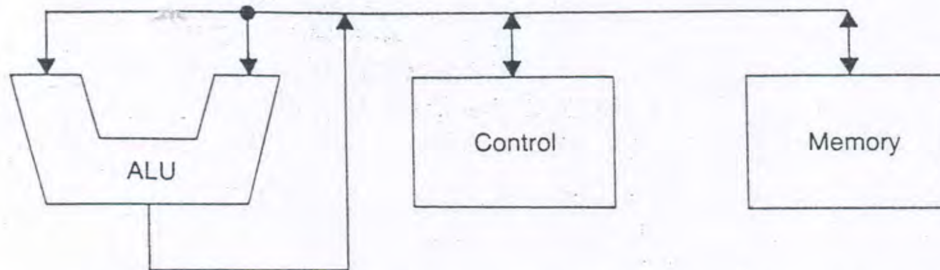


Fig. 3.78 Architecture of a simple microprocessor to be studied in Problem 3.15.

3.16. A common interconnection technique is a crossbar switch, an implementation of which is illustrated in Fig. 3.79. The implementation of Fig 3.79 uses a two-input/two-output switching cell, which is also shown in the figure. The switching cell has two modes of operation: the crossing mode and the bending mode. In the crossing mode, input 1 is connected to output 2 and input 2 is connected to output 1. In the bending mode, input 1 is connected to output 1 and input 2 is connected to output 2. The crossbar switch allows any processor to be directly connected to any memory, if desired. The crossbar switch has been used extensively in interconnection networks for transactions processing systems. The problem is that a single switch failure can make it impossible for certain connections to be realized. Assume that each switching cell can fail in only one of two ways: the cell is stuck in the bending mode or the cell is stuck in the crossing mode. In both cases, the data is not corrupted, but the cell simply loses its switching capability. Develop a design for a crossbar switch that uses the basic switching cell and is capable of tolerating any single switch failure. Assume that your crossbar switch must connect four processors to four memories with each processor being able to connect, one at a time, to any of the memories.

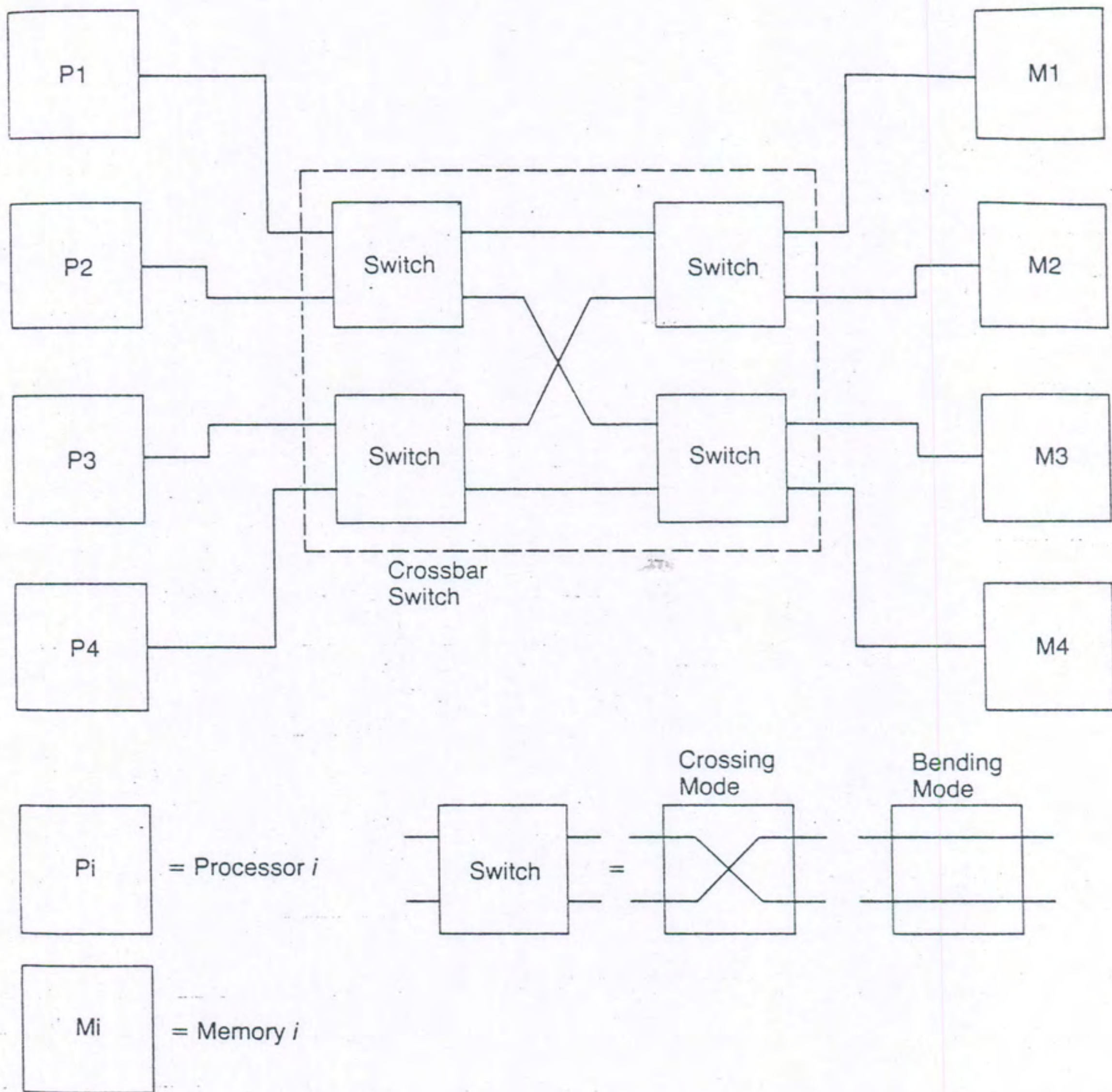


Fig. 3.79 Implementation of a crossbar switch using basic switching elements.