ROBERT S. LEDLEY

Digital Computer and Control Engineering

McGRAW-HILL

DIGITAL COMPUTER AND CONTROL ENGINEERING

McGRAW-HILL ELECTRICAL AND ELECTRONIC ENGINEERING SERIES

Frederick Emmons Terman, *Consulting Editor*
W. W. Harman and J. G. Truxal,
Associate Consulting Editors

Ahrendt and Savant · Servomechanism Practice
Angelo · Electronic Circuits
Aseltine · Transform Method in Linear System Analysis
Atwater · Introduction to Microwave Theory
Bailey and Gault · Alternating-current Machinery
Beranek · Acoustics
Bracewell · The Fourier Transform and Its Application
Brenner and Javid · Analysis of Electric Circuits
Brown · Analysis of Linear Time-invariant Systems
Bruns and Saunders · Analysis of Feedback Control Systems
Cage · Theory and Application of Industrial Electronics
Cauer · Synthesis of Linear Communication Networks
Chen · The Analysis of Linear Systems
Chen · Linear Network Design and Synthesis
Chirlian · Analysis and Design of Electronic Circuits
Chirlian and Zemanian · Electronics
Clement and Johnson · Electrical Engineering Science
Cote and Oakes · Linear Vacuum-tube and Transistor Circuits
Cuccia · Harmonics, Sidebands, and Transients in Communication Engineering
Cunningham · Introduction to Nonlinear Analysis
D'Azzo and Houpis · Feedback Control System Analysis and Synthesis
Eastman · Fundamentals of Vacuum Tubes
Feinstein · Foundations of Information Theory
Fitzgerald and Higginbotham · Basic Electrical Engineering
Fitzgerald and Kingsley · Electric Machinery
Frank · Electrical Measurement Analysis
Friedland, Wing, and Ash · Principles of Linear Networks
Ghausi · Principles and Design of Linear Active Circuits
Ghose · Microwave Circuit Theory and Analysis
Greiner · Semiconductor Devices and Applications
Hammond · Electrical Engineering
Hancock · An Introduction to the Principles of Communication Theory
Happell and Hesselberth · Engineering Electronics
Harman · Fundamentals of Electronic Motion
Harman · Principles of the Statistical Theory of Communication
Harman and Lytle · Electrical and Mechanical Networks
Harrington · Introduction to Electromagnetic Engineering
Harrington · Time-harmonic Electromagnetic Fields
Hayashi · Nonlinear Oscillations in Physical Systems
Hayt · Engineering Electromagnetics
Hayt and Kemmerly · Engineering Circuit Analysis
Hill · Electronics in Engineering
Javid and Brenner · Analysis, Transmission, and Filtering of Signals
Javid and Brown · Field Analysis and Electromagnetics
Johnson · Transmission Lines and Networks
Koenig and Blackwell · Electromechanical System Theory
Kraus · Antennas
Kraus · Electromagnetics
Kuh and Pederson · Principles of Circuit Synthesis

Ledley · Digital Computer and Control Engineering
LePage · Complex Variables and the Laplace Transform for Engineering
LePage and Seely · General Network Analysis
Ley, Lutz, and Rehberg · Linear Circuit Analysis
Linvill and Gibbons · Transistors and Active Circuits
Littauer · Pulse Electronics
Lynch and Truxal · Introductory System Analysis
Lynch and Truxal · Principles of Electronic Instrumentation
Lynch and Truxal · Signals and Systems in Electrical Engineering
Millman · Vacuum-tube and Semiconductor Electronics
Millman and Seely · Electronics
Millman and Taub · Pulse and Digital Circuits
Millman and Taub · Pulse, Digital, and Switching Waveforms
Mishkin and Braun · Adaptive Control Systems
Moore · Traveling-wave Engineering
Nanavati · An Introduction to Semiconductor Electronics
Pettit · Electronic Switching, Timing, and Pulse Circuits
Pettit and McWhorter · Electronic Amplifier Circuits
Pettit and McWhorter · Concepts of Probability Theory
Pfeiffer · Linear Systems Analysis
Reza · An Introduction to Information Theory
Reza and Seely · Modern Network Analysis
Rogers · Introduction to Electric Fields
Ryder · Engineering Electronics
Schwartz · Information Transmission, Modulation, and Noise
Schwarz and Friedland · Linear Systems
Seely · Electromechanical Energy Conversion
Seely · Electron-tube Circuits
Seely · Electronic Engineering
Seely · Introduction to Electromagnetic Fields
Seely · Radio Electronics
Seifert and Steeg · Control Systems Engineering
Siskind · Direct-current Machinery
Skilling · Electric Transmission Lines
Skilling · Transient Electric Currents
Spangenberg · Fundamentals of Electron Devices
Spangenberg · Vacuum Tubes
Stevenson · Elements of Power System Analysis
Stewart · Fundamentals of Signal Theory
Storer · Passive Network Synthesis
Strauss · Wave Generation and Shaping
Su · Active Network Synthesis
Terman · Electronic and Radio Engineering
Terman and Pettit · Electronic Measurements
Thaler · Elements of Servomechanism Theory
Thaler and Brown · Analysis and Design of Feedback Control Systems
Thaler and Pastel · Analysis and Design of Nonlinear Feedback Control Systems
Thompson · Alternating-current and Transient Circuit Analysis
Tou · Digital and Sampled-data Control Systems
Tou · Modern Control Theory
Truxal · Automatic Feedback Control System Synthesis
Valdes · The Physical Theory of Transistors
Van Bladel · Electromagnetic Fields
Weinberg · Network Analysis and Synthesis
Williams and Young · Electrical Engineering Problems

DIGITAL COMPUTER AND CONTROL ENGINEERING

ROBERT STEVEN LEDLEY

ASSOCIATE PROFESSOR OF ELECTRICAL ENGINEERING
THE GEORGE WASHINGTON UNIVERSITY
CONSULTANT MATHEMATICIAN
TO THE NATIONAL BUREAU OF STANDARDS

WRITTEN WITH THE ASSISTANCE OF

LOUIS S. ROTOLO JAMES BRUCE WILSON

Research Scientist *Research Associate*
The George Washington University

McGRAW-HILL BOOK COMPANY, INC.
New York Toronto London
1960

DIGITAL COMPUTER AND CONTROL ENGINEERING

Library of Congress Catalog Card Number 59-15055

VII

36981

THE MAPLE PRESS COMPANY, YORK, PA.

TO MY SONS
FREDDY AND GARY

FOREWORD

Until less than a century ago, men toiled manually to produce the very clothes they wore, shelters they lived in, and food they ate. The industrial revolution—derived from the invention of machines that automatically make commodities—was a revolution that *vastly enlarged man's productive capabilities.* The effect is observed upon comparing the daily life of a man of a century ago with that of a modern man, in his highly mechanized economic and interdependent social civilization. Now we are on the threshold of a new kind of revolution, a revolution that may in the future have even more far-reaching effects, *vastly increasing man's "thinking" capabilities* of planning, analyzing, computing, controlling. This new revolution derives from the availability of machines that automatically compute and control. We know its effects will be great, but we now can only speculate on the forms they will take. Not only will its influence be directly marked in the physical sciences and in technology, but, perhaps even more significantly, it will have a tremendous effect on the biological sciences and on the economic, political, and social aspects of our civilization. Its effects during the productive life span of the infant of 1960 will certainly be greater than those the industrial revolution has had during the life span of the infant of 1900. Pivotal in the growth and development of this newest of mankind's capabilities will be the digital-computer and -control engineer, whose primary occupation is the creation —the research, development, and production—of these new machines.

There is another quite remarkable aspect of digital-computer and control engineering. Never before in the history of human endeavor has a new development of such scope and complexity emerged so rapidly. Within a single decade this entirely new field, constructing and utilizing thousands of new computers, has already penetrated almost all phases of our modern society, from nuclear-energy production and missile design to the processing of bank checks, business invoices, and medical diagnoses. And in the research stages there are already components that might make it feasible to build computers manyfold more complex than present-day computers.

Beyond the unique potential to mankind and the phenomenal growth of the field is yet a third relatively unique aspect of digital-computer and control engineering. It is fast becoming one of the broadest in scope and most demanding of all engineering fields, encompassing fundamental aspects of logic, mathematics, systems engineering, as well as solid-state physics and electrical engineering. Since an error of only a single bit in a program or the breakdown of a single gate or component of a digital computer or control can result in the failure of the entire system, high meticulousness is demanded of the engineer. Thus his training must not only cover a wide range of topics but must as well emphasize precise attention to detail.

This book is the first comprehensive elementary engineering text in the digital-computer- and digital-control-engineering field (although there are several excellent advanced books in more specialized aspects of the field). The book seeks to present a new synthesis of educational material possessing a unity and breadth arising from the organization of the various aspects of digital-computer and -control engineering as a whole. It provides the material of a basic field of study for all students of electrical engineering, regardless of their ultimate specialty. However, it is hoped that this book may help stimulate a number of young engineers to enter this dynamic and vitally young field.

Samuel N. Alexander
Chief, Data Processing System Division
The National Bureau of Standards

PREFACE

General Information. The purpose of this book is to fill the need for a comprehensive elementary-engineering textbook in the large and still rapidly growing field of digital computers and controls. (The term *control* is used to emphasize that digital control and digital computers are based on the same principles.) The need for such a text is not confined to engineering schools; in industry as well, the graduate engineer with a few sporadic encounters with digital circuitry needs a sound introduction to the burgeoning literature on all phases of digital computers and controls.

Within the first five years after he earns his degree almost every electrical engineer will deal with some phase of digital circuitry. Therefore a course on digital computers is required for all candidates for the bachelor of electrical engineering degree at The George Washington University. This is indicative of a trend in engineering schools throughout the country.

This text is based on experience I gained in teaching courses on digital computers at The George Washington University School of Engineering. It is directed to senior undergraduate engineering students and first-year graduate students and is intended primarily for a year's course. Prerequisites are college physics, calculus, and at least a first course in electronic circuits, although no great proficiency in these subjects is required. Since it is an elementary exposition of the principles of digital-computer and control engineering, the book covers topics in all three phases of the subject: the over-all design of digital systems (Parts 1 and 2); the logical design of digital circuitry (Parts 3 and 4); and the electronic design of digital circuits (Part 5).

An introductory exposition of a field as large and complex as this can never hope to treat all subjects exhaustively. Full treatment must be left to specialized source books, handbooks, and journal articles. Almost without exception, each chapter of this book could be used as the basis for an entire book in itself—this is in fact true of some sections as well. Each chapter is designed to introduce the student to certain fundamental concepts and techniques of development. The method of teaching is by example rather than by generalized exposition. It is felt that the student is more capable of grasping abstractions from specifics than of extracting concepts from discussions based on generalizations alone. Hence I have tried wherever it was possible to guide the student by means of specific, concrete examples. From experience I have found that this pedagogical method is particularly well suited to engineering students.

Since there has been no previous comprehensive text in the field, there is no precedent for the choice of topics. The subjects covered were chosen to present a continuous, natural development of the major aspects of the field in the limited time available to the engineering student. Of course in any field of this size there will exist differences of opinion as to which topics should and should not be included. I feel that I have chosen those topics of greatest importance and that this comprehensive treatment will satisfy the needs of the largest number of readers.

Much attention was paid to the exercises, of which more than 750 appear in the book. The exercises at the end of each section serve to illustrate the material of the section, to enable the student to gain computational facility, and to extend the material to closely related topics not covered in the section. Almost every chapter ends with Additional Topics, a section designed to introduce the reader to new material not covered in the chapter and to stimulate his further reading in the field. The text includes many new results of original research never previously compiled into book form, some of which here appear in publication for the first time (see Special Technical Features below).

Outline of the Text. Perhaps the most outstanding capability afforded by digital techniques is that of decision making, as exemplified by the programmed system. Thus Part 1, the first of two parts on digital systems, is concerned with the digital programmed system. Its first two chapters are introductory in nature: the first is intended to *motivate* the student by delineating the wide range of applications of digital systems and controls; the second is designed to *orient* the student with respect to the digital-computer-engineering field. The next three chapters are directly concerned with programming; their purpose is to expose the engineering student to a large variety of instruction and operation formats, as well as to the practice of coding. Sequencing of instructions in four-, three-, two-, and one-address-system instruction formats is considered. The fundamental concepts of the loop, or iteration, and the subroutine are introduced, followed by a discussion of the various kinds of operations that instructions can involve. At the end of Part 1 we consider the formulation of program-checking and computer-maintenance programs. Further automatic-programming techniques lay the groundwork for a simplified presentation of the international algebraic automatic program called ALGOL.

Part 2 is concerned with the functional approach to digital-systems design; here again the purpose is to expose the student to a variety of possible systems-design concepts. Numerical analysis is considered, as the basis of the systems design of the general-purpose digital computer. Here the concept of the polynomial-approximation approach is stressed as a fundamental method for reducing most mathematical computation to additions, subtractions, multiplications, and divisions. Next are described computational methods other than those of classical numerical analysis, methods which are, however, of fundamental importance in business and in other activities requiring data reduction. Besides techniques for searching and sorting, examples of methods for redundant and irredundant coding are considered. In order to demonstrate clearly the possibility of other than general-purpose computers, the digital differential analyzer is considered, along with real-time control and other techniques. Also a general discussion is included of the concepts underlying the "super" computers now in the research and developmental stages. Finally I introduce the Pedagac, a small general-purpose computer intended to provide the necessary *thread of continuity* to the study of digital-computer engineering (see below for further discussion of the Pedagac).

Probably the "newest" of the concepts confronting the uninitiated reader in digital-computer and -control engineering is that concerned with Boolean algebra as the basis for the logical design of digital circuits. Thus Part 3, the first of two parts on the logical design of digital circuitry, is concerned with the mathematical foundations of Boolean algebra. First Boolean algebra is introduced in terms of propositions, for it is advisable that the engineer understand the relation of Boolean algebra to other concepts as well as digital circuitry. Hence we take up the propositional-calculus representation and the class, or set, representation before the digital-circuit representation. The method of using bases and associated designation numbers in the succeeding chapters (which was first fully developed by the author, although it appeared implicitly in some earlier writings) has been found admirably suited to the teaching of the logical design of digital circuits. On the basis of this method, several modern procedures for the simplification of Boolean functions are explained, leading into the design of digital circuits to compute elementary synchronous recursive functions. In the final three chapters of this part, digital computational methods of importance in logical circuit design are considered. With few exceptions the methods presented are based on the author's original research. Elementary algorithms, including methods for solving Boolean equations and their application to circuit design, are given. Chapters 13 and 14 consider computations with Boolean matrices, the former being concerned with the theoretical development of the methods, the latter with applications of these results. (Should it be desired to study the applications of the computational methods before delving into the details of the proofs, Chap. 13 has been written so that the applications of Chap. 14 may be considered directly after Sec. 13-3 with no loss of continuity.)

Part 4 is concerned with the logical design of specific computer components. First the serial arithmetic unit is considered, then parallel and rapid arithmetic operations. Since the philosophy of teaching general principles by specific example is used, a survey of methods of performing arithmetic operations is not given. Instead the discussions are centered round a few illustrations, carefully chosen for their suitability in demonstrating clearly the concepts involved in circuits that perform arithmetic operations. Next the various problems are considered that arise in the computation of control functions, beginning with a general discussion of minimum decoding procedures. Finally the concept of packaging is introduced, and with the logical design of the Pedagac as an illustrative example, many of the important aspects of the final logical design of a complete digital system are elucidated.

The last part, Part 5, is concerned with what is probably the most active field in digital-computer and -control engineering, the electronic design of digital circuits. The goals of the electronic design of circuits for use in digital computers and controls are derived from considerations of the digital-systems design and the logical design already covered. Here again the attempt is made to teach concepts through selected examples of circuits. Clearly, in these rapidly developing and dynamic fields, detailed discussions of many different specific circuits are not justified—many of them would certainly be obsolescent before publication of the book. Hence specific circuits are used only as concrete illustrations of the more general underlying principles. The experienced reader will observe that the topics covered in many of the individual sections can well afford entire books to themselves. The first chapter of this part is concerned with the two most important problems involved in the transition from abstract systems and logical design to the electronic realization of digital circuits, namely, timing, or clock phasing, and reliability. Then aspects of the use of semiconductor devices, diodes and transistors, in electronic-digital-circuit design are considered. Here the most important concept to be learned is a thorough understanding of the use of the devices; hence much space is devoted to discussion of the physical operation and the equivalent circuit of transistors. Consideration of magnetic elements in digital-circuit design follows, encompassing magnetic-core and multiaperture gating devices as well as magnetic amplifiers. The discussion ends with a section on two most promising modern circuit developments, Cryotron and microwave circuits. The next chapter is concerned with the closely related memory methods and input-output techniques. Only core and film high-speed memories are considered, since the use of other high-speed memory forms is rare; for the same reason only tape and drum low-speed memories are described. Many of the most important input-output methods depend on analog-to-digital and digital-to-analog conversion, and these are accordingly included in this chapter. In this context a full explanation of the important sampling theorem is given. The final chapter illustrates some exceedingly important topics in the final design of a computer, through the electronics and wiring diagram of the Pedagac.

Special Technical Features. In Part 1 of the text Sec. 5-9 is concerned with the International Algebraic Language (ALGOL). This general automatic algebraic programming language contains the primary features of other automatic programs, such as FORTRAN, IT, etc., and has the advantage of becoming widely accepted. Sponsored by the Association for Computing Machinery and several foreign societies, the universal acceptance of ALGOL would undoubtedly have a great beneficial effect on the national and international exchange of ideas and methods of programming.

Special features of Part 2 include J. H. Wegstein's general method for accelerating the convergence of iterative solutions to equations, in Sec. 6-3. The Tabledex method and the techniques of searching with relaxed conditions given in Secs. 7-3 and 7-4, as well as the new, more accurate formulas for evaluating superimposed coding given in Sec. 7-7, are the original work of the author. The specific method of Sec. 8-3 for coding the control computer was developed by the author, while the method for coding the logistics and business computer, appearing in Sec. 8-5, is based on the work of W. H. Marlow.

In Part 3 the computational methods for constraints, logical dependence and independence, solution to Boolean algebraic equations, and transformation to the absolute simplest form, given in Secs. 12-2 through 12-10, are the author's original developments. The method of antecedence and consequence solutions, the fundamental Boolean matrix formulas and their proofs, given in Secs. 14-1 through 14-6, are the result of the author's original research, as are also the extension of the use of designation numbers to three (or more) -valued logic and their application to the design of three-valued digital circuits (Sec. 12-11). The extension of the fundamental Boolean matrix formulas to multivalued logic, as given in Sec. 14-7, was developed by W. R. Smith and N. F. J. Matthews; the general solution to the logical matrix equation given in that section was developed by W. R. Smith. R. D. Elbourn deserves credit for the direct method of finding prime implicants given in Sec. 11-7.

The discussions of the parallel adder and of rapid multiplication, in Secs. 16-2 and 16-4 of Part 4, are based on the work of A. Weinberger and J. L. Smith. The rapid-division method of Sec. 16-5 was developed by the author and J. B. Wilson. The treatment of minimum decoding methods is based on a development of C. H. Page. In Part 5 the probabilistic-logic approach to increasing circuit reliability (Sec. 19-4) is the author's adaptation of a development of W. S. McCulloch. The transistor equivalent circuit, described in Sec. 20-4 for the most commonly used grounded-emitter voltage-drive configuration, is based on an original development of S. B. Geller. The magnetic amplifier of E. W. Hogue in Sec. 21-3 presents a new technique for magnetic amplification. Section 23-4 on minimum-wiring theory is based on the works of H. Loberman and A. Weinberger.

Special Pedagogical Feature: The Pedagac. It has been my experience that, although students might understand most of the various isolated aspects of computer engineering, they still might not visualize the complete process of designing and building a computer. In order to provide the *necessary thread of continuity* to the study of computer engineering, a simple computer, the Pedagac (*"Pedagogic Automatic Computer"*) is designed from start to finish in the book (see Chaps. 9, 18, and 23). At first glance the Pedagac may seem unduly specific. However, this is far from true, for the Pedagac serves as a concrete example by means of which many exceedingly important points are illustrated. Many of these points would be completely meaningless to the student if they were not developed in the context of an entire system. The Pedagac was specially designed to illustrate pertinent subject matter most clearly from a pedagogical (rather than cost or production, etc.) point of view. The arithmetic unit is serial, since a parallel arithmetic unit would have overwhelmed the student with primarily repetitive circuitry. The registers are static flip-flops, for these are the most easily understood in relation to arithmetic operations. The control is of a parallel nature, which is in general easier to comprehend. Eight clock phases are used to simplify the transition from logical to electronic design. The specially designed packages were made as simple in concept as possible.

Possible Course Structures. This text may be used for college courses in several different ways. For a *two-term* course the first term may cover Parts 1 and 2, and Part 3 through Chap. 12; the second term, the remainder of the book. Or the book may be used for a two-term *composite* elementary-advanced course: the elementary course may include much of the fundamentals of digital-computer engineering, and then those students who wish to pursue the subject in more detail may continue in the more advanced course. One suggested elementary course would consider Chaps. 1 to 4 of Part 1, Chaps. 6 and 7 of Part 2, Chaps. 10 and 11 of Part 3, Chaps. 15 and 16 of Part 4, and Chaps. 19 and 20 of Part 5; the more advanced course that follows would complete the text. A more leisurely treatment of the subject may be given in a *three-term* course, with more attention being paid to the selected readings and other projects and subjects mentioned in the Additional Topics sections. Here Parts 1 and 2 may be covered the first term, Part 3 the second term, and Parts 4 and 5 the third term. Other, perhaps quite eclectic, arrangements will certainly occur to the experienced instructor.

Acknowledgments. It gives me the greatest pleasure to acknowledge the help and cooperation I have received from many persons during the preparation of this book. My sincerest thanks go to James B. Wilson, who carefully and tirelessly edited the entire manuscript, contributing many important and valuable improvements to the style and technical presentation; to Louis S. Rotolo, who assisted in the designing of the Pedagac and with the Pedagac chapters, was extremely helpful in the preparation of the manuscript, and contributed many valuable suggestions; and to William R. Smith, who critically read the entire manuscript, prepared most of the technical illustrations, and helped in numerous other ways. I would like also to express my gratitude to R. D. Elbourn for reviewing the manuscript, making many constructive criticisms. The encouragement and support given by E. Frank and N. T. Grisamore have been greatly appreciated. Many colleagues helped by supplying advice and information on various technical points. Of these the author particularly wishes to mention A. Weinberger, J. A. Cunning-

ham, S. B. Geller, E. W. Hogue, J. H. Wegstein, the late M. Abramowitz, H. Loberman, S. Greenwald, E. R. Toense, W. W. Youden, J. L. Smith, D. R. Boyle, C. H. Page, W. H. Marlow, N. F. J. Matthews, and J. Rabinow. The author is indebted to G. U. Uyehara, N. T. Grisamore, and R. A. Toense for assisting in the design of the Pedagac package. The quality of the illustrations is in large part due to the skill and talent of J. E. Ozefovich, D. K. Anand, and A. Bucek, as well as W. R. Smith. The author wishes to thank Eva March Cuddy, Edna Crum, and Judith Holsberg for typing most of the manuscript. He gratefully acknowledges the partial support of the Mathematics Division of the Air Force Office of Scientific Research, Air Research and Development Command, and the Information Systems Branch of the Office of Naval Research, without which the preparation of the manuscript would have been impossible.

Robert Steven Ledley

CONTENTS

PART 1. INTRODUCTION TO DIGITAL PROGRAMMED SYSTEMS

PART 2. FUNCTIONAL APPROACH TO SYSTEMS DESIGN

PART 3. FOUNDATIONS FOR THE LOGICAL DESIGN OF DIGITAL CIRCUITRY

PART 4. LOGICAL DESIGN OF DIGITAL-COMPUTER CIRCUITRY

PART 5. ELECTRONIC DESIGN OF DIGITAL CIRCUITS

PART 1

INTRODUCTION TO DIGITAL PROGRAMMED SYSTEMS

CHAPTER 1

APPLICATIONS OF DIGITAL COMPUTERS AND CONTROL

1-1. Introduction

The purpose of this chapter is to stimulate the reader to want to know what is in the rest of the book. By illustrating how computers can be applied, we hope to interest the reader in how he might help design and use these remarkable machines. The wide range of potential applications of digital computers and control in all phases of science, industry, and government is truly amazing. In the sciences applications range from mathematics and physics to biology and medicine. The early applications were in engineering and physics, but perhaps the greatest scientific use of computers yet to come is in the biological, medical, and social sciences. In business and industry applications range from automatic banking, inventory control, and process control to the precise control of milling machines to produce parts requiring complicated high-tolerance machining and to missile-testing processes. In government applications range from automatic patent searching and post-office sorting to large-scale control of defense weapon systems. Present applications are already too numerous to list in a single volume, but they are infinitesimal compared with the potential future applications. We hope, however, that by presenting brief descriptions of just a *few* applications we may impart to the reader a better insight into how digital computers and controls are used.

In general a computer can be conceived as a numerical-transformation machine. Numbers are the *inputs* to it, and the computer transforms these numbers into new numbers, which appear as the *outputs:*

$$\text{Input numbers} \rightarrow \boxed{\text{computer}} \rightarrow \text{output numbers}$$

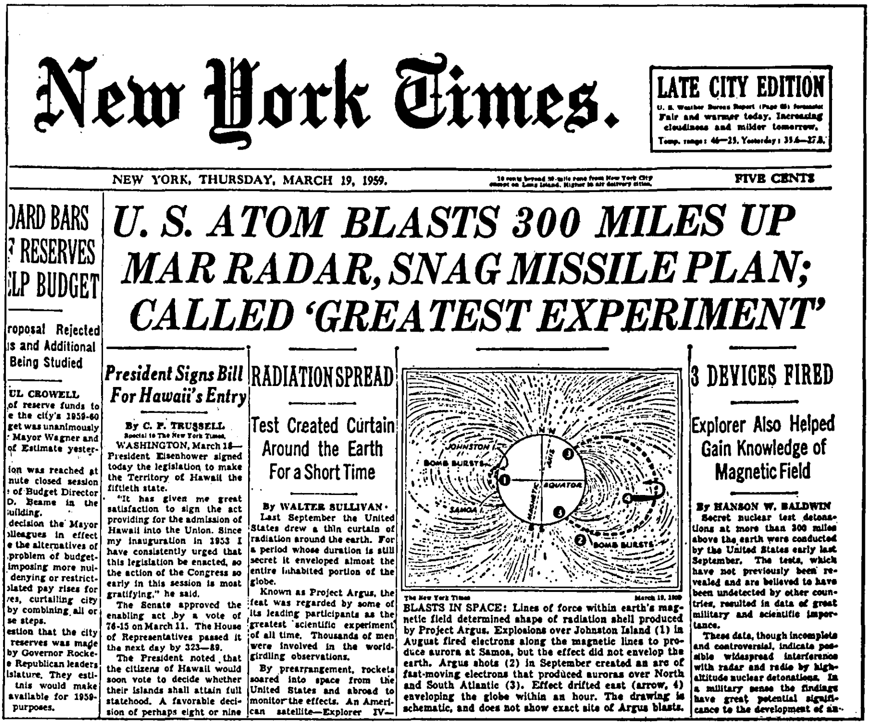
For instance, the input numbers may be the initial conditions of a differential equation; the output numbers will be a table of the functional solutions. Or the input numbers may be readings from an engineering drawing, and the output numbers will be coded instructions to direct a special milling machine. Or the input numbers may be codes that represent a patient's symptoms; the output numbers may be codes that represent possible alternative disease diagnoses.

There are two main kinds of electronic computers: *digital* and *analog.* In digital computers the numbers (digits) themselves are handled explicitly by the computer; an analog computer instead deals with a representation of the numbers, for instance, by voltages, lengths, etc. In this book we are concerned with digital computers only. These can be classified as *general-purpose* or *special-purpose* computers. In the first part of this book we shall describe the general-purpose computer; in Part 2 we shall consider some of the special-purpose computers. Special-purpose digital computers are sometimes referred to as *digital controls.* The same engineering principles are involved in the design of either general-purpose computers or special-purpose digital-control computers. Therefore we have used the two terms in the title of the book.

For the purposes of this chapter we have arbitrarily classified computer applications into four categories; many applications can fall into more than one of our categories. *First* we shall consider *numerical solutions* to *equations.* This represents an important class of applications, and undoubtedly it is these applications which makers of the first computers had in mind. The first of these applications described is based on principles familiar to every electrical engineer, the motion of charged particles in a vacuum. However, the context of the application discussed here has far greater significance. Problems involving the solution to equations are not limited to the physical sciences; equations can be written to describe many biological phenomena as well. The *second* category is *process control:* Here the need for digital computers with specialized capabilities becomes apparent. It is probably not an exaggeration to say that the *third* category came into being because of the great capabilities of the computers: this is *simulation.* Before the advent of computers, simulations (except perhaps laboriously calculated military war games) were rarely even discussed because they were clearly not feasible to perform. However, with the advent of high-speed computers simulations have become practical, and great new fields of scientific research have been opened. Most of the present-day computer applications can probably be classed in the *fourth* category, *data processing:* Business accounting, statistical reductions, information retrieval, and many other important but routine procedures can be handled by computers.

1-2. Numerical Solution to Equations

Motion of Charged Particles in the Earth's Magnetic Field. Project Argus was a major scientific and military experiment conducted by the United States (Fig. 1-1). Rockets carried a small atomic bomb more than 300 miles above the earth, at this altitude the bomb was exploded, and a resulting thin layer of radiation spread quickly round the globe. Nuclear and atomic particles were propelled by the exploding bomb and streaked through the vacuum of the universe under forces almost entirely due to the earth's magnetic field. Some particles moved toward the earth's poles and caused artificial auroras (northern and southern lights). In order to measure the characteristics of this radiation layer, an earth satellite and other rockets were launched with paths passing through the layer (see Fig. 1-2). Instruments in the satellite and rockets transmitted measurements back to earth. Scientifically the shots rank among mankind's foremost experiments. They showed that radiation introduced into space by man can have significant effects over those of natural radiation. Militarily the Argus shots indicated that the additional radiation added to the natural radiation interfered with man's electronic communications. The grave consequence is that such interference can be caused at will.



New York Times.

LATE CITY EDITION
U. S. Weather Bureau Report (Page 65) forecasts: Fair and warmer today. Increasing cloudiness and milder tomorrow.
Temp. range: 46—23. Yesterday: 35.6—27.8.

NEW YORK, THURSDAY, MARCH 19, 1959. 10 cents beyond 50-mile zone from New York City except on Long Island. Higher in air delivery cities. FIVE CENTS

U. S. ATOM BLASTS 300 MILES UP MAR RADAR, SNAG MISSILE PLAN; CALLED 'GREATEST EXPERIMENT'

)ARD BARS
F RESERVES
LP BUDGET

roposal Rejected
is and Additional
Being Studied

UL CROWELL
of reserve funds to e the city's 1959-60 get was unanimously Mayor Wagner and of Estimate yester-

ion was reached at nute closed session of Budget Director D. Beame in the uilding.
decision the Mayor lleagues in effect e the alternatives of problem of budget-imposing more nui-denying or restrict-lated pay rises for es, curtailing city by combining all or se steps.
estion that the city reserves was made by Governor Rocke-e Republican leaders islature. They esti-this would make available for 1959-purposes.

President Signs Bill For Hawaii's Entry

By C. P. TRUSSELL
Special to The New York Times.

WASHINGTON, March 18—President Eisenhower signed today the legislation to make the Territory of Hawaii the fiftieth state.

"It has given me great satisfaction to sign the act providing for the admission of Hawaii into the Union. Since my inauguration in 1953 I have consistently urged that this legislation be enacted, so the action of the Congress so early in this session is most gratifying," he said.

The Senate approved the enabling act by a vote of 76-15 on March 11. The House of Representatives passed it the next day by 323—89.

The President noted that the citizens of Hawaii would soon vote to decide whether their islands shall attain full statehood. A favorable decision of perhaps eight or nine

RADIATION SPREAD

Test Created Curtain Around the Earth For a Short Time

By WALTER SULLIVAN

Last September the United States drew a thin curtain of radiation around the earth. For a period whose duration is still secret it enveloped almost the entire inhabited portion of the globe.

Known as Project Argus, the feat was regarded by some of its leading participants as the greatest scientific experiment of all time. Thousands of men were involved in the world-girdling observations.

By prearrangement, rockets soared into space from the United States and abroad to monitor the effects. An American satellite—Explorer IV—

The New York Times March 19, 1959

BLASTS IN SPACE: Lines of force within earth's magnetic field determined shape of radiation shell produced by Project Argus. Explosions over Johnston Island (1) in August fired electrons along the magnetic lines to produce aurora at Samoa, but the effect did not envelop the earth. Argus shots (2) in September created an arc of fast-moving electrons that produced auroras over North and South Atlantic (3). Effect drifted east (arrow, 4) enveloping the globe within an hour. The drawing is schematic, and does not show exact site of Argus blasts.

3 DEVICES FIRED

Explorer Also Helped Gain Knowledge of Magnetic Field

By HANSON W. BALDWIN

Secret nuclear test detonations at more than 300 miles above the earth were conducted by the United States early last September. The tests, which have not previously been revealed and are believed to have been undetected by other countries, resulted in data of great military and scientific importance.

These data, though incomplete and controversial, indicate possible widespread interference with radar and radio by high-altitude nuclear detonations. In a military sense the findings have great potential significance to the development of an-

FIG. 1-1. Announcement of Argus shots.

The Argus project tested theoretical calculations of what would take place when the small atomic bomb was exploded above the earth's atmosphere. Where would the electrons and other particles go? This was a problem in the study of the motion of charged particles in a magnetic field (the earth's) and is in principle, except for the shape of the field, the same as the problems encountered in magnetically directing an electron beam in a television tube, or accelerating particles in a betatron, or circulating particles in a cyclotron, etc. If the earth's magnetic field is represented as that of a magnetic dipole, and if x, y, and z are cartesian coordinates with the z axis directed along the earth's magnetic axis from south to north, then from electromagnetic field theory it can be shown that the trajectories of the charged particles are given by the solutions to the following differential equations (where the superscript dots denote differentiation with respect to time):

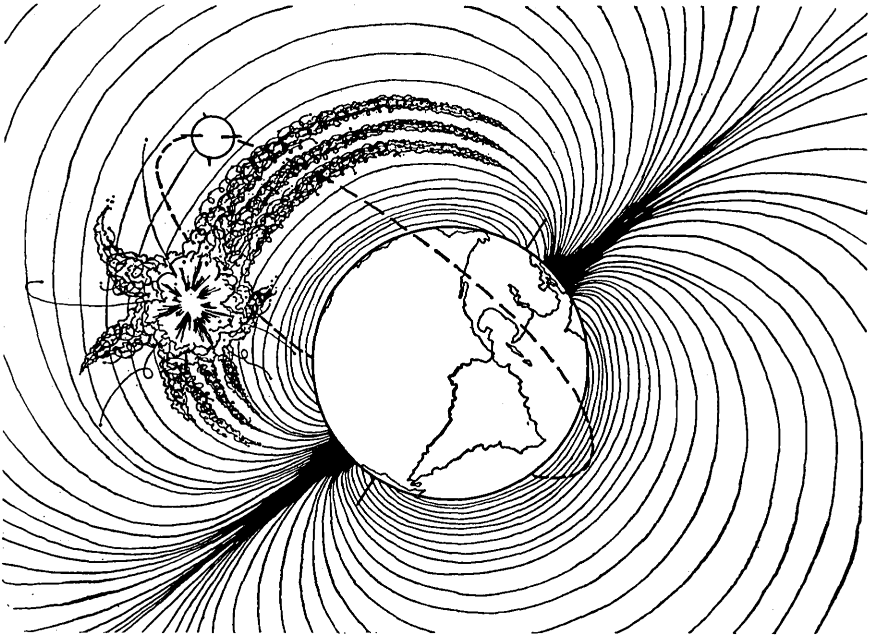


FIG. 1-2. Earth satellite measuring characteristics of radiation layer.

$$\ddot{x} = \frac{Mq}{mc}\{-(x^2 + y^2 + z^2)^{-5/2}[(2z^2 - x^2 - y^2)\dot{y} - 3yz\dot{z}]\}$$

$$\ddot{y} = \frac{Mq}{mc}\{-(x^2 + y^2 + z^2)^{-5/2}[(x^2 + y^2 - 2z^2)\dot{x} + 3xz\dot{z}]\}$$

$$\ddot{z} = \frac{Mq}{mc}\{+3z[k(x^2 + y^2 + z^2)^{-5/2} + (x^2 + y^2)(x^2 + y^2 + z^2)^{-4}]\}$$

where $k = (x\dot{y} - y\dot{x}) - (x^2 + y^2)(x^2 + y^2 + z^2)^{-3/2}$
M = earth's magnetic dipole strength
m = relativistic mass of particle
q = charge of particle
c = speed of light

The initial conditions are the initial coordinates (x_0,y_0,z_0) of the particle and the components $(\dot{x}_0,\dot{y}_0,\dot{z}_0)$ of its velocity just after the explosion.

These equations can be used to plot the paths of the charged particles given their initial velocities and directions. The distribution of the initial velocities and directions can be calculated from nuclear physics; the trajectories of different kinds of particles can then be computed for many different initial conditions in the distribution. The complexity of these computations requires the use of a digital computer. The basic importance of such a calculation preliminary to the actual experiment is evident: by predicting the motion of the particles, proper and adequate plans can be made concerning the instrumentations of the satellite and the measurements to be made, the best geographical location of the experiment, the optimum altitude for exploding the bomb, and so forth.

Hydrodynamics of Blood Flow. The partial differential equations that describe biological phenomena more often than not defy analytical solution. Attempts are usually made either to introduce simplifying assumptions or to isolate special aspects of the problem for investigation. Both these alternatives involve certain compromises on the part of a researcher. Such compromises can often be avoided, however, by using an electronic computer to solve the original partial differential equations numerically.

Consider the problem of blood flow through a large artery—an instance of pulsed viscous fluid flow in an elastic tube. The first step in an investigation of this problem is the formulation of hypotheses about the hydrodynamic principles involved and the derivation of the corresponding partial differential equations. The second step is to devise an experiment from which boundary conditions for the equations may be determined. The third step is to solve the equations on a computer, comparing the solutions with the data obtained experimentally. Probably adjustments will be required in the hypotheses and the equations, and further experiments and computations will have to be devised. In this way the phenomenon can be studied in its entirety.

For example, the following equations can be derived, assuming laminar viscous flow and an exponential functional form for the elasticity of the blood vessel:

$$V = -\frac{1}{8\mu}\frac{dP}{dx}A^2 \qquad F = -\frac{\pi A^4}{8\mu}\frac{dP}{dx}$$

$$\frac{\partial}{\partial t}A^2 + \frac{\partial}{\partial x}A^2V = 0 \qquad \frac{dP}{dA} = CA^m$$

where P = pressure
F = average flow through a cross section
V = average blood velocity over a cross section
μ = coefficient of blood viscosity
A = radius of aorta
x = distance along aorta
t = time
C, m = constants

The experiment was to measure the pressure, as a function of time, at equally spaced intervals along the aorta of an anesthetized dog (see Fig. 1-3). The intercostal arteries (between the ribs) were tied, so that there was no significant outflow from the aorta, and a flowmeter was placed at one end of the aorta. The boundary conditions chosen were the flow and the pressure at one end of the aorta; the equations were used to predict the pressure at the other points along the aorta. The accuracy of the predictions with respect to the experimental results would indicate the degree of validity of the original hypotheses upon which the equations were based. Such experiments can have important applications to the prevention and cure of heart diseases in man.

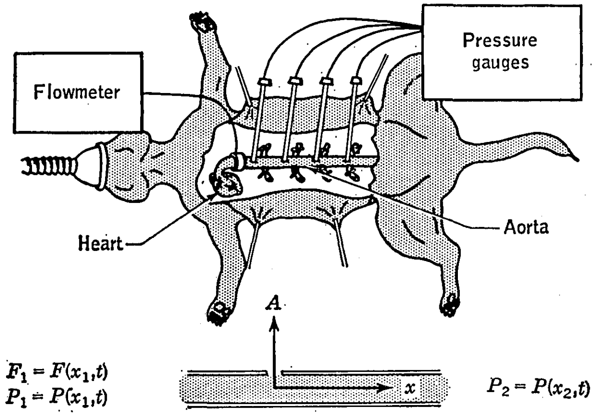


FIG. 1-3. Hydrodynamics of blood flow.

The Design of a Lens System. The need for accurate high-altitude photography has gained importance with the suggestion that aerial inspections be used as a means of averting atomic wars. For this work lenses of very high resolving power must be designed for aerial cameras. Figure 1-4 shows the detail that can be obtained from high-altitude photographs using modern lens systems. The scientific design of lens systems has become feasible only through the use of electronic computers.

The problem of designing a lens system consists in tracing the paths of many rays, *emanating from a single object point*, through the lens system to the final image plane (see Fig. 1-5). The extent of deviations in the image plane of these rays is a measure of the resolving power of the system. The computer computes the path of each ray as it would be refracted on passing through each spherical lens surface of the system. The initial direction of each ray is taken from the object point under consideration, and the position of impingement of this ray on the first lens surface is computed. Then the new direction of the ray as it leaves this first surface is calculated, and from this its position on the second lens surface is computed. In this way the ray is traced through the entire lens system, and its position on the final image plane is determined

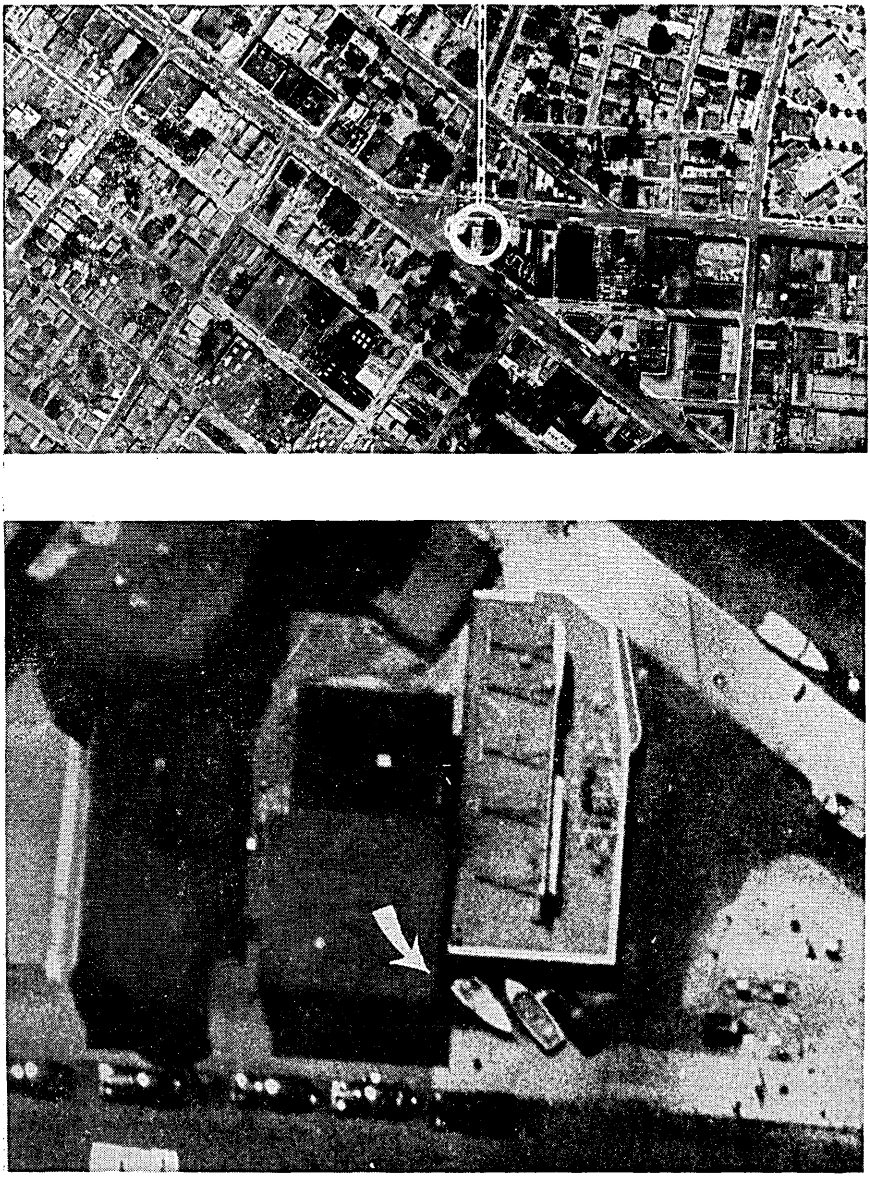


FIG. 1-4. Aerial photography (25,000 ft), with enlargement of circled portion of photograph. [*Photographs courtesy U.S. Air Force Air Photographic and Charting Service (MATS).*]

TABLE 1-1. FLOW DIAGRAM OF COMPUTATIONS FOR TRACING A RAY THROUGH A LENS SYSTEM†

Read into the computer three constants for the lens surface presently under consideration:

C_1 = curvature of spherical lens surface

t = distance between this and previous lens surface

$\mu_1 = \dfrac{N}{N_1}$ = ratio of indices of refractions of lens surface to left and right, respectively

↓

Read into the computer six constants for the ray being traced, namely:

$T = (x,y,z)$ = three coordinates of ray on previous lens surface

$Q = (X,Y,Z)$ = three components of unit direction vector of ray leaving previous surface

↓

Compute $T_1 = (x_1,y_1,z_1)$, the coordinates of the point where the ray strikes this lens surface, by means of the following incidence equations:

$$x_1 = x + LX - t$$
$$y_1 = y + LY$$
$$z_1 = z + LZ$$

where

$$L = e + \frac{C_1M^2 - 2M_x}{X + \xi}$$

and

$$\xi = [X^2 - C_1(C_1M^2 - 2M_x)]^{1/2}$$
$$M^2 = x^2 + y^2 + z^2 - e^2 + t^2 - 2tx$$
$$M_x = x + ex - t$$
$$e = tx - (xX + yY + zZ)$$

↓

Compute $Q_1 = (X_1,Y_1,Z_1)$, the components of the unit direction vector of the refracted ray leaving this surface, by means of the following refraction equations:

$$X_1 = \mu_1X - gC_1x_1 + g$$
$$Y_1 = \mu_1Y - gC_1y_1$$
$$Z_1 = \mu_1Z - gC_1z_1$$

where

$$g = \xi' - \mu_1\xi$$

and

$$\xi' = [1 - \mu_1^2(1 - \xi^2)]^{1/2}$$

and ξ was found in the previous box.

Print out $T_1 = (x_1,y_1,z_1)$ and $Q_1 = (X_1,Y_1,Z_1)$.

↓

Repeat the computation for each lens surface of the system in sequence from left to right until the ray position on the final image plane has been determined.

† The subscripted variables refer to the lens surface under consideration; the nonsubscripted variables refer to the previous surface.

(see Table 1-1 and Fig. 1-6). The results of a set of such computations from one object point are shown to the right of Fig. 1-5, where each dot represents the final position of a ray on the image plane. In an ideal lens system all the rays would fall on the same point in the final image plane. The dispersion of the dots indicates the resolving power of the real lens system.

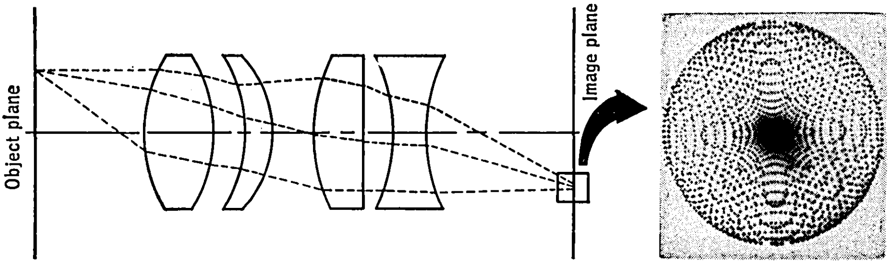


FIG. 1-5. Ray tracing.

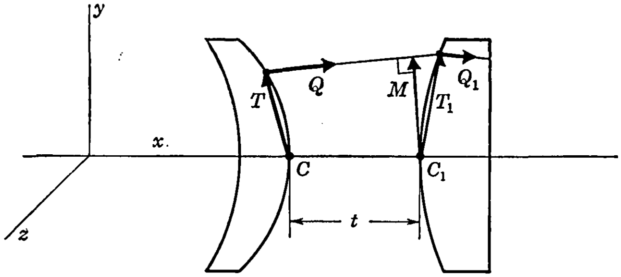


FIG. 1-6. Geometry of ray tracing.

1-3. Process Control

Machine-tool Control.† Significant improvements in many manufacturing processes can be obtained with the application of digital data-processing and control techniques. An example is digital machine-tool control, used to improve the over-all process of producing many different and complicated parts needed in our modern technology. Greater accuracy, reproducibility, and versatility are obtained as tooling costs and skilled-manpower requirements are reduced. It increases our capabilities to produce greater quantities and varieties of precision-machined parts, thus revolutionizing their potential use.

Consider a milling machine cutting a two-dimensional contour. The cutting tool is moved in the x and y directions by means of high-performance hydraulic servo drives. These servo drives are controlled by rapidly sequenced pulses that tell the servos precisely where to move the

† See Y. C. Ho and E. C. Johnson, Design of a Numerical Milling Machine System, *Proc. Eastern Joint Computer Conf.*, 1957, Washington, D.C.

tool in order to obtain the contour finally desired. For instance, each time the x-direction control receives a pulse, it might advance the tool 0.0002 in. in the x direction; if we assume that there can be up to 20,000 such pulses per second, the tool can be made to advance in the x direction a maximum of $0.0002 \times 20{,}000 = 4$ ips = 20 fpm. By varying the number of pulses per second, up to the maximum, the *speed* with which the tool moves in the x direction can be precisely controlled. The control of the y-direction motion of the tool is similar. Thus the tool can be directed to move round any desired contour. The precision of machining the desired part is of course limited by the smallest increment of distance of the control.

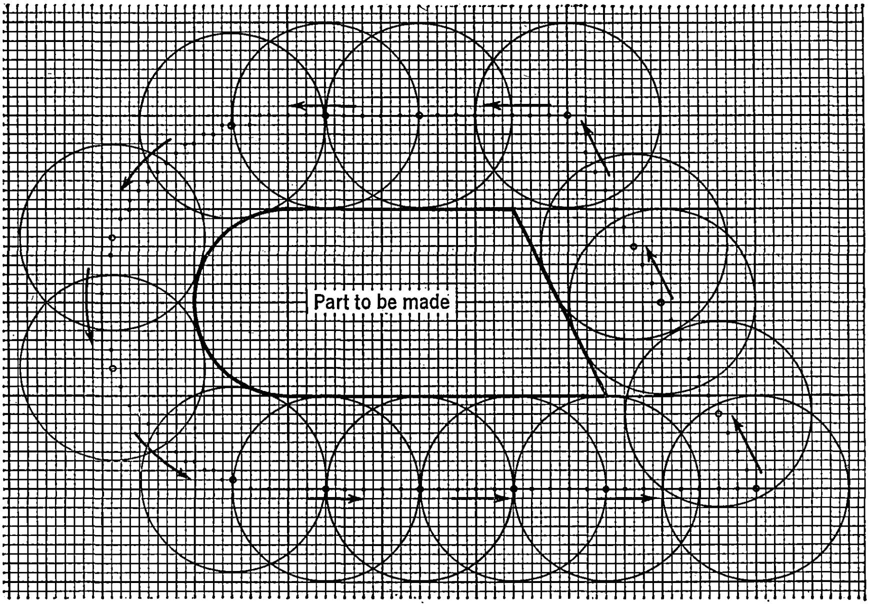


FIG. 1-7. Cutter offset path.

The problem of digital machine-tool control reduces to that of transforming information on a blueprint design into an appropriately timed sequence of x and y input control pulses. The control pulses direct the center of the cutting tool, but the desired contour is formed by the edge of the cutter; hence the proper *cutter offset* must be calculated from the blueprint. See, for example, Fig. 1-7, where the circles indicate some of the successive positions of the cutter. If we assume in Fig. 1-7 that the graph lines are 0.0002 in. apart, then the dots represent the successive positions of the offset path of the cutter's center that should be directed by the control pulses. (For example, the diagonal cut would be made by two pulses in the y direction for each one pulse in the x direction.) Of course, in a realistic case the part would be many times the size illustrated, and hence the number of input pulses required would be very large. Thus it would be a difficult job to start from the blueprint and manually determine the detailed sequences of control pulses. The method of interpreting the blueprint design for the automatically controlled milling machine is divided into two steps. *First* the cutter's offset path is determined grossly in terms of the end points of straight-line path segments, and radii of circular contours, and so forth, as shown in Fig. 1-8. This in itself can present difficult problems of curve fitting. (Can you determine the equations of the offset path for the contour illustrated in Fig. 1-8?) The *second step* is to interpolate between these points and from the radii, etc., to determine finally the precise 0.0002-in. *input-pulse fine structure.*

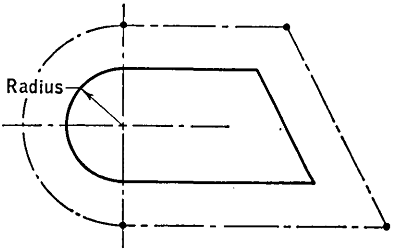


FIG. 1-8. Gross determination of cutter's offset path.

Two computers are used for this process. The first is a general-purpose digital computer into which are put the specifications for the part contour taken from the blueprint. These might take the forms "diagonal at arctan 2," "straight 4 inches," "horizontal 3 inches," "radius 2 inches," etc. Or the input to the digital computer might be the blueprint itself, put into the computer by means of a "picture-reading" machine. The computer is programmed to compute from such input data the gross data of the offset cutter path. This is perhaps put out in the form of punched paper tape, like the rolls of a player piano, or in the form of a deck of punched cards. The second computer is used to control the milling machine directly. The punched paper tape or card deck is put into this control computer, which interpolates, etc., to produce the fine-structure input pulses to the servo drives. The entire process is illustrated in Fig. 1-9.

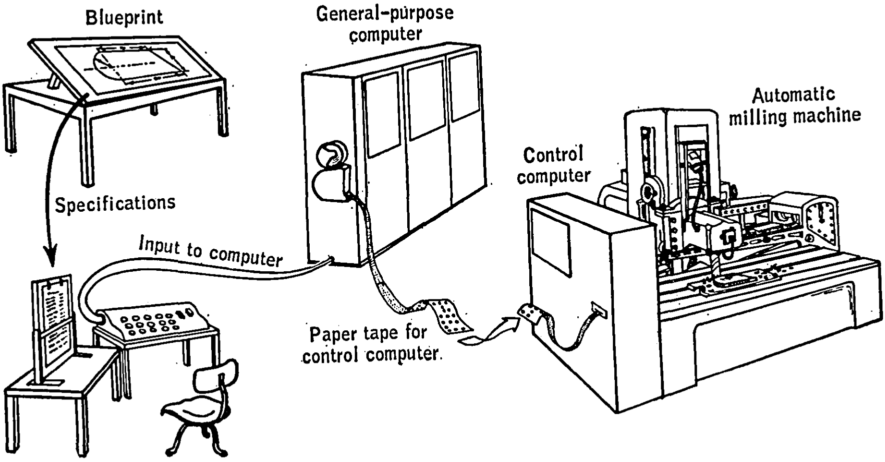


FIG. 1-9. Machine-tool control process.

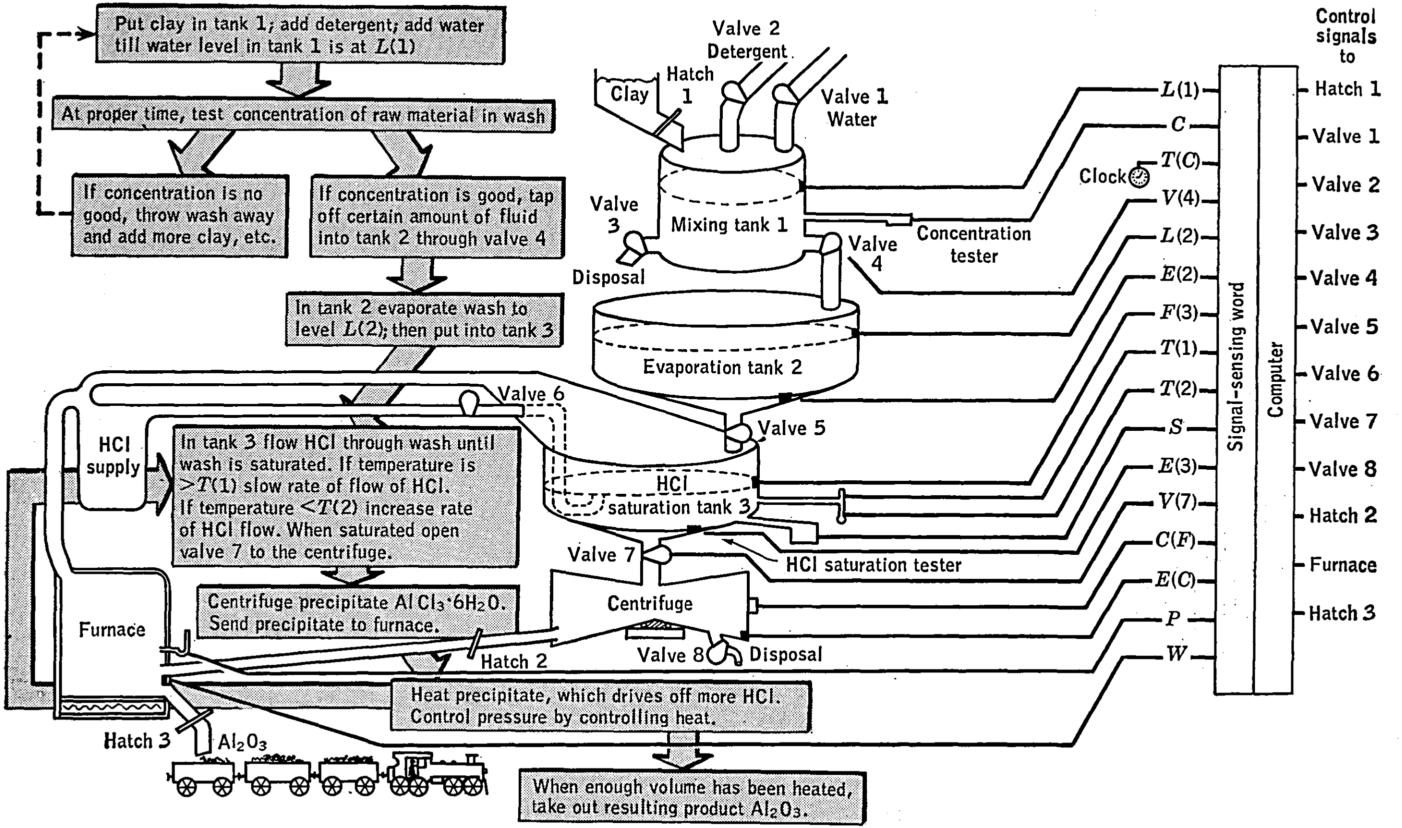

Put clay in tank 1; add detergent; add water till water level in tank 1 is at $L(1)$
At proper time, test concentration of raw material in wash
If concentration is no good, throw wash away and add more clay, etc.
If concentration is good, tap off certain amount of fluid into tank 2 through valve 4
In tank 2 evaporate wash to level $L(2)$; then put into tank 3
In tank 3 flow HCl through wash until wash is saturated. If temperature is $>T(1)$ slow rate of flow of HCl. If temperature $<T(2)$ increase rate of HCl flow. When saturated open valve 7 to the centrifuge.
Centrifuge precipitate $AlCl_3 \cdot 6H_2O$. Send precipitate to furnace.
Heat precipitate, which drives off more HCl. Control pressure by controlling heat.
When enough volume has been heated, take out resulting product Al_2O_3.
Valve 2
Detergent
Hatch 1
Clay
Valve 1
Water
Valve 3
Mixing tank 1
Disposal
Valve 4
Concentration tester
Clock
Evaporation tank 2
Valve 6
Valve 5
HCl supply
HCl saturation tank 3
Valve 7
HCl saturation tester
Centrifuge
Furnace
Hatch 2
Valve 8
Disposal
Hatch 3
Al_2O_3
$L(1)$
C
$T(C)$
$V(4)$
$L(2)$
$E(2)$
$F(3)$
$T(1)$
$T(2)$
S
$E(3)$
$V(7)$
$C(F)$
$E(C)$
P
W
Signal-sensing word
Computer
Control signals to
Hatch 1
Valve 1
Valve 2
Valve 3
Valve 4
Valve 5
Valve 6
Valve 7
Valve 8
Hatch 2
Furnace
Hatch 3

FIG. 1-10. Control of chemical plant.

Control of Chemical Plant. In many factories the processing of raw materials frequently involves sequences of decisions for which all alternative possibilities are known. By means of digital computers the entire processing can be made completely automatic. Control of this kind has the advantages that the decision processes themselves become more reliable when performed electronically and that the design of more complicated processes becomes feasible. As a specific example, consider a simplified version of a chemical factory that produces Al_2O_3 (see Fig. 1-10).† Table 8-2 summarizes the conditions which initiate various actions in the plant. The column entitled "Sensed-signal conditions" represents the list of conditions requiring some operation to be performed. The column entitled "Corresponding signals to be generated" indicates the actions to be taken.

Input signals to the control computer are transmitted by wires from critical places in the factory (see Fig. 1-10). These signals, defined in Table 8-1, consist essentially of two-valued (on or off) test results which in combination tell the computer the *present state of the factory.* For example, signal $L(1)$ will be a pulse when the mixture in mixing tank 1 is at or above a predetermined level and no pulse if the mixture is below this level. Each signal is initiated by a *transducer,* a device that converts mechanical information into electrical information, or vice versa. The transducers for our example convert fluid levels in tanks, temperatures, chemical concentrations, volumes of precipitate in a centrifuge, pressures, and so forth, into the electrical pulses that are the inputs to the computer. The outputs from the control computer are electrical signals supplied to transducers (servomechanisms) that open and close valves or hatches and otherwise operate to change the state of the factory. Table 8-2 shows under what conditions such changes of state will be initiated.

1-4. Simulations

Industrial Dynamics. The term industrial dynamics refers to the processes and factors influencing the economic development of an industrial corporation considered as an interacting part of a free competitive economy. The study of industrial dynamics involves so many factors that no complete analytical mathematical analysis is yet possible. Many universities in the United States, in cooperation with many industrial corporations, are attempting to simulate aspects of the industrial-dynamics problem. Representatives of management take part in such simulations, making decisions; the digital computer then computes the effects of these decisions and produces a report to the management representatives, who make new decisions, etc. (see Fig. 1-11). One advantage of such simulations is that they aid in training management personnel by presenting them with many and different industrial circum-

† The control of this particular process will be considered in greater detail in Sec. 8-3; it suffices here to outline briefly the kind of input information supplied to the computer and the way the computer then controls the factory. (See pp. 262 to 267.)

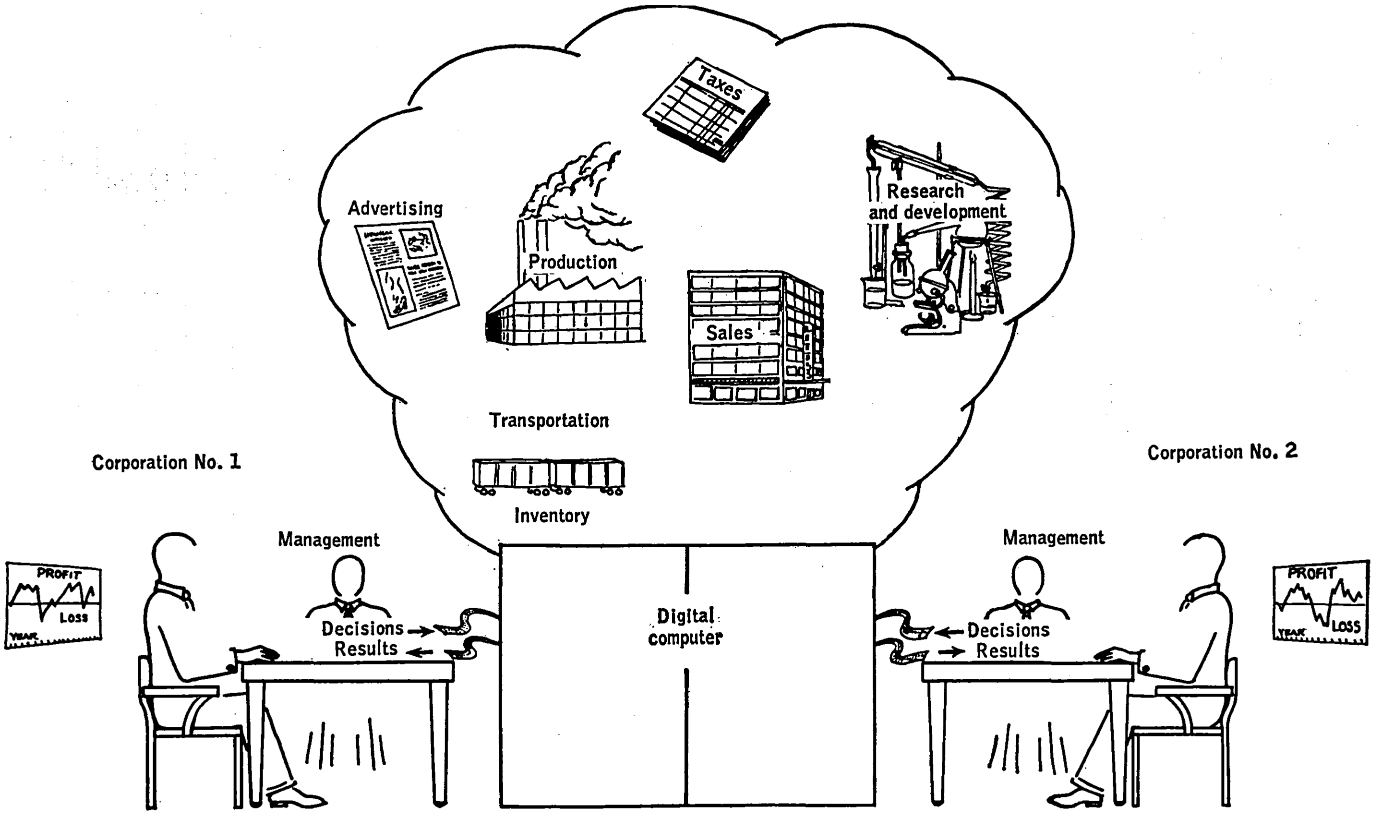


FIG. 1-11. Business simulation.

stances analogous to real situations. A second advantage is that the simulations aid the study of industrial dynamics by bringing before the scientists involved situations not encountered through purely academic study. And, third, important generalizations can be deduced from the progress and results of the simulations.

As an illustration, consider a multifirm one-product competitive industry. The simulation is designed in such a way that the results of 5 years of operations can be simulated in 1 day. Each firm in the simulation is represented by a team of executives, who must make the following set of "management" decisions:

1. Price of product
2. Producing volume
3. Advertising expenditures
4. Research and development expenditures
5. Amount of new investment in plant and equipment
6. Dividends to be paid, etc.

One set of such decisions is made by each team firm for each quarter of a year. The computer then computes the interactions and effects of these decisions on the market; i.e., it simulates what may happen in reality and produces for each team a report giving:

1. Sales volume
2. Per cent of total products sales handled by this firm
3. Production capacity of this firm for the next quarter
4. Statement of profit and loss
5. Current inventory quantities
6. Statement of receipts and disbursements
7. Statement of financial conditions, etc.

Based on the study of its report, each team makes decisions as to how the firm will operate in the next quarter. These decisions are fed into the computer, and the simulations progress in this way from quarter to quarter.

The computer processes the decisions made by the teams (firms) and produces reports by means of mathematical models. The model is essentially a set of hypothetical equations relating the input information to the desired output information. For example, increased advertising bears a functional relation to increased sales, but this in turn depends on the advertising decisions of competitive firms. Increased research and development can produce an improved product which may give one firm some advantage. Price rises of course offset sales, but advertising can to a certain extent offset an increase in price. Obviously the gross profit is the product of sales volume and price per unit product, minus administrative costs, overhead, manufacturing costs, advertising, research and development, etc. The mathematical model embodying such considerations can become very complicated, depending on the level at which the simulation is intended. Evidently a great deal of study and research must be carried out in connection with the derivation of the specific equations that enter into the computer simulation. For a simulation to be at all realistic, the complications will be so numerous as to be impossible to perform without digital computers.

Man-Machine Simulation System. Man-machine simulation is now recognized as an essential tool for the development of complex automatic systems that involve both electronic and manual controls. Such is the case, for example, in air-traffic control, ground control of interceptors, ground-controlled approach and landing, weapon assignment, or missile guidance systems. The interactions between the electronic control system being designed and the team of human operators can be studied *before* the system is actually constructed. This eliminates the only other alternative, the frequently prohibitively expensive process of actually building the proposed system and carrying out an extensive program of tests and modifications.

Figure 1-12 represents a simulation system that includes both a digital and an analog computer, linked with display and control equipment which permit human beings to be coupled into the system. The analog-computer equipment is used for such tasks as accepting continuous control information and solving complex dynamic equations in real time, for which they are suited. The digital computer is used for controlling the experiment, for sequencing the stages depending on the outcome of each stage, for generating simulated data, for calculating with high precision where necessary, for handling variables with large dynamic ranges, for making logical decisions, and for statistically analyzing experimental results.

An example of a simulation is a simplified ground-controlled interceptor system (Fig. 1-12). The digital computer simulates a situation in which a large number of aircraft are observed by the search radar. A console displays the locations of these aircraft on a cathode-ray tube for the ground crew; with each aircraft is also displayed an associated identification (i.e., friendly, enemy, unknown, interceptor, bomber, commercial, etc.). The ground crew manually assigns targets to the interceptors; the targets are automatically tracked by the digital computer. A model of an interceptor cockpit fully equipped with instruments and pilot flying controls enables a human operator to imitate the actions of the interceptor pilot in flight. The pilot of the interceptor receives his flight-control information either directly from the digital computer or from the ground-crew commands. The pilot's control movements are interpreted by the analog computer, which then simulates the dynamics of the interceptor, operating the instruments in the cockpit. The velocity components of the interceptor are sent to the digital computer, which computes the path of the interceptor. The digital computer also displays the computed positions of both interceptor and target on a monitor scope for the ground crew. It is this display which gives the ground crew its information for the simulated data link with the pilot. The research control console enables a research team to set up the initial conditions, vary the target aircraft maneuvers, and determine the interceptor closure tactics. Records are kept of information generated by the

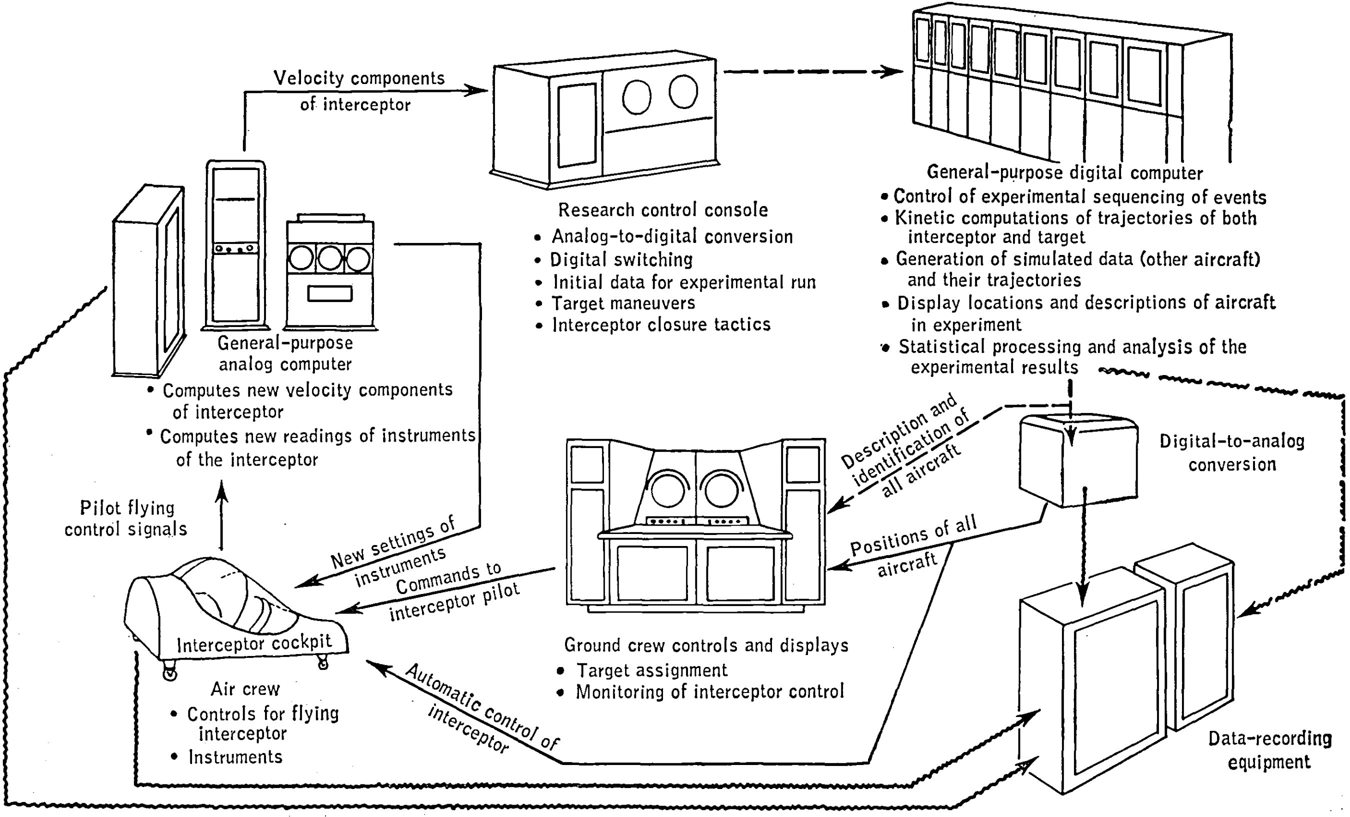

Velocity components of interceptor
Research control console
• Analog-to-digital conversion
• Digital switching
• Initial data for experimental run
• Target maneuvers
• Interceptor closure tactics
General-purpose digital computer
• Control of experimental sequencing of events
• Kinetic computations of trajectories of both interceptor and target
• Generation of simulated data (other aircraft) and their trajectories
• Display locations and descriptions of aircraft in experiment
• Statistical processing and analysis of the experimental results
General-purpose analog computer
• Computes new velocity components of interceptor
• Computes new readings of instruments of the interceptor
Pilot flying control signals
New settings of instruments
Commands to interceptor pilot
Description and identification of all aircraft
Positions of all aircraft
Digital-to-analog conversion
Interceptor cockpit
Air crew
• Controls for flying interceptor
• Instruments
Ground crew controls and displays
• Target assignment
• Monitoring of interceptor control
Automatic control of interceptor
Data-recording equipment

FIG. 1-12. Man-machine simulation.

interceptor pilot, analog computer, and digital computer during the experiment. These can then be analyzed on the digital computer after an experimental simulation has been completed.

1-5. Data Processing

Inventory Control. The annual industrial inventory of the United States has been estimated at 30 billion dollars; the cost of supporting this inventory is probably about 3 billion dollars annually in interest lost, tied-up money, depreciation, obsolescence, spoilage, storage, handling charges, and so forth. It has been estimated that the use of computer control and information-processing methods could reduce this inventory and its associated cost of support by more than 25 per cent. (A special-purpose inventory-control computer will be discussed in a later chapter.) Figure 1-13 indicates how a general-purpose computer could be used in an inventory-control system. Here the inventory files are electronically recorded in the computer. Inputs to the computer are items initiated outside the inventory records, such as receipts, issues, orders, shipping reports, etc. These inputs indicate the alterations in the inventory records that must be made by the computer. The outputs of the computer are items that arise from an examination of the inventory records, such as the need for more parts, orders to restock, bills or invoices for items shipped, accounting records, and other data about the current inventory records that may aid management in planning, etc. The analysis of an inventory-control problem for the use of a digital computer includes the preparation of a complete report on all the processing requirements, process charts of the data flow, and definitions of the controls. These reports and charts correspond to the equations describing a scientific problem.

Consider, for example, a concern that manufactures and sells certain products, and suppose that the company keeps the following files (see Fig. 1-13): (*a*) a customer master file that contains all customers' names, addresses, purchases, balances due, credits, etc.; (*b*) a products inventory master file that contains the kind and number of each product in stock as well as its location, price, and other identifying descriptions and numbers; (*c*) a factory parts and raw-materials inventory file that contains a list of all parts and materials stocked that enter into the manufacturing process, together with descriptive identifications and numbers; and (*d*) a suppliers master file that contains all suppliers' names, addresses, supply records, balances due, credits, orders not yet received, and so forth. The procedure by which the computer keeps such files up to date and generates information from them will now be described; the reader should keep in mind that our intention is to give a gross idea of the problems involved, and we have hence oversimplified the circumstances. More realistically, these problems are characterized by huge masses of detailed considerations.

The customer file would be updated each time an order, payment, or return was received from the customer. At the same time the computer

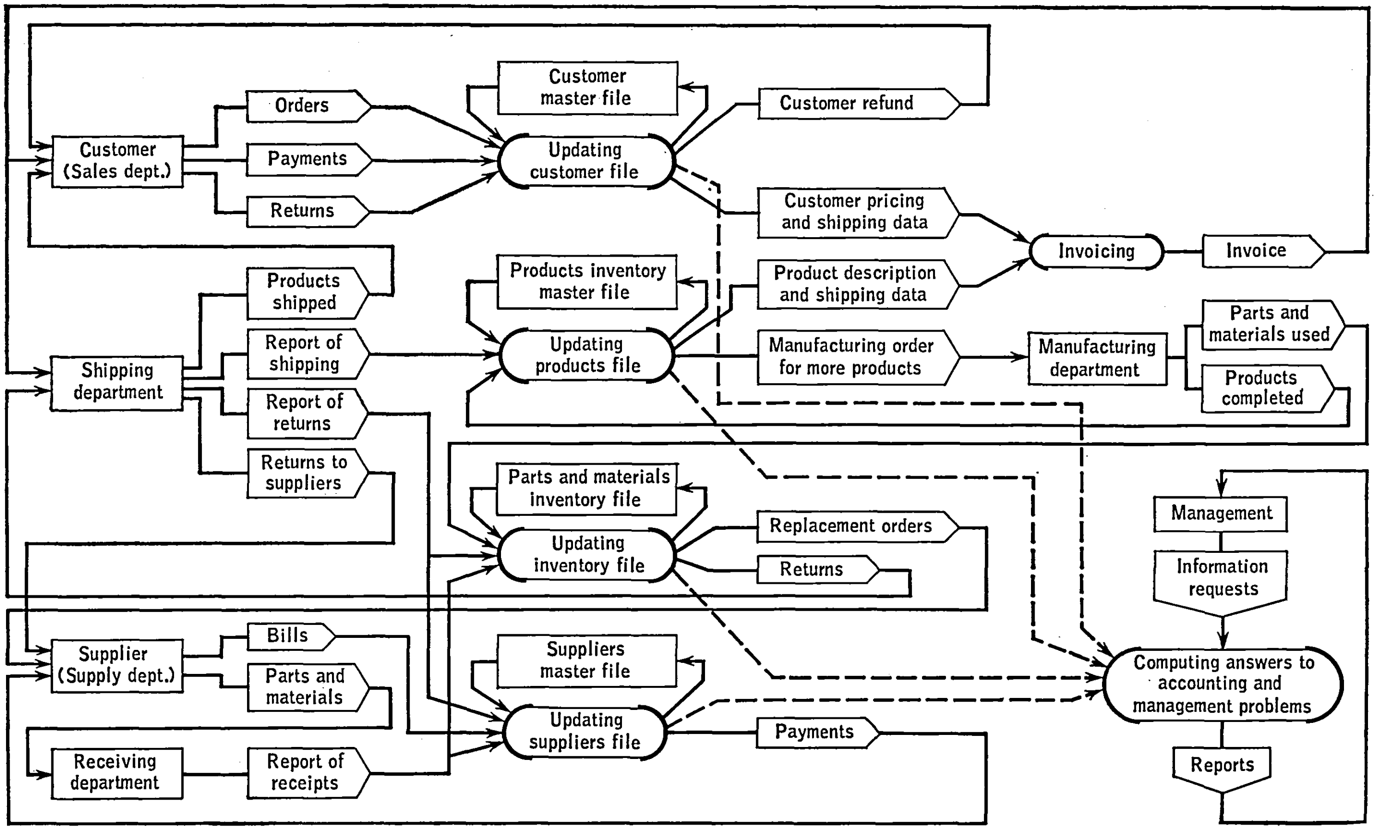


FIG. 1-13. Inventory-control system.

would produce the necessary pricing and shipping information (i.e., customer discount, location, etc.) if an order was made, or the customer's refund order if a return was made. The products inventory file would be updated each time a report was received of shipping or completing a product. If a product was to be shipped, the computer would produce the product description number and warehouse location, and it would also produce an order for more products to be manufactured if the shipment reduced the inventory below a predetermined level. The computer can also produce the invoice for a sale from the proper information received from the files. The file of parts and raw materials used in the manufacturing processes would be updated each time a report was made of parts and materials used, returned, or received. The computer would determine what replacement orders and returns should be made as well as update the inventory. The supplier file would be updated each time a bill was received, or report of return or receipt was made. The computer would produce the payment orders to the supplier. Finally the computer would produce reports requested by management, on the basis of the various file records.

Airline Reservation System. In the reservation system of a nationwide airline, every minute hundreds of inquiries about seats available, actual sales, cancellations, and unsatisfied demands must be processed. Each ticket office in the United States of a particular airline must be able to interrogate the same centrally located computer concerning seats available on particular flights and to make reservations on particular flights; the processing of each query must be completed within a few seconds. The main advantages of such a system are that the customer is provided with an immediate reservation service and the airline is provided with an accurate inventory system almost devoid of errors.

A simplified reservation system is shown in Fig. 1-14. Each ticket office contains a reservation interrogation and sales machine, which is connected by transmission lines to a central service area. At the central service area, records are electronically stored on the availability and disposition *of all seats on all airplanes* to be flown by the airline, for about a month in the future. The processing of a reservation starts at the ticket-office machine. Let us say that a customer desires two seats from Washington, D.C., to Chicago on Apr. 30 on the 3 P.M. flight. The sales clerk observes that Flight 83 leaves Washington, D.C., at 3 P.M. and goes to Seattle via Chicago, and chooses the card for Flight 83 from a file box. Each flight card contains information about two flights, one on each of the long edges. Along the edge are printed the scheduled stops, leaving a margin which is punched with a code representing the flight number. The clerk puts the punched margin of the card into a slot in the ticket-office machine, then presses buttons indicating the number of seats (two) and the date (Apr. 30) desired. The names of the cities on the card appear opposite a row of buttons on the machine, and the clerk pushes the buttons next to the names of the desired cities, Washington and Chicago. Then he presses the *I* (interrogation) button.

Meanwhile the remote scanner (see Fig. 1-14) is searching for a ticket-office machine with a card in it. When one is located, the remote scanner connects the ticket-office machine with the computer. The marginal punches on the card are sensed by the computer and tell it which flight is being interrogated; the date, number of seats, etc., are also transmitted to the computer. If two seats are available on Flight 83 for Apr. 30, the computer transmits the information to the ticket-office machine by illuminating the proper lights on it. If the customer desires to make the reservations, the *R* (reservation) button is pushed and the computer subtracts two from the number of seats available on Flight 83.

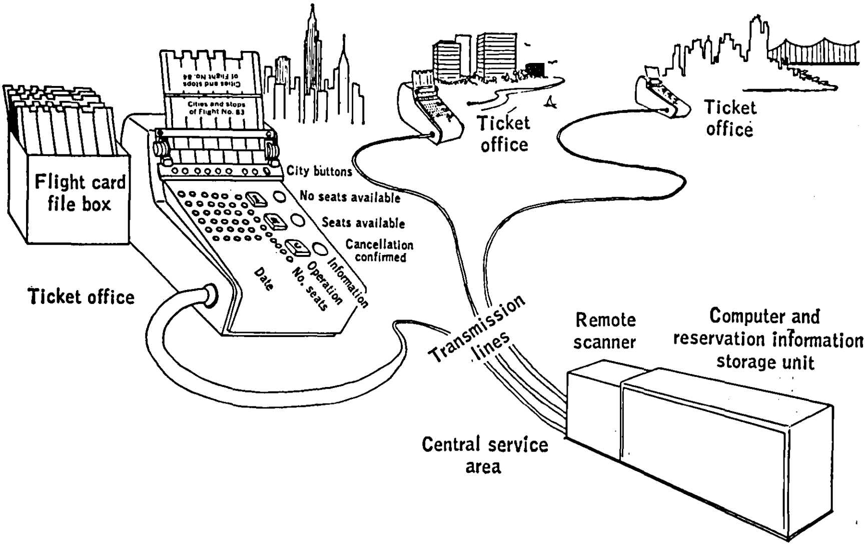


FIG. 1-14. Airline reservation system.

If no seats were available, the no-seats-available light would have been illuminated. Similarly a cancellation can be made, and so forth. Of course our illustration has been greatly simplified, but the essential operations performed by the computer and the problems involved have been demonstrated.

Aids to Medical Diagnosis. It has been recognized that electronic computers can aid certain aspects of medical diagnosis. For example, the computer can (1) produce a list of possible diagnoses, consistent with medical knowledge, for a given set of symptoms presented by a patient; (2) indicate further diagnostic tests which best differentiate between remaining diagnostic possibilities; (3) calculate the probabilities for the alternate diagnostic possibilities; and (4) enable a more precise statement and analysis of the value decisions which may be associated with treatment planning. Such computer applications must be based on extensive medical data; hence a further use of computers is (5) to compile statistics that (*a*) relate symptom-disease combinations and that (*b*) evaluate disease-treatment-prognosis results. These statistics are then used in future diagnosis, treatment, and prognosis determination. In addition, the computer can serve as an indispensable aid, under certain special circumstances: The computer can (6) enable criteria of a more quantitative nature to be utilized in evaluating the results of certain diagnostic procedures (such as electrocardiograms, electroencephalograms, etc.) and can perform the complicated calculations necessary for the proper interpretation of certain clinical measurements. Finally, the computer can significantly aid the important aspect of information retrieval in, for example, (7) the rapid retrieval of current information about new preventive measures, diagnostic techniques, and specific treatments, and (8) the accumulation and retrieval of desired aspects of a particular patient's total medical history (such as total radiation dosage received, previous allergic reactions, individual biochemical and physiologic norms and deviations, etc.).

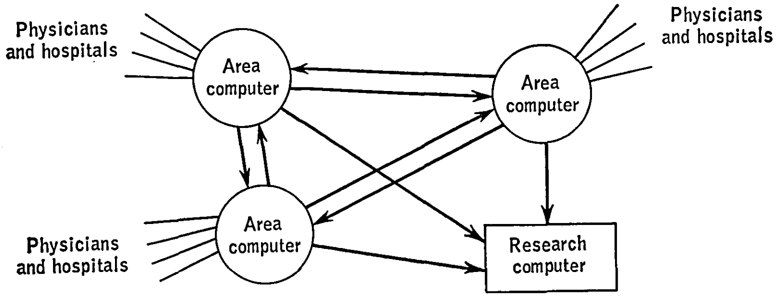


FIG. 1-15. Hypothetical health-computing system.

Consider a modern high-speed electronic digital computer, and suppose that it has been programmed to aid medical diagnosis. Let us assume further that the physician can directly communicate with the computer by telephone, teletype, radio, etc. The value of such a computer interrogation arises from three factors: (1) the ability of the computer to formulate a treatment plan that will maximize the chance of curing the patient; (2) the ability to determine the minimum number of necessary medical laboratory tests or other diagnostic procedures for the particular patient; and (3) the ability to evaluate more accurately diagnostic-test results for a particular patient based upon his previously recorded health records.

A network of such computers could form a hypothetical health-computing system (see Fig. 1-15). Here each computer can communicate with individual physicians and hospitals within its area, receiving, transmitting, and computing medical information as required. However, the area computers must be capable of communicating with each other as well, since approximately 20 per cent of Americans change addresses each year, and probably most of us go on at least one trip per year. Also, all the area computers could communicate with a special research computer that could sample data as required for various research and public health-control investigations.

The great significance and importance of such a health-computer network cannot be overestimated as an aid to increasing individual good health and longevity and as a vast new source of medical information concerning mankind.

1-6. Additional Topics

a. The following references are not intended to be a complete bibliography on the use of digital computers; rather they are intended as sources of past and future applications.

Books

Alt, Franz L.: "Electronic Digital Computers—Their Use in Science and Engineering," Academic Press, Inc., New York, 1958.

Bowden, B. V.: "Faster than Thought," Pitman Publishing Corporation, New York, 1953.

Canning, R. G.: "Electronic Data Processing for Business and Industry," John Wiley & Sons, Inc., New York, 1956.

Chapin, Ned: "Automatic Computers—A System Approach for Business," D. Van Nostrand Company, Inc., Princeton, N.J., 1959.

Eckert, W. J., and Rebecca Jones: "Faster, Faster," McGraw-Hill Book Company, Inc., New York, 1956.

Edmundson, H. P., et al.: Studies in Machine Translation, *Repts.* 1–9, Rand Corp., 1958.

Gass, Saul I.: "Linear Programming Methods and Applications," McGraw-Hill Book Company, Inc., New York, 1958.

Gotlieb, C. C., and J. N. P. Hume: "High-speed Data Processing," McGraw-Hill Book Company, Inc., New York, 1958.

Jeenel, Joachim: "Programming for Digital Computers," McGraw-Hill Book Company, Inc., New York, 1959.

Ledley, R. S., and J. B. Wilson: "Programming and Utilizing Digital Computers," McGraw-Hill Book Company, Inc., New York, in press.

Livesley, R. K.: "Digital Computers," Cambridge University Press, New York, 1957.

Locke, W. N., and A. D. Booth (eds.): "Machine Translation of Languages," John Wiley & Sons, Inc., New York, 1955.

McCracken, D. D.: "Digital Computer Programming," John Wiley & Sons, Inc., New York, 1957.

Periodicals

The Computer Journal, British Computer Society, Ltd., London.

Computers and Automation, Edmund C. Berkeley and Associates, New York.

Computing News, J. W. Granholm, Seattle.

Control Engineering, monthly, McGraw-Hill Publishing Company, New York.

Data Processing Digest, Canning, Sisson, & Associates, Los Angeles.

Datamation (Research and Engineering, the Magazine of Datamation), bimonthly, Relyea Publishing Co., New York.

IBM Journal of Research and Development, International Business Machines Corporation.

IRE Transactions on Electronic Computers, Professional Group on Electronic Computers, Institute of Radio Engineers, Inc., New York.

Journal of the Association for Computing Machinery, New York.

Journal of Operations Research Society of America, Baltimore.

Journal of the Society for Industrial and Applied Mathematics, Philadelphia.

Machine Translation, Massachusetts Institute of Technology, Cambridge, Mass.

Mathematical Tables and Other Aids to Computation, National Academy of Sciences–National Research Council, Washington, D.C.

Proceedings of the . . . Joint Computer Conferences of the Professional Group on Electronic Computers (IRE), the Association for Computing Machinery, and the American Institute of Electrical Engineers (AIEE) Committee on Computing Devices.

b. References to Analog-computer Applications. The following are but a few of the many available texts describing analog-computer applications:

Johnson, C. L.: "Analog Computer Techniques," McGraw-Hill Book Company, Inc., New York, 1956.

Korn, G. A., and T. M. Korn: "Electronic Analog Computers," 2d ed., McGraw-Hill Book Company, Inc., New York, 1956.

Murray, F. J.: "The Theory of Mathematical Machines," 2d ed., King's Crown Press, New York, 1948.

CHAPTER 2

PRINCIPLES AND BLOCK DIAGRAM OF A DIGITAL COMPUTER

2-1. Introduction

Recapitulation. In the previous chapter we gave some idea of *what* computers can do. We observed that, in general, initial conditions, constants, and other numbers are first read into the computer and that the computer then proceeds with the calculations, finally printing out or otherwise displaying the results. In addition, the examples presented illustrated the kinds of calculations that can be carried out by a computer. These concepts are important foundations for an appreciation and understanding of the present chapter.

Orientation. The purpose of this chapter is to *orient* the student in the field of digital-computer engineering; it endeavors to present a bird's-eye view of the field. Thus, as the student continues into the detailed material of this book, he should always appreciate the relationship and significance of each topic in the over-all pattern. Specific, concrete examples are used as vehicles for transmitting the over-all ideas of the design and construction of digital computers. The student should pay strict attention to the general concepts and should use the specific examples only as a *basis for understanding* these generalizations.

This chapter begins with a description of the block diagram of a digital computer. There follows a more detailed discussion of the functioning of the parts of the computer. From this "macroscopic" point of view we go to the "microscopic" picture of a computer, glancing at various kinds of circuitry techniques. Thus we consider briefly the methods by which a number is recorded, by which digital circuitry can perform specific functions, and by which internal controls are obtained, and describe how numbers are memorized by and read into and out of a computer. The examples have been chosen solely to be appropriate for this orientation phase of study and therefore constitute simpler and in many cases less advanced techniques. Finally, we summarize our discussion of the main features of digital-computer and -control engineering.

A large part of the orientation in any subject must necessarily be concerned with the specialized vocabulary used to describe basic concepts. Therefore there are in this chapter many such basic discussions.

2-2. Block Diagram of a Computer

The purpose of this section is to acquaint the student briefly with the parts of a digital computer. To this end the block diagram of a computer will be developed in steps, from the simple to the more detailed.

The Computer. As was seen in Chap. 1, a computer handles numbers; it takes numbers that are fed into it and operates on them to form new numbers—the output numbers. Hence the simplest block diagram of a computer is as shown in Fig. 2-1.

Input numbers
Computer
Output numbers

FIG. 2-1. Simplest block diagram.

The computer can perform many different kinds of *operations* on numbers. Most of these operations combine two numbers to produce a third number. The two numbers so combined are called the *arguments* of the operation, and the number produced is called the *result.* Many operations that a computer can perform are similar to the familiar operations used in hand computation, such as adding two numbers to form the sum, multiplying two numbers to form the product, etc. Also analogous to hand computing is the fact that the computer can perform *only one such operation at a time.*† Hence, when many operations are to be performed, *they must be performed in some appropriate sequence.*

The computer is told what operations to perform by means of *instructions.* An instruction not only contains information about an operation to be performed but also designates the two arguments (usually) for the desired operation. When the operation specified by an instruction has been completed by the computer, i.e., when the result has been obtained, the instruction is said to have been *executed.*

For example, suppose that we wanted to compute the value of

$$y = a(b + c)$$

where a, b, and c represent some specific real numbers. A sequence of instructions that would tell a computer to form $a(b + c)$ is as follows:

	Operation	1st argument	2d argument	Result
Instruction 1.............	Add	b	c	$x = b + c$
Instruction 2.............	Multiply	a	x	$y = ax$

The computer would first execute instruction 1 and then instruction 2.

The Memory. The computer itself is composed of two main parts, a *memory,* which stores numbers and instructions, and a *computing unit,* in which the actual computations take place (see Fig. 2-2). The com-

† Modern computers are being developed to perform several operations simultaneously.

puter's memory stores the numbers to be operated on; it stores intermediate results that are generated during the course of a computation; and it stores the final results. For instance, the numbers represented by *a*, *b*, and *c* of our example are initially stored in the memory. After the first instruction is executed, the intermediate result *x* is stored in the memory. After the second instruction is executed, the final result *y* is stored in the memory.

The instructions themselves are also stored in the computer's memory. Each instruction is transmitted to the computing unit when it is to be executed. In our example, instruction 1 is first transmitted to the computing unit and executed, and then instruction 2 is transmitted to the computing unit and executed.

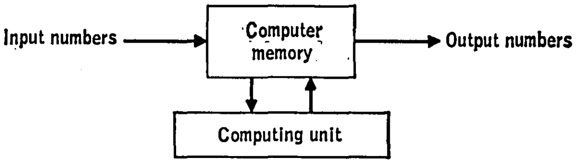


FIG. 2-2. Memory and computing unit.

The reason for having an internal memory in the computer is that numbers and instructions can be transmitted between such a memory and the computing unit more rapidly than by other means. This enables the computations to proceed more quickly. High-speed computers exist such that a number or instruction in the memory can be transmitted to the computing unit in as short a time as 1 μsec. On the other hand, arithmetic operations such as addition can be performed in less than 1 μsec. Hence, it is generally the speed of the memory that limits the speed of a computer.

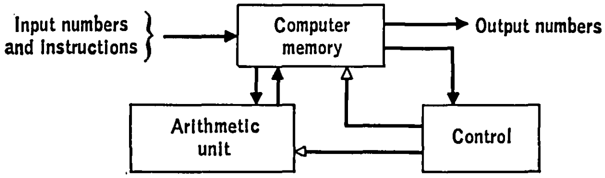


FIG. 2-3. Arithmetic unit and control.

The Computing Unit. It is seen from the above discussion that the computing unit has two functions: it must (1) obtain instructions from the memory and interpret them, as well as (2) perform the actual operations. Hence it is composed of two parts: the *control*, concerned with the former, and the *arithmetic unit*, concerned with the latter function (see Fig. 2-3).

An analogy can be made between the functioning of an electronic computer and the performance of hand computation. The computer's memory is analogous to the tabulation of numbers often kept during hand computations. The function of the computing unit is analogous to the actual performance of the hand-computational operations. Carrying our analogy with hand computation a little further, the arithmetic unit is analogous to the pencil and paper, abacus, slide rule, etc., while the control is analogous to the mental processes that take place in determining what operation is to be performed, choosing the proper arguments of the operation, keeping the proper sequence of operations, etc.

The Control. The control itself must perform two functions. It must (1) interpret the instruction; then, based on this interpretation, it must (2) tell the arithmetic unit what to do. The latter function is accomplished through the use of electronic control signals.

In accordance with these two functions, we can separate the part of the control that interprets, or *decodes,* the instructions, called the *instruction decoder,* from the part that generates the control signals, called the *control generator* (see Fig. 2-4).

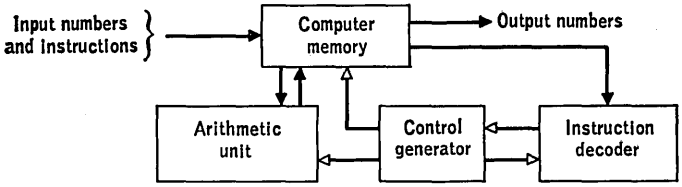


FIG. 2-4. Instruction decoder and control generator. (Solid-headed arrows indicate information; hollow-headed arrows indicate control signals.)

After an instruction has been transmitted to the instruction decoder, where it is interpreted, the control generator senses this interpretation and then produces signals that tell the arithmetic unit which operation to perform. It also generates signals that choose the proper numbers (arguments) from the memory and sends them to the arithmetic unit at the proper time; and when the operation has been performed, still other control signals take the result from the arithmetic unit back to the memory. After an instruction has been executed, the control generator produces signals that enable the next instruction to go from the memory to the instruction decoder. In this way the instructions are performed *sequentially.*

Input-Output. There are many means by which numbers and instructions can be initially introduced into and finally read out of the computer memory. They can be typed directly into the computer on specially adapted electric typewriters, read into the computer by means of punched paper tape on a teletypelike machine, punched on cards and read into the computer by means of a punched-card machine, memorized on magnetic tape and read into the computer by a specially built tape recorder, etc. When the computations have been completed, the resulting numbers can be read out of the computer by having the computer operate a typewriter, punch paper tape, punch cards, put pulses on magnetic recording tape, etc. Most computers have several such *input units* and *output units.* The *in-out selector* determines which unit will read information into or out of the computer; it is controlled by signals from the control generator (see Fig. 2-5). Each of the boxes in Fig. 2-5 represents electronic or electromechanical equipment. It is the purpose of this book to describe the function, use, and design of this equipment.

Summary. The computer's memory stores the instructions and the initial, intermediate, and final numbers involved in a computation. Instructions tell the computer what computations to perform. An instruction includes information about the kind of operation to be performed and the arguments involved in the operation.

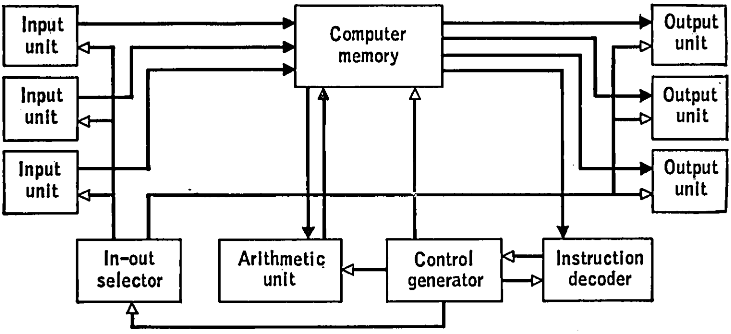


FIG. 2-5. Input-output.

The instructor decoder interprets an instruction. This interpretation is sensed by the control generator. Based on that information, the control generator produces control signals that transmit the proper numbers from the memory to the arithmetic unit and that tell the arithmetic unit what operations to perform on these numbers or arguments. When the results have been computed by the arithmetic unit, the control signals guide them back to the memory. Then the control signals bring the next instruction into the instruction decoder, and the cycle continues. When new data is to be read into the memory by means of the input units, the control generator directs the in-out selector to choose the proper input unit. The control generator takes similar action when computed data is to be read out of the memory by means of one of the output units.

EXERCISES

(*a*) What are the general purpose and function of an operation? An instruction? The computer memory? The computing unit? The arithmetic unit? The control? The instruction decoder? The control generator? Control signals? Input units? Output units? The in-out selector?

(*b*) If 2 and 3 are added to obtain 5, what are the arguments of this operation? The result?

(*c*) Write a sequence of instructions by means of which $a(b + cd)/f$ can be computed. Letting $a = 6$, $b = 4$, $c = 2$, $d = 3$, and $f = 5$, perform the sequence of instructions.

(*d*) Write a sequence of instructions by means of which $(1 - a^2)^3$ can be computed.

(*e*) What *four* functions do the control signals perform?

2-3. Functional Description of a Computer

The functional description of a computer involves consideration of further details about the computer's memory and the relation of an instruction to the memory. Two fundamental concepts of modern digital computers are involved in this relationship.

Words. The memory of a computer is composed of a large number of memory boxes, analogous to mailboxes. The information, or items to be memorized, is placed into these boxes as mail is placed into mailboxes. Each memory box, like a mailbox, has a name, or *address.* These addresses are usually numbers. For example, if the computer contains 100 memory boxes, their respective addresses might be the numbers from 1 to 100 (or 0 through 99). The information or items put into the memory boxes are called *words* and are analogous to the mail itself. However, *only one word can be stored in a memory box at a time.* As was seen in Sec. 2-2, there are two basic types of information, or words, that can be put into the boxes: words that are *numerical quantities* and words that are *computer instructions.*

At this point a short digression on computer terminology will prove helpful. The *memory box* of the previous paragraph is more correctly called an *address,* or alternatively a *memory address, memory location, storage location,* or *cell.* Instead of saying, "A word is in a memory box," computer personnel say, "The *contents* of an address is a word." Instructions are sometimes called commands; quantities are sometimes called data.†

An instruction word looks like a number, and there is no way to tell from the word itself whether it is a quantity or an instruction. The computer must be told explicitly which addresses contain instructions and which contain quantities. Ordinarily a word is treated by the control as an instruction and by the arithmetic unit as a quantity. The fact that it is only the *interpretation of a word* that distinguishes instructions from quantities is the first fundamental concept of modern digital computers. This is fundamental because it enables the computer to *operate* on instructions. That is, the computer itself can formulate or change an instruction by treating it as though it were a quantity; and then, at a later time during the computations, this changed instruction can actually be executed. In a certain sense, this gives the computer the capability of actually *writing* instructions as well as *executing* them; i.e., the computer can "tell" itself what to do—which is the closest that a computer probably ever comes to "thinking." More will be said about this in a later chapter (see Chap. 5).

Instructions and Addresses. To see the relationship between an instruction and the computer's memory, consider, for example, an instruction that tells the computer to *add* two numbers. The instruction does

† See First Glossary of Programming Terminology, *J. Assoc. Computing Machinery,* June, 1954; I.R.E. Standards on Electronic Computers: Definitions of Terms, 1956, *Proc. IRE,* vol. 44, no. 9, September, 1956.

not directly tell the computer to add the numbers themselves but rather tells the computer to add the numbers that are found as the *contents of addresses specified in the instruction.* If the numbers to be added are located in address α and address β, the instruction literally says, "Add the *contents of address* α to the *contents of address* β." Of course, the proper numbers to be added must have been previously placed into the addresses α and β. The same instruction might go on to tell the computer to put their sum into address γ and that the *next instruction* will be found as the contents of address δ. Hence, after our *add* instruction has been executed, the number that is the sum will be found as the contents of address γ. It must be certain that the contents of address δ is actually an instruction and not a number because the contents of address δ *will be interpreted* as if it were an instruction in any case.

The fact that instructions deal directly with addresses whose contents are to be operated on, rather than with the numbers themselves, is the second fundamental concept of digital-computing machines. This gives the computer an advantage that is exactly analogous to the advantage of performing algebra on symbols $x, y, \ldots$.

As an example of an instruction, suppose that we wanted to add 125 and 412 and that we put 125 into address 32 and 412 into address 34. Suppose that we want the sum to appear in address 36 and that the next instruction is the contents of address 41. Then our *add* instruction, when decoded, would say, "Add the contents of address 32 to the contents of address 34, and put the sum into address 36; then the next instruction will be found as the contents of address 41."

The Accumulator. Consider now what happens in the arithmetic unit while an instruction is being executed. In most computers only one word at a time can be transferred between the arithmetic unit and the memory. Hence, to perform an operation involving two arguments, the first argument must be transferred from the memory to the arithmetic unit and stored there temporarily while the second argument is being transferred. The special memory cell in the arithmetic unit for this purpose is called the *accumulator* (see Fig. 2-6). When the operation is performed, the result is formed in the accumulator before it is transmitted back to the memory.

Thus, in the above example, the contents of address 32, namely, 125, would be brought into the accumulator first; then this would be added to the contents of address 34, namely, 412, and the result, namely, 537, would be formed in the accumulator. Then 537 would be transmitted from the accumulator into address 36.

The Instruction Register and the Current-address Register. Next consider the instruction decoder that interprets an instruction. In order that the instruction decoder perform its function, it must constantly refer to the instruction being interpreted during the time control signals are being set up. To facilitate this, while an instruction is being executed it is stored in a special memory cell, called the *instruction register,* located in the instruction decoder (see Fig. 2-6).

There is another special memory cell located in the instruction decoder, called the *current-address register* (see Fig. 2-6). The contents of this register is (nearly) always the memory address from which the instruction being executed came. The reason for this is related to the fact that the address of the present instruction was given as part of the *previous* instruction.

For example, consider the *add* instruction described above. Suppose that this instruction were originally the contents of address 40. Then, while it is being executed, the contents of the instruction register will be the *add* instruction itself, and the contents of the current-address register will be 40. After the add operation has been executed (i.e., after the sum of 125 and 412 has been put into address 36) and just before the next instruction is read into the instruction register, the address of the next instruction, namely, 41, is recorded in the current-address register.

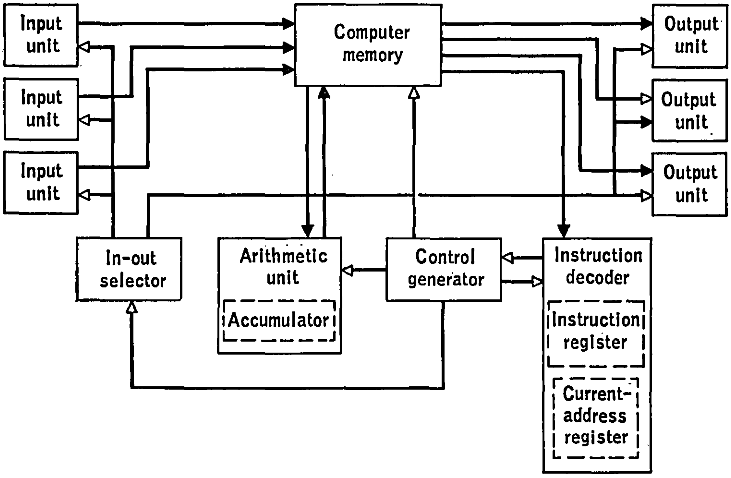


FIG. 2-6. Accumulator, instruction register, and current-address register.

The Four Phases. The functioning of a computer during the execution of an instruction is often summarized in terms of four phases. Assuming that an instruction has already been transmitted into the instruction register, *phase* 1 involves the transmitting of the first argument from the memory into the accumulator. During this phase the instruction decoder determines the address of the first argument from the instruction, and the control generator produces control signals that transmit the contents of this memory address into the accumulator. During *phase* 2 the second argument is brought to the arithmetic unit and the operation is performed. Both these functions are included in the same phase because most often the operation is performed as the second argument enters the arithmetic unit. In this phase the instruction decoder determines from the instruction the address of the second argument and what operation is to be performed; the control generator generates signals that set up the arithmetic unit to perform this operation and signals that transmit the contents of the address of the second argument into the arithmetic unit; the arithmetic unit performs the operation, generally leaving the result as the contents of the accumulator. During *phase* 3 the contents of the accumulator is transmitted into the memory address specified by the instruction. During *phase* 4 the address of the next instruction is ascertained, and the contents of the current-address register is changed to the new address; the contents of this new address is then transmitted into the instruction register as the next instruction. Then phase 1 for this instruction is initiated, and the process is repeated.

It should be observed that computers are often designed with phase 1 or 3 omitted. That is, in some computers the memory address of both arguments need not necessarily be specified in a single instruction, nor need the memory address into which the result is transmitted always be specified. In such computers multiple instructions would be equivalent in function to a single instruction as described above. This aspect of instructions is discussed in considerable detail in the next chapter, and we shall not press the point here, except to note that, although the principles described hold in general, the specific details vary with different computers.

Summary. There must be an instruction in the instruction register initially so that the whole process may begin, and there must be a way to stop the computer. These details will be considered in Chap. 4. The important points we wish to emphasize in this section are: (1) *instructions and quantities* are memorized by the computer as the contents of memory *addresses;* (2) they are indistinguishable from each other except by interpretation, and (3) instructions explicitly involve only addresses and tell the computer what to do with the *contents* of these addresses; (4) the functioning of a computer during the execution of an instruction can be summarized in terms of the four phases as described.

EXERCISES

(*a*) Discuss the first two fundamental concepts of modern digital computers, using accepted computer terminology.

(*b*) Consider the example of Sec. 2-2. Suppose that the value of a is found as the contents of address 50, of b as the contents of address 51, and of c as the contents of address 52; the value of the intermediate result x is to be put into address 53, and the final result y is to be put into address 54. How would instructions 1 and 2 of that example be written if these instructions themselves were found as the contents of addresses 55 and 56, respectively?

(*c*) Describe by naming specific addresses and special memory cells what happens during each of the four phases during the execution of the instruction given as an example in this section. What is the contents of the instruction register? What is the contents of the current-address register? What is the contents of the contents of the current-address register? (HINT: The contents of the current-address register is an address itself.)

(*d*) During phase 1 what is the relation between the contents of the instruction register and the contents of the current-address register?

2-4. Words and Pulses

Binary Numbers and Bits. Words are memorized in the computer as *binary numbers.* Hence it is necessary to divert our attention for a moment to a description of the binary notation for numbers. Numbers written in the binary system make use of only the symbols 0 and 1, called *bits,* just as numbers in the decimal system use the symbols 0, 1, 2, 3, 4, 5, 6, 7, 8, and 9, called *digits.* We say, for example, that 1101 is a binary number composed of 4 bits, just as 8,675 is a decimal number composed of 4 digits. In the decimal system the number 8,675 really means $(8 \times 10^3) + (6 \times 10^2) + (7 \times 10^1) + (5 \times 10^0)$, where $10^0 = 1$; analogously, in the binary system the number 1101 really means $(1 \times 2^3) + (1 \times 2^2) + (0 \times 2^1) + (1 \times 2^0)$, where $2^0 = 1$. Since $2^3 = 8$, $2^2 = 4$, $2^1 = 2$, and $2^0 = 1$, 1101 is $(1 \times 8) + (1 \times 4) + (0 \times 2) + (1 \times 1) = 13$ in decimal. A more general and thorough discussion of the binary-number system appears in Chap. 3. Our purpose here is simply to show that binary numbers are made up of bits, 0 and 1, and that a binary number is just another notation for our familiar decimal numbers.

Words as Pulse Trains. As can probably be foreseen by the reader at this point, the reason for using the binary numbers in digital computers is to enable the use of simplified electronic circuitry. For then only two signal voltage levels need be distinguished, the voltage level that is to represent 1 and the voltage level that is to represent 0. A word is memorized by the computer as a series, or train, of 1 and 0 signal voltage levels, often called *pulses,* that represent the bits of the word. The number of bits in a single word is called the *length* of the word. All the words within a particular computer are of the *same length;*† hence, so are the contents of all addresses in the computer memory. When the proper signals are generated by the instruction decoder, the train of pulses that represents a word goes from the memory address to the arithmetic unit. In some computers this is accomplished in a serial manner in time; i.e., the pulses flow past a fixed point at some fixed rate, say, 1 pulse/μsec. The *speed* of a computer is usually denoted by this pulse rate, or equivalently by the pulse frequency, i.e., 1 megacycle for our illustration.

It is often important to be able to denote a particular bit in a word by means of its position in the word. For this purpose we number the positions from *right to left.* Then Pn represents the nth bit of a word. For example, consider the 30-bit word in Fig. 2-7. Here $P1 = 0$, $P2 = 0, \ldots, P6 = 1$, $P7 = 0$, $P8 = 1, \ldots, P23 = 1$, $P24 = 0$, etc.

Time Synchronization and Clock Pulses. In the arithmetic unit two word pulse trains are operated on by electronic circuits to form a new word pulse train that represents the result of the operation. In the control the word pulse train is decoded and interpreted as an instruction by other electronic circuits. The circuits that accomplish these func-

† Modern computers are being developed to use different word lengths in different sections of the machine.

tions are called *gates*, or digital switching circuits. We shall have a little more to say about such gates in Sec. 2-5, and much more in the rest of the book. However, for the purposes of this section it suffices to mention that everything that happens to a word during the four phases takes place in these switching circuits, or gates.

It has been implied in the above discussion that there is some kind of synchronization of the pulses in a computer. In most computers this is indeed the case: all pulses are synchronized with respect to each other by means of *clock pulses* generated by a clock within the computer. The clock is an oscillator, with a frequency that is usually some integral multiple of the speed of the computer. Every gate is so connected to the clock of the computer that pulses are allowed to enter or leave the gates only at those times at which a clock pulse is received. The effect is analogous to an orchestra where the conductor beats out the timing, like the clock—the individual musicians, the gates, play notes, the pulses, according to this rhythm as directed by their music sheets, the design of the gating connections. At each beat of the clock all the pulses advance in the computer according to the gating connections, just as the musicians go on to the next musical beat. The purpose of this is to keep the phases of the pulses coordinated, in a manner to be described in detail in a later chapter. In addition, the clock pulses usually play an important role in reshaping the pulses, which become somewhat attenuated and distorted in passing through the gating circuits.

P30	P29	P28	P27	P26	P25	P24	P23	P22	P21	P20	P19	P18	P17	P16	P15	P14	P13	P12	P11	P10	P9	P8	P7	P6	P5	P4	P3	P2	P1
1	0	1	0	1	1	0	1	1	0	1	0	0	1	1	1	0	0	0	1	1	1	1	0	1	0	0	0	0	0

FIG. 2-7. Example of a 30-bit word.

Example of Adding. Let us consider what happens in the arithmetic unit when two numbers are added. To be specific, suppose that these numbers (where *An* and *Bn* label the bit positions) are

A30	A29	A28	A27	A26	A25	A24	A23	A22	A21	A20	A19	A18	A17	A16	A15	A14	A13	A12	A11	A10	A9	A8	A7	A6	A5	A4	A3	A2	A1
0	0	0	0	1	0	1	1	0	0	0	0	0	0	1	1	1	1	1	1	1	0	0	1	1	1	0	0	1	1

B30	B29	B28	B27	B26	B25	B24	B23	B22	B21	B20	B19	B18	B17	B16	B15	B14	B13	B12	B11	B10	B9	B8	B7	B6	B5	B4	B3	B2	B1
0	0	0	0	0	1	0	1	1	1	0	1	1	1	0	1	0	1	1	1	1	0	1	0	1	0	0	0	0	0

(The decimal equivalents are, for *A*, 46,202,483 and, for *B*, 24,600,224. See Chap. 3 for decimal-to-binary conversion methods.) Suppose that we are in phase 2; i.e., the first argument has already been transmitted into the accumulator. Suppose also that the pulse rate of the computer is 1 μsec; then the bits advanced once each microsecond. Figure 2-8 shows the situation at the start of phase 2; Fig. 2-9 shows the situation 1 μsec later; Fig. 2-10 after 2 μsec; Fig. 2-11 after 10 μsec; Fig. 2-12 after

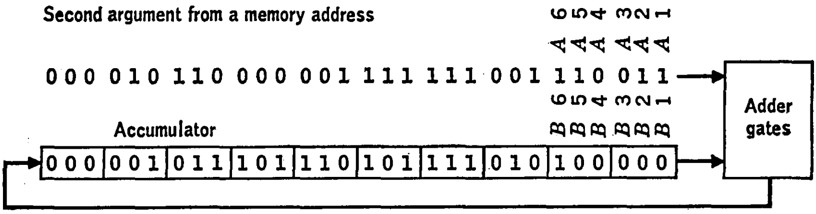


FIG. 2-8. Start of phase 2.

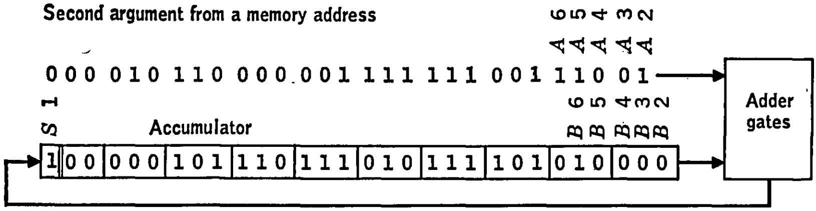


FIG. 2-9. One microsecond after start of phase 2.

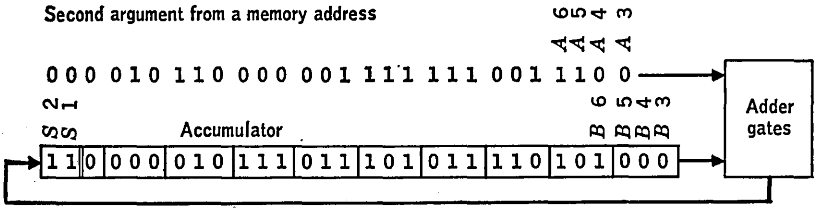


FIG. 2-10. Two microseconds after start of phase 2.

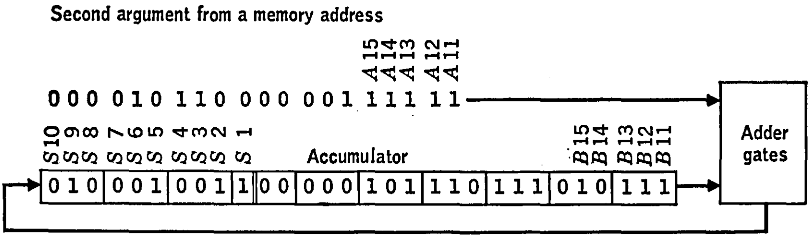


FIG. 2-11. Ten microseconds after start of phase 2.

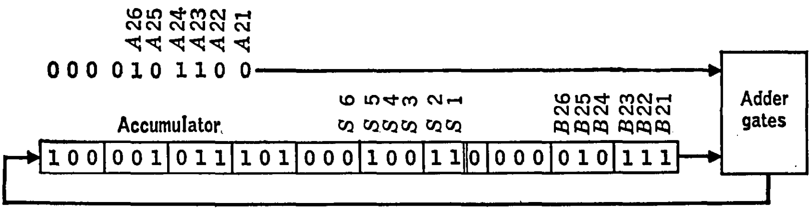


FIG. 2-12. Twenty microseconds after start of phase 2.

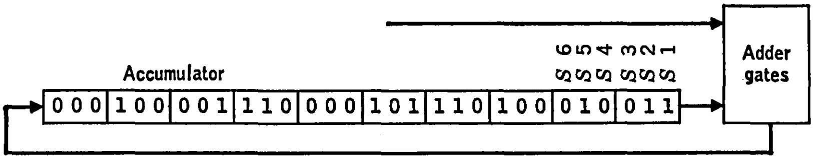


FIG. 2-13. Thirty microseconds after start of phase 2.

20 μsec, and Fig. 2-13 after 30 μsec. As the two arguments advance bit by bit into the adder, the result is formed bit by bit; the partial results are stored in the accumulator behind the partially used first argument as shown. The double line separates the partial first argument from the partial result so far formed in the accumulator. (The partial result is to the left of the double line; the remaining first argument is to its right.) In Fig. 2-13 the complete result has been formed in the accumulator, namely:

S30 S29 S28	S27 S26 S25	S24 S23 S22	S21 S20 S19	S18 S17 S16	S15 S14 S13	S12 S11 S10	S9 S8 S7	S6 S5 S4	S3 S2 S1
0 0 0	1 0 0	0 0 1	1 1 0	0 0 0	1 0 1	1 1 0	1 0 0	0 1 0	0 1 1

(which in decimal is 70,802,707). The computer is then ready to start on phase 3.

Figures 2-8 to 2-13 illustrate what would happen in a so-called "serial" computer. However, there are other, "parallel" methods for adding. Both serial and parallel adders are discussed in detail in later chapters. Our purpose here was merely to give a general idea of how word pulse trains travel through gates in a synchronous fashion during a computation.

Instruction Decoding. We have given an example of how numerical quantities are gated. Next let us illustrate the method by which an instruction word is written and decoded. Suppose that the words of a hypothetical computer are each 30 bits in length, as above. Let the leftmost six bits, *P*30, *P*29, *P*28, *P*27, *P*26, *P*25, represent the operation code, i.e., tell what the instruction is to do: add two numbers, or form the product of two numbers, etc. Suppose, for concreteness, that the binary number 101 011 is the code for "add" (see Fig. 2-14). Let the next six bits, *P*24, . . . , *P*19, represent the address of one of the numbers to be added; *P*18, . . . , *P*13 the address of the other number; and *P*12, . . . , *P*7 the address in which the sum is to be stored. The

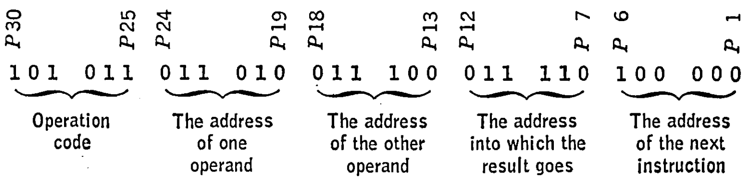

P30
P25
P24
P19
P18
P13
P12
P 7
P 6
P 1
101 011
011 010
011 100
011 110
100 000
Operation code
The address of one operand
The address of the other operand
The address into which the result goes
The address of the next instruction

FIG. 2-14. Example of an instruction word.

last six bits, $P6, \ldots, P1$, tell where the next instruction is located in the memory. When the instruction word goes into the instruction decoder, it is stored in the instruction register. This register has a circuit for each bit being stored, with an output wire so that, if a 1 is stored, the circuit puts out the 1 signal voltage level on the wire and the 0 signal voltage level for a zero. The outputs on these wires are sensed by gating circuits, and the instruction is thus decoded (see Fig. 2-15).

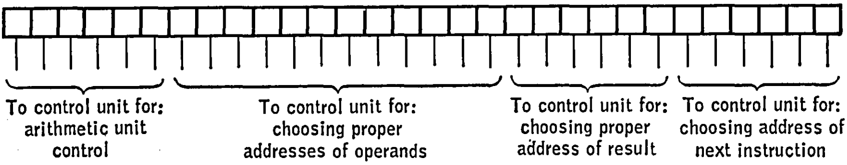

To control unit for: arithmetic unit control
To control unit for: choosing proper addresses of operands
To control unit for: choosing proper address of result
To control unit for: choosing address of next instruction

FIG. 2-15. Instruction register.

The control unit generates control pulses which are also *binary in character*, i.e., are composed only of the 0 and 1 voltage levels. In fact, almost all the circuitry in a digital computer is concerned with only these two voltage levels. This distinction can be made as to the function of pulses in the computer: some pulses represent words (numbers or instructions); other pulses represent control signals. Such a distinction was made in the block diagram of a computer, where solid-headed arrows represented the flow of word pulses and hollow arrows represented the flow of control-signal pulses.

EXERCISES

(*a*) Define: length of word; pulses; speed of computer; gates; clock pulses.

(*b*) In the addition example given in this section what is the contents of the accumulator 15 μsec after the start of phase 2? 25 μsec after?

2-5. Logical Building Blocks

Almost the entire computer is composed of only three basic kinds of simple circuits, often called building blocks. Many of each kind of these circuits are connected to make the computer. The functioning of the computer depends only on *how* these basic circuits are *wired together*. Hence the electronic aspect of computers can be divided into two fields: (1) the design of these three very basic circuits; and (2) the design of the wiring connections between these basic circuits for proper functioning of the computer. The three kinds of basic circuits are called *and gates*, *or gates*, and *not gates*. Although the internal electronic design of these building blocks and the theory of how to design wiring connections between them comprise the latter parts of this book, it is felt that some background material on these subjects will aid the reader to visualize in a general way the over-all structure of computers.

The Three Building Blocks. An *and gate* is a circuit with two (or more) inputs and one output so that a 1 signal voltage level at the output will

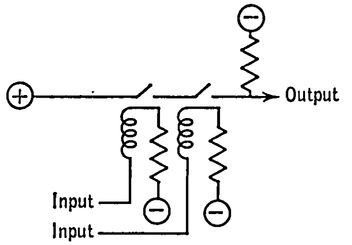
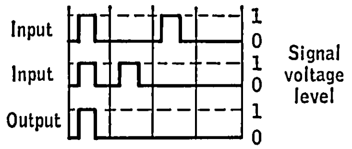
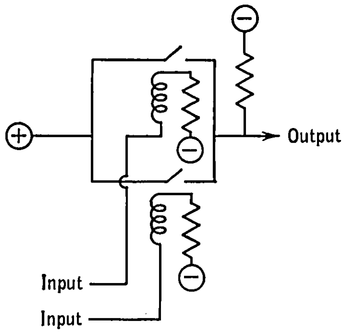
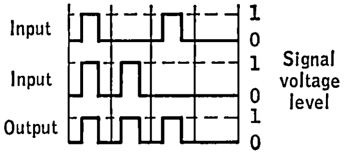
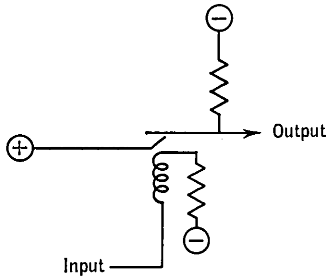
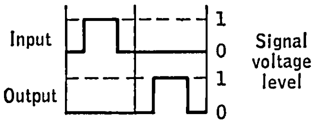
	Circuit diagram	Voltage-level diagram
And gate		 Input Input Output 1 0 1 0 1 0 Signal voltage level
	Only when current is flowing in both relay windings will the voltage level of the output rise	The four possible combinations of input-signal voltage levels are indicated with the resulting output voltage level
Or gate		 Input Input Output 1 0 1 0 1 0 Signal voltage level
	Only when current is flowing in either or both of the relay windings will the voltage level of the output rise	The four possible combinations of input-signal voltage levels are indicated with the resulting output voltage level
Not gate		
	Only when current is not flowing in the relay windings will the voltage level of the output rise	The two possible input-signal voltage levels are indicated with the resulting output voltage level

FIG. 2-16. The three building blocks as constructed using relays.

occur only when a 1 signal voltage level appears on all inputs. An *or gate* has two (or more) inputs and one output; it will have a 1 signal-voltage-level output only whenever a 1 signal voltage level appears on any one or more inputs. A *not gate* has one output and one input and will have a 1 signal-voltage-level output only whenever a 0 signal voltage level appears on its input. Such circuits can be designed using relays, electron tubes, diodes, magnetic cores, transistors, etc. Figure 2-16 illustrates the possible construction of such circuits using relays.

Flip-flops. There is one other type of circuit, called a *flip-flop,* commonly used in computers. A flip-flop is a circuit that temporarily stores a single bit. Its single output can be either a unit or zero voltage level, and this level will remain fixed until some change occurs at its inputs. Generally a flip-flop has two inputs: a pulse on one input will cause the output to rise to the unit signal level; a pulse on the other input will cause the output to fall to the zero signal level. Registers are often made of a number of these flip-flops, one flip-flop being used for each bit of the word (see Fig. 2-15). Flip-flops are ordinarily used for registers owing to their high speed of reaction as compared with other memory devices (see below). They are rarely used for the computer's memory because of their high cost and bulk. These points will be discussed in considerable detail in Part 5.

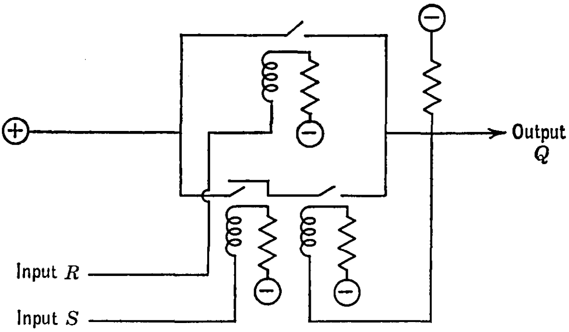


FIG. 2-17. Relay flip-flop.

However, it may prove helpful here to illustrate how a flip-flop can be made from relays (see Fig. 2-17). When input R is a unit voltage level and closes its relay, the output Q becomes a unit voltage level. In this case the output relay Q closes, and the unit-voltage-level output is continued independently of relay R until relay S is activated by a unit voltage level at input S. When relay S is activated, the output Q falls to the zero voltage level and relay Q reopens; the zero voltage level is maintained until input R is a unit voltage level. In the circuit shown the simultaneous occurrence of unit voltages on both R and S inputs will result in the R input overriding the S input.

Example of Gating Numerical Quantities. Now we shall show, in a quite preliminary way, how the three building-block circuits can be connected so as to perform some function. Consider as an illustration the design of a circuit that might be found as part of the arithmetic unit of a computer. The circuit uses two numbers to make a third number, in a serial fashion. That is, the circuit has two inputs such that *P*1 of one number enters on one input wire at the same time as *P*1 of the other enters on the other input wire. The output corresponding to these inputs is *P*1 of the output number. Then, say 1 μsec later, *P*2 of each number enters the input wires of the circuit, and *P*2 of the output number is made, etc. In this way, the 30 bits of the output number are made serially as the corresponding bits of each of the input words pass into the circuit. In our example the output number is to be constructed according to the following rule: *Pn* of the output number is 1 if *Pn* of both the input numbers is the same, i.e., either both 1 or both 0; otherwise *Pn* of the output number is 0. Thus the circuit is to behave as follows:

$$\left.\begin{array}{l}\text{Input 1}\\\text{Input 1}\end{array}\right\}\text{output 1}\qquad\qquad\left.\begin{array}{l}\text{Input 1}\\\text{Input 0}\end{array}\right\}\text{output 0}$$

$$\left.\begin{array}{l}\text{Input 0}\\\text{Input 0}\end{array}\right\}\text{output 1}\qquad\qquad\left.\begin{array}{l}\text{Input 0}\\\text{Input 1}\end{array}\right\}\text{output 0}$$

For example,

If one input number is	101110	011001	101101	011011	001100
and the other is	011100	100001	111111	000000	101010
then the output number is	001101	000111	101101	100100	011001

Such a circuit is called an "equalizer." Figure 2-18 indicates the connections that are necessary to make this circuit.

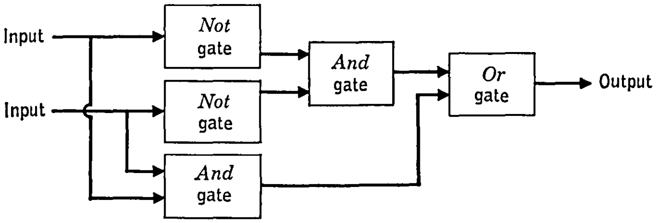


FIG. 2-18. Equalizer circuit.

In order to see that this circuit actually performs the task, examine the four diagrams of Fig. 2-19, one corresponding to each of the four input possibilities, where the signal voltage level is indicated at each connection.

Example of Gating Control Pulses. The above example illustrated gating of word pulses. We shall now illustrate gating of control pulses (see Fig. 2-20). Suppose that there were at least two circuits in the arithmetic unit, an equalizer and an adder. Then, according to Fig. 2-20, the two arguments to be operated on (i.e., the one coming from the mem-

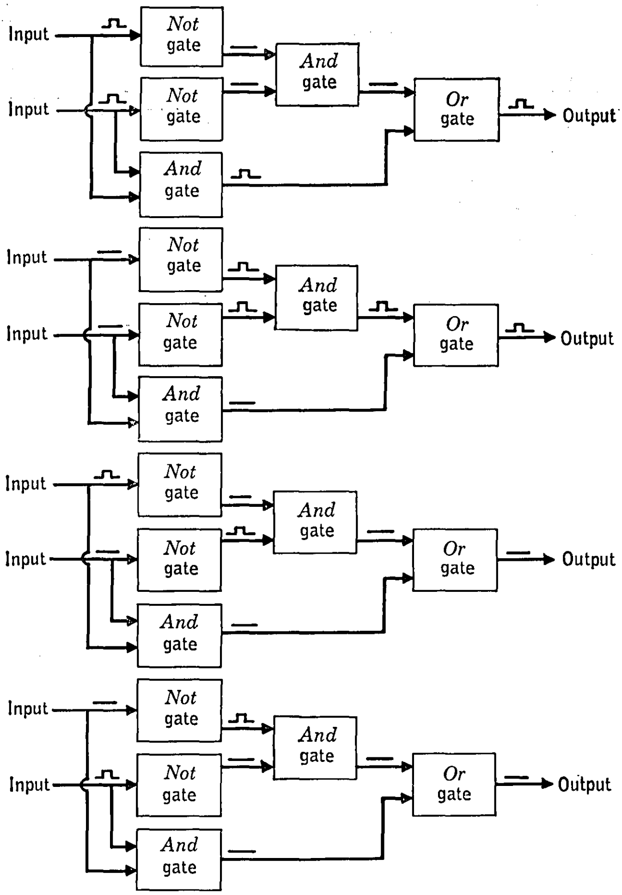


FIG. 2-19. Analysis of equalizer circuit.

ory and the other already in the accumulator) will enter both the adder and the equalizer. But, because of the control signals, the result of only one of these circuits will be brought back to the accumulator. For example, if an instruction says that the numbers are to be added, then a 1 signal voltage level must be in the control wire going to the adder's output gate and a 0 signal voltage level must be in the control wire going to the equalizer's output gate. Then, as can be clearly seen, no signals from the equalizer's result will get through the equalizer's output gate, but the adder result will go through the adder's output gate.

Next let us illustrate how these control signals are generated. Suppose that

101 011

is the operation code for addition of the instruction. And suppose that

001 010

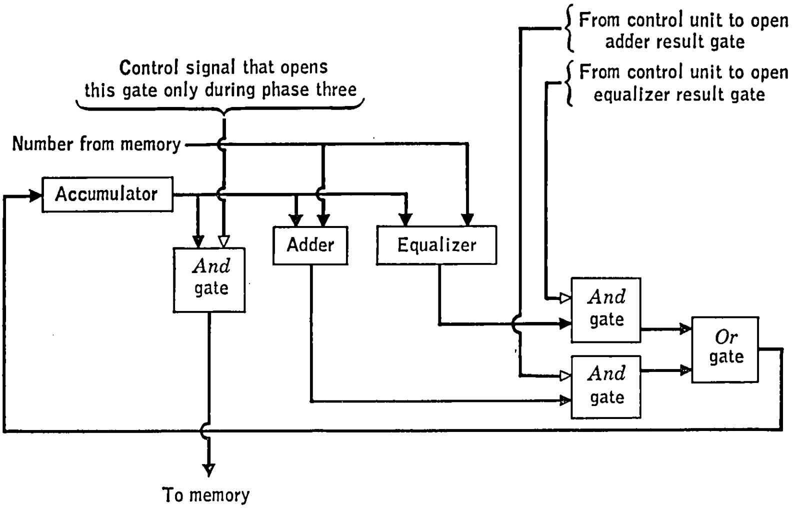

From control unit to open adder result gate
From control unit to open equalizer result gate
Control signal that opens this gate only during phase three
Number from memory
Accumulator
And gate
Adder
Equalizer
And gate
And gate
Or gate
To memory

FIG. 2-20. Example of gating of control pulses.

is the operation code for "equalize." The signals for these positions of the instruction register (that is, *P*30–*P*25) are gated by means of an *and* gate that has more than two inputs. Such a gate will produce a 1 signal voltage level *only* when there is a 1 *signal voltage level on all its inputs.* From Fig. 2-21 it can be seen that the *add* control signal will occur only when 101 011 is in *P*30–*P*25 of the instruction register; and the equalizer control signal will occur only when *P*30–*P*25 is 001 010. The wires marked *CP* represent clock-pulse wires. We have included these in the diagram to illustrate how they enter every *and* gate. Thus we have illustrated the way in which signals can be generated to control which operation result will be allowed to go back to the accumulator.

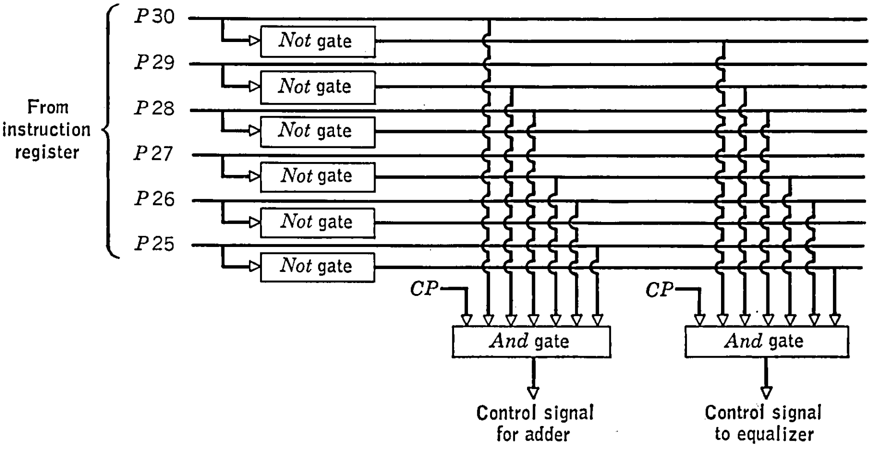


FIG. 2-21. Generation of *add* and *equalizer* control signals.

EXERCISES

(*a*) How can the flip-flop of Fig. 2-17 be constructed from *and*, *or*, and *not* gates?

(*b*) Draw the voltage-level diagram for the flip-flop of Fig. 2-17.

(*c*) Suppose that in Fig. 2-20 the number from the memory is

000 010 110 000 001 111 111 001 110 010

and that the contents of the accumulator at the start of phase 2 is

000 001 011 101 110 101 111 010 100 000

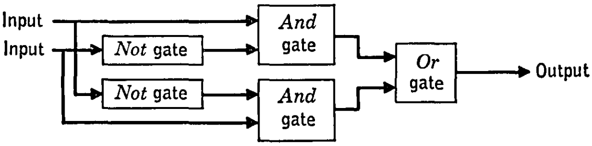
Then, if the adder result gate is open during phase 2 (and the equalizer gate closed), what will be the contents of the accumulator at the end of phase 2? (HINT: See Sec. 2-4.) What will it be if the equalizer result gate is open and the adder gate is closed?

(*d*) Why do we need a gate that is open only during phase 3, as shown in Fig. 2-20?

(*e*) Why do we need the clock pulses shown in Fig. 2-21?

(*f*) How is Fig. 2-21 related to Fig. 2-15? Draw the proper composite figure.

(*g*) Analyze the output of the given circuit by a method similar to that used in Fig. 2-19.



EXERCISE *g*

(*h*) If the following numbers are inputs to the circuit of Exercise *g*, then what will be the output number?

Input number	101110	011001	101101	011011	001100
Other input number	011100	100001	111111	000000	101010

2-6. Input, Output, and Memory Systems

Input Information. As we have seen, there are many ways to get information into and obtain information from a digital computer. In order to give a more concrete picture of the process, we shall describe in some detail the procedures a person would follow in order to put information into a computer. The input information usually consists of a list of numbers and a list of instructions telling the computer what operations to perform on these numbers. For example, a problem might be to compute the trajectory of a missile (i.e., to construct a table giving the position of the missile at each second during its flight) given its initial velocity and angle of take-off. Suppose that the equation of the trajectory is

$$x = (V_0 \cos \theta)t \qquad y = (V_0 \sin \theta)t - \tfrac{1}{2}gt^2$$

where x,y = coordinates of missile at any time
V_0 = initial velocity
θ = angle of take-off from horizontal

Among the list of numbers that must be entered into the computer are the values of V_0, cos θ, sin θ, g, and the constant 1 (the time interval of 1 sec). A list of instructions must also be read into the computer telling it how to compute x and y. All these numbers and instructions must be entered into the computer in binary form. The first problem under consideration in this section is just how this might be accomplished.

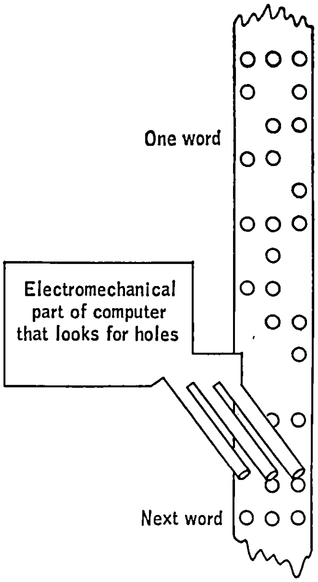


FIG. 2-22. Punched paper tape.

Punched Paper Tape as an Input Vehicle. Suppose that punched paper tape is to be used for this process. An electromechanical device exists so that a hole in the tape is interpreted by the computer as a 1, no hole as a 0. Consider a paper tape with positions for three holes per line. If there are 30 bits to a word, then 10 lines are needed per word. Figure 2-22 illustrates how the following word would look on a tape:

111 101 011 110 001
111 010 110 011 001

One method for getting the proper holes in the paper tape is to use a paper-tape hand punch machine. This manual punch machine has keys: pushing a single key will punch one line. The keys might be numbered from 0 to 7, and the array of holes corresponding to each key might be as given in Fig. 2-23. Hence the word above would be punched by pushing the following keys in succession:

7, 5, 3, 6, 1, 7, 2, 6, 3, 1

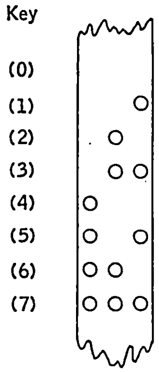


FIG. 2-23. Punched-paper-tape code.

The person who prepares the list of numbers and instructions for a problem is called a *programmer*, or *coder*, and the list itself is called a *coding sheet.* The *code* is written as a list of computer words. However, the coder does not write the words in binary form but instead puts down, as a shorthand, the numbers of the keys that are to be punched. Hence the coding sheet, instead of the word above, would have

75 36 17 26 31

The punch operator would push these keys in succession, and the paper tape would be punched in a binary mode. After the punching is complete, the paper tape is run through the computer paper-tape reader, which translates the holes into the units and zeros that are electronically stored in the computer's memory. The computer then calculates the results, and they are read out—

perhaps by having the computer punch a paper tape. This paper tape would then be put into a machine that does the reverse of punching, i.e., that looks at a paper tape and prints on paper the numbers that appear there, in a manner similar to a ticker-tape machine. Figure 2-24 is a photograph of a paper-tape punch machine and a computer paper-tape reader.

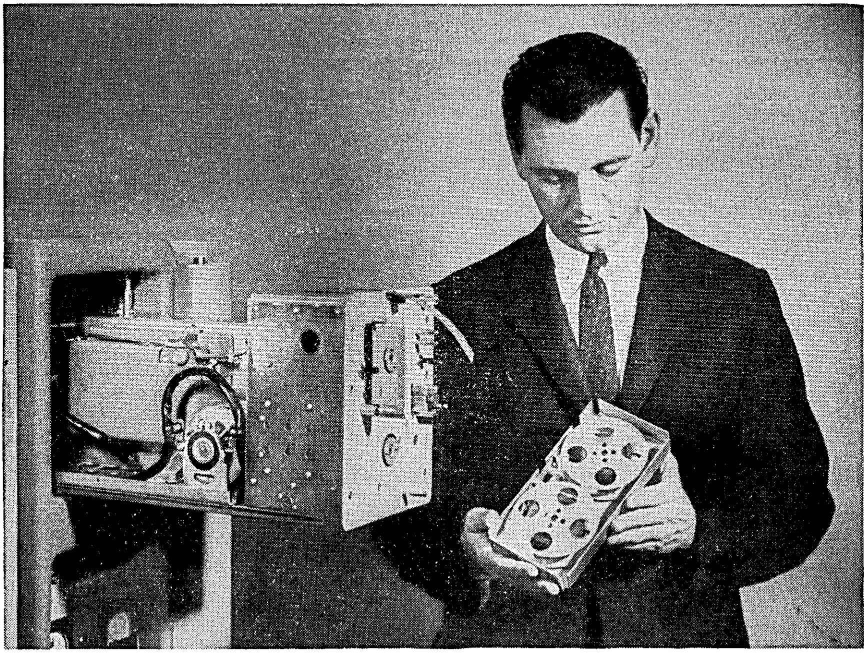


FIG. 2-24. Paper-tape punch and reader. (*Photograph courtesy Bendix Computer Division of Bendix Aviation Corp.*)

Other Input-Output Methods. Although in Part 5 of the book we devote most of Chap. 22 to input-output methods, mentioning some of these here will serve to orient the reader. Besides punched paper tape there are punched cards: the cards are punched in much the same manner as the paper tape, and the device that reads the cards into the computer likewise senses the holes and interprets these in terms of electronic pulses (see Fig. 2-25). A typewriter may be rigged with relays so that when a key is pushed a code is produced by the relays and transmitted directly into the computer. Such methods are often found too slow. A faster method for putting a program or code into a computer is by means of magnetic tape, like sound-recording tape (see Fig. 2-26): here a small magnetized area on the tape is interpreted by the computer as a unit, no magnetized area as a zero. However, before this magnetic tape is read into the computer, the program must be first "written" onto the tape. This may be accomplished on a machine not associated with the computer itself, often called an "inscriber." There are many varieties of inscribers; some enable the code to be put on the magnetic tape directly from a typewriter, others from previously punched paper tape or cards. For example, one procedure may require that the code first be punched on paper tape and the paper tape "inscribed" onto magnetic tape, which is finally read into the computer. Table 2-1 gives some idea of the speeds in bits per second at which information can be transformed from one medium to another. Knowing the number of bits per word, one can then calculate the number of words per minute that can be so transformed.

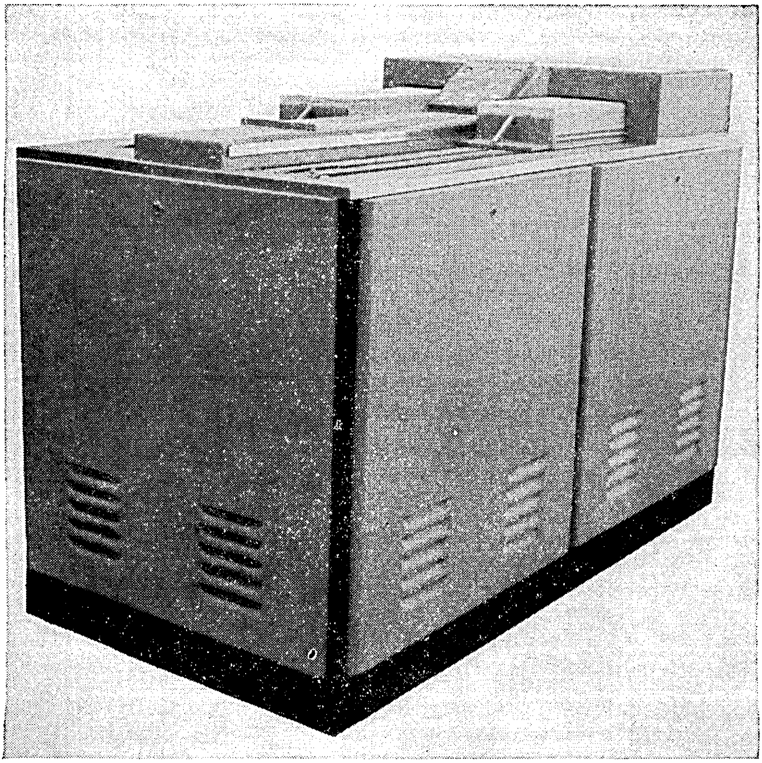
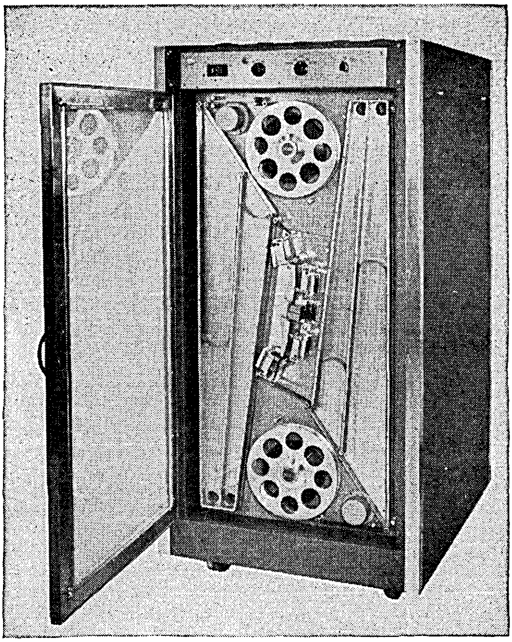


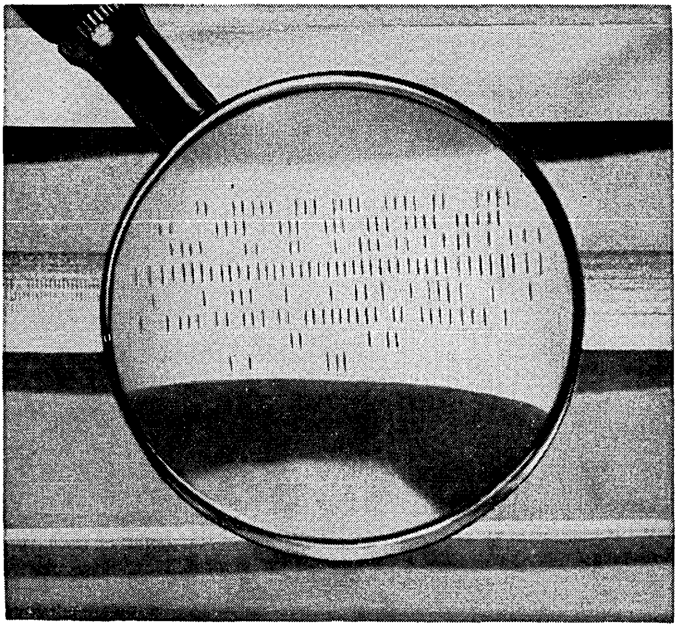
FIG. 2-25. Punched-card reader. (*Photograph courtesy National Cash Register Co.*)

There are also some unusual means for putting information into the computer. One is a reading machine that can translate digits and letters, as they appear on a printed page, into a binary code and write this code into the memory.

Most of the methods for taking information from the outside world into the computer's memory can be reversed to serve as methods for reading information out from the computer's memory to the outside world in a form that is readily understandable. Hence the computer can work a typewriter, punch paper tape, punch cards, or write onto



(*a*)



(*b*)

FIG. 2-26. (*a*) Magnetic-tape unit; (*b*) magnetic tape, showing location of bits. (*Photographs courtesy Potter Instrument Company, Inc.*)

TABLE 2-1. ORDER OF MAGNITUDE OF RATES

From	To	
Paper tape	Computer memory	2,000 bits/sec
Punched cards	Computer memory	3,000
Typewriter	Computer memory	40
Magnetic tape	Computer memory	100,000
Paper tape	Magnetic tape	2,000
Punched cards	Magnetic tape	3,000
Typewriter	Magnetic tape	40
Computer memory	Paper tape	300
Computer memory	Punched cards	2,000
Computer memory	Magnetic tape	100,000
Computer memory	Typewriter	600 characters/min
Paper tape	Typewriter	600
Punched cards	Printer	10,000
Magnetic tape	High-speed printer	100,000
Computer memory	High-speed printer	100,000
Computer memory	Cathode-ray-tube printer	500,000

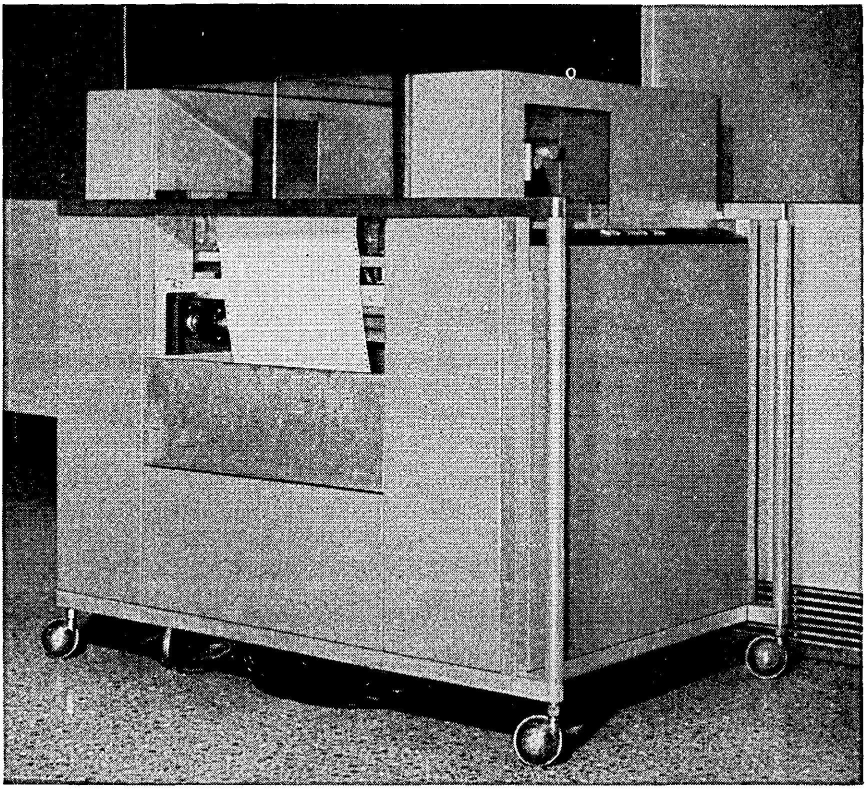


FIG. 2-27. High-speed printer (600 lines per minute, 120 characters to the line). (*Photograph courtesy Radio Corporation of America.*)

magnetic tape. Of course the "*out*scriber" is required to transform the information from punched paper tape, punched cards, or magnetic tape into the letter and number characters we are more familiar with. In addition there are methods using so-called "high-speed" printers and specially made cathode-ray tubes (see Figs. 2-27 and 2-28) that, like the typewriter, display numbers and letters directly from the computer.

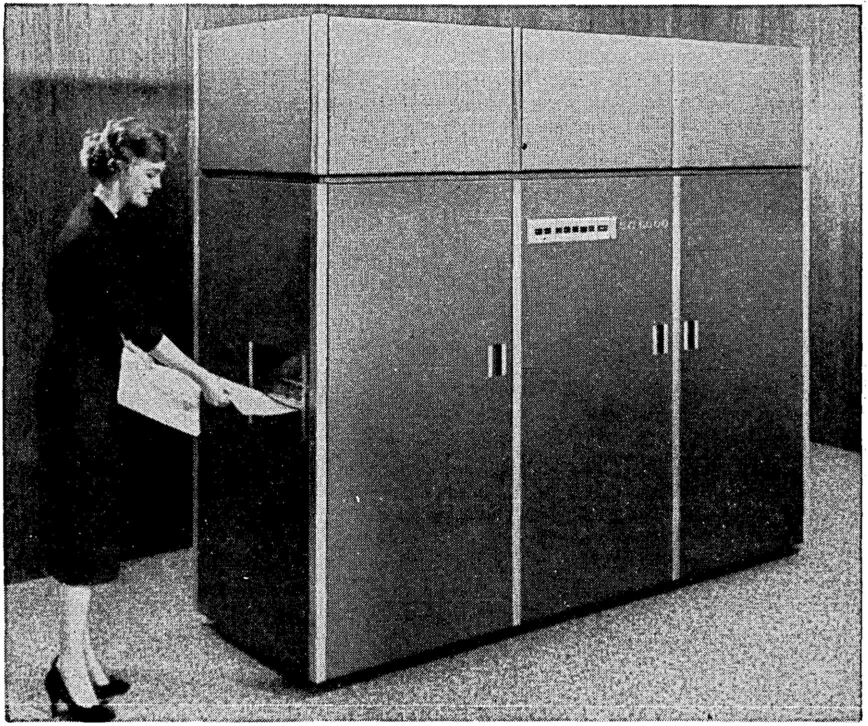


FIG. 2-28. High-speed printer using Charactron tube (4,680 lines per minute, 120 characters to the line). (*Photograph courtesy Stromberg-Carlson Division of General Dynamics Corp.*)

Other outputs from a computer may be obtained from attaching oscilloscope or television tubes to the computer's memory. Then the computer literally can draw maps, can indicate the movements of dots representing aircraft, etc., or can display other kinds of pictures.

Acoustic-delay-line Memory. Words read into the computer go into the computer's memory. The function of a computer memory is to store the pulses of each word in such a way that they may be called into the arithmetic unit rapidly, upon generation of the proper control signals. In addition, the computer memory must be able to accept rapidly the pulse train of a word for storage. The speed of modern computers is essentially limited by the speeds of access to their memories.

There are many ways for memorizing words within a computer, and some of these will be considered in detail in Part 5 of this book. However, in order to gain a broader point of view at this time, it might be helpful to describe briefly one of these memory methods. Consider as an example the acoustic mercury delay-line memory. This memory consists of a set of glass tubes, each about 2 ft long and ¾ in. in diameter (see Fig. 2-29). Each tube is filled with mercury and has a quartz crystal at each end. Electrical pulses make the crystal at one end vibrate; the disturbances propagate down the tube in the mercury and vibrate the other crystal; these latter vibrations are changed by the crystal into electrical impulses, which are then sent to an amplifier. The amplifier reshapes the pulses and sends them to the other crystal, etc. Thus the pulses of a word travel serially, i.e., consecutively, down the mercury, through the amplifier, and around. The proper control signals will allow the word pulse train to go to the arithmetic unit or

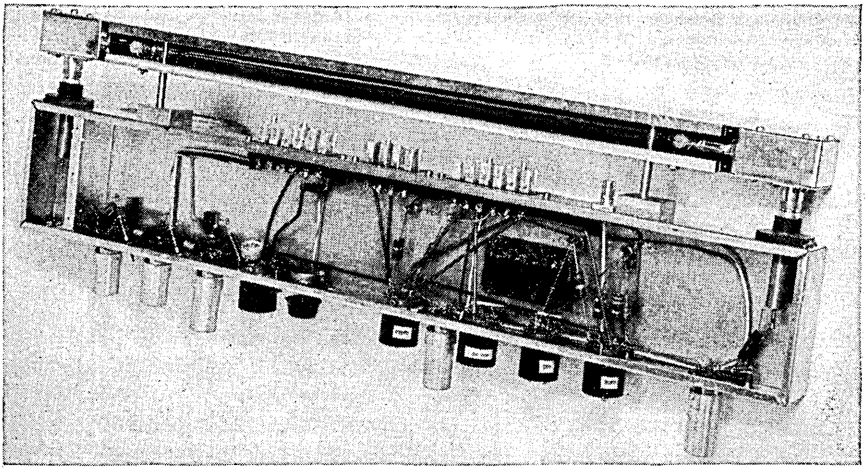


FIG. 2-29. Acoustic mercury delay-line memory. (*Photograph courtesy National Bureau of Standards.*)

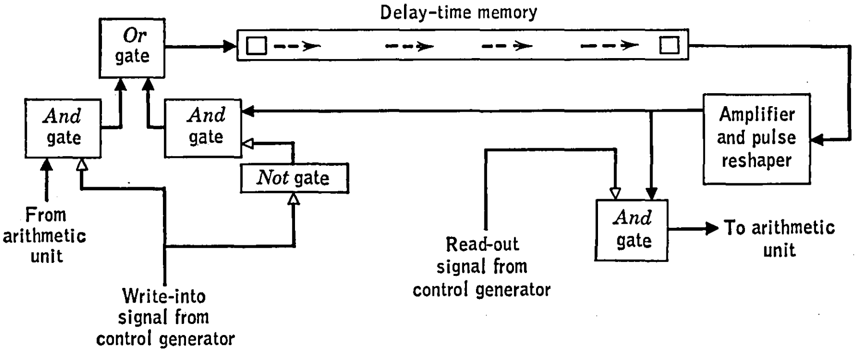

Delay-time memory
Or gate
And gate
And gate
Not gate
Amplifier and pulse reshaper
And gate
From arithmetic unit
Write-into signal from control generator
Read-out signal from control generator
To arithmetic unit

FIG. 2-30. Read and write control signals.

come from the arithmetic unit into the circuit. Of course the timing of these processes is critical, because when it is desired to read a word out of the memory, the gating must operate precisely as $P1$ of that exact word is picked up by the second crystal. To maintain the proper speed of propagation in the mercury, the temperature of the line must be held within a narrow band.

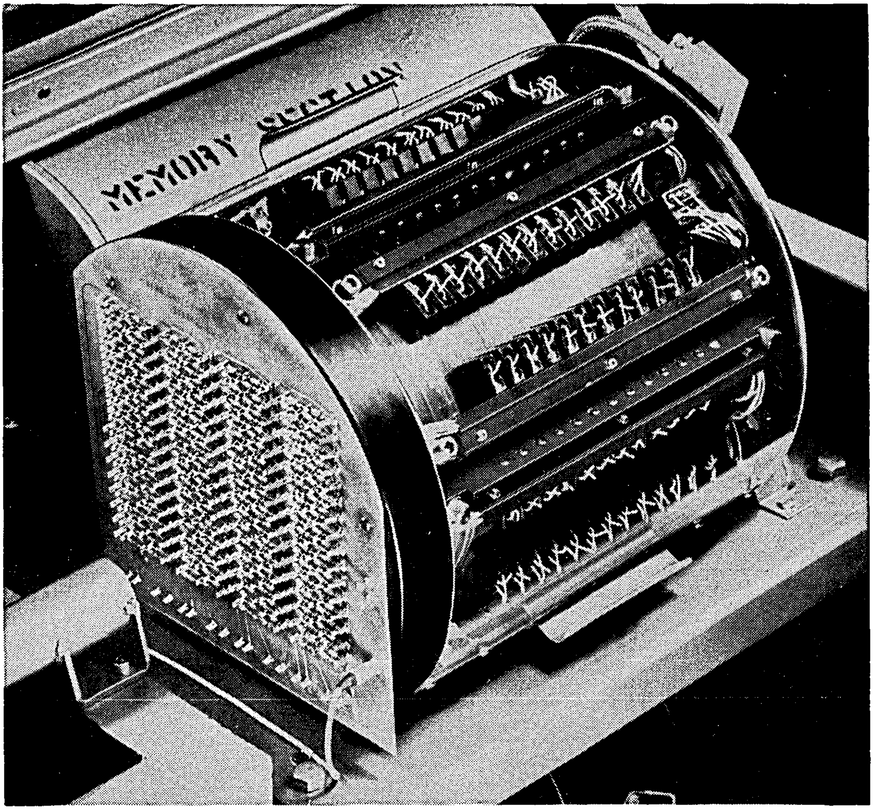


FIG. 2-31. Magnetic drum. (*Photograph courtesy Royal McBee Corporation.*)

An important point that is true for practically all kinds of memories is that when a word is *read out of a memory* the word is *still stored in the memory* (unless otherwise specifically stated by the instruction—see Chap. 4). Thus in Fig. 2-30 the read-out control signal opens the gate to the arithmetic unit but does not prevent the recirculation of the word through the memory. On the other hand, when a word is written *into the memory* from the arithmetic unit, *the previous word of that address is automatically erased and replaced by the new word.* In Fig. 2-30 the write-into control signal opens the gate from the arithmetic unit at the same time it closes the recirculation gate.

Other Memory Systems. There are several other memory systems that will be described in Chap. 22 of Part 5. However, we shall mention some of these here for orientation. Magnetic tape, described above as an input-output system, can as well be used for a computer memory. A similar magnetic-pulse technique is incorporated into magnetic drums, which are simply rapidly rotating cylinders with magnetizable surfaces (see Fig. 2-31). Faster memories are constructed from magnetic cores, specially prepared from ferrites having a square hysteresis loop, thereby presenting two stable states. Each core then represents one bit of a word (see Fig. 2-32). An even faster memory has been constructed, using diodes and capacitors. As mentioned above, a memory with the fastest reaction time is one constructed using flip-flops to record bits. The time required to transmit one computer word out of the memory to where it will be used is called the *memory access time* (see Table 2-2).

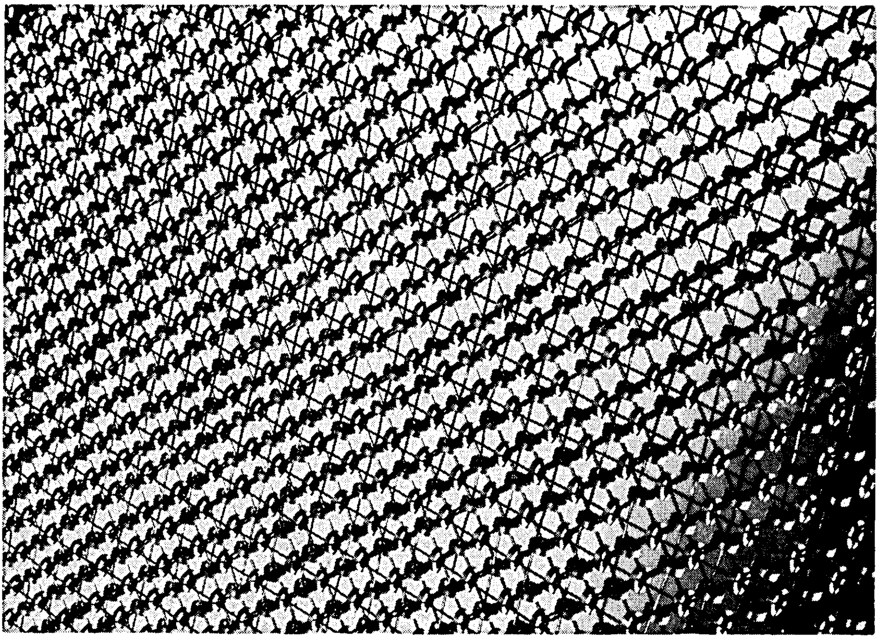


FIG. 2-32. Magnetic cores. *(Photograph courtesy International Business Machines Corp.)*

In an acoustic-delay-line memory the stored bits come out serially. If more than one word is stored in a single line, the computer must wait

TABLE 2-2. ORDER OF MAGNITUDE OF MEMORY ACCESS TIME

Memory system	*Time required to transmit a 30-bit word*
Magnetic tape	5 msec + time necessary to position tape
Magnetic drum	10 msec
Acoustic delay line	100 μsec
Magnetic core	5 μsec
Diode capacitor	1 μsec
Flip-flop register	1 μsec

for the proper word to pass the read-out gate. In the magnetic-tape memory the whole tape must be searched, and a great deal of time is lost just winding and rewinding tape. For the drum the computer must wait for the word to rotate to the proper read-out position. The magnetic-core arrangement enables any word to be read out with no delays except those inherent in the circuitry involved. A word is read from a core memory in parallel; i.e., all the bits are transmitted to the arithmetic unit at the same time. A similar situation occurs with diode-capacitor memory or with a memory composed of a set of flip-flop registers.

Unfortunately, the faster the memory, the more it costs per bit. Hence a computer usually has several kinds of memory. Most commonly, magnetic tapes and drums are used for the so-called "low-speed" memory of a computer, while a magnetic-core memory or diode-capacitor memory is used as the "high-speed" memory. Often there is also included in the larger computers a battery of flip-flop registers for the really high-speed work. In this way, various parts of a large program can be stored in the relatively low-speed memory system of a computer and then transmitted to the high-speed memory when the actual computations of this part of the program are to occur. We shall have more to say about this later (Chaps. 5, 7, and 8).

EXERCISES

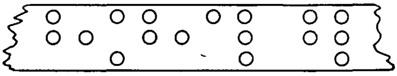
(*a*) What keys of a paper-tape puncher would be pushed to punch the number

101 110 011 001 101 101 011 011 001 100

(Use the table of Fig. 2-23.)

(*b*) Draw a picture of paper tape with the number in Exercise *a* punched on it.

(*c*) What number does the given hole pattern represent?



EXERCISE *c*

(*d*) During what phases might the *write-into* signal, shown in Fig. 2-30, be "on," i.e., have a unit signal voltage level? During what phases must this signal be "off," i.e., have a zero signal voltage?

(*e*) During what phases might the *read-out* signal shown in Fig. 2-30 be on? During what phases must this signal be off?

2-7. Digital-computer and -control Engineering

In this chapter the attempt was made to give a broad picture of how a digital computer works. The *three* disciplines that comprise digital-computer and -control engineering were introduced. These were (1) *coding and programming*, or the study of writing sequences of instructions that direct the computer to perform particular tasks; (2) *logical design of computer circuitry*, or the study of designing proper connections between the three building-block gates so that the resultant circuit will perform specified functions; and (3) *electronic design of computer circuitry*, or the study of specific electronic-circuit representations of the building-block gates as well as the electronic design of the input and output equipment associated with a computer. It has been seen how the methods and techniques by which a computer is programmed are intimately associated with the logical design of the computer, which in turn must satisfy the requirements set by the electronic design of the computer gate circuits. Hence, in dealing with almost any aspect of computers, the engineer must have a thorough grasp of all three of these disciplines.

In designing a digital computer, perhaps the engineer's first consideration is choosing the proper coding and programming system which best suits the purpose and function for which the computer is to be used. Although the basic concepts of programming are invariant, the computer engineer can choose from a wide range of possibilities of programming systems. This choice is extremely important, for it forms the basis of the entire structure of a computer design. The rest of the chapters in Part 1 of this book are concerned with this phase of computer engineering. Part 2 is concerned with the operation and use of computers pertaining to various problems—numerical analysis, searching and sorting, etc.

If we assume that the electronic design of the basic building-block gating circuits has been chosen, the fundamental problem of computer engineering becomes that of the logical design of the computer. Part 3 of the book lays the mathematical foundations for such design techniques, and Part 4 applies these to the logical design of common computer components. This aspect of computer engineering will perhaps appear most unusual to electrical engineers. The laborious analysis carried out in Sec. 2-5 is replaced by systematic computational methods based on a rather different kind of algebra—called *Boolean algebra*.

Part 5 of the book considers the electronic design of the basic building-block gating circuits, using various hardware components such as diodes, transistors, ferrite cores, etc. The usefulness of a computer depends to a large extent on the speed, reliability, stability, and other such characteristics of the circuits from which it is made. Hence the careful design, analysis, and testing of these circuits and their components comprise a large part of computer engineering.

2-8. Additional Topics

a. Analogy with Desk Calculator. Make a detailed analog between a digital electronic computer and a desk calculator, paying attention to function, structure, sequence of steps in their operation, etc. That is, what are analogous to accumulator, memory, instruction register, control unit, arithmetic unit, etc.?

b. Constructing a Cardiac Computer. A *Cardiac* computer is made by assigning each person of the class as one of the units or parts of a computer. Each person so assigned becomes responsible for performing all the functions of this assigned part of the computer, but no other functions. Then a program or sequence of instructions is written, and these instructions are executed by means of this human simulation of a computer. Words and control signals are transmitted by passing around cards or pieces of paper.

c. Memories. For each of five commercially available high-speed digital electronic computers list the kinds of memories they include and their memory access times, and the kinds of input-output equipment they have and the speeds of these units. For example, get commercial advertising from the manufacturer, and consult Martin H. Weik, A Second Survey of Domestic Electronic Digital Computing Systems, *BRL Rept.* 1010, June, 1957.

d. Background. There are several subjects the student may find interesting as background material to his study of digital-computer engineering. These are cybernetics, information theory, certain aspects of neurophysiology, and the so-called "theory of automata." Readings relating to these subjects can be found in:

Ashby, W. Ross: "Design for a Brain," John Wiley & Sons, Inc., New York, 1952.

Brillouin, L.: "Science and Information Theory," Academic Press, Inc., New York, 1956.

De Latil, Pierre: "Thinking Machines," Houghton Mifflin Company, Boston, 1957.

McCulloch, W. S., H. Quaster, G. A. Miller, and L. S. Fogel: Human Beings as Computers, a collection of four papers, *IRE Trans. on Electronic Computers,* vol. EC-6, no. 3, pp. 190–202, September, 1957.

Newman, James R. (ed.): "The World of Mathematics," vol. 4, pp. 2070–2133, Simon and Schuster, Inc., New York, 1956.

Shannon, C. E., and J. McCarthy (eds.): "Automata Studies," Princeton University Press, Princeton, N.J., 1956.

Wiener, Norbert: "Cybernetics," John Wiley & Sons, Inc., New York, 1948. (Omit chaps. II–IV.)

The student may find it profitable to keep the following questions in mind during the course of his readings: What is the definition of information? Of what importance is information theory to the study of computers? What is the relation between the functioning of a neuron and those of the three building-block gates? In what way is a digital computer like the human brain? In what way does the human brain differ from a digital computer? Compare and contrast the computer's memory and that of the human brain. Can a digital computer think?

e. Terminology. Discuss the problems involved in standardizing computer terminology; i.e., compare definitions and usages of words as found, for example, in First Glossary of Programming Terminology, *J. Assoc. Computing Machinery,* June, 1954, and The Institute of Radio Engineers Standards on Electronic Computers: Definitions of Terms, 1956, *Proc. IRE,* vol. 44, no. 9, September, 1956.

f. History. Discuss the history of digital computers, identifying each advance in computer engineering with particular computers; include Babbage's computer, the ENIAC, EDVAC, SEAC, DYSEAC, UNIVAC, Whirlwind, IAS computer, Remington Rand 1103, IBM 704, and NORC, and the modern supercomputers such as LARC and Stretch [see B. V. Bowden (ed.), Faster than Thought, Pitman Publishing Corporation, New York, 1953, and other references in *J. Assoc. Computing Machinery* and *IRE Trans. on Electronic Computers*]. Figure 2-33 represents an evolutionary tree of digital computers that may be of aid in the discussion. We cannot claim that this tree is complete, but we hope that the major trends are reflected. Figure 2-34 is a montage of present-day computers. These photographs have been included to give the student an idea of the variety of modern computers.

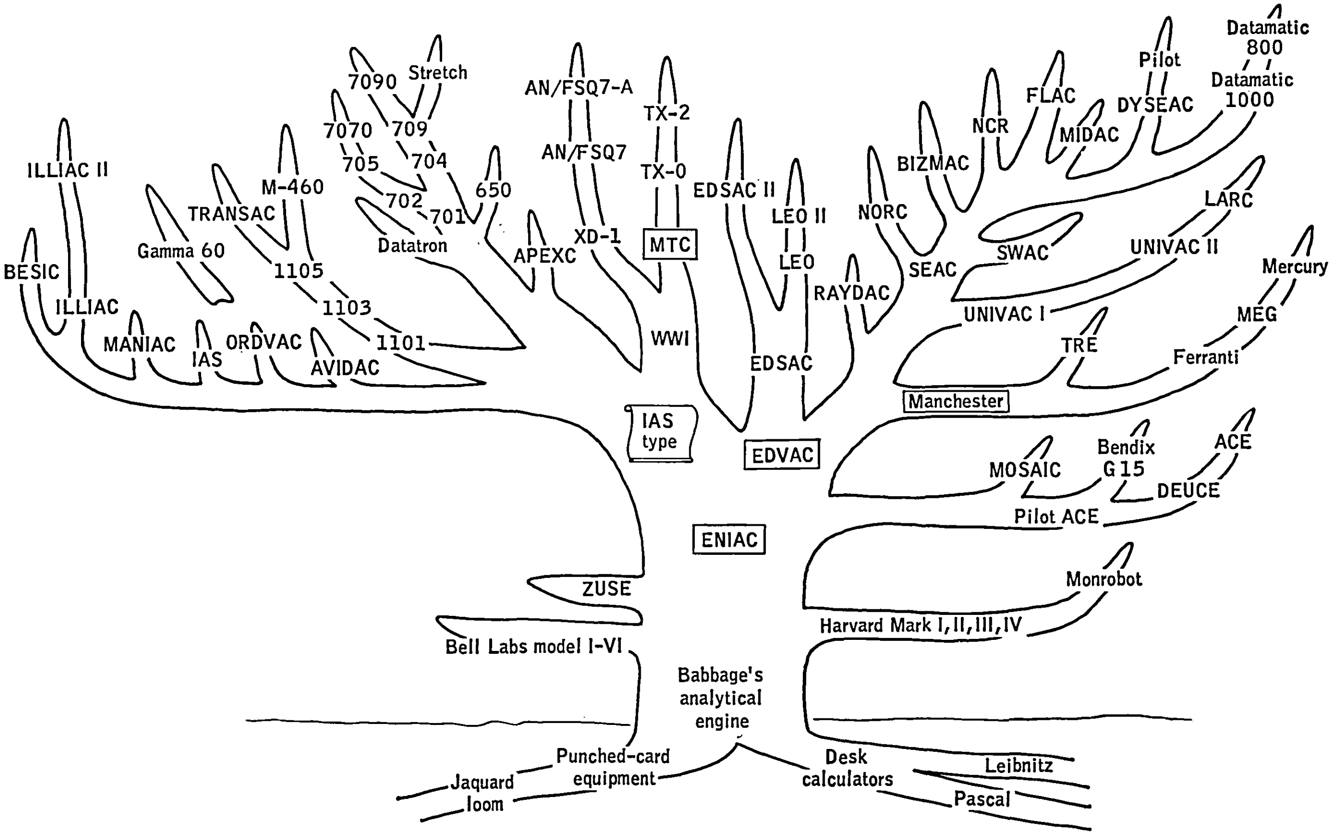


FIG. 2-33. The evolutionary tree of computer development.

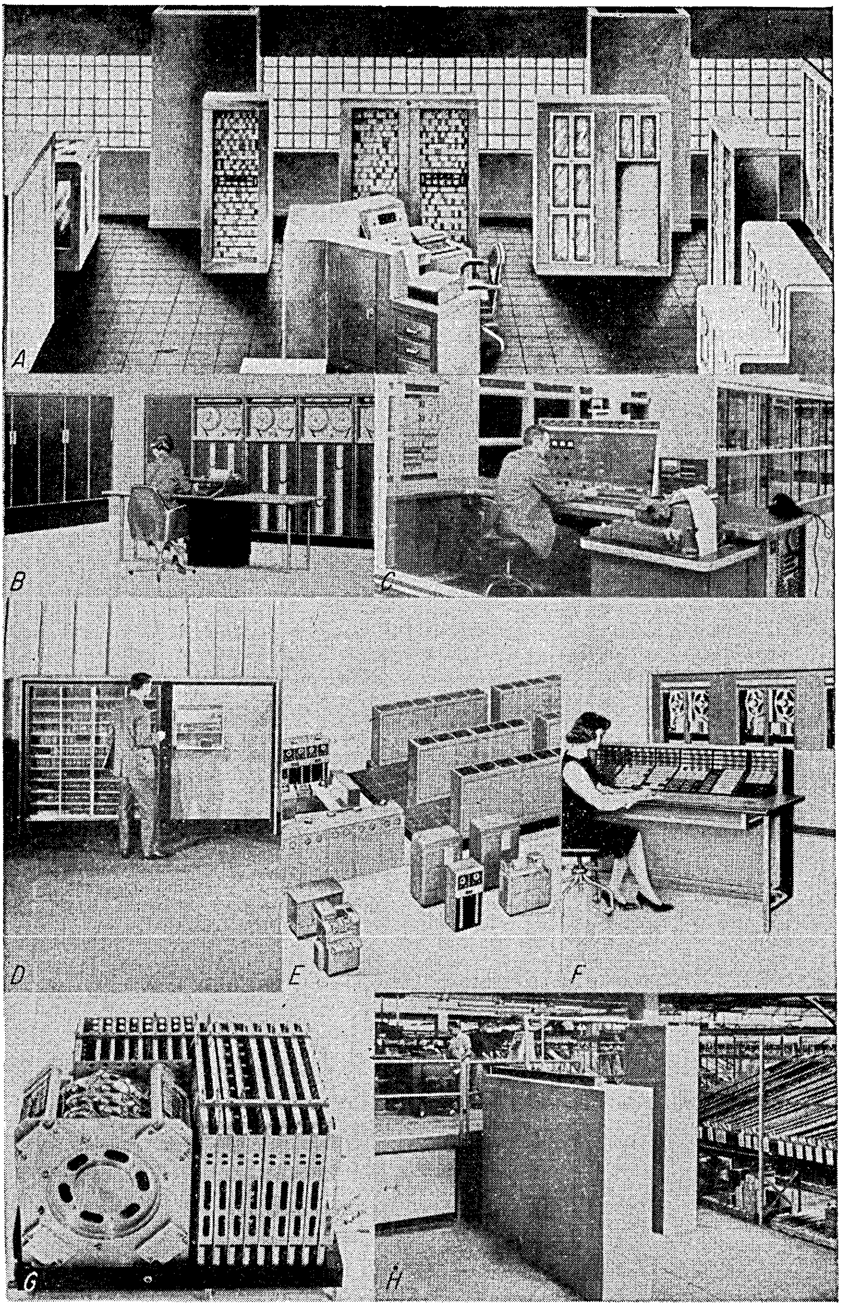


FIG. 2-34. Present-day computers. (*A*) FLAC computer for biomedical research; (*B*) Honeywell 800; (*C*) Datamatic 1000; (*D*) UNIVAC solid-state computer; (*E*) UNIVAC 1105; (*F*) RCA 501; (*G*) air-borne computer; (*H*) RVS-Mark I (an automatic order-filling system). [*Courtesy* (*B*, *C*) *Minneapolis-Honeywell;* (*D*, *E*) *Remington Rand;* (*F*) *Radio Corp. of America;* (*G*) *Librascope, Inc.;* (*H*) *Industrial Electronics Engineers, Inc.*]

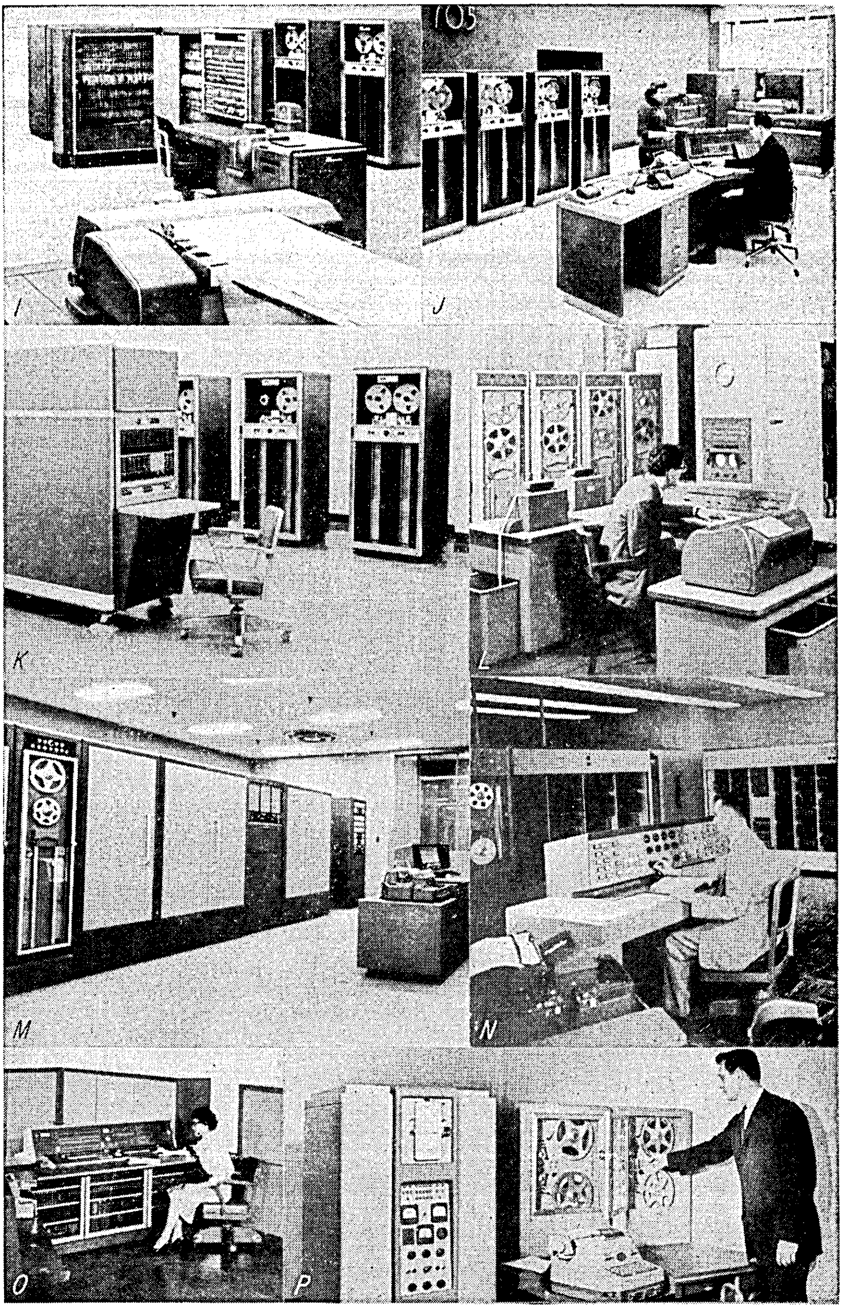


FIG. 2-34. Present-day computers (*continued*). (*I*) IBM 709; (*J*) IBM 705; (*K*) IBM 650; (*L*) Perseus G/7578/1; (*M*) Burroughs 205; (*N*) NCR 304; (*O*) TRANSAC-S-2000; (*P*) Bendix G-15. [*Courtesy* (*I, J, K*) *International Business Machines Corp.;* (*L*) *Ferranti, Ltd.;* (*M*) *Burroughs Corp.;* (*N*) *National Cash Register;* (*O*) *Philco Corp.;* (*P*) *Bendix Computer Division.*]

CHAPTER 3

CODING AND PROGRAMMING A DIGITAL COMPUTER

3-1. Introduction

Recapitulation. The previous chapter has given the basic foundations necessary for a detailed discussion of how to direct a computer to perform desired computations. A code was defined as a list of instructions and numbers, the instructions being the specific means by which a person tells the computer what to do. The fundamental concept that instructions are written in terms of addresses, the contents of which are the numbers to be operated upon, was emphasized. We described the procedure by which the instructions are called into the instruction register, one instruction at a time and in sequence; and we indicated how the operations are performed by the computer. The use of the binary number system inside the computer and of some kind of shorthand for coding purposes was discussed briefly. We described the processes involved in transforming numbers written on paper into binary pulses inside the computer's memory. And it was noted that a similar process is involved in the reverse transformation from pulses in the computer's memory to typed numbers on a working paper outside the computer.

Importance of Coding to the Engineer. It is important for the digital-computer engineer to be well versed in coding and programming because, first, such knowledge is essential for a clear understanding of the principles of digital computers; second, occasions often arise when the computer engineer will have to run a problem of his own associated with his engineering duties; third, such knowledge is essential in maintaining an existing computer or checking out a new computer; and, finally, the engineer must be capable of evaluating various designs with respect to the final use and application of the computer.

Material Directed to Engineering Needs. The present chapter is the first of three chapters devoted exclusively to programming and coding. This chapter is concerned mainly with an introduction to coding and instruction formats. The next chapter (Chap. 4) is concerned with subroutines and the basis of instruction definitions. The final chapter of this group (Chap. 5) considers the principles of automatic programming. These chapters are directed to the digital-computer engineer and the problems he will face in his work. The engineer must grasp the fundamental ideas involved, and his training cannot be restricted to a single type of computer instruction format. He must understand the principles behind the definitions of many kinds of instructions, so that he may design new ones and evaluate proposed ones. He must also understand the basis for automatic programming so that he may use or write such programs intelligently in solving engineering problems or in performing computer maintenance. Programming and coding are therefore a fundamental aspect of computer engineering.

Manner of Presentation. In successive sections of this chapter we consider instructions that include explicitly first four addresses, then only three, then two, and finally only one address. We begin with the four-address system because this format includes *all the necessary ingredients* of an instruction. Then, by omitting one address in the instruction format, we arrive quite naturally at a three-address system, and so forth. As each address is omitted from the instruction format, it must be compensated for by corresponding additional computer circuitry or by using several instructions where one sufficed before. In this way the advantages and disadvantages of various *basic instruction formats* are clearly observed.

Examples will show that, the smaller the number of addresses per instruction, the greater the ingenuity and work required of the coder, the larger the list of instructions that must be available to the coder, and the longer the equivalent codes. On the other hand, as the number of addresses per instruction is increased, the number of possible addressable memory locations is decreased and the words are made longer. In designing a computer system all these factors must be carefully evaluated with respect to the intended use of the computer, the amount of hardware required, the speed of the computer, the cost of the computer and of programming and coding, etc.

In the current chapter only very basic instruction lists are used in order that the material be as simple as possible at first. Only sequences of instructions and decisions are illustrated. This leaves the student free to concentrate on the fundamental properties and uses of the instruction formats. Consideration of those most important and fundamental programming techniques of loops (i.e., recursions, iterations) and subroutines is left to Chap. 4. In that chapter also, a large number of more complicated and specialized instruction or operation types are considered in detail. We believe that the student will more easily understand the principles of programming if consideration of these somewhat sophisticated ideas is postponed. Any discussion of coding and programming must be preceded by a study of number systems, as will be found in the next sections of this chapter.

Stages in Coding. As will be seen, there are five stages in coding. First the computations to be performed must be clearly and precisely defined. The over-all plan of the computations is diagramed by means of a so-called "flow chart." The second stage is the actual coding. It is often best to write a code in terms of a symbolic language first, for then changes are easily made. Numbers are assigned to the symbols, and the final code is prepared. In the third stage some procedure is used to get the code into the memory of the computer. The fourth stage consists in "debugging" the code, i.e., detecting and correcting any errors. The fifth and final stage involves the running of the code on the computer and tabulating the results. In this chapter and the next only the first two stages are considered. Techniques for the fourth stage are described in Chap. 5.

Philosophy of Coding. As will become abundantly clear in the following sections, a single error in one instruction invalidates the entire code. Hence coding is an exacting technique, requiring attention to details without losing sight of the over-all plan. Learning to code must be developed from within a person. Following through each instruction of the examples given in this chapter is important, but it is not sufficient. The student must code problems by himself if he is to become aware of all the detailed thinking and reasoning associated with the coding technique. The method of teaching in this text is through example. The student should try to do the examples himself, using the text only as a guide to the methods involved.

3-2. Number Systems: Conversion

Number Systems and the Radix. Preliminary to any discussion of coding should come an analysis of various number systems. In Sec. 2-4 we noted that 8,675 really means $8 \times 10^3 + 6 \times 10^2 + 7 \times 10^1 + 5 \times 10^0$, where $10^0 = 1$. Since decimal numbers are expanded as illustrated in powers of ten, we say that the decimal number system has a *radix* of ten. This concept can be generalized to define number systems based on any positive-integer radix q: if a number N is based on radix q, then it can be expanded as

$$N = a_nq^n + a_{n-1}q^{n-1} + \cdots + a_2q^2 + a_1q^1 + a_0q^0$$

for integer $N \geq 1$, or

$$N = a_{-1}q^{-1} + a_{-2}q^{-2} + a_{-3}q^{-3} + \cdots + a_{-m}q^{-m}$$

for $0 < N < 1$, where $a_0, a_1, \ldots, a_{-1}, a_{-2}, \ldots$ are nonnegative integers each less than q. Of course, for noninteger $N > 1$, the expansion contains both the positive and negative powers,

$$\begin{aligned} N = a_nq^n + \cdots + a_2q^2 + a_1q^1 + a_0q^0 \\ + a_{-1}q^{-1} + a_{-2}q^{-2} + \cdots + a_{-m}q^{-m} \end{aligned}$$

For example,

$$\begin{aligned} 8{,}675.8675 = (8 \times 10^3) + (6 \times 10^2) + (7 \times 10^1) + (5 \times 10^0) \\ + (8 \times 10^{-1}) + (6 \times 10^{-2}) + (7 \times 10^{-3}) + (5 \times 10^{-4}) \end{aligned}$$

that is, $a_3 = 8$, $a_2 = 6$, $a_1 = 7$, $a_0 = 5$, $a_{-1} = 8$, $a_{-2} = 6$, $a_{-3} = 7$, and $a_{-4} = 5$. Even though we are most familiar with the system for q = ten, history records ancient civilizations that used q = six, q = twelve, and even q = sixty! For our purposes we are most interested in $q = 2$ (*binary* system), $q = 8$ (*octal* system), and $q = 16$ (*sexadecimal*† system).

First note that for radix q the number system must involve q symbols. For example, for q = ten, the ten symbols are *0, 1, 2, 3, 4, 5, 6, 7, 8,* and *9*; for $q = 2$, the 2 symbols are *0* and *1*; for $q = 16$, the 16 symbols most often used are *0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E,* and *F*. Then, if $N = a_nq^n + a_{n-1}q^{n-1} + \cdots + a_2q^2 + a_1q^1 + a_0q^0$, where the a_i are of course chosen from the q symbols, the number is conventionally written by juxtaposition of the coefficients a_i as follows: $N = a_na_{n-1} \cdots a_2a_1a_0$. If $N = a_{-1}q^{-1} + \cdots + a_{-m}q^{-m}$, then the number is usually written by juxtaposition of the a_i preceded by [.]—that is, $.a_{-1}a_{-2} \cdots a_{-m}$. For noninteger $N > 1$ we would have $a_n \cdots a_0.a_{-1} \cdots a_{-m}$. With this in mind, *counting* in the four different systems mentioned above is accomplished as in Table 3-1.

Conversion from One Number System to Another. Using number systems based on different radices is like talking in different languages. Just as one can translate one language into another, so also one can "convert" from one radix to another. For example, in Table 3-1 numbers on the same row are equivalent—that is,

$$28 \text{ (decimal)} = 11100 \text{ (binary)} = 34 \text{ (octal)} = 1C \text{ (sexadecimal)}$$

The binary, octal, and sexadecimal equivalents of the decimal numbers from 0 to 15 should be memorized.

In order to derive a systematic method for converting from one number system to another, we make use of the well-known result that if N and q are positive integers then there always exists a unique (nonnegative) integer r less than q and a unique integer S such that

$$\frac{N}{q} = S + \frac{r}{q} \qquad 0 < r < q$$

Let us apply this to convert an integer number with radix p, say N_p, to its equivalent with radix q, say N_q. In other words, we wish to determine the nonnegative integers $a_0, a_1, \ldots, a_n$, each less than q, such that

$$N_p = a_nq^n + \cdots + a_2q^2 + a_1q^1 + a_0q^0$$

Observe that

$$\begin{aligned} \frac{N_p}{q} &= \underbrace{a_nq^{n-1} + \cdots + a_2q^1 + a_1q^0}_{\text{Integer}} + \underbrace{\frac{a_0}{q}}_{\text{Fraction}} \\ &= \qquad\qquad S_0 \qquad\qquad + \quad \frac{r_0}{q} \end{aligned}$$

Since the quotient and remainder *are each unique*, equating the fractional

† Also less correctly called *hexadecimal.*

TABLE 3-1. CONVERSION TABLE

Decimal ($q = 10$)	Binary ($q = 2$)	Octal ($q = 8$)	Sexadecimal ($q = 16$)
0	0	0	0
1	1	1	1
2	10	2	2
3	11	3	3
4	100	4	4
5	101	5	5
6	110	6	6
7	111	7	7
8	1000	10	8
9	1001	11	9
10	1010	12	*A*
11	1011	13	*B*
12	1100	14	*C*
13	1101	15	*D*
14	1110	16	*E*
15	1111	17	*F*
16	10000	20	10
17	10001	21	11
18	10010	22	12
19	10011	23	13
20	10100	24	14
21	10101	25	15
22	10110	26	16
23	10111	27	17
24	11000	30	18
25	11001	31	19
26	11010	32	1*A*
27	11011	33	1*B*
28	11100	34	1*C*
29	11101	35	1*D*
30	11110	36	1*E*
31	11111	37	1*F*
32	100000	40	20
33	100001	41	21
34	100010	42	22
35	100011	43	23
36	100100	44	24
. . .	. . . . . .	. . .	. . .

parts we have $a_0 = r_0$ and $S_0 = a_n q^{n-1} + \cdots + a_2 q^1 + a_1 q^0$. Dividing S_0 by q and using our above results, we obtain

$$\frac{S_0}{q} = a_n q^{n-2} + \cdots + a_2 q^0 + \frac{a_1}{q} = S_1 + \frac{r_1}{q}$$

whence $a_1 = r_1$, and so forth.

Hence the systematic conversion method is as follows: Divide N_p by q; the remainder is a_0. Divide the resulting quotient by q; the remainder is a_1. Divide the resulting quotient by q; the remainder is a_2; etc. As an example, we have converted 28 (*decimal*) into binary ($q = 2$) in Table 3-2. Hence,

28 (*decimal*) = 11100 (*binary*)

Table 3-2. Decimal to Binary Conversion ($N_p \geq 1$)

	Remainders
$2\underline{\lvert 28}$	
$2\underline{\lvert 14}$	$0 = a_0$
$2\underline{\lvert 7}$	$0 = a_1$
$2\underline{\lvert 3}$	$1 = a_2$
$2\underline{\lvert 1}$	$1 = a_3$
0	$1 = a_4$

The work can be more conveniently arranged from right to left, as follows:

$$\boxed{\begin{array}{c}\text{Divide by 2}\\ \leftarrow \\ \downarrow \text{ Place remainder}\end{array}}$$

$$\begin{array}{cccccccccccl} 0 & \leftarrow & 1 & \leftarrow & 3 & \leftarrow & 7 & \leftarrow & 14 & \leftarrow & 28\underline{\lvert 2} & \\ \downarrow & & \downarrow & & \downarrow & & \downarrow & & \downarrow & & & \\ 1 & & 1 & & 1 & & 0 & & 0 & & & \text{Remainders} \end{array}$$

As another example, let us convert 28 (*decimal*) to octal ($q = 8$),

$$\begin{array}{ccccccl} 0 & \leftarrow & 3 & \leftarrow & 28\underline{\lvert 8} & & \\ \downarrow & & \downarrow & & & & \\ 3 & & 4 & & & & \text{Remainders} \end{array}$$

whence the converted octal number is 34.

To convert 28 (*decimal*) to sexadecimal ($q = 16$),

$$\begin{array}{ccccc} 0 & \leftarrow & 1 & \leftarrow & 28\underline{\lvert 16} \\ 1 & & C & & \end{array}$$

Hence the converted sexadecimal number is $1C$. [Here, of course, $28 - 16 = 12$ (*decimal*) $= C$ (*sexadecimal*) from Table 3-1.]

Next consider $0 < N_p < 1$. We wish to find nonnegative integer coefficients $a_{-1}, a_{-2}, \ldots, a_{-m}$, each less than q, such that

$$\begin{array}{lrl} & N_p &= a_{-1}q^{-1} + a_{-2}q^{-2} + \cdots + a_{-m}q^{-m} \\ \text{Here} & qN_p &= a_{-1} + a_{-2}q^{-1} + \cdots + a_{-m}q^{-m+1} \\ & &= u_{-1} + v_{-1} \end{array}$$

where u_{-1} is the integral part and v_{-1} is the fractional part of qN_p; hence

$$\begin{array}{lrl} & u_{-1} &= a_{-1} \\ \text{and} & v_{-1} &= a_{-2}q^{-1} + \cdots + a_{-m}q^{-m+1} \\ \text{Again} & qv_{-1} &= a_{-2} + a_{-3}q^{-1} + \cdots + a_{-m}q^{-m+2} = u_{-2} + v_{-2} \\ \text{whence} & u_{-2} &= a_{-2}, \text{ etc.} \end{array}$$

The conversion method thus is as follows: Multiply N_p by q; the integral part is a_{-1}. Multiply the resulting fractional part by q; the integral part is a_{-2}; etc.

As an example, we have converted 0.28 (*decimal*) to binary ($q = 2$) in Table 3-3. Hence 0.28 (*decimal*) = .01000111 · · · (*binary*). The work can be more conveniently arranged from left to right, as follows:

TABLE 3-3. DECIMAL TO BINARY CONVERSION ($0 < N_p < 1$)

Integral parts	
	0.28
	× 2
$a_{-1} = 0$	.56
	× 2
$a_{-2} = 1$	.12
	× 2
$a_{-3} = 0$	.24
	× 2
$a_{-4} = 0$	.48
	× 2
$a_{-5} = 0$	.96
	× 2
$a_{-6} = 1$	.92
	× 2
$a_{-7} = 1$	.84
	× 2
$a_{-8} = 1$	.68
	. . . .

$$\boxed{\begin{array}{l} \text{Multiply by 2} \\ \qquad \to \\ \text{Place} \\ \text{integral} \ \downarrow \\ \text{part} \end{array}}$$

$$\begin{array}{llllllllll} 2\times & 0.28 \to & .56 \to & .12 \to & .24 \to & .48 \to & .96 \to & .92 \to & .84 \to & .68 \\ & & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow \\ \textit{Integral parts} & & 0 & 1 & 0 & 0 & 0 & 1 & 1 & 1 \end{array}$$

Note that the process does not necessarily end, as it must for the integral numbers, but may be carried out to any number of significant figures. Conversion to the octal and sexadecimal systems is similarly accomplished.

For nonintegral numbers greater than unity the integral part and the fractional part are done separately as just described.

Note that in converting from binary to octal (or sexadecimal) we divide, or multiply, the binary number by binary 100 (or 1000), which is equivalent to shifting the fraction point to the left, or to the right, by three (four) binary places. Hence for binary to octal conversion we start at the point and count off the binary bits in groups of three, to the left and to the right. Then we write the octal equivalent for each of these groups of three bits. For example, if the binary number is 11100.110100011, the octal equivalent is

$$\begin{array}{cccc} 3 & 4\,.\,6 & 4 & 3 \\ 011 & 100.110 & 100 & 011 \end{array}$$

To convert to sexadecimal, we count off similarly by groups of four. For the same binary number the sexadecimal number is

$$\begin{array}{ccccc} 1 & C\,.\,D & 1 & \cdots \\ 0001 & 1100.1101 & 0001 & \cdots \end{array}$$

(See Table 3-1 for binary-octal and binary-sexadecimal equivalents.) The converse will of course hold.

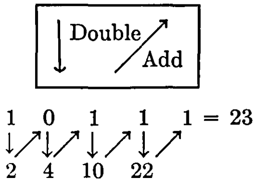
Converting from Binary to Decimal. Another conversion method, particularly useful for converting from binary to decimal, is based on the following factorization, illustrated for $n = 4$:

$$\begin{aligned} N &= a_4q^4 + a_3q^3 + a_2q^2 + a_1q^1 + a_0q^0 \\ &= (((a_4q + a_3)q + a_2)q + a_1)q + a_0 \end{aligned}$$

If N is a binary integer, then $q = 2$ and the a's are either 0 or 1, but a_4, the most significant coefficient, must be 1. Hence, to convert N to decimal, start out with 1, and double it, obtaining 2; then add a_3, and double the result; add a_2, and double the result; add a_1, and double the result; finally add a_0. For example,

$$\begin{array}{cccccl} a_4 & a_3 & a_2 & a_1 & a_0 & \\ 1 & 0 & 1 & 1 & 1 & = (((2 + 0)2 + 1)2 + 1)2 + 1 \\ & & & & & = ((4 + 1)2 + 1)2 + 1 \\ & & & & & = (10 + 1)2 + 1 \\ & & & & & = 22 + 1 = 23 \end{array}$$

The method is to accumulate the result while calculating, and the work from left to right might look like this:



Similarly for N such that

$$N = a_{-1}q^{-1} + a_{-2}q^{-2} + a_{-3}q^{-3} + a_{-4}q^{-4}$$

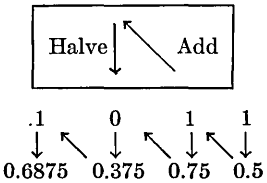
we have the factorization

$$N = \frac{1}{q}\left(a_{-1} + \frac{1}{q}\left(a_{-2} + \frac{1}{q}\left(a_{-3} + \frac{1}{q}a_{-4}\right)\right)\right)$$

If N is given in binary, then $q = 2$ and the a's are either 0 or 1. The decimal conversion is calculated by starting with the least significant unit (for example, a_{-4} in our case), working toward the left as follows: Add a_{-3} to .5, and halve the result; add a_{-2}, and halve the result; add a_{-1}, and halve the result. For example,

$$\begin{aligned} .1011 &= \tfrac{1}{2}(1 + \tfrac{1}{2}(0 + \tfrac{1}{2}(1 + 0.5))) \\ &= \tfrac{1}{2}(1 + \tfrac{1}{2}(0 + 0.75)) \\ &= \tfrac{1}{2}(1 + 0.375) = 0.6875 \end{aligned}$$

Of course the method is to accumulate while calculating, and the work from right to left might be arranged as follows:



This method can be used for octal and sexadecimal conversion to decimal simply by converting the octal or sexadecimal to binary first; e.g., for octal,

$$53.53 \ (\textit{octal}) = 101011.101011 \ (\textit{binary})$$

EXERCISES

Convert the following decimal numbers to binary, octal, and sexadecimal. Carry the conversion far enough so that the converted number differs from the original decimal number by less than $\frac{1}{500}$.

Answers

	Decimal	*Binary*	*Octal*	*Sexadecimal*
(*a*)	56	111000	70	38
(*b*)	0.79	.110010100	.624	.*CA*
(*c*)	56.79	111000.110010100	70.624	38.*CA*
(*d*)	0.732	.101110110	.566	.*BB*
(*e*)	87,231	10101010010111111	252277	154*BF*
(*f*)	87,231.732	10101010010111111.101110110	252277.566	154*BF*.*BB*

(*g*) Convert −56.79 to binary, octal, and sexadecimal. (HINT: A negative number is converted as if it were positive, and then the negative sign is affixed to the result. Why?)

Convert to decimal:
(*h*) 70 (*octal*).
(*i*) .624 (*octal*).
(*j*) *BB* (*sexadecimal*).
(*k*) 70.624 (*octal*).
(*l*) Compare the results of Exercise *k* with those of Exercise *c*, and explain.

3-3. Number Systems: Arithmetic

Addition and Multiplication Tables. We have already considered the arithmetic operation of *counting* in different number systems. Although all arithmetic can be accomplished by counting, we are more familiar with the laborsaving operations of addition, multiplication, subtraction, and division. Of course these operations may also be carried out in other than the decimal number system. Addition and multiplication are essentially carried out by means of tables. For decimal, binary, and octal the tables are shown in Table 3-4.

Arithmetic Operations. In both addition and subtraction the "carry" or "borrow" is handled in other number systems exactly as it is handled in the familiar decimal system. Of course the addition tables are used during the calculation. For example,

$$\begin{array}{rrrrrl} & 1 & 1 & 0 & 1 & (\textit{binary}) \\ + & {}_{,} & 1 & 0_{,} & 1 & (\textit{binary}) \\ \hline 1 & 0 & 0 & 1 & 0 & (\textit{binary}) \end{array} \qquad \begin{array}{rrrrl} & 7 & 5 & 7 & (\textit{octal}) \\ + & {}_{,} & 5_{,} & 6 & (\textit{octal}) \\ \hline 1 & 0 & 3 & 5 & (\textit{octal}) \end{array}$$

$$\begin{array}{rrrrl} 1 & 1 & 0 & 1 & (\textit{binary}) \\ -\,{}_{,} & 1_{,} & 1 & 0 & (\textit{binary}) \\ \hline 0 & 1 & 1 & 1 & (\textit{binary}) \end{array} \qquad \begin{array}{rrrl} 7 & 5 & 2 & (\textit{octal}) \\ -\,{}_{,} & 6_{,} & 5 & (\textit{octal}) \\ \hline 6 & 6 & 5 & (\textit{octal}) \end{array}$$

Multiplication and division in other systems also follow the rules familiar to us in the decimal system. Of course the multiplication and

TABLE 3-4. ARITHMETIC TABLES

DECIMAL ADDITION TABLE

+	0	1	2	3	4	5	6	7	8	9
0	0	1	2	3	4	5	6	7	8	9
1	1	2	3	4	5	6	7	8	9	10
2	2	3	4	5	6	7	8	9	10	11
3	3	4	5	6	7	8	9	10	11	12
4	4	5	6	7	8	9	10	11	12	13
5	5	6	7	8	9	10	11	12	13	14
6	6	7	8	9	10	11	12	13	14	15
7	7	8	9	10	11	12	13	14	15	16
8	8	9	10	11	12	13	14	15	16	17
9	9	10	11	12	13	14	15	16	17	18

DECIMAL MULTIPLICATION TABLE

×	0	1	2	3	4	5	6	7	8	9
0	0	0	0	0	0	0	0	0	0	0
1	0	1	2	3	4	5	6	7	8	9
2	0	2	4	6	8	10	12	14	16	18
3	0	3	6	9	12	15	18	21	24	27
4	0	4	8	12	16	20	24	28	32	36
5	0	5	10	15	20	25	30	35	40	45
6	0	6	12	18	24	30	36	42	48	54
7	0	7	14	21	28	35	42	49	56	63
8	0	8	16	24	32	40	48	56	64	72
9	0	9	18	27	36	45	54	63	72	81

BINARY ADDITION TABLE

+	0	1
0	0	1
1	1	10

BINARY MULTIPLICATION TABLE

×	0	1
0	0	0
1	0	1

OCTAL ADDITION TABLE

+	0	1	2	3	4	5	6	7
0	0	1	2	3	4	5	6	7
1	1	2	3	4	5	6	7	10
2	2	3	4	5	6	7	10	11
3	3	4	5	6	7	10	11	12
4	4	5	6	7	10	11	12	13
5	5	6	7	10	11	12	13	14
6	6	7	10	11	12	13	14	15
7	7	10	11	12	13	14	15	16

OCTAL MULTIPLICATION TABLE

×	0	1	2	3	4	5	6	7
0	0	0	0	0	0	0	0	0
1	0	1	2	3	4	5	6	7
2	0	2	4	6	10	12	14	16
3	0	3	6	11	14	17	22	25
4	0	4	10	14	20	24	30	34
5	0	5	12	17	24	31	36	43
6	0	6	14	22	30	36	44	52
7	0	7	16	25	34	43	52	61

addition or subtraction tables are used during the calculations. For example,

```
     1011  (binary)           775  (octal)
×     101  (binary)       ×    56  (octal)
  -------                   -----
     1011                    5756
    0000                    4761
   1011                     -----
  -------                   55566  (octal)
   110111  (binary)
```

```
       1011  (binary)          775  (octal)
 101|110111  (binary)       56|55566  (octal)
     101                       502
     ---                       ---
      111                       536
      101                       502
      ---                       ---
       101                       346
       101                       346
```

EXERCISES

(*a*) Make an addition and multiplication table for sexadecimal numbers. [HINT: In the addition table the rows (and columns) count by 1; the row (or column) of the multiplication counts by *n* if *n* is the number of the row (column).]

Perform the following operations in binary:

	Answers
(*b*) 111 + 1	1000
(*c*) 1010 + 111	10001
(*d*) 1110 − 1	1101
(*e*) 1 − 1110	−1101
(*f*) 1101 × 1011	10001111
(*g*) 11011 ÷ 110	100.1

Perform the following operations in octal:
(*h*) 6,754 + 777.
(*i*) 73 − 56.
(*j*) 555 − 62.
(*k*) 5,715 ÷ 65.

Using the tables of Exercise *a*, calculate in the sexadecimal system:

	Answers
(*l*) *AFCB* + 1*AD*8	*CAA*3
(*m*) 842*D* − 65*AE*	1*E*7*F*
(*n*) 3*C* × *D*1	30*FC*
(*o*) 1*A*8*E* ÷ 21	*CE*

3-4. Coding: Sequences of Instructions

We shall initiate our detailed discussion of coding using the *four-address* instruction system. That is, in this section we shall consider instructions which explicitly include four addresses, since a four-address instruction system includes all the essential ideas or basic ingredients of an instruction.

Instruction Format. As we have previously seen, some bits of the instruction are set aside for the operation code designation—i.e., they tell the computer the instruction is add, multiply, divide, etc. The rest of the bits usually denote the four addresses. For the more usual operations that involve two operands, such as addition, multiplication, etc., two of the addresses are the *addresses of the operands.* The third address tells *where the result is to be put;* the fourth address, where to obtain the *next instruction.* Hence a typical four-address *instruction format* is as follows (where the four addresses are denoted by α, β, γ, and δ):

Operation code	First operand address α	Second operand address β	Put result into address γ	Address of next instruction δ

For any specific computer using a four-address instruction format the number of bits in each address and in the operation code must be given, as well as the actual operation code itself. In the design of a computer the number of bits reserved for an address presents an upper limit to the number of words in the addressable memory of the computer. If an address is denoted by n bits, no more than 2^n words can be contained in the addressable memory.

Octal Shorthand. In order to illustrate the coding techniques developed in this section, we assume that there are 9 bits in each address ($2^9 = 512$ addressable memory words) and 6 bits in the instruction code, a total of $(4 \times 9) + 6 = 42$ bits in a word. This brings up the first important detail of coding: the actual bits in an instruction are not written out; rather, some shorthand is written instead. For example, in our case the bits would be divided into 14 groups of 3 bits each, and the octal equivalent of each group would be written out. In other words, two octal numbers would represent the instruction, and each address would be represented by three octal numbers. Thus, if 101 011 is the binary code for add, then the instruction that says, "Add the contents of address 011 010 110 to the contents of address 011 100 101, put the result into address 011 110 100, and take the next instruction from 100 000 001," is written in octal notation as

Operation	α	β	γ	δ
53	326	345	364	401

In such cases it evidently facilitates matters to call the addresses in the memory by their *octal numbers.* It will be assumed throughout the rest of this section that such is the case. Also, numerical quantities will be written on the code sheet in *octal* (i.e., they will have to be converted from decimal to octal before being written on the code sheet).

The Computer Manual. For our computer we must have a "computer manual" that gives the operation codes of the different instructions and also precisely defines the meaning of the addresses for each instruction type. For example, see Table 3-5. As is clearly seen, the coding manual must always be at the coder's side. Two further observations must be reemphasized before we can proceed with an example. First, *when a word is called into the arithmetic unit from the memory, it is* not *erased from* *its memory address*, but *remains* there also. Second, *when a word is put into a memory address, it replaces the previous contents of this address*, i.e., it erases what had been there.

TABLE 3-5. OPERATION CODES

Code	Operation	Meaning
53	Add	Add contents of address α to contents of address β; put sum into address γ; take next instruction from address δ
42	Multiply	Multiply contents of address α by contents of address β; put product into address γ; take next instruction from address δ
54	Subtract	Subtract contents of address β from contents of address α; put difference into address γ; take next instruction from address δ
41	Divide	Divide contents of address α by contents of address β; put quotient into address γ; take next instruction from address δ

Example 3-1. As we have noted above, a *sequence* of instructions is required for a problem. We shall illustrate such a sequence for a code to evaluate

$$Y = V_{0_y}t - \tfrac{1}{2}gt^2$$

where Y = altitude of a missile projected with an initial Y velocity component of V_{0_y} computed at time t. Suppose that $V_{0_y} = 1{,}000$ fps, $g = 32$ ft/sec^2, and $t = 53$ sec. Now 1,000 (*decimal*) = 1750 (*octal*), 32 (*decimal*) = 40 (*octal*), and 53 (*decimal*) = 65 (*octal*). Let us put the numbers

1750 into address 10
40 into address 11
65 into address 12
and 2 into address 13

Let us reserve address 14 for the result of the product $V_{0_y}t$, address 15 for t^2, address 16 for gt^2, address 17 for $\tfrac{1}{2}gt^2$, and finally address 20 for $V_{0_y}t - \tfrac{1}{2}gt^2$. The first instruction would be "Multiply the contents of address 10 by the contents of address 12, and put the result into address 14," where we have yet to indicate where the next instruction will be found. This is written as

Operation	α	β	γ	δ
42	010	012	014	?

Now let us put this instruction into address 1. Next we would write an instruction to form t^2 and put this instruction into address 2. Now we can finish the first instruction,

42 010 012 014 002

The third instruction would multiply t^2 by g; the fourth instruction would divide gt^2 by 2; and the last instruction would subtract $\frac{1}{2}gt^2$ from $V_{0y}t$. A code is written on a coding sheet, which usually has the left-hand column labeled with addresses and the remaining columns labeled with the parts of an instruction. In this way a clear account is made of exactly which instructions are located in which addresses. On a coding sheet our code would look like Example 3-1*a*. The student should *carefully check* each instruction in this code to be sure that each does what the "remark" says it does. There are several things missing from this code: How do we get the computer started, and how do we stop it when the job is finished?

EXAMPLE 3-1*a*

Address	Instruction					Remarks
	Operation	α	β	γ	δ	
000						
001	42	010	012	014	002	Form $V_{0y}t$
002	42	012	012	015	003	Form t^2
003	42	015	011	016	004	Form gt^2
004	41	016	013	017	005	Form $\frac{1}{2}gt^2$
005	54	014	017	020	006	Form Y
006			See text			
007						
010	00	000	000	001	750	V_{0y} constant
011	00	000	000	000	040	g constant
012	00	000	000	000	065	t constant
013	00	000	000	000	002	2 constant
014						$V_{0y}t$ temporary
015						t^2 temporary
016						gt^2 temporary
017						$\frac{1}{2}gt^2$ temporary
020						$Y = V_{0y}t - \frac{1}{2}gt^2$

There are many systems in use for initiating a code on a computer. For example, after the code is read into the computer, the computer might *automatically* take the very first instruction from address 000. In our example we would then have a *dummy instruction*, say "Subtract contents of 007 from contents of 007, put result in 007, and *take next instruction from* 001." We used 007 because our particular code does not involve 007. In order to stop the computer when the computation has been completed, the last instruction should be one that says simply "Stop the computer." For instance, suppose that 60 is the operation code for such an instruction; then we could put

60 000 000 000 000

in address 006. In such an instruction the addresses α, β, γ, and δ have no use.

The Used Temporaries. Memory space in a computer is often at a premium and should be conserved. From this point of view the above code is not satisfactory. For note that, after t^2 has been computed, address 015 is not used again. In other words, we could have computed gt^2 and put this result back into 015, and then computed $\frac{1}{2}gt^2$ and put this back into 015 also, eliminating the necessity for using 016 and 017. Similarly, after computing Y, we could put this result into 015, eliminating the use of 020 also.

This brings up an important point about addresses reserved for numbers. There are two kinds of numbers contained in addresses: "permanent," or constant, numbers, such as 2 and g, and "temporary," or transient, numbers, i.e., numbers that appear only in intermediate stages of the computation. These temporary numbers can share their address with other temporary numbers that appear at some other time. Hence we call such memory locations *temporaries.* In our example above, 015 would be such a temporary. The adjusted code becomes Example 3-1*b*.

EXAMPLE 3-1*b*

Address	Instruction					Remarks
	Operation	α	β	γ	δ	
000	54	007	007	007	001	Take first instruction from 001
001	42	010	012	014	002	Form $V_{0y}t$
002	42	012	012	015	003	Form t^2
003	42	015	011	015	004	Form gt^2
004	41	015	013	015	005	Form $\frac{1}{2}gt^2$
005	54	014	015	015	006	Form Y
006	60	000	000	000	000	Stop computer
007						
010	00	000	000	001	750	V_{0y} constant
011	00	000	000	000	040	g constant
012	00	000	000	000	065	t constant
013	00	000	000	000	002	2 constant
014						$V_{0y}t$ temporary
015						Temporary, where final result is found

EXERCISES

(*a*) Using the instructions defined in this section, write a code that calculates $x = a^3 - bc$, where $a = 313$ (*decimal*), $b = 983$ (*decimal*), and $c = 838$ (*decimal*).

(*b*) In the example given in this section why are two temporaries needed, that is, 014 and 015? Write a code using only one temporary. [HINT: $V_0t - \frac{1}{2}gt^2 = (V_0 - \frac{1}{2}gt)t$.]

(c) If an instruction format for a computer allowed 12 bits for each address, what is the maximum number of addressable words possible for the memory of that computer? [*Ans.* 2^{12} = 4,096 (*decimal*) words.] If the first address of this memory is 0000, what is the last memory address? [*Ans.* 7777 (*octal*) (that is, $2^{12} - 1$ written in octal).]

3-5. Coding Decisions

The Comparison Instruction. We shall next give an example to be coded where a simple decision is made by the computer. Such decisions are usually made by comparison instructions. For example, a comparison instruction might operate thus: "Compare the contents of α with the contents of β; if the contents of address α is greater than the contents of address β, then take the next instruction from address γ; otherwise, take the next instruction from address δ." A few simple conventions can greatly shorten such a statement: Let (α) stand for "the contents of address α"; and let the word *take* mean "take the next instruction from." Then the above statement becomes "If $(\alpha) > (\beta)$, take γ; if $(\alpha) \leq (\beta)$, take δ." Let the operation code for this instruction be 43.

Example 3-2. Now consider the code for the following problem: Three numbers, U, V, and W, are found as the contents of addresses 015, 016, and 017, respectively. We wish to write a code that will put the largest of these numbers into address 020. First note that in any case we shall need three instructions that transfer the contents of addresses 015, 016, and 017, respectively, into address 020, and the rest of the code is to decide which of these three instructions to execute. The transfer problem is easily taken care of by adding the contents of the desired address to the constant zero and putting the result into 020. (Hence we need an address whose contents is zero.) Let us put the three such instructions into 005, 006, and 007 (see the code below). To determine which of these to execute, let us first compare $(015) \overset{?}{>} (016)$. If this is the case, then we would compare $(015) \overset{?}{>} (017)$. If this is also the case, then $(015) >$ both (016) and (017), that is, $U > V$ and $U > W$, and hence we would execute (005), i.e., transfer (015) to (020). If $(015) \leq (017)$, then $(017) \geq (015) > (016)$, that is, $W \geq U > V$; hence we transfer (017) to (020), i.e., execute (007). If $(015) \leq (016)$, then compare $(016) \overset{?}{>} (017)$. If this is the case, $(016) \geq$ both (015) and (017), that is, $V \geq U$ and $V > W$; hence we transfer (016) to (020), i.e., execute (006). If $(016) \leq (017)$, then $(017) \geq (016) \geq (015)$, that is, $W \geq V \geq U$; hence transfer (017) to (020), i.e., execute (007). The code is shown in Example 3-2*a*.

Several general observations can be made that are exemplified by this code. First, coding is a very exacting technique: every path of every instruction must be followed carefully, for a *single* error invalidates the *entire* code. Second, note that when using a comparison instruction the significance of the $\leq$ must be carefully considered, as well as that of the $>$. Reading a code does *not* teach one how to code—actually writing

EXAMPLE 3-2*a*

Address	Instruction					Remarks
	Operation	α	β	γ	δ	
000	54	021	021	021	001	Take first instruction from 001
001	43	015	016	002	003	$U > V$? Yes, try $U > W$; i.e., take 002 No, try $V > W$; i.e., take 003
002	43	015	017	005	007	$U > W$? Yes, U is largest number; i.e., take 005 No, W is largest number; i.e., take 007
003	43	016	017	006	007	$V > W$? Yes, V is largest number; i.e., take 006 No, W is largest number; i.e., take 007
004						
005	53	015	021	020	010	Transfer U to largest number cell
006	53	016	021	020	010	Transfer V to largest number cell
007	53	017	021	020	010	Transfer W to largest number cell
010	60	000	000	000	000	Stop computer
011						
012						
013						
014						
015						U
016						V three given numbers
017						W
020						Largest number cell
021	00	000	000	000	000	0 constant

a code does. Most students will undoubtedly feel uneasy about the above example until they rewrite it, doing the detailed thinking themselves, using the example only as an occasional guide and final check.

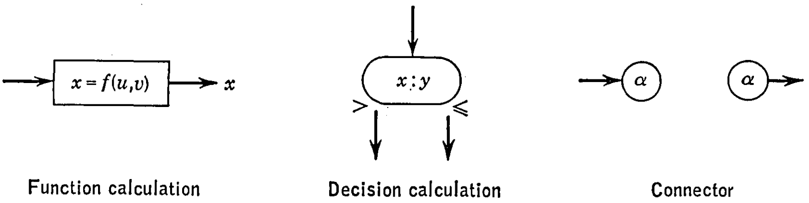
The Sign Bit. Up to now we have been making two tacit assumptions, namely, (1) that all numbers are whole numbers (i.e., have no fractional part) and (2) that all numbers and results are positive. Both the assumptions are not necessarily true. They were made in order to concentrate on the coding problems so far considered and avoid unnecessary complication in those discussions. We shall defer full consideration of the first of these problems for a later section, remarking here only that a computer can consider numbers whose magnitudes lie within a restricted range which depends on the construction of the computer. We have been illustrating a computer with a range of integers from 0 through $2^{42} - 1$. Hence, before putting a problem on a computer, it must be "scaled" so that all numbers that appear will be whole numbers. Similarly problems must be scaled to fit on other computers, depending on their ranges. The second problem is solved as follows: A single bit is added on the left to the length of the word, making the word 43 bits long. If this bit is a zero, then the machine is wired to interpret the number as being positive; if this bit is a unit, then the machine will interpret the number as being negative. The circuitry in the arithmetic unit must sense the signs of the operands and compute the proper values for this so-called *sign bit* in the results. The meaning attached to this 43d bit, as far as an instruction is concerned, is different for different computers and is discussed below; it is never a fundamental ingredient of the instruction. On the coding sheet the sign bit is noted by affixing a + to the left of the number if it is positive or zero, and a — if it is negative.

EXERCISE

(*a*) Using the instructions defined in this and the preceding section, write a code that calculates X = minimum (U,V,W), where U, V, and W are the contents of addresses 010, 011, and 012, respectively.

3-6. Coding: Flow Charting and Symbolic Code Aids

A *flow chart* is a diagram, or picture, of a code that is often helpful for visualizing interrelationships between various parts of a code. Such a diagram is almost always made before the specific instructions are written. There are essentially three kinds of symbols used in a flow chart (see Fig. 3-1). The first represents *function calculation;* the second represents *decisions* and the various associated alternatives; and the third, called a (variable) *connector,* is simply a way to eliminate too many crossing lines in the picture or to indicate which lines to follow when one has to continue the diagram on another page.



Function calculation

Decision calculation

Connector

FIG. 3-1. Flow-chart symbols.

For illustration, consider the example of Sec. 3-4: The functions $V_{0y}t$ and $\frac{1}{2}gt^2$ are to be calculated, and their difference is the final desired result. Hence we would draw the flow diagram in Fig. 3-2. The first (left) box represents the instruction in address 001; the second box represents the set of instructions in addresses 002 to 004; the third box represents the instruction in address 005. The arrows indicate the order in which the computer does the indicated computing. Note that the symbols *representing intermediate results appearing on the left-hand side of the equations*, namely, x and y, *appear later in the chart on the right-hand side of an equation.* Only symbols that signify end results, like Y, will not appear on the right-hand side of an equation. This observation gives a superficial way to check that all intermediate results are used appropriately.

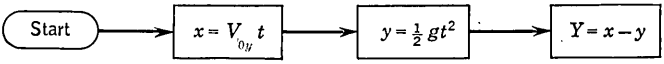


FIG. 3-2. Simple flow chart.

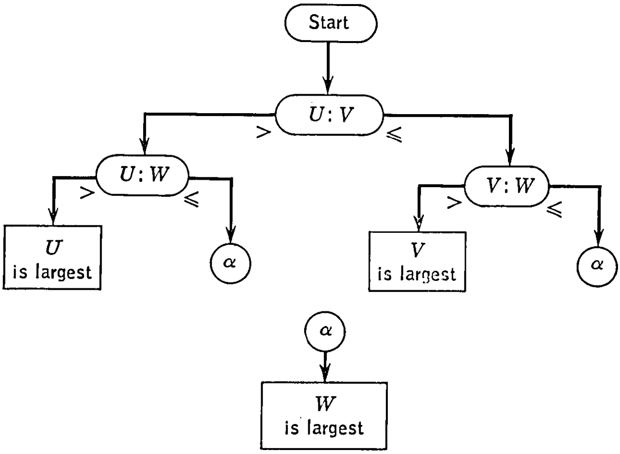


FIG. 3-3. Flow chart including decision boxes.

A more complicated flow diagram is associated with the problems of Sec. 3-5. Here decision boxes are used (see Fig. 3-3). The lines leading away from the ovals represent the different paths the computer will take, depending on the outcomes of the decisions. The little symbols ($>$ and $\leq$) near the lines represent the conditions under which the program will go in the corresponding directions. *The proper corresponding condition is indicated when the symbol replaces the colon;* i.e., the computer will follow the left-hand path when $U > V$ and the right-hand path when $U \leq V$. The use of the connector is clear. The top decision box represents (001); on the next lower level the left-hand box represents (002), the right-hand box (003). On the next lower level the left-hand function box represents (005), the right-hand box (006), and the lowest box (007).

Symbolic coding is another intermediate aid between the statement of the problem and the final code. Symbolic coding consists in writing a code, *not in terms of specific numerical addresses*, but rather in terms of some name description or other symbolism to represent the addresses. Then at a later time specific addresses can be assigned for these symbols, or names, to produce the actual code. The intermediate code in terms of symbols is called the *symbolic code.* This technique is extremely useful, particularly in those cases where one must write instructions involving addresses of constants or of other instructions that have not yet been specifically assigned. For example, this difficulty is encountered in writing the code for the second example of the previous section, since when writing the instruction for $U:V$, the coder must refer to the not as yet written or located instructions for $U:W$ and $V:W$. A symbolic code for this example might be as shown in Example 3-3*a*. The symbolism of this code is as follows: **U** is the address whose contents is the number *U*—that is, (**U**) = U. **V** is the address such that (**V**) = V. **W** is the address such that (**W**) = W. **U:V** is the address whose contents has the *compare* instruction concerning U and V, and similarly for **U:W** and **V:W**; **ULN** is the address of the instruction that puts the number U into the largest number cell, and similarly for **VLN** and **WLN**. **LNC** is the address of the largest number cell; **ϕ** is the address whose contents is the constant zero; and **S** is the address of the stop instruction. **CMP** stands for the *compare* operation code, and **ADD** stands for the *add* instruction code. The assignment of addresses then is shown in Example 3-3*b*.

EXAMPLE 3-3*a*

Address	Instruction				
	Operation	α	β	γ	δ
U:V	CMP	U	V	U:W	V:W
U:W	CMP	U	W	ULN	WLN
V:W	CMP	V	W	VLN	WLN
ULN	ADD	U	ϕ	LNC	S
VLN	ADD	V	ϕ	LNC	S
WLN	ADD	W	ϕ	LNC	S
S					
U					
V					
W					
LNC					
ϕ					

Steps in Writing a Code. The procedure for programming or coding is as follows: (1) Understand the statement of the problem. (2) Draw a flow diagram for the problem. (3) Code symbolically each box of the flow diagram. (4) Assign addresses. (5) Write out the final detailed code. The flow diagram presents an *over-all visual picture* of the prob-

EXAMPLE 3-3*b*

Symbolic name	*Address (or code)*
U:V	001
U:W	002
V:W	003
ULN	005
VLN	006
WLN	007
S	010
U	015
V	016
W	017
LNC	020
φ	021
CMP	43
ADD	53

lem, *organizing the coding effort* so that it can be concentrated on one box at a time. Symbolic coding enables an instruction to be written before the coder knows the addresses of constants and other instructions referenced by this instruction. In addition, since the symbolic code names for these addresses are suggestive of their contents, the symbolic code is easier to write and follow.

EXERCISES

Using the instructions defined in the previous section, draw the flow diagram, write a symbolic code, assign addresses, and finally write the final codes to calculate x [where $a = 313$ (*decimal*), $b = 983$ (*decimal*), and $c = 838$ (*decimal*)]:

(*a*) $x = a^3 - b \cdot c$.

(*b*) $x = b(a + c)$.

(*c*) $x = \dfrac{a^3 - b \cdot c}{b(a + c)}$.

If U, V, and W are the contents of addresses 010, 011, and 012, respectively, calculate:

(*d*) $x = \text{minimum } (U,V,W)$.

(*e*) $x = \text{maximum } (U^2,V^2,W^2)$.

(*f*) $x = \text{maximum } (U^3 - VW,\ V(U + W))$.

HINT: The same symbolic code for Exercises *a* and *b* appears as parts of the symbolic code for Exercises *c* and *f*.

3-7. Three-address Instruction Systems

Using Less than Four Addresses. Up to now we have considered an instruction format containing four addresses. It was remarked in Sec. 3-1 that each of the four addresses is a necessary ingredient to coding and that computers using fewer addresses must compensate by (1) having the omitted information wired in and by (2) using more instructions to achieve the same ends. From the engineering point of view the primary advantage of having fewer than four addresses in the instruction is that for the same word length each address will comprise more bits and hence give the computer a larger addressable memory capability. In the 43-bit format described in Sec. 3-4, 36 bits were reserved for the four addresses, making each address 9 bits long, for a maximum of 512 words in the addressable memory. If three addresses were used, each would comprise 12 bits, allowing a 4,096-word memory; two addresses, 18 bits per instruction, a 262,144-word memory; one address, 36 bits, 68,719,476,736 words. Of course shorter words can be designed for machines with one- or two-address instructions, and this in itself can, as we shall see, save a great deal of hardware.

The Three-address Instruction. The usual† three-address instruction is similar to the four-address instruction except that the δ address is omitted from the format. Hence the format becomes

Operation code	Address of first operand, α	Address of second operand, β	Address into which result is put, γ

How does the computer know in what address to find the next instruction? Except for special instructions the computer will always take as the next instruction the contents of the *next consecutive address* following that of the present instruction. That is, the computer will normally take the instructions consecutively in the order in which they are written into the memory. This is accomplished in the *current-address register.*

EXAMPLE 3-1c. THREE-ADDRESS

Address	Instruction				Remarks
	Opera-tion	α	β	γ	
0000	53	0007	0007	0007	Take first instruction from 0001
0001	42	0010	0012	0014	Form $V_{0y}t$
0002	42	0012	0012	0015	Form t^2
0003	42	0015	0011	0015	Form gt^2
0004	41	0015	0013	0015	Form $\frac{1}{2}gt^2$
0005	54	0014	0015	0015	Form Y
0006	60	0000	0000	0000	Stop computer
0007					
0010	00	0000	0000	1750	V_{0y}
0011	00	0000	0000	0040	g
0012	00	0000	0000	0065	t
0013	00	0000	0000	0002	2 constant
0014					$V_{0y}t$ temporary
0015					Temporary, where final result is found

† Any of the addresses might be omitted at the whim of the designer.

This register is first set at the address of the first instruction; then, each time an instruction is executed, the contents of this register normally is automatically increased by 1 and is taken by the computer as the address of the next instruction. (Because of this the current-address register is often called the *current-address counter.*) With this understanding of where the computer will find the next instruction the only change necessary in the *add*, *multiply*, *divide*, and *subtract* instructions given in Sec. 3-4 is the deletion of the δ address. However, the comparison instruction will need redefining thus: If $(\alpha) > (\beta)$, take next instruction from γ; otherwise take the next consecutive instruction. If $(\alpha) > (\beta)$, then the current-address counter will automatically be set at address γ; if $(\alpha) \leq (\beta)$, the counter will behave as usual, just increasing by 1.

Examples. For instance, the examples coded in Secs. 3-4 and 3-5 would have to be rewritten as shown in Examples 3-1*c* and 3-2*b*.

There is not much difference between Example 3-1*c* and its four-

Example 3-2*b*. Three-address

Address	Instruction				Remarks
	Operation	α	β	γ	
0000	54	0021	0021	0021	Take first instruction from 0001
0001	43	0015	0016	0005	$U:V$ If $U > V$, take 0005 If $U \leq V$, take 0002
0002	43	0016	0017	0007	$V:W$ If $V > W$, take 0007 If $V \leq W$, take 0003
0003	53	0017	0021	0020	Transfer W to largest number cell
0004	43	0022	0021	0012	Jump to stop
0005	43	0015	0017	0011	$U:W$ If $U > W$, take 0011 If $U \leq W$, take 0006
0006	43	0022	0021	0003	Jump to transfer W instruction
0007	53	0016	0021	0020	Transfer V to largest number cell
0010	43	0022	0021	0012	Jump to stop
0011	53	0015	0021	0020	Transfer U to largest number cell
0012	60	0000	0000	0000	Stop
0013					
0014					
0015					$\left.\begin{matrix}U\\V\\W\end{matrix}\right\}$ given numbers
0016					
0017					
0020					Largest number cell
0021	00	0000	0000	0000	0 constant
0022	00	0000	0000	0001	1 constant

address version. Consider, however, Example 3-2*b*. Each branch of this code should be carefully reconstructed by the student so that he may grasp just what problems arise from ordering the instructions and how these problems are solved. Note that there appear in addresses 0004 and 0010 *dummy comparisons,* which artificially force a jump to address 0012, where the stop instruction is located. Also, a dummy comparison was made in address 0006 in order to reuse (0003). A total of 11 instructions were necessary in the three-address system, while 8 sufficed in the four-address system. The placement of the instructions was somewhat complicated; this difficulty is relieved by *symbolic coding.*

EXERCISES

(*a*) By rearranging the order of one of the comparisons together with the order of some of the instructions in Example 3-2*b* above, one instruction can be eliminated—i.e., the code will take only 10 instructions instead of 11. Can you find this comparison and rearrange the instructions properly? (HINT: In address 0005 the instruction can be changed so as to jump directly to make W the largest number. But then some other instructions must be reordered.)

(*b*) Redefine in detail for a three-address system the *add, subtract, multiply,* and *divide* instructions given in Sec. 3-4.

(*c*) Draw flow diagrams for the above examples, and state which sets of instructions belong to each box.

(*d–i*) Do Exercises *a* to *f* of Sec. 3-6, using the three-address instructions defined in this section. Carry out each of the five steps of coding in every problem.

3-8. Two-address Instruction Systems

The Two-address Instruction. In the two-address instruction usually both the γ and δ addresses are eliminated. The format is simply

Operation code	Operand address α	Operand address β

The computer finds the location of the next instruction as in the three-address system, taking care of the function of δ. Where does the result of the operation go? This problem is solved in a two-address system by having the result of every arithmetic operation automatically left in a very special memory location, called the *accumulator.* This special memory cell is located in the arithmetic unit; it has been discussed in Chap. 2 of this book. It suffices to note here that γ is most often understood to be the accumulator, although as we shall see in Chap. 4 there can be many variations to this theme. Hence the definitions of our instructions must be rewritten as shown in Table 3-6.

Of course, after the sum is in the accumulator, it must be transferred to an ordinary memory location. Hence we need a transfer instruction. Other versions which might be used are also listed (cf. Chap. 4).

Examples. As mentioned above, the word length is often shortened for two- or one-address systems; so let us, for illustration purposes, use a

TABLE 3-6. TWO-ADDRESS INSTRUCTIONS

Code	Operation	Meaning
53	Add	Add (α) to (β), and put result in accumulator
42	Multiply	Multiply (α) by (β), and put result in accumulator
54	Subtract	Subtract (β) from (α), and put result in accumulator
41	Divide	Divide (α) by (β), and put result in accumulator
43	Compare	If (α) > (acc), take β; otherwise go to next consecutive instruction (as usual)
52	Transfer	Transfer (acc) into α, and take β
51	Add and transfer	Add (α) to (acc), and put result into β
40	Multiply and transfer	Multiply (α) by (acc), and put result into β

word length of 24 bits plus one sign bit. In order to increase flexibility, *the accumulator is often given an address*, say, for example, 777. The example codes as revised are shown as Examples 3-1*d* and 3-2*c*. Note that two more instructions were needed in Example 3-1*d* than for the three- or four-address codings. The contents of the accumulator must be carefully watched, or errors might creep in.

In Example 3-2*c* note how the comparisons were arranged so that the next instruction in sequence could be a transfer instruction direct from the accumulator. This was successfully accomplished for V and U. However, W had to be placed in the accumulator especially for its trans-

EXAMPLE 3-1*d*. TWO-ADDRESS

Address	Instruction			Remarks
	Operation	α	β	
000	54	017	017	Take first instruction from 001
001	42	010	012	Form $V_{0y}t$ in accumulator
002	52	014	003	Put $V_{0y}t$ into 014
003	42	012	012	Form t^2 in accumulator
004	42	011	777	Form gt^2 in accumulator
005	41	777	013	Form $\frac{1}{2}gt^2$ in accumulator
006	54	014	777	Form Y
007	52	015	016	Put Y into 015
010	00	001	750	V_{0y}
011	00	000	040	g
012	00	000	065	t
013	00	000	002	2
014				$V_{0y}t$ temporary
015				Temporary, where result is found
016	60	000	000	Stop computer
017	00	000	000	0 constant

fer to the largest number cell. In any event it is to be noted that the placement of instructions and the order of the comparisons become more complicated in a two-address system. Note also that 11 instructions were needed, one more than was absolutely necessary for the three-address system (see Exercise *a* of Sec. 3-7) and three more than for the four-address system.

EXAMPLE 3-2*c*. TWO-ADDRESS

Address	Instruction			Remarks
	Operation	α	β	
000	54	021	021	Take first instruction from 001
001	53	016	021	Put V into accumulator
002	43	015	005	$U:V$
				If $U > V$, take 005
				If $U \leq V$, take 003
003	43	017	010	$W:V$ (since V is already in accumulator)
				If $W > V$, take 010
				If $W \leq V$, take 004
004	52	020	012	Transfer V from accumulator into 020
005	53	015	021	Put U into accumulator
006	43	017	010	$W:U$
				If $W > U$, take 010
				If $W \leq U$, take 007
007	52	020	012	Transfer U from accumulator into 020
010	53	017	021	Transfer W into accumulator
011	52	020	012	Transfer W from accumulator into 020
012	60	000	000	Stop computer
013				
014				
015				U
016				V } given numbers
017				W
020				Largest number cell
021	00	000	000	0 constant

EXERCISES

(*a*) Relate sets of instructions of the above examples to their respective boxes of the flow diagrams.

(*b–g*) Do Exercises *a* to *f* of Sec. 3-6, using the two-address instructions defined in this section.

3-9. One-address Instruction Systems

The One-address Instruction. In the one-address instruction only one operand address, α, remains. The accumulator does a dual job, standing in for both the second operand address β and the address γ into which the result is put. Address δ is handled as in the three- and two-address systems. There are a large number of variations on this theme, and often additional registers or special memory locations are used. However, in presenting the principles of one-address coding it suffices here to consider only one register, the accumulator. In a one-address system many more kinds of instructions are found essential. For example, in a typical modern one-address computer there are 88 different instructions. This should be compared with the 46 and 16 instructions defined for typical modern two- and three-address computers. It is evident that, the more complex and numerous the instructions that have to be learned by the coder, the harder it is for him to code. In order to help the coder who deals with a machine with, say, 88 instructions, advanced programming techniques are resorted to; these will be discussed in Chap. 5. The necessity for advanced programming techniques to aid the coder prepare programs for a one-address computer system becomes even more striking when one reads the rather tortuous techniques needed to accomplish the simplest of complete operations, such as comparisons, as illustrated by the examples to be presented below. The list of one-address instructions in Table 3-7 is sufficient to meet our needs for the examples.

TABLE 3-7. ONE-ADDRESS INSTRUCTIONS

Code	Operation	Meaning
53	Add	Add (α) to (acc); put result in accumulator
42	Multiply	Multiply (α) by (acc); put result in accumulator
54	Subtract	Subtract (α) from (acc); put result in accumulator
41	Divide	Divide (acc) by (α); put result in accumulator
43	Conditional jump	If (acc) is negative, take α; otherwise take next instruction in sequence as usual
52	Transfer	Transfer (acc) into α
51	Replace add	Add (α) to (acc); put result into α
40	Replace multiply	Multiply (α) by (acc); put result into α
44	Jump	Take next instruction from α

The transition from two to one address for the *add*, *multiply*, *subtract*, and *divide* instructions is natural enough, but how is comparison performed? To perform a comparison, one of the numbers is put into the accumulator and the other subtracted from it; then a *conditional jump* is used. (A computer with more than one register may compare the contents of two registers.)

Examples. Let us shorten our word to 15 bits plus a sign bit, for convenience in this presentation; then the codes for our examples can be written as shown in Examples 3-1*e* and 3-2*d*.

Note, for the first, that it is necessary to make sure that the contents of the accumulator was zero before using the *add* instruction to transfer a number into the accumulator. Also we calculate $\frac{1}{2}gt^2$ first so that after forming $V_{0y}t$ in the accumulator we can subtract $\frac{1}{2}gt^2$ directly from the accumulator, thus avoiding an extra clearing, etc., of the accumulator. More instructions are needed than for the other systems: one-address, 12; two-address, 7; three-address, 7; four-address, 7. One less instruction than used will suffice: the student should check this for himself, revising the code to require one less instruction.

EXAMPLE 3-1*e*. ONE-ADDRESS

Address	Instruction		Remarks
	Operation	α	
000	44	001	Take first instruction from 001
001	42	016	Clear accumulator, and form t^2
002	53	021	
003	42	021	
004	42	020	Form gt^2
005	41	022	Form $\frac{1}{2}gt^2$
006	52	023	Put $\frac{1}{2}gt^2$ into 023
007	42	016	Clear accumulator, and form $V_{0y}t$
010	53	017	
011	42	021	
012	54	023	Form $Y = V_{0y}t - \frac{1}{2}gt^2$
013	52	023	Put Y into 023
014	60	000	Stop computer
015			
016	00	000	0 constant
017	01	750	V_{0y}
020	00	040	g
021	00	065	t
022	00	002	2
023			$\frac{1}{2}gt^2$ temporary and result

There are 25 instructions in the code of the second example as compared with 11 for the corresponding code in the two-address system, 10 in the three-address system, and 8 in the four-address system. By choosing a more convenient set of one-address instructions (cf. Chap. 4) we might have reduced the length of the code somewhat, but not significantly.

One-plus-one-address Systems. There is another kind of two-address instruction that deserves mention. It is a two-address instruction where the α and δ addresses are retained, instead of the α and β addresses. In other words, an addition instruction might read: Add (α) to the accumulator, and take the next instruction from δ. Such a system has the coding characteristics of a one-address system; the purpose of the second address is to allow minimum access coding when a drum is being used

EXAMPLE 3-2*d*. ONE-ADDRESS

Address	Instruction		Remarks
	Operation	α	
000	44	001	Jump to first instruction of code, 001
001	42	035	Clear accumulator
002	53	032	Put V into accumulator } set up comparison
003	54	031	Form $V - U$ in accumulator
004	43	015	If (acc) negative, $U > V$, try $U{:}W$; if positive, $U \leq V$, try $V{:}W$
005	42	035	Clear accumulator
006	53	032	Put V into accumulator } set up comparison
007	54	033	Form $V - W$ in accumulator
010	43	025	If (acc) negative, $W > V$, W largest; if positive, $W \leq V$, V largest
011	42	035	Clear accumulator
012	53	032	Put V into accumulator } V is largest number
013	52	034	Put V into largest number cell
014	44	030	Jump to stop
015	42	035	Clear accumulator
016	53	031	Put U into accumulator } set up comparison
017	54	033	Form $U - W$ in accumulator
020	43	025	If (acc) negative, $W > U$, W largest; if positive, $W \leq U$, U largest
021	42	035	Clear accumulator
022	53	031	Put U into accumulator } U is largest number
023	52	034	Put U into largest number cell
024	44	030	Jump to stop
025	42	035	Clear accumulator
026	53	033	Put W into accumulator } W is largest number
027	52	034	Put W into largest number cell
030	60	000	Stop computer
031			U
032			V } given numbers
033			W
034			Largest number cell
035	00	000	0 constant

(see Chap. 2). That is, the second address presents some further flexibility in the placing of the instructions in the drum memory. This is often called a *one-plus-one-address system.*

EXERCISES

(*a*) Relate sets of instructions of the above codes with their respective boxes in the flow diagrams.

(*b–g*) Do Exercises *a* to *f* of Sec. 3-6, using the one-address instructions defined in this section.

(*h*) Define one-plus-one instructions corresponding to the one-address instructions given above; then redo the second example. Have a significant number of instructions been saved?

3-10. Decimal Systems

Decimal-coded Binary. The only variations in the coding system so far described were with respect to the instruction format and, in particular, to the number of addresses that were explicitly displayed in this format. In all these systems numbers were written in *octal-coded binary;* i.e., a set of octal characters represented a binary number. In fact the three bits represented by each octal character corresponded to the same number with radix 2: 3 (*octal*) represented 011, 5 (*octal*) represented 101, etc. The instructions were written with the same shorthand, the octal character representing the real equivalent 3-bit binary number.

There is another system for coding numbers and an associated shorthand for representing the bits in an instruction, called *decimal-coded binary.* Of course, either octal-coded binary or decimal-coded binary can be used with four-, three-, two-, and one-address instruction formats.

Usually, in decimal-coded binary, 6 bits are associated with a single decimal number. Since there are 64 different combinations that can be formed with 6 bits, the letters of the alphabet are also assigned binary codes. These assigned codes depend on the logical design of the computer. For example, one such system (UNIVAC) is as shown in Table 3-8.

TABLE 3-8. UNIVAC X-3 CODE

Symbol	*Binary code*	*Symbol*	*Binary code*	*Symbol*	*Binary code*	*Symbol*	*Binary code*
0	00 0011	*A*	01 0100	*J*	10 0100	*S*	11 0101
1	00 0100	*B*	01 0101	*K*	10 0101	*T*	11 0110
2	00 0101	*C*	01 0110	*L*	10 0110	*U*	11 0111
3	00 0110	*D*	01 0111	*M*	10 0111	*V*	11 1000
4	00 0111	*E*	01 1000	*N*	10 1000	*W*	11 1001
5	00 1000	*F*	01 1001	*O*	10 1001	*X*	11 1010
6	00 1001	*G*	01 1010	*P*	10 1010	*Y*	11 1011
7	00 1010	*H*	01 1011	*Q*	10 1011	*Z*	11 1100
8	00 1011	*I*	01 1100	*R*	10 1100		
9	00 1100						

In such decimal computers a decimal number is written into a word "as is," with no conversion required. Of course the resulting binary number is not a number at all, but a binary code. The arithmetic unit then handles 6 bits at a time, and is wired so as to produce the correct results. For example, in octal-coded binary we would have

Octal	*Corresponding binary operation as it would appear in computer words*
5	000 101
+ 6	+000 110
13	001 011

In decimal-coded binary we would have

Decimal	
5	000 011 001 000
+ 6	+000 011 001 001
11	000 100 000 100

In a later chapter examples of decimal adders, etc., will be presented.

Note that a word that contains letters *cannot* be added to another word since addition is defined only for numbers. Also, an address must be denoted by pure *numbers*, not letters (for as we shall see in Chap. 4, we must be able to add to or subtract from the addresses). The letters can be used only for the operation code or for some letter constant.

The advantages of a decimal system are simply that no conversion need be made for numbers and that letters can be read into and out of the computer directly. However, the disadvantage of a decimal system is that it is difficult to write codes where the value of a particular single bit of a word has significance, as is necessary, for instance, in switch sensing; while there are methods for determining the value of some bit of the word, in general the bits are used in the computer in groups of six.

Example. As an illustration of a code in such a system, consider the three-address instructions of Table 3-9, where one decimal symbol is used for the operation code and three decimal digits are used for each address. Here $(\alpha) + (\beta) \rightarrow \gamma$ means, "The contents of α are added to the contents of β, and the sum is put into address γ," etc.

TABLE 3-9. THREE-ADDRESS DECIMAL INSTRUCTIONS

Code	Operation	Meaning
A	Add	$(\alpha) + (\beta) \rightarrow \gamma$
M	Multiply	$(\alpha) \cdot (\beta) \rightarrow \gamma$
S	Subtract	$(\alpha) - (\beta) \rightarrow \gamma$
D	Divide	$(\alpha) \div (\beta) \rightarrow \gamma$
C	Compare	If $(\alpha) > (\beta)$, take γ
P	Stop computer	

The code for the first of our examples is shown in Example 3-1*f*.

EXAMPLE 3-1*f*. DECIMAL-CODED BINARY

Address	Instruction				Remarks
	Operation	α	β	γ	
000	*S*	007	007	007	Take first instruction from 001
001	*M*	008	010	012	Form $V_{0y}t$
002	*M*	010	010	013	Form t^2
003	*M*	013	009	013	Form gt^2
004	*D*	013	011	013	Form $\frac{1}{2}gt^2$
005	*S*	012	013	013	Form *Y*
006	*P*	000	000	000	Stop computer
007					
008	0	000	001	000	V_{0y}
009	0	000	000	032	g
010	0	000	000	053	t
011	0	000	000	002	2
012					$V_{0y}t$ temporary
013					Temporary where final result is found

EXERCISE

(*a*) Recode the second of our illustrative examples in three-address decimal-coded binary.

3-11. Additional Topics

a. General Background. Read Arithmetic Numbers and the Art of Counting, in James R. Newman (ed.), "The World of Mathematics," vol. 1, pp. 418–453, Simon and Schuster, Inc., New York, 1956.

b. The Real Numbers. The axiomatic development of real numbers is one of the highlights of modern mathematics. A good description of this development is given by H. A. Thurston, "The Number System," pt. I, Interscience Publishers, Inc., New York, 1956.

c. The Division Algorithm. In Sec. 3-2 we used the reasonable result that, if N and q are positive integers, then there always exist a unique nonnegative integer r less than q and a unique integer S such that

$$\frac{N}{q} = S + \frac{r}{q} \qquad 0 \leq r < q$$

This is called the *division algorithm.* For a "proof," see G. Birkhoff and F. MacLane, "A Survey of Modern Algebra," pp. 1–20, The Macmillan Company, New York, 1948.

d. Infinite Decimals: Rational and Irrational Numbers. Observe that the result of converting 0.3 (*decimal*) into binary is .01001100110011 · · · . Of course in a physical situation one might be told, for example, that 0.3 is good to two significant places, i.e., it would be written as 0.30; then there would be no purpose in carrying the binary conversion certainly to more than, say, eight places, i.e., to .01001101 (why?). On the other hand, suppose that we consider 0.3 purely as a number; then we can never write its binary conversion precisely, for it is an infinite binary fraction. Of course we could write 0.3 (*decimal*) $= \frac{11\ (binary)}{1010\ (binary)}$ precisely. It is easy to tell by looking at 0.3 (*decimal*) that it can be written as $\frac{3\ (decimal)}{10\ (decimal)}$. But how could we tell directly by looking at .010011001100 · · · (*binary*) that it can be written as $\frac{11\ (binary)}{1010\ (binary)}$? This problem brings up the observation that there are two kinds of real numbers, rational and irrational. A rational number can be written as the quotient of two integers; an irrational number cannot be so written. The problem stated above thus resolves itself into the following: By looking at an infinite radical fraction (in a system of any radix q) how can rational numbers be distinguished from irrational numbers? For a further discussion of irrational numbers see Edward Kasner and James Newman, "Mathematics for the Imagination," pp. 65–111, Simon and Schuster, Inc., New York, 1940.

e. Computer Manuals. For a short description of both a four-address system (SEAC) and a three-address system (DYSEAC) see Computer Development at the National Bureau of Standards, *NBS Circ.* 551. For a two-address system see the Remington Rand 1103 computer manual, and for a one-address system see the IBM 704 computer manual.

f. Cardiac Computer. Using the Cardiac computer (see Exercise *b* of Sec. 2-8) as a four-address system, compute the codes given in the examples of Secs. 3-4 and 3-5. Do the same for the three-, two-, and one-address coding examples given in Secs. 3-7 to 3-9.

g. Decision Processes. Decision processes require criteria by which the decisions in any particular case are made. If the criteria are sufficient for a particular set of decisions, then a computer can be coded to make them. In fact, it is remarkable that a single comparison instruction can always effectuate this process. For the criteria can be used by the computer to set up numbers representing the various alternatives, and comparison instructions can then choose between them. In recent years the analysis of decision-making processes has received much thought. The student will profit by reading, for example, J. D. Williams, "The Complete Strategyst," McGraw-Hill Book Company, Inc., New York, 1954. This deals with the *theory of games,* which is essentially a theory of value decisions or strategy. See also Herman H. Goldstine and John von Neumann, "Planning and Coding of Problems for an Electronic Computing Instrument," pt. II, vol. I, Institute for Advanced Study, Princeton, N.J., 1947; R. D. Luce and H. Raiffa, "Games and Decisions—Introduction and Critical Survey," John Wiley & Sons, Inc., New York, 1957.

CHAPTER 4

PROGRAMMING FUNDAMENTALS

4-1. Introduction

In the previous chapter we have introduced the concept of an instruction in several different instruction systems. We have discussed in some detail the concept of a sequence of instructions and the significance of the order in which instructions are written. Now we turn to extended and more complicated codes, or sequences of instructions, that are often called programs. As we shall note in a future section, almost any computation requires a fairly intricate sequence of instructions. Although each individual instruction is simple enough, the total collection or sequence of instructions, the program, can become considerably involved.

The present chapter considers first recursion codes, or loops, and instruction modification (Sec. 4-2). These concepts, together with those developed in the previous chapter, comprise all the fundamental concepts necessary for coding and programming. The subsequent material simply amplifies and presents further details of these methods. The discussion of recursion codes leads naturally into a consideration of subroutines (Sec. 4-3). It is in the consideration of subroutines that the elements of automatic programming arise, as considered in detail in Chap. 5.

With the material on recursion codes and subroutines in mind, one can readily appreciate the importance of the large variety of possible instruction types that can be designed for a computer. In the previous chapter the definitions of instructions were kept as simple as possible so as not to interfere with the basic concepts presented there. However, we are now in a position to consider the many possible special-purpose instructions that prove so convenient in many coding situations. The method of presentation of instruction types (Secs. 4-4 and 4-5) is not the giving of precise definitions but rather concentration on the basic concepts involved and on the wealth of combinations and variations that can be obtained. In designing a computer the final definition of the instructions ultimately must be up to the engineer, because of technical electronic and logical design problems. Hence it is important for the engineer to understand the various ingredients that go into the make-up of various instruction types and the special purposes for which they are conceived.

To appreciate fully the significance of some of the instruction types, the engineer should see how they are used in specific codes. In Sec. 4-6 some special coding problems are considered; in each case some of the special instruction types considered in the previous section can be used to full advantage. In almost all the problems considered, the specific codes are left for the student to write. However, in order to write these codes, the student is expected to define precisely those instructions he needs that best suit the purposes of the application. It is felt that the experience gained in defining instructions based on the principles of Secs. 4-4 and 4-5, and then using these instructions in a code, is one of the best possible ways for an engineer to gain flexibility and mastery of instruction design from a computational point of view.

Finally, in Sec. 4-7 the operation of the computer through the control panel is discussed. There the principles of communicating with the computer through the control panel are primarily considered.

4-2. Recursion Codes and Instruction Modification

Loops in the Flow Diagram. In Sec. 3-4 we considered the problem of determining the altitude at a time t of a missile projected with an initial y component of velocity V_{0y}. A more realistic problem would be to determine the trajectory of the missile, i.e., the position of the missile, say, after each 10 sec of its flight. For such a problem we use a *loop*, or *recursion code*, i.e., a code which uses a certain set of instructions over again several times. For example, at time t_i the x and y components of position will be

$$x_i = V_{0x}t_i$$
$$y_i = V_{0y}t_i - \tfrac{1}{2}gt_i^2$$

where V_{0x} is the initial x component of the velocity. To be concrete, suppose that $V_{0x} = 2{,}000$ fps, $V_{0y} = 1{,}000$ fps, and $g = 32$ ft/sec^2. Then at time t_1 ($= 10$ sec) we would have $x_1 = 20{,}000$ ft and

$$y_1 = 10{,}000 - 1{,}600 = 8{,}400 \text{ ft}$$

At time t_2 ($= 20$ sec) $x_2 = 2{,}000 \times 20 = 40{,}000$ ft,

$$y_2 = 1{,}000 \times 20 - 16 \times 20^2 = 13{,}600 \text{ ft}$$

At time t_3 ($= 30$ sec) $x_3 = 2{,}000 \times 30 = 60{,}000$ ft,

$$y_3 = 1{,}000 \times 30 - 16 \times 30^2 = 15{,}600 \text{ ft}$$

and so forth. During such a computation it is clear that the same formulas are used over again, each time increasing our t_i by 10 sec. However, the computation should stop when the missile hits the ground, i.e., when y_i is no longer positive. In our case it is easily seen that this is when $t = 70$ and $y_7 = 70{,}000 - 16 \times 70^2 = -8{,}400$ ft. Table 4-1 represents the final results, and Fig. 4-1 is the flow chart of the process. In the flow chart we have introduced the i notation, where t_{i+1} represents the next time around *and* $i + 1 \rightarrow i$ *means that for the next iteration we replace the old ith values with the new* $(i + 1)$*st values.*

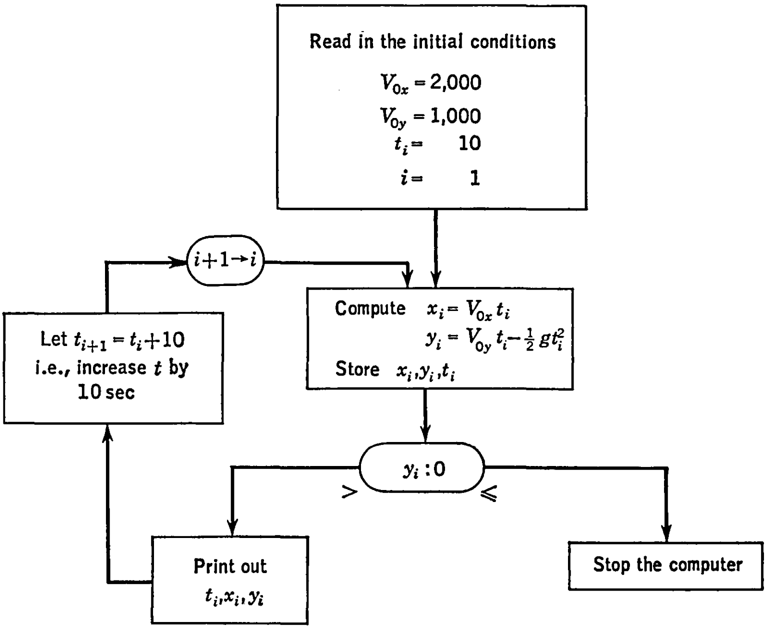
With this illustration it is seen that there are four basic ingredients to a recursion code. It must contain (1) a set of instructions, called the

TABLE 4-1. COMPUTATION OF TRAJECTORY OF MISSILE

i	t_i	x_i	y_i
1	10	20,000	8,400
2	20	40,000	13,600
3	30	60,000	15,600
4	40	80,000	14,400
5	50	100,000	10,000
6	60	120,000	2,400
7	70	140,000	−8,400

iteration instructions, that are to be reused; (2) another set of instructions that modifies the original set each time around; (3) a set of instructions, often called a *tally*, that determines when to *exit*, or break out of the loop, and appropriately notifies the computer; and (4) a set of instructions that sets up the initial conditions and starts the loop. In addition a recursion code often contains (5) a set of instructions that *resets* the loop so that it may be used again by the computer at some future time. A generalized loop can be indicated by the flow diagram of Fig. 4-2.

Sometimes the tally consists of instructions determining whether or not the result of each iteration is smaller than some given number, as occurs often in function computations; or the tally may just count the



Read in the initial conditions
$V_{0x} = 2{,}000$
$V_{0y} = 1{,}000$
$t_i = 10$
$i = 1$
$i+1 \rightarrow i$
Let $t_{i+1} = t_i + 10$ i.e., increase t by 10 sec
Compute $x_i = V_{0x} t_i$
$y_i = V_{0y} t_i - \frac{1}{2} g t_i^2$
Store x_i, y_i, t_i
$y_i : 0$
>
⩽
Print out t_i, x_i, y_i
Stop the computer

FIG. 4-1. Flow chart for computation of missile trajectory.

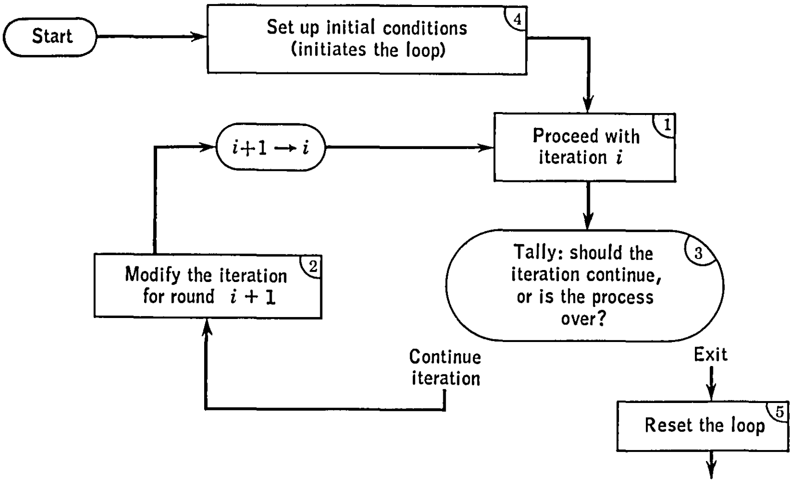

Start
Set up initial conditions (initiates the loop)
4
Proceed with iteration i
1
$i+1 \rightarrow i$
Tally: should the iteration continue, or is the process over?
3
Modify the iteration for round $i+1$
2
Continue iteration
Exit
Reset the loop
5

FIG. 4-2. Flow chart of generalized loop.

number of iterations until the desired number have been accomplished. As an example of the former tally, consider the computation of $\sqrt{N}$ to within an error of ϵ by the following process (see Fig. 4-3): First note that $\sqrt{N}$ is the x (and the y) coordinate where the curves $x = y$ and $N = xy$ intersect. We approach this point by successive approximations starting from (x_0,y_0), where $x_0 = N$, $y_0 = 1$, moving along a perpendicular to $x = y$, dropping an altitude to point (x_1,y_1) on $N = xy$, moving from this

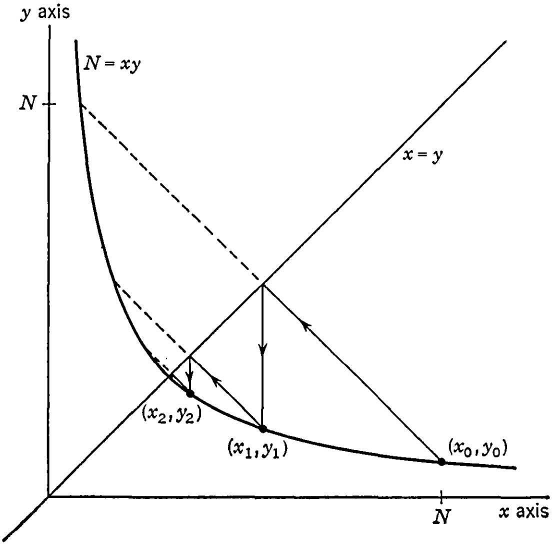


FIG. 4-3. Computation of $\sqrt{N}$.

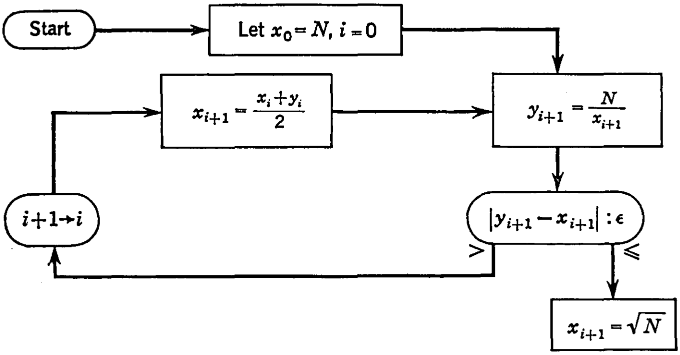

Start
Let $x_0 = N, i = 0$
$x_{i+1} = \frac{x_i + y_i}{2}$
$y_{i+1} = \frac{N}{x_{i+1}}$
$i+1 \to i$
$|y_{i+1} - x_{i+1}| : \epsilon$
$>$
$\leqslant$
$x_{i+1} = \sqrt{N}$

FIG. 4-4. Flow chart for computation of $\sqrt{N}$.

point along a perpendicular to $x = y$, dropping an altitude to point (x_2,y_2) on $N = xy$, and so forth. The next point (x_{i+1},y_{i+1}) is found in terms of the present point (x_i,y_i) as follows:

$$x_{i+1} = \frac{x_i + y_i}{2} \qquad y_{i+1} = \frac{N}{x_{i+1}}$$

(see Fig. 4-3). When $x_i - y_i \leq \epsilon$, the desired accuracy has been obtained. The flow diagram for this loop is shown in Fig. 4-4. A symbolic code for this appears in Example 4-1.

EXAMPLE 4-1. THE SQUARE-ROOT ROUTINE

Address	Operation	α	β	γ	Remarks
000					
001	Divide	N	x	y	$y_{i+1} = \frac{N}{x_{i+1}}$
002	Subtract	y	x	Δ	$x_{i+1} - y_{i+1}$
003	Compare	Δ	ϵ	005	$\Delta : \epsilon$
004	Stop				$x = \sqrt{N}$
005	Add	x	y	x	$\left.\right\} x_{i+1} = \frac{x_i + y_i}{2}$
006	Divide	x	2	x	
007	Compare	1	0	001	
010					N
011					x
012					y
013					ϵ
014					2
015					1
016					0

As an example of the latter tally, consider a code that computes x^n, where x and n are known. Here, after each multiplication, 1 is added to the tally temporary, and comparison is made with the constant n. A symbolic code for this is given in Example 4-2. Note that initially the contents of the x^n temporary must be 1, and the contents of the *tally* temporary must be 0 (why?).

EXAMPLE 4-2. x^n ROUTINE

Address	Operation	α	β	γ	Remarks
000					
001	Multiply	x	x^n temporary	x^n temporary	Form partial product
002	Add	1	Tally temporary	Tally temporary	Adjust tally temporary
003	Compare	n	Tally temporary	001	
004	Stop				
005					x^n temporary
006					Tally temporary
007					n
010					x
011					1

The importance of the recursion code cannot be overemphasized. In fact, as we shall see in a future section, special instructions are included in many computers to aid the coding of such codes.

Instruction Modification. One of the fundamental concepts upon which modern computers are based is that of instruction modification, i.e., having a program automatically modify itself. Let us illustrate how instructions can modify instructions by means of the following examples:

Suppose that, during the course of computing for a problem, the computer often needed to know the value of the binomial distribution function $p(x) = \binom{100}{x} (0.1)^x (0.9)^{100-x}$, where $\binom{100}{x}$ is the binomial coefficient, for $x = 1, 2, 3, 4,$ or 5. One method for performing such a calculation is simply to write a table into the memory of the computer and "look up" the value each time it is needed.

Suppose such a table was written into addresses 001 to 005; i.e., in address 001 is located $10^4p(1)$, in address 002 is located $10^4p(2)$, . . . , in 005 is located $10^4p(5)$. Note that we must *scale* the values of $p(x)$, since $p(x)$ is less than 1 (for all values of x) but our computer interprets all numbers as greater than 1. Of course the numbers in addresses 001 to 005 will be written in octal.

Suppose also that the value of x (which for this problem can only be 1, 2, 3, 4, or 5) is found in address 006 and that the result, namely, $10^4p(x)$, corresponding to this value, is to be put into 007. To perform this task, the code must have a set of instructions that will transfer the contents of one of the addresses 001, 002, 003, 004, or 005 into 007. Using the one-address system devised in Sec. 3-9, these instructions (where the contents of 010 is the constant zero) are:

Operation	α	Remarks
42	010	Clear accumulator
53	?	Put contents of ? into accumulator
52	007	Transfer (acc) into 007

If the contents of 006 were 1, then 001 should replace the ?; if (006) = 2, then 002 should replace the ?; etc. The problem therefore resolves itself into modifying the instruction with the ? so that ? is replaced by the contents of 006. One way of doing this is to write the instruction first as 53 000. Then add (006) to it, the result being 53 001, etc.

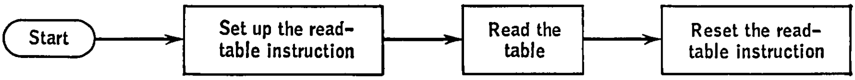

Start
Set up the read-table instruction
Read the table
Reset the read-table instruction

FIG. 4-5. Flow chart of table look-up routine, indicating *reset* instructions.

EXAMPLE 4-3. BINOMIAL-COEFFICIENT LOOK-UP ROUTINE

Address	Instruction		Remarks	
	Operation	α		
001	00	003	$10^4p(1)$	
002	00	020	$10^4p(2)$	
003	00	073	$10^4p(3)$	table
004	00	237	$10^4p(4)$	
005	00	523	$10^4p(5)$	
006	00	004	$x = 4$ (or 1, 2, 3, 5)	
007			$10^4p(x)$ = result	
010	00	000	0 constant	
011	42	010	*Start here*, and clear accumulator	
012	53	006	Put (006) into accumulator	Set up ? instruction
013	51	015	*Add (acc) to ? instruction*	
014	42	010	Clear accumulator	
015	53	?	Now do ? instruction [(015) reads 53004 by time code progresses this far]	
016	52	007	Transfer (acc) into 007	
017	42	010	Clear accumulator	
020	53	023	Put instruction constant into accumulator	Reset ? instruction
021	52	015	Transfer (acc) into 015	
022	44		*End here*—jump to another part of program	
023	53	000	Instruction constant	

But after the code has been used, an additional set of instructions is needed to reset it back to 53 000. The flow diagram is as in Fig. 4-5. The entire code, then, is as in Example 4-3.

Instruction Modification in Loops. Frequently instruction modification is necessary for the coding of loops. For example, suppose that it is desired to form the sum of a long column of numbers which are located in consecutive addresses of the computer memory from address 030 to 077. We could use the same *add* instruction for the successive additions provided that the *add* instruction were modified before each addition so as to add the contents of the *next* successive address to the partial sum each time round. The flow diagram is as in Fig. 4-6 and the code as in Example 4-4. The arrows in Fig. 4-6 indicate the path of the computer through the instructions. The sum is formed in address 001. The student should be sure to understand the purpose of each of the five sets of instructions of Example 4-4.

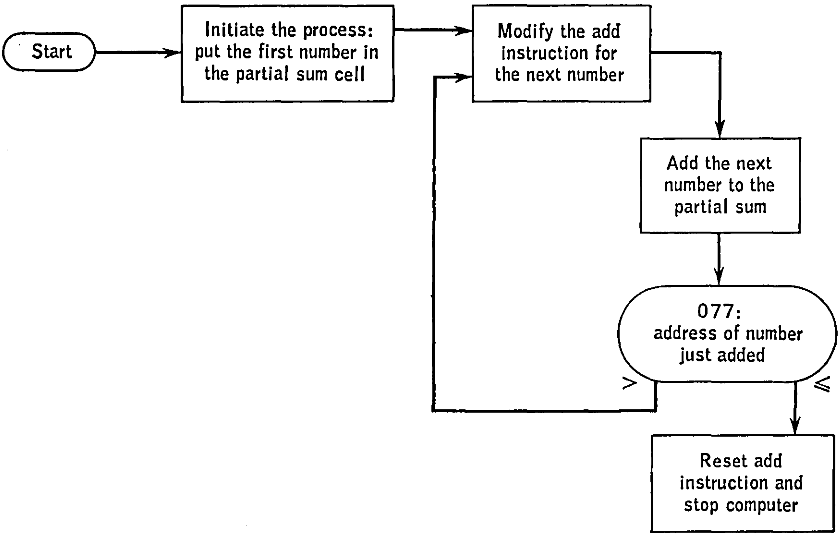

Start
Initiate the process: put the first number in the partial sum cell
Modify the add instruction for the next number
Add the next number to the partial sum
077: address of number just added
>
≤
Reset add instruction and stop computer

FIG. 4-6. Flow chart for instruction modification.

Loops within Loops. As an example of multiple loops, or loops within loops, consider the computation of a table of sin x by means of the infinite series

$$\sin x = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \frac{x^9}{9!} - \frac{x^{11}}{11!} + \cdots$$

for each $\frac{1}{100}$ radian from 0 to $\pi/2$, to eight decimal places (see Sec. 6-11). First we would have a loop that formed $x^n/n!$ by multiplying a partial product successively by x/P_i (loop A); then we would need a loop that adds or subtracts this result to or from the partial sum and increases

EXAMPLE 4-4. COLUMN-ADDING ROUTINE

Address	Instruction		Remarks	
	Operation	α		
000	44	002	Start at 002	
001			Final sum (temporary partial sum)	
002	42	024	Clear accumulator	initiate process: put first number into partial-sum cell
003	53	030	Put (030) into accumulator	
004	52	001	Put (acc) into 001	
005	42	024	Clear accumulator	instruction modification
006	53	025	Increase α of *add* instruction by 1	
007	51	012		
010	42	024	Clear accumulator	add next number to partial sum
011	53	001	Put partial sum in accumulator	
012	53	030	*Add* instruction	
013	52	001	Put partial sum into its temporary	
014	42	024	Clear accumulator	tally
015	53	012	Put (012) into accumulator	
016	54	026	Subtract (026) from accumulator	
017	43	005	53077 > (012)? Yes: add another number No: sum is complete	
020	42	024	Clear accumulator	reset and stop computer
021	53	027		
022	52	012	Put 53030 into 012	
023	60	000		
024	00	000	0 constant	
025	00	001	1 constant	
026	53	077	53077 constant	
027	53	030	53030 constant	
030 . . . 077			Numbers to be added	

n by 2 until the partial sum becomes correct to eight significant figures (loop B); and finally we need a loop that increases x by 0.01 and continues to compute the next value of sin x (loop C). The flow diagram for this process appears in Fig. 4-7.

EXERCISES

(*a*) Code the flow diagram of Fig. 4-1 for a one-address system as described above.

(*b*) The symbolic codes of Examples 4-1 and 4-2 are incomplete in that they do not reset the computer for additional runs. Complete these codes accordingly and assign specific addresses.

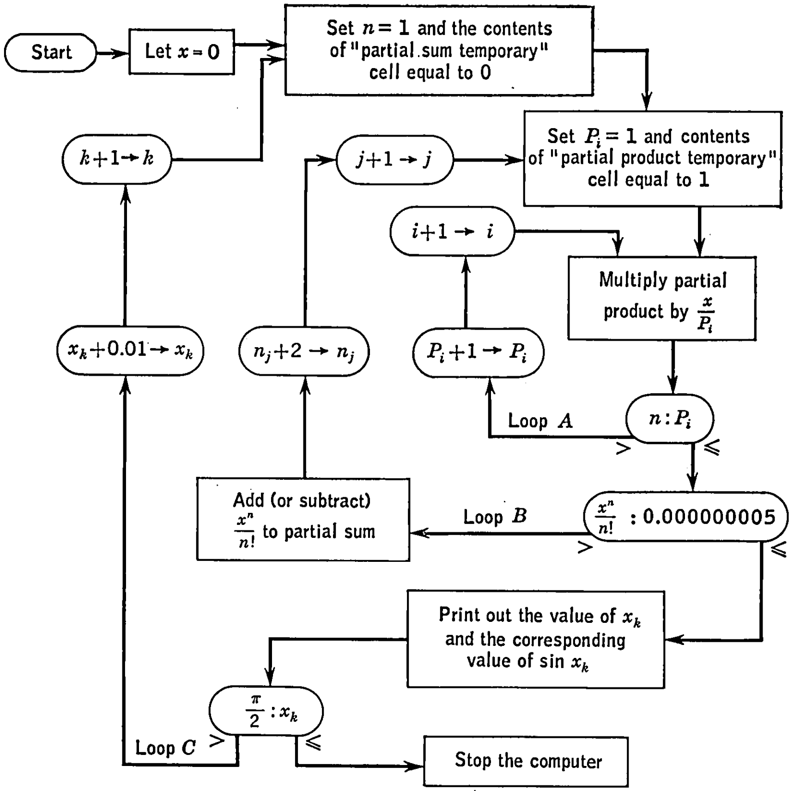

Start
Let $x = 0$
Set $n = 1$ and the contents of "partial sum temporary" cell equal to 0
Set $P_i = 1$ and contents of "partial product temporary" cell equal to 1
$k+1 \rightarrow k$
$j+1 \rightarrow j$
$i+1 \rightarrow i$
Multiply partial product by $\frac{x}{P_i}$
$x_k+0.01 \rightarrow x_k$
$n_j+2 \rightarrow n_j$
$P_i+1 \rightarrow P_i$
Loop A
$n : P_i$
$>$
$\leqslant$
Add (or subtract) $\frac{x^n}{n!}$ to partial sum
Loop B
$\frac{x^n}{n!}$: 0.000000005
$>$
$\leqslant$
Print out the value of x_k and the corresponding value of sin x_k
$\frac{\pi}{2} : x_k$
Loop C
$>$
$\leqslant$
Stop the computer

FIG. 4-7. Flow chart of loops within loops. The notation $P_i + 1 \rightarrow P_i$, $n_j + 2 \rightarrow n_j$, and $x_k + 0.01 \rightarrow x_k$, etc., is often used to indicate that in the $(i + 1)$st iteration $P_{i+1} = P_i + 1$, $n_{j+1} = n_j + 2$, and $x_{k+1} = x_k + 0.01$. etc.

If x is found in 001 and n is in 002, draw flow diagrams and write codes that will compute the following functions using the one-address system and also the three-address system as described above:

(*c*) $x^n/n!$. [HINT: Form a partial product by multiplying by $x/(n - i)$.]

(*d*) $e^x = 1 + x + x^2/2! + x^3/3! + \cdots$. (HINT: Compute enough terms so that $x^n/n! < \epsilon$; this test will tally the breaking of the loop.)

(*e*) The binomial coefficient $n!/x!(n - x)!$.

(*f*) The first moment of the binomial expansion, namely,

$$u = \sum_{x=0}^{n} x \frac{n!}{x!(n-x)!} p^x q^{n-x}$$

(*g*) Write a routine that forms the sum of the squares of numbers found in addresses 030 to 077.

(*h*) Write a routine that puts the numbers found in 030 to 077 in numerical order. HINT: Successively interchange locations of numbers in adjacent addresses if they are not in order; keep track of the number of interchanges each time through; when no interchanges are necessary, then the numbers are in order; that is, for example,

1st time through:	$\overset{\times}{4, 2}, 3, 1$	$2, \overset{\times}{4, 3}, 1$	$2, 3, \overset{\times}{4, 1}$	2, 3, 1, 4
2d time through:	$\overset{\frown}{2, 3}, 1, 4$	$2, \overset{\times}{3, 1}, 4$	$2, 1, \overset{\frown}{3, 4}$	2, 1, 3, 4
3d time through:	$\overset{\times}{2, 1}, 3, 4$	$1, \overset{\frown}{2, 3}, 4$	$1, 2, \overset{\frown}{3, 4}$	1, 2, 3, 4
4th time through:	$\overset{\frown}{1, 2}, 3, 4$	$1, \overset{\frown}{2, 3}, 4$	$1, 2, \overset{\frown}{3, 4}$	1, 2, 3, 4

(no interchanges; so numbers are in proper order).

4-3. The Subroutine

Use of a Subroutine. A *subroutine* is a subcode that may be used many times during the computation of a program but is written only once in the whole code. As the computer proceeds down the main program, the control will occasionally jump to this subroutine and then, after doing the subroutine, will jump back to the main program where it left off. This detour from the main program through the subroutine may occur several times during the computation of the program. Hence a subroutine must have an *entrance*, a way of getting into it, and an *exit*, a way of getting out of it. Each time an entrance is made to a subroutine, some initial conditions must be set up that are characteristic of the place in the main program from which the entrance was made. For instance, if the subroutine calculates some function, the initial values of the independent variables at that point in the main program must be given to the subroutine. In addition, as an entrance to a subroutine is made, the exit must be set up; i.e., the subroutine must be told *where* to transfer control back to the main program. For each time the subroutine is used, its exit is usually to some different part of the main program. Hence, in order to use a subroutine, the coder must know (1) the entrance, i.e., the address of the first instruction; (2) the addresses of the temporaries in which the initial conditions are to be set up; (3) the addresses of the temporaries whose contents will be the results of the subroutine computation; and (4) the exit, i.e., the address of some jump instruction that is to be preset (when entering the subroutine) so that, when the computation of the subroutine has been completed, the computer will transfer control back to the proper address of the main program.

Consider, for instance, the flow diagram in Fig. 4-8, which illustrates the setting up of the initial conditions and different exits of the subroutine. The A in the circle is a connector and of course means that the program jumps to the subroutine at that point. The B connector is called a *variable connector;* i.e., it indicates that the program jumps either to B_1 or to B_2, depending on how the "variable" connector is set. When

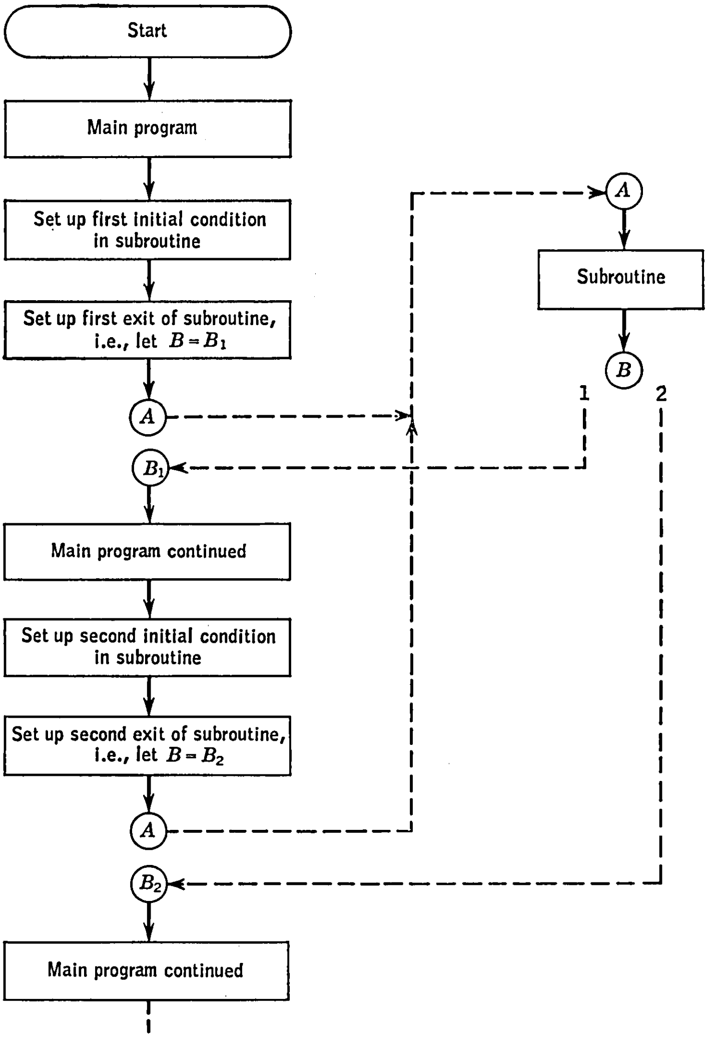

Start
Main program
Set up first initial condition in subroutine
Set up first exit of subroutine, i.e., let $B = B_1$
A
B_1
Main program continued
Set up second initial condition in subroutine
Set up second exit of subroutine, i.e., let $B = B_2$
A
B_2
Main program continued
A
Subroutine
B
1
2

FIG. 4-8. Flow chart for setting up initial conditions and different exits of a subroutine.

we say that $B = B_1$, we mean that the B connector is set so that it will go from B to B_1; and when we say that $B = B_2$, then the B is set to go from B to B_2. The rest of the flow diagram is self-explanatory but should be studied carefully by the student.

Example. As a specific example of the use of subroutines, consider a program which is to compute the radiation intensity S of an antenna at a point in space as a function of the directional angles θ and φ and the radius R given by the equation†

$$S = \frac{30I_0^2}{2\pi R^2}\left[\frac{\cos\left(\frac{\pi}{2}\cos\theta\right)}{\sin\theta}\,\frac{\sin\left(\frac{n\pi}{2}\sin\theta\cos\varphi\right)}{\sin\left(\frac{\pi}{2}\sin\theta\cos\varphi\right)}\right]^2$$

where n = a given integer
I_0 = a known constant

Now the sine and cosine functions are evaluated by means of their power series. Since these functions appear several times as functions of different arguments, it would be wise to use their respective power-series codes as subroutines. Figure 4-9 is a flow chart for the calculation of S and indicates the use of the subroutines.

This flow diagram can be used to illustrate some of the details involved in using subroutines. For example, suppose that the first instruction of the cos x subroutine is located in address 700. Then, using a one-address system, the jump instruction to this subroutine would be 44 700, which completes the first step in using a subroutine. The second step is to set up the exit from the subroutine. One method is to have in address 677 (the address preceding 700) a jump instruction 44 000, where α is to be designated. Suppose that the main program will continue after the subroutine, from address 017. Then we would want (677) to be 44 017. Then at the end of the subroutine would appear a jump to 677, whence the computer will jump to 017. To set up (677), the main program would put the constant 44 017 into the accumulator and then perform a transfer into 677. Another method of accomplishing this return jump to the main program is through some sort of special instruction designed for this purpose. In this case the jump to the first instruction, for example, 44 700, automatically memorizes the address of this instruction in some counter or index register. Then a special jump instruction is placed into address 677 that looks at this counter or register and jumps back to the main program.

For the third step in using a subroutine the main routine must somehow tell the subroutine the values of the initial data, i.e., the independent variables. For this purpose a common agreement is made between the person who writes the subroutine and the user of the subroutine about certain addresses set aside for this purpose. For example, suppose that address 676 were agreed upon to be the address of the initial value of x in the cos x subroutine. Then, before jumping to the subroutine, the main program would transfer the initial value of x into 676 for the use of the subroutine. Similarly an agreement might be made that the final computed value of cos x be placed in 675; when the computer exits from the subroutine back to the main program, the main program first transfers (675) to some temporary that the main program uses. Of course, there

† See, for example, J. A. Stratton, "Electromagnetic Theory," p. 451, eq. (53), McGraw-Hill Book Company, Inc., New York, 1941.

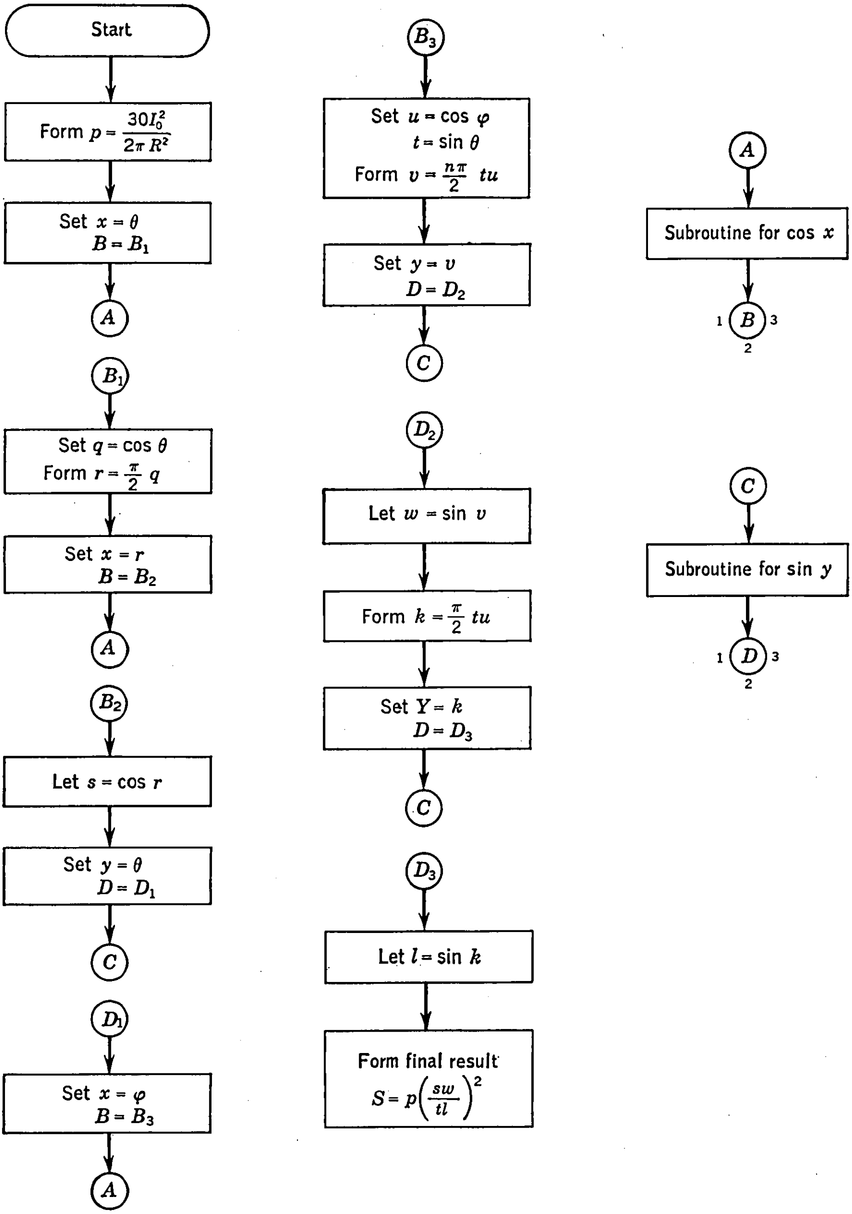


FIG. 4-9. Flow chart for computation of radiation intensity.

is an alternative method for performing these two tasks. The main program could first tell the subroutine where the initial value of x is located and where the final computed value of $\cos x$ should be placed, and the subroutine could include the transfer instructions. In such a case the instructions of the main program that "tell" the subroutine would do so by modifying some instructions of the subroutine.

Library of Subroutines. As illustrated by the example, several subroutines may be used in one main program. In fact it is found that many common subroutines are used quite often, such as $\sin x$, $\cos x$, e^x, $\sqrt{x}$, etc. Hence it often becomes profitable to have a *library of subroutines* available to the programmer, stored at all times in some part of the computer memory. There would also be some catalogue kept outside the computer that the programmer can consult when he wants to use a subroutine. This catalogue would tell where each subroutine is located and all pertinent data about how to use it, such as where to put the initial values of the independent variables, where the computed values of the dependent variables are found, etc. In addition, as we shall discuss in a later section, automatic programming aids are devised to make such a library of subroutines easier to use.

EXERCISES

Construct a library of subroutines, using our three-address system given above, as follows:

(*a*) Write a subroutine for $\sqrt{x}$ for your library.

(*b*) Write a subroutine for $x^m/m!$ for your library.

(*c*) Write a subroutine for $\sin x$ for your library. (HINT: This subroutine will itself use subroutine *b*.)

(*d*) Write a subroutine for $\cos x$ for your library.

(*e*) Write a manual describing how to use your library of subroutines.

(*f*) Code the example given in this section, using your library of subroutines.

4-4. Instruction Types

In this section we present many of the important kinds of instructions that are found in modern digital computers. The object is to describe the *concepts* upon which the various kinds of instruction are based rather than specific details of the instructions themselves. The concepts are chosen from those which are most commonly used. Instructions based on these concepts may take many different forms in the many different computers. For purposes of clarity in this presentation these instruction concepts have been classified into six major categories: (1) the use of auxiliary bits in an instruction; (2) arithmetic instructions; (3) logical and bit-handling instructions; (4) decision instructions; (5) recursion-aiding instructions; (6) *read* and *write* instructions. Although there is some overlapping, it is most convenient to consider each instruction as falling into one, and only one, of these categories.

1. Auxiliary Bits. Auxiliary bits are bit positions that are included in the format of the instruction words of a particular computer, but they are *not* part of the operation code or the addresses. Two common uses of such bits are (*a*) to halt the computer or to make the program jump to some standard memory location, called *break points*, and (*b*) to indicate *relative* addresses—both of which will now be described.

a. BREAK POINTS. As was pointed out above, the "sign" bit is an auxiliary bit. The sign bit can be given meaning for instructions as follows: if it is 0, the computer continues to compute as usual; if it is 1, then the computer halts. Important use is made of such a system in detecting errors in a program. Suppose that a coder were running a program on a computer for the first time; he might want to stop the computer at intermediate stages in the program to see whether or not the code were correct up to those points. In order to do this, he would put 1 in the sign bits of those instructions at which he wants to stop the computer so that he could examine the partial results. An alternative to this procedure, which serves the same purpose, is to have the computer take the next instruction from a standard, predetermined location in the memory whenever a 1 appears in the sign bit of an instruction. At the standard position a special code might be located that can do various things to aid the coder detect errors and correct this code.

b. RELATIVE ADDRESSES. Another important use for auxiliary bits is concerned with *relative addresses.* Relative addresses are associated with so-called *relative counters*, sometimes called *index registers*, and with auxiliary bits. A relative counter is a very special memory cell or register that can contain only a single address; i.e., it is a "short" memory cell. The computer must also have special instructions that can put a number into such a relative counter, add to the number in the relative counter, etc. Such instructions will be considered later in this section.

An address appearing in an instruction is said to be interpreted "relative" to a counter if the computer, instead of considering the contents of the address indicated, considers instead the contents of the word whose address is the sum of the counter reading and the indicated address. For example, if the index register read 002 and the address in the instruction were 053, then the computer would consider the contents of

$$002 + 053 = 055$$

The role of the auxiliary bits in connection with relative addresses is as follows: The auxiliary bits of an instruction can indicate whether an address is to be considered relative or not. For example, consider a three-address instruction format that has three auxiliary, i.e., extra, bits on the right:

Operation	α	β	γ	Auxiliary bits $P3$ $P2$ $P1$

Suppose that there is one *relative counter*. Then the following meaning can be attached to these bits: If $P1$ is 1, then γ is to be interpreted as relative to the counter; if $P2$ is 1, then β is relative; if $P3$ is 1, then α is relative. Hence, if the auxiliary bits were 101 and the instruction were "add," then the contents of "α + (counter)" would be added to the contents of β and the result put into the address "γ + (counter)."

The use of such relative addresses is to aid the coding of iterations, or loops. As illustrated by some of the problems of the previous sections, each time around a loop it is often necessary to increase some addresses of the main set of instructions by 1 (or some other integer). In such cases a set of instructions modifying the instruction must precede the main set of instructions. But with relative addresses all that need be done is to increase the counter by 1 (or whatever is the necessary integer for a particular case) by means of the counter-setting instruction. One variation of this system is to have several such relative counters or registers, which can be used in many different ways. For example, an instruction preceding the use of relative addresses can determine to which counter the addresses are to be considered relative. Or the addresses may be considered relative to the sum or some other function of the numbers in the counters.

2. Arithmetic Instructions. These instructions can be divided into four classes: add, subtract, multiply, and divide. One of the important concepts that appears in arithmetic instructions is the *automatic floating decimal point.* As was seen in some of the above examples and problems, in ordinary arithmetic instructions the coder was limited to handling numbers within the range of the word length. For example, if the word length is 15 bits plus a sign bit, and if all numbers are interpreted as being nonfractional, that is, if the binary point is considered to be on the right, then only *integers* that lie between 0 and 2^{15}, or -1 and -2^{15}, can be handled by the computer. Obviously there must be some way out of such a totally unacceptable situation. The method is to multiply all numbers involved in a code by an appropriate *scaling factor* so that they fall within the desired range during the computer computations. Of course the results put out by the computer must be readjusted to their actual values. However, it suffices to mention here that scaling presents more work for the coder and is a disadvantage. Many computers avoid this disadvantage by having a so-called *floating point* for arithmetic operations.

In computers with floating-point arithmetic operations a number is memorized in a word composed of two parts: a mantissa, which contains the significant figures of the number, and an *exponent.* For example, one computer that has a 45-bit word uses 36 bits for the significant figures and their sign and 9 bits for the exponent and its sign. Hence this computer can handle numbers whose *absolute values* are as large as $(2^{35} - 1) \times 2^{255}$ (since $2^8 = 256$) or as small as 2^{-255}, that is, numbers within this range with no more than 35 significant bits. A range such as this covers most cases that usually occur and hence relieves the coder of all scaling considerations.

The terms *normalized* and *unnormalized* are often used in connection with floating-point arithmetic operations. These terms are concerned with the problem of determining the appropriate exponent of the result of an operation. The result of an operation is called *normalized* if the exponent is so adjusted that the leftmost bit of the mantissa is a unit bit. This of course maximizes the number of significant figures that can be recorded in a word. A computer operation that presents normalized results is called *normalized;* e.g., we would say *add-normalized.* As an example, suppose that a computer had a 24-bit (8 octal position) mantissa and we multiply-normalized the (octal) numbers (with octal exponent): .00033333 × 2^6 by .04444444 × 2^3. The unnormalized result (double length) is .0000 1753 0442 4714 × 2^{11}. Of course this is a binary number in the computer, and the problem is to determine that exponent which will move the binary number to the left until a unit bit appears in the leftmost position. The number of bit positions the mantissa will have to be moved to the left in our example is $3 \times 4 + 2$ [where 3×4 is the number of bit positions corresponding to four zeros; and since 1 (octal) is 001 (binary), a move two further positions to the left is necessary]. Figure 4-10 illustrates the binary and octal forms of the double-length mantissa before and after the move.

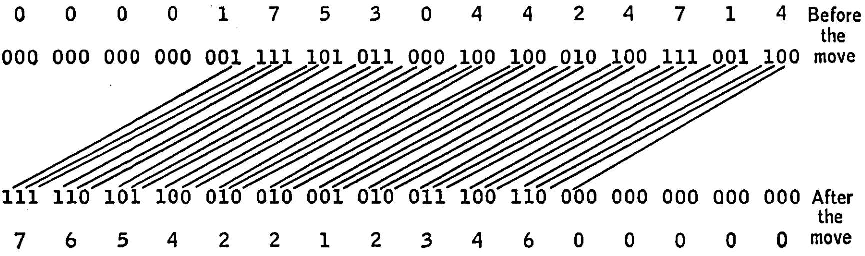


FIG. 4-10. Normalized and unnormalized mantissa.

The final normalized result will therefore be .7654 2212 × 2^{-5} [where $11 - (3 \times 4 + 2) = -5$]. Of course the computer will do this automatically and will display only the "single-length" result. (See below for further discussion of single- and double-length results of multiplication.) Note that, if the computer binary point is understood to be to the right, the above multiplication would be

$$00033333 \times 2^{-22} \times 04444444 \times 2^{-25} = 00001753 \quad 04424714 \times 2^{-47}$$

The product could here be normalized by shifting right

$$(3 \times 3) + 1 = 12 \text{ octal places}$$

to give 76542212 × 2^{-35} $(-47 + 12 = -35)$. If the product were normalized by shifting left, dropping the last eight places would be equivalent to multiplying by 2^{-30} and the exponent would have to be corrected accordingly: $-47 - [(3 \times 4) + 2] + 30 = -35$.

When an unnormalized operation is defined, the method for determining the exponent must be clearly stated.

We can now return to a description of the kinds of arithmetic instructions. Consider "add" first.

a. ADD. 1. *Ordinary addition* instruction, as described in previous sections.

2. *Floating addition—unnormalized.* The exponent of the result will be that of the largest exponent of the two operands.

3. *Floating addition—normalized.* The exponent of the result will be adjusted so that the most significant bit of the number is a unit.

4. *Absolute-value "add."* In a two-, three-, or four-address system the result would be $= |(\alpha)| + |(\beta)|$.

5. *Add into accumulator.* In a one-address system (acc) + (α) is put into the accumulator.

6. *Add into memory.* In a one-address system (acc) + (α) is put into α.

7. *Add relative.* The instruction code itself tells that the address is to be considered relative to some relative counter.

8. *Add with overflow check.* If the sum is greater than the largest number that can be stored in a word, then *overflow* is said to have occurred. If overflow occurs, then the computer is to break the sequence of instructions; e.g., in a three-address system, if no overflow, then take next instruction; if overflow, take γ.

9. *Multiply-add.* In a two-address system, form $(\alpha) \cdot$ (acc) + (β), and put result into accumulator.

10. *Partial add-α address.* In a two or more address system, only the α portion of the words is added. Similarly there are add-β, -γ address instructions.

b. SUBTRACT. The kinds of *subtract* instructions are analogous to the kinds of *add* instructions but include also:

11. *Change sign of register.* This is *not* a *subtract* instruction but only changes the sign of the accumulator.

c. MULTIPLY. Before describing the kinds of *multiply* instructions we must first describe briefly how multiplication takes place in an arithmetic unit. First note that, if an n digit is multiplied by an m digit number, the result can have as many as $n + m$ digits. In a computer, therefore, the result of a multiplication can contain twice as many bits as appears in a word. Hence there are usually two registers in the arithmetic unit (or one double-word-length register) to record the result of the multiplication. If a word has m bits, then the least significant m bits (which are found in one of the registers, called the *minor register*) are called the *minor multiplication result,* while the rest of the bits are called the *major multiplication result* (and are found in the other register, called the *major register*).

For example, if we consider a 24-bit (8 octal position) word, and multiply 00033333 by 04444444, we obtain as a result 00001753 04424714. Here 00001753 is the major product, and 04424714 is the minor product.

1. *Regular multiplication.* Leaving both major and minor results in the registers.

2. *Major multiplication—unrounded.* The memorized result is the major multiplication result.

3. *Major multiplication—rounded.* The memorized result is the major multiplication result with a least significant unit added to the major result *if* the most significant bit of the minor multiplication result is a unit. If the multiplication is normalized also, the normalization is done first, then the rounding.

4. *Floating multiplication—unnormalized.* The exponents are added, and the mantissas are multiplied.

5. *Floating multiplication—normalized.* After the multiplication the product is shifted and the sum of the exponents adjusted so that the most significant bit is a unit.

6. *Rounding only.* This is *not* a multiplication instruction, but the number in the major register is rounded with respect to the number in the minor register.

d. DIVIDE. In dividing, the quotient appears in the major register, the remainder in the minor register. This brings up some variations in the *divide* instruction.

1. *Quotient divide.* The quotient is the memorized result.
2. *Remainder divide.* The remainder is the memorized result.
3. *Quotient divide—rounded.*
4. *Floating divide—unnormalized.*
5. *Floating divide—normalized.*
6. *Divide with overflow check on quotient.*

EXERCISES

(*a*) Consider the example illustrated by Fig. 4-5. Code this in a three-address system in which the instruction format includes three auxiliary bits as described in Sec. 4-4, under Relative Addresses. Assume a counter-setting instruction that sets the counter to the number α and takes the next instruction from γ (here β is not used).

(*b*) What would a counter-setting instruction be in a one-address system?

(*c*) Code the same problem as in (*a*), using a one-address system where the instruction format has a single auxiliary bit.

(*d*) Find the unnormalized and normalized sum of 77770000 $\times 2^5$ and 00007777 $\times 2^6$ (where, of course, the numbers are in octal).

(*e*) Find the unnormalized difference of 0000 7777 $\times 2^6$ and 7777 0000 $\times 2^5$.

(*f*) For the numbers 7777 0000 $\times 2^5$ and 0000 7777 $\times 2^6$ find:

The unnormalized major product, unrounded.
The unnormalized major product, rounded.
The normalized major product, unrounded.
The normalized major product, rounded.
The unnormalized minor product.
The normalized minor product.

4-5. Instruction Types (*Continued*)

3. Logical and Bit-handling Instructions. These instructions are used for two main purposes. First they are used to modify instructions; second they are used when it is desired to find whether the bit in a *Pi* position is zero or a unit, as is often necessary when bits represent switch inputs in a machine.

a. LOGICAL ARITHMETIC. 1. *Logical addition.* There is *no carry*, and the bits in the corresponding positions of the two operands are "added" according to the rule $1 \underline{|+} 1 = 1$, $1 \underline{|+} 0 = 1$, $0 \underline{|+} 1 = 1$, $0 \underline{|+} 0 = 0$. For example,

$$\text{Logical} + \quad \begin{array}{ll} 0101 & A \\ \underline{0011} & B \\ 0111 & A \underline{|+} B \end{array}$$

which can, of course, be generalized to a word of any number of bits in length.

2. *Logical multiplication.* Similar to logical addition except that $1 \underline{|\cdot} 1 = 1$, $1 \underline{|\cdot} 0 = 0$, $0 \underline{|\cdot} 1 = 0$, $0 \underline{|\cdot} 0 = 0$. For example,

$$\text{Logical} \times \quad \begin{array}{ll} 0101 & A \\ \underline{0011} & B \\ 0001 & A \underline{|\cdot} B \end{array}$$

which can be generalized as above.

3. *Logical ring add* (*or logical inequality*). Similar to logical addition, except that $1 \underline{|\neq} 1 = 0$, $1 \underline{|\neq} 0 = 1$, $0 \underline{|\neq} 1 = 1$, $0 \underline{|\neq} 0 = 0$. For example,

$$\text{Logical ring} + \quad \begin{array}{ll} 0101 & A \\ \underline{0011} & B \\ 0110 & A \underline{|\neq} B \end{array}$$

which can be generalized as above.

4. *Logical compare* (*or logical equality*). Similar to logical addition, except that $1 \underline{|=} 1 = 1$, $0 \underline{|=} 0 = 1$, $1 \underline{|=} 0 = 0$, $0 \underline{|=} 1 = 0$. For example,

$$\text{Logical compare} \quad \begin{array}{ll} 0101 & A \\ \underline{0011} & B \\ 1001 & A \underline{|=} B \end{array}$$

which can be generalized as above.

5. *Complement.* Units are changed to zeros, and zeros to units; e.g., if $(\alpha) = 0101$, the complement of (α) is 1010.

6. *Extract.* For a four- or three-address instruction, where a bit ot (α) is a unit, replace the corresponding bit of (γ) with the corresponding

bits of (β). For example,

(α)	000	111	111	000	011
(β)	010	001	101	110	110
(γ)	011	101	110	010	101
Result, found in γ	011	001	101	010	110

7. *Odd-even extractor* (used for decimal computers). Replace those digits of (γ) which correspond (in position) to odd digits of (α) with the corresponding digits of (β). For example,

(α)	222	555	793	246	277
(β)	134	682	304	765	310
(γ)	867	524	116	243	876
Result	867	682	304	243	810

b. SHIFT INSTRUCTIONS. 1. *Right shift.* The whole (α) is shifted to the right by the number of positions specified in the instruction. If the shift is n positions, the effect is the same as multiplying by 2^{-n}. Those bits which run off the right end of the word are lost; those which come into the left end are zeros.

2. *Left shift.* Like the right shift, but in the other direction, i.e., like multiplying by 2^n.

3. *Circular shifts.* Shifts where the bits that go off one end come back at the other as if the ends of the words were connected together. (These are also called "end-around shifts.")

4. *Left significant shift.* Shift till there is a unit in the most significant place. This differs from the above shift instructions in that the number of positions to be shifted is not given by the instruction but is determined during the operation. The number of positions shifted is often recorded in β; the shift is performed in the accumulator for a one-address system.

4. Decision Instructions. When a flow chart of a program comes to a branch point, some form of decision instruction must be used to determine which branch should be taken. As far as the segmented execution of the instructions in a program is concerned, a branch point means that the program either will continue on normally or will *jump* to another part of the program. The term *jump* is used to indicate that at such a branch point the next instruction is taken from some part of the memory other than the usual sequence.

a. COMPARISON OF WORDS. 1. *Usual comparisons.* If (α) > (β), then take γ for four- or three-address computer; if (α) > (acc), take β for two-address computers.

2. *Absolute comparison.* Same as 1, except that the absolute values of the words are compared.

3. *Negative jump.* If (acc) is negative, take α.

4. *Positive jump.* If (acc) is positive, take α.

5. *Zero jump.* If (acc) = 0, take α.

6. *Nonzero jump.* If (acc) ≠ 0, take α.

7. *High jump.* If (acc) > (α), skip next instruction.

8. *Low jump.* If (acc) < (α), skip next instruction.

9. *Equal jump.* If (acc) = (α), skip next instruction.

10. *Least significant jump.* If least significant bit of (acc) is 1, then jump to α.

Of course items 3 to 10 are for one-address systems; in items 7 to 9 the number of instructions skipped may be more than one but would be fixed for a given machine.

b. SENSING OF SIGNAL SWITCHES. 1. *Special signal transfer.* If a certain signal specified by the operation is on, then take α. This instruction is used in conjunction with an instruction that turns on the signal. This latter instruction may be separated in the program from the special-signal-transfer instruction so that the jump occurs some time after the signal was turned on. There may be several signals and associated sets of instructions.

2. *Manual jump.* Here a jump will occur only when a manual external switch has been set before the program reaches the instruction.

3. *Manual halt and transfer.* The computer will stop if the external switch specified by the instruction has been turned on. Then, when the start button is pushed, the jump to α will occur. If the switch is off, the computer jumps without stopping.

4. *Return jump.* In a two-address system the program jumps to (β) for the next instruction; at the same time the contents of the current instruction register (i.e., the address of this particular return-jump instruction) is recorded as (α). The purpose of this instruction is that it is often desirable to jump *back* to the place from which the original program jumped.

5. Recursion, or Loop-aiding, Instructions. These instructions are designed to aid in the programming of loops. In the examples and problems given above each time around the loop the main set of instructions usually needs to be modified by means of additional instructions. As was seen above, relative counters or index registers can aid in eliminating these additional instructions. Also, loop codes require some sort of tally to be kept to allow the computer to jump out of the loop. The purpose of recursion, or loop-aiding, instructions is to facilitate the use of the relative counters and to aid in keeping the tally automatically.

a. INDEX-REGISTER MODIFIER. Adds 1 to the index register and jumps to α. Here the loop tally is not considered.

b. FILE AND MODIFY RELATIVE COUNTER. This instruction stores the contents of one of the relative counters (assuming that there are several relative counters) as (α); then another designated by the instruction is set to the *number* β (that is, *not* the contents of β, but β itself), and this counter is then considered the relative counter until another *file* instruction occurs. Since resetting and going to another relative counter essentially constitutes a jump, this instruction can act as a return jump. However, if the number β of the *file* instruction itself is considered relative to the same counter that the file instruction is setting, then this counter is set to its original reading + β; that is, the counter is increased by β. As we have seen in the index-type instructions, resetting a counter is an aid to recursion codes; however, *no* tally is considered.

c. TALLY JUMP. Subtracts 1 from the *number* α of the instruction; if the result (that is, $\alpha - 1$) is positive, replaces the number α by the number $\alpha - 1$ (in the α-address position) of the instruction, and jumps to take as the next instruction the contents of β; otherwise the program continues sequentially as usual. This instruction does not consider the relative counters.

d. MODIFY INDEX REGISTER AND TALLY. Compares the contents of the index register with the number α. If the contents of the index register is greater than the number α, then the number α is subtracted from the index register and the difference placed in the index register. Then a jump is made to β. Otherwise the program continues in sequence. Note that the index register acts both as tally *and* as a relative counter. Of course this must be used in conjunction with an instruction that initially sets the contents of the index register.

6. *Read and Write Instructions.* These instructions operate the input and output electromechanical units.

a. MAGNETIC TAPES. These instructions can enable the computer to move the magnetic tapes forward or backward, or completely to rewind them. They enable the computer to read from the magnetic tapes into the higher-speed memory. The number of words written onto or read from the magnetic tape is specified either in terms of individual words or, more usually, in terms of the number of *blocks of words,* where by a block of words we mean some predetermined number of words. If, for example, there are eight words in a block, then only multiples of eight words can be handled. Similarly, in just moving the magnetic tapes, the distance to be moved is specified in either single words or blocks of words. The instruction code tells whether to read or write, or move the tapes backward or forward; the number α tells how many words or blocks are to be considered. For reading and writing the number β is the address that the first word is written out of or read into.

b. PUNCHED TAPES AND CARDS. Instructions that tell the machine to read in from paper tape or to punch paper tape either give the initial and final addresses to be read into or out of or else give the first address and the number of words to be considered. Sometimes in reading-in paper tape, only the initial address is given, and a signal is put on the paper tape to tell the computer when to stop reading. Punched-card input-output is similar.

c. DISPLAY. For displays on an oscilloscope each instruction may tell the display to exhibit one dot or other geometrical pattern, and the instruction gives the coordinates of that dot or of some point on the pattern.

d. CONCURRENT IN-OUT. Some computers have concurrent input and output. In these cases the computer can compute at the same time as words are being read into or out of the computer.

In addition it should be mentioned that some computers have such complicated methods of reading in and out that special codes must be written to facilitate these processes, and these codes must be available at all times inside the computer's memory. Often these codes must be put into the computer at first by hand methods, in a sort of bootstrap operation. Once the routines are in the computer, the operation may never need to be repeated. However, these methods are so peculiar to the particular computer involved that it is not worthwhile considering them here.

EXERCISES

(*a*) What combination of other logical instructions would produce the same result as an *extract* instruction, in a binary computer?

(*b*) How can *return-jump* instructions facilitate the use of subroutines?

(*c*) Use the *modify-index-register-and-tally* instruction to code the problem described in Fig. 4-5. Write the code in a two-address system.

4-6. Special Coding Techniques

Multiway Branch. Our instruction systems described above allow for only a two-way branch in a code. That is, by means of a comparison instruction, the computer can determine along which of two possible directions to continue in the code. However, it often occurs that a many-way branch is needed in a code; this must be programmed. Just as the decision process for a two-way branch was reduced to a comparison of numbers, so a many-way branch may be reduced to comparisons of numbers. For example, if a code is to have a four-way branch, then, to determine which branch is taken in any case, the code might determine a number N which can take only the values 1, 2, 3, and 4. Then one way of coding the four-way branch would be a *split tree* of successive comparisons as in Fig. 4-11.

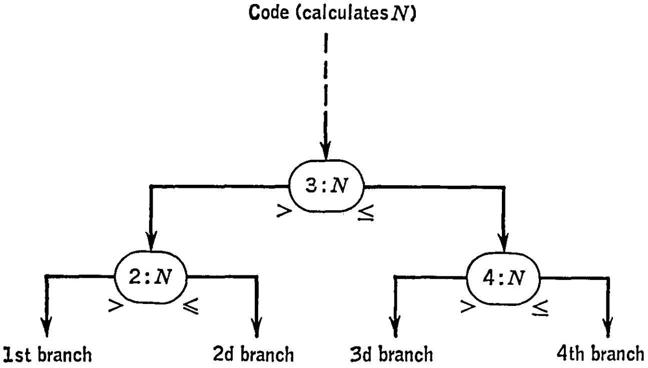
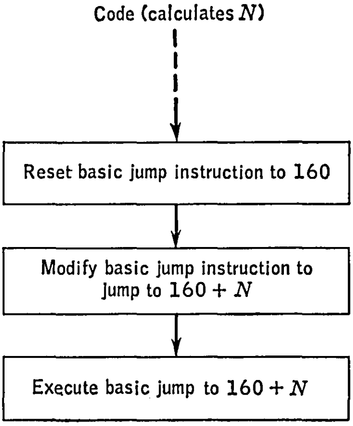


FIG. 4-11. Split-tree comparisons for multiway branch.

However, there is another, more efficient method for coding a many-way branch, using a so-called *jump table.* Here the number N that decides which branch the computer should take is used as an instruction modifier (see Fig. 4-12). The basic jump instruction is modified so that, when executed, the computer will jump to the appropriate-branch jump instruction of the jump table. This branch jump then jumps to the proper branch of the code.


Code (calculates N)
Reset basic jump instruction to 160
Modify basic jump instruction to jump to 160 + N
Execute basic jump to 160 + N

The jump table

Address	Branch jump
161	Jump to 1st branch
162	Jump to 2d branch
163	Jump to 3d branch
164	Jump to 4th branch

FIG. 4-12. Jump table for multiway branch.

Switch Sensing. In real-time applications (see Chap. 1) of digital computers it is often convenient to connect the outside world to the computer by means of a battery of switches. These switches are attached to a very specially built word in the computer's memory, and each switch controls a single bit of this word. If a switch is on, then the computer reads a unit in the position of the word corresponding to this switch; if the switch is off, then the computer reads a zero. Thus this special word gives at any time the status of the corresponding battery of switches. Let us in addition assume that this special word behaves the same as any other word of the computer's memory, except that the computer *cannot* read *into* it.

There are two ways of interpreting these switches, called *on-off* switch sensing and *one-shot* switch sensing. For on-off switch sensing the computer is to determine for each switch whether it is on or off. The nth switch can be sensed in this way as follows: Shift the switch word left until the nth bit is in the most significant place; then compare this shifted word with a word of all units except in the most significant place. For example, consider a one-address system with the following *shift* instruction: Shift left (acc) the number of positions α. Thus suppose that the switch word was 011 110 101 110 001, and that we wanted to determine whether $P8$ (the eighth bit from the right) were 0 or 1. We would place this word into the accumulator and perform: Shift left 7; i.e., shift the word left seven positions. To compare, a negative jump would be used after subtracting (acc) from 011 111 111 111 111. If the result is negative, then the shifted word had a unit in the most significant position, that is, $P8 = 1$; if the result is zero or positive, $P8 = 0$.

For "one-shot" switch sensing the computer senses the switch word periodically at a rapid rate. The computer is only to determine which switches have been *turned on.* The one-shot switch is not to be sensed when it is either left on or turned off. Thus one-shot switches are sensed according to Table 4-2. To code this we can use the *extract* instruction.

TABLE 4-2. WHEN ONE-SHOT SWITCHES ARE SENSED

New information about switch	Old information about switch	Sensed
0	0	No
1	0	Yes
0	1	No
1	1	No

The old information is extracted with the new onto all units; then in the resulting word a zero will occur in the position corresponding to the switch that was just turned on. Hence, each time the switching word is sensed, it will be recorded in some other memory location. Then both the previous and the present switching word will be available at all times. To see why the extract instruction works, consider the following:

$$\begin{array}{rcccc} \text{New word } (\alpha) & 0 & 1 & 0 & 1 \\ \text{Old word } (\beta) & 0 & \textcircled{0} & 1 & \textcircled{1} \\ (\gamma) & 1 & 1 & 1 & 1 \\ \hline & 1 & 0 & 1 & 1 \end{array}$$

Hence only when 0 in the old word corresponds to a 1 in the new word will a 0 result. Of course next the position of this zero must be determined. This can be accomplished by shifting left one position at a time, keeping track of the number of times shifted, until a zero is sensed in the most significant position.

Input-Output. Very often the input-output instructions of a computer are not adequate to read in a whole code or read out all of a given portion of the memory. In such a case a subroutine must be used for reading in and out of the computer. For example, consider the following read-in instruction for a one-address computer: Read one word into α. If an entire code is to be read into successive addresses of the computer's memory, then clearly a subroutine that iterates this instruction is needed. For example, using such a subroutine, the following procedure might be followed in reading in a code, where it is assumed that the subroutine is already in the computer's memory (see Fig. 4-13): The first instruction of the code to be read in will be a jump to the read-in subroutine. Then, when this instruction is read into, say, address 000 and executed, the computer will jump to the read-in subroutine. The read-in subroutine will read in the next two words of the code; these words will contain the initial and final address where the code is to be put in the computer's memory. Then the subroutine loads the read-in instruction with this initial read-in address and the tally with the final read-in address, and the iteration of the read-in instruction begins. When the complete code has been read in, the subroutine will jump to the initial read-in address and computation will proceed. It is probably wise to have the computer stop before computation proceeds, to give the computer operator a chance to turn off the input equipment, etc. Then, when the run button is pushed (see the next section), the computations will proceed.

Table Look-up. Consider the problem of coding a computer to look up a function table that has previously been stored in the computer's memory. That is, given a particular value of the independent variable, we desire a code that will select from the computer's memory the corresponding value of the dependent variable as recorded in the table.

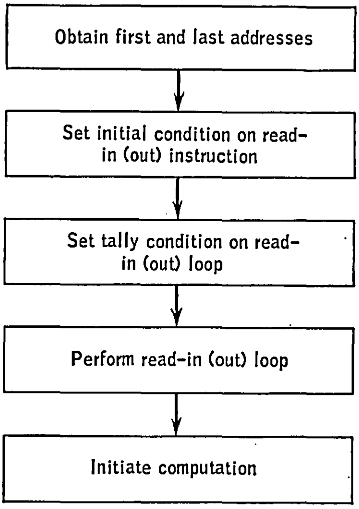
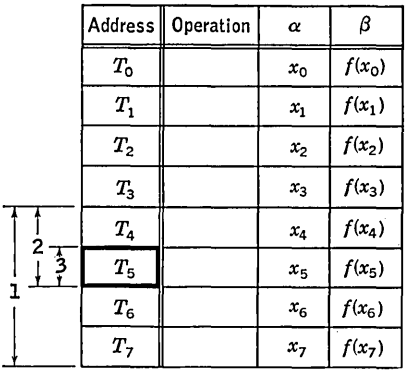

Obtain first and last addresses
Set initial condition on read-in (out) instruction
Set tally condition on read-in (out) loop
Perform read-in (out) loop
Initiate computation

FIG. 4-13. Flow chart of input-output routine.



Address	Operation	α	β
T_0		x_0	$f(x_0)$
T_1		x_1	$f(x_1)$
T_2		x_2	$f(x_2)$
T_3		x_3	$f(x_3)$
T_4		x_4	$f(x_4)$
T_5		x_5	$f(x_5)$
T_6		x_6	$f(x_6)$
T_7		x_7	$f(x_7)$

FIG. 4-14. Setup of table for table look-up routine.

METHOD 1. We have already illustrated above (see Sec. 4-2 under Instruction Modification) one simple technique that can be used when the independent variable in the table takes on successive integer values.

METHOD 2. Consider now the case of nonintegral values for the independent variable, but where the intervals between successive values of the independent variable given in the table are all equal. In this case, given the value of the independent variable for which the value of the function is desired, simply divide it by the interval to obtain an integer; thence this case reduces to the one considered in method 1.

The interesting case occurs when the values of the independent variable given in the table do not have equal intervals between them. Here, to be specific, suppose that we are in a two-address system and the table is recorded in successive cells of the computer's memory as follows: The value of the independent variable is found in the α position of a word, and the corresponding value of the dependent variable is found in the β position of the same word (see Fig. 4-14). Let us call the successive values of the independent variable $x_0, x_1, x_2, \ldots, x_f$, where of course

$x_0 < x_1 < x_2 < \cdots < x_f$, and of the dependent variable $f(x_0)$, $f(x_1)$, $f(x_2)$, . . . , $f(x_f)$, and suppose that they are recorded in addresses T_0, T_1, T_2, . . . , T_f. We desire to find $f(N)$ for some given value N, where it is assumed that $N = x_i$ for some i; that is, no interpolation is considered here.

METHOD 3. One simple procedure is to compare successively $N:x_0$, $N:x_1$, $N:x_2$, . . . , until, for some i, $N = x_i$; then $f(N) = f(x_i)$ can be read from the computer's memory as desired. This, of course, can be accomplished by a simple iteration.

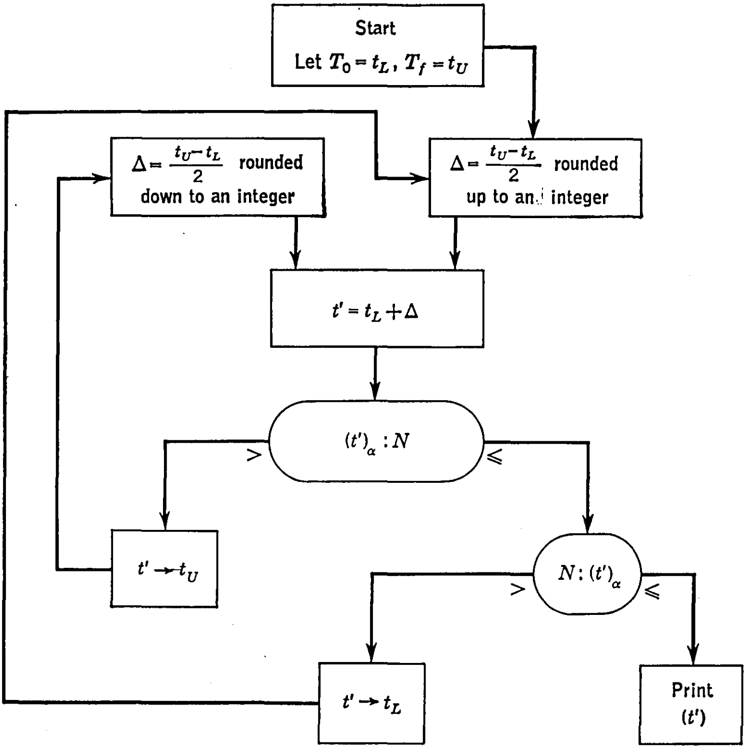

Start
Let $T_0 = t_L$, $T_f = t_U$
$\Delta = \frac{t_U - t_L}{2}$ rounded down to an integer
$\Delta = \frac{t_U - t_L}{2}$ rounded up to an integer
$t' = t_L + \Delta$
$(t')_\alpha : N$
$>$
$\leqslant$
$t' \rightarrow t_U$
$N : (t')_\alpha$
$>$
$\leqslant$
$t' \rightarrow t_L$
Print (t')

FIG. 4-15. Flow chart for table look-up, method 4.

METHOD 4. There is another method that on the average will enable $f(N)$ to be obtained faster. The flow chart for this method appears in Fig. 4-15. The method consists in determining in which half of the table $x_i = N$ is located; then the halving procedure is again applied to this smaller part of the table, etc., until a single cell is left. For example, consider Fig. 4-14, and suppose that N is x_5. Then the first step of this method is to determine that N is included in T_4 through T_7. The second step locates N further, as being in T_4 or T_5. The third step in this case locates N as being in T_5. The code would be written as follows: At any step let t_L and t_U be the lower and upper addresses, respectively, of the range of addresses in the table under consideration. Then form $t' = t_L + \Delta$, the address that divides this range of address into two parts, where $\Delta = (t_U - t_L)/2$. Then we look to see whether N lies in the upper or lower half of the range; i.e., is the contents of the α part of t' greater or less than or equal to N, that is, $(t')_\alpha{:}N$? If $(t')_\alpha > N$, then t' becomes the upper address of the new range; that is, $t' \to t_U$. If $(t')_\alpha < N$, then t' becomes the lower address of the new range; that is, $t' \to t_L$. Of course, if $(t')_\alpha = N$, we have the desired result. There is one little complication, however. When $\Delta = (t_U - t_L)/2$ is formed, it may not be an integer; but $t' = t_L + \Delta$ is an address and therefore must be an integer, hence must be rounded to an integer. A little reflection will show that, if $(t')_\alpha > N$, then the next Δ should be the nearest *smaller* integer to $(t_U - t_L)/2$, while if $(t')_\alpha < N$, the next Δ should be the nearest *larger* integer to $(t_U - t_L)/2$.

For example, consider Fig. 4-14 in connection with the flow chart of Fig. 4-15, and suppose for argument's sake that $N = x_5$ and that $T_0 = 100$, $T_1 = 101$, $T_2 = 102$, . . . , $T_7 = 107$. Then we would start with $t_L = 100$, $t_U = 107$, whence $\Delta = (107 - 100)/2 = 3\frac{1}{2}$, rounded up = 4. Hence $t' = 100 + 4 = 104$. Comparing $(104)_\alpha{:}N$, we find $N > (104)_\alpha$. Hence 104 is t_L, and we form $\Delta = (107 - 104)/2 = 1\frac{1}{2}$, rounded up = 2. Thus $t' = 104 + 2 = 106$. Comparing $(106)_\alpha{:}N$, find $N < (106)_\alpha$. Hence 106 becomes t_U. Thus $\Delta = (106 - 104)/2 = 1$, whence $t' = 104 + 1 = 105$. Comparing $(105)_\alpha{:}N$, find $N = (105)_\alpha$, and the result has been obtained.

EXERCISES

In all the exercises below the student should feel free to define explicitly any instructions that he may require or that may help simplify the code (particularly in the one-address system), based on the material of the previous two sections (Secs. 4-4 and 4-5), unless specifically stated otherwise by the exercise. These instruction definitions are to be listed preceding the code.

(*a*) Define appropriate one-address instructions, and code both the split-tree and jump-table method for a four-way branch.

(*b*) Suppose that only a single on-off switch could be on at one time. Define appropriate one-address instructions, and code a subroutine that will determine the position of a switch that is turned on.

(*c*) Write a code for the same problem as (*b*), but now consider one-shot switches.

(*d*) Write an input subroutine, using the one-address input instruction given in this section.

(*e*) Using a one-address system, with an input instruction that reads in one word at a time as described in this section, and assuming that the word format has a single auxiliary bit, define appropriate counter-setting instructions, etc., and write a read-in subroutine.

(*f*) Write a code for method 3 of the table look-up problem. Make use of an *extract* instruction to extract the successive x_i onto all zeros to compare with N; then extract $f(N)$ onto zeros for the read-out. Also assume that there are two auxiliary bits and one relative counter to aid the iteration.

(*g*) Code method 4 of the table look-up problem in a two-address system.

4-7. The Control Panel

Purpose of the Control Panel. The control panel has three main functions: (1) to enable the operation and monitoring of the computer during the computation of a program; (2) to aid the checking, or "debugging," of a new program; and (3) to aid the maintenance of the computer by the engineer. The control panel and the program itself are the main communication links between the operator and the computer. By means of the control panel the computer can be started and stopped, computations can be initiated, magnetic tapes can be properly positioned and other input-output equipment controlled, computer voltage and current levels, temperature, power input, etc., can be checked and adjusted, and so forth. There are wide variations in the features of control panels of different computers. In this section we shall discuss some of the possible features that a control panel might have (see Fig. 4-16).

In general, information is transmitted between the operator and computer by means of lights and buttons on the control panel. However, there is one method for monitoring a program during the computations using neither lights nor buttons that deserves special mention. It consists simply of a radio placed near the computer tuned to pick up signals generated during the computation. The computer, operating at a megacycle or more, presents a radio frequency; recursion loops generate audiofrequency modulations that can be detected. Various sound patterns will correspond to the different sequences or loops of instructions in the program. A coder who is computing many reruns of the same code, as would be the case in computing mathematical tables, can frequently relate these sounds to the various iterations of his program. A trained ear can then often detect errors or other troubles that his computations may have, just by "listening" to the program while it is being computed.

Lights and Signals. Usually one first notices the lights and signals displayed by the control panel. By means of these signals the computer can tell the operator something about its internal state. There are in general three kinds of lights and signals: the first kind tells why the computer stops during the course of a computation; the second presents a visual picture of the contents of various registers and memory cells; the third gives the electrical state of the computer.

When the computer stops or halts, usually one of several lights comes on, indicating the *cause for the stop;* i.e., a *stop* instruction has been executed (called a final stop), or an instruction being executed may contain an erroneous operation code not meaningful to the computer, or an instruction being executed may have reference to an erroneous memory address not legal for the computer,† or a check stop due to an auxiliary bit may have occurred, and so forth. Control panels are often constructed so that, when the computer has stopped or halted, the *contents*

† Frequently the full complement of memory address is not constructed for a computer. Hence an erroneous instruction would refer to an address that the computer does not have.

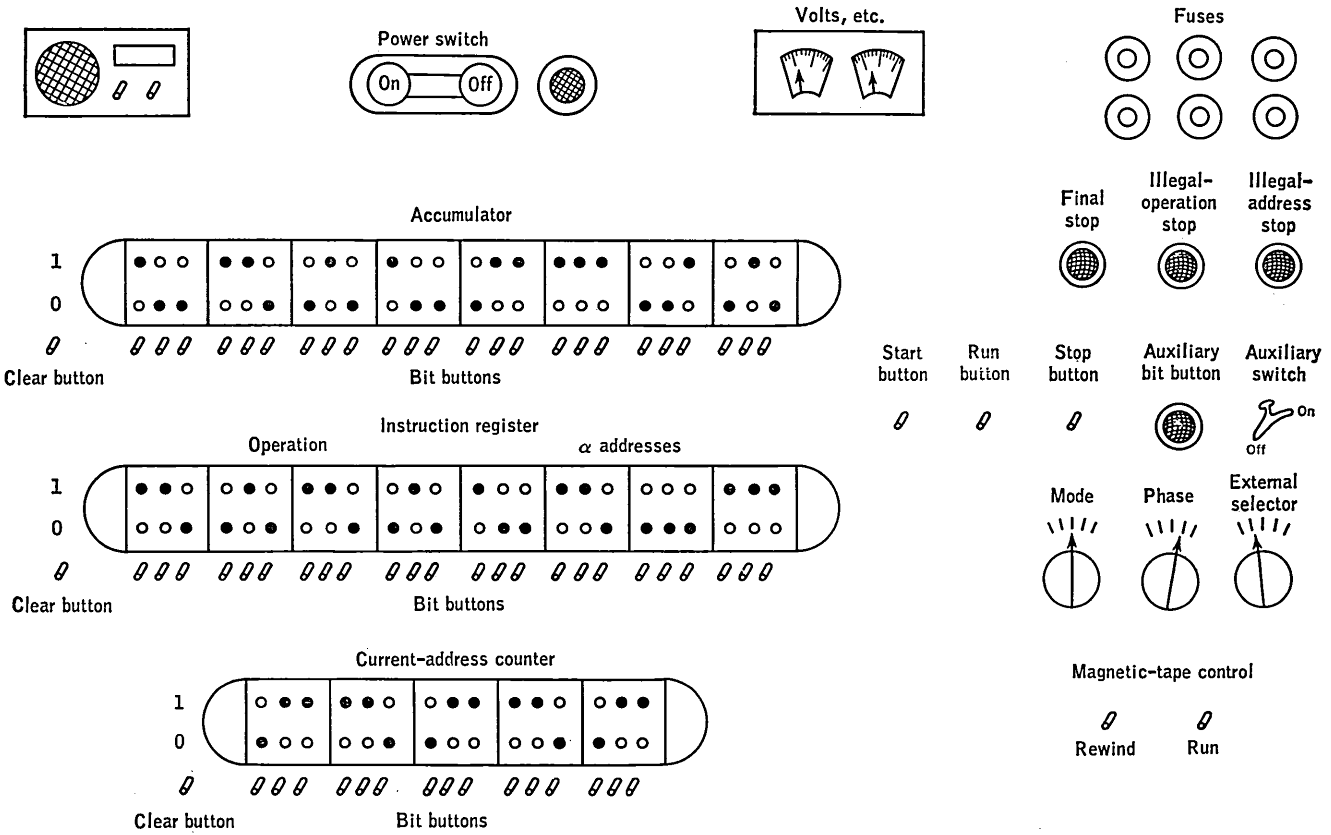


FIG. 4-16. A simple control panel. In the illustration the accumulator reads 46243712, the instruction register reads 62624607, and the current-address counter reads 36363.

of various registers can be ascertained from sets of lights. For instance, there may be two small "grain-of-wheat" neon bulbs on the control panel corresponding to each bit of the accumulator. When the top bulb of the pair is on, the corresponding bit might be a unit; when the bottom bulb is on, the corresponding bit would be a zero. Similarly the contents of the instruction register of the current-address counter and of other registers and counters can be displayed. If an *electronic failure* has occurred, such as a blown fuse or a change in voltage or current readings beyond an allowable limit, appropriate signal lights will come on. Of course there will be a pilot light indicating that the power is on, etc.

Buttons and Switches. The only means the computer operator has to control the computer, other than by the code itself, is by the buttons and switches on the control panel. Besides the *on-off switch* for the power there is a *start* button, which initiates the computations. For example, when the start button is pushed, the computer may automatically take as the first instruction the contents of address 000. There is also a *stop* button; when pushed, the computations cease. In addition to the start button there may be a *run* button, which is used to restart computations after a halt has occurred in the middle of a computation. In this case the difference between the start and run button is that the run button tells the computer to continue to perform the instruction already in the instruction register, whereas the start button tells the computer to initiate a particular standard predetermined sequence of events, e.g., take 000 as the next instruction.

The computer can usually be operated in several different *modes.* For example, by setting a switch the computer can be made to execute only a single instruction each time the start or run button is pushed. In this way a sequence of instructions can be "stepped through" slowly by successively pushing the run button. This would be done, for example, if one desired to see several of the instructions and their results on the lights of the control panel that display the contents of various registers. On the other hand a different setting of this mode switch can make the computer sequence itself through the instructions of a program at a very much reduced rate. Finally of course the mode switch can be set so that the computations proceed at full speed. Often in the single instruction mode, there is a *phase-selector* switch which makes the computer proceed through just *one* phase each time the run button is pushed.

So much for the special methods of starting the computer; consider next special methods for stopping or halting the computer. In previous sections we have observed how the operation of manual switches can be used to halt the computation when auxiliary bits are present. These switches are called *auxiliary switches.* There can be several auxiliary switches, each associated with a different auxiliary bit of the instruction word. Then, by setting the proper auxiliary switch, the computer can be made to stop at different sets of instructions. Other switches may make the computer halt at a particular phase when the stop button is pushed.

One of the most important uses of a control panel is to insert words manually into particular memory locations. Such an occasion may arise, for example, when it is necessary to correct only one or two words in a long program that is already in the computer's memory. There are several methods used to accomplish this. One method is to have one button associated with each bit of the registers that may be displayed by lights on the control panel. When the computer has been halted, the contents of a register may be changed by means of these buttons. For example, there may be one "clear" button for each register that, when pushed, makes the contents of the respective register all zeros. Then the bit buttons of this register are pushed to insert units in the desired bit locations. Under these conditions the procedure for inserting a word into a particular memory position would be as follows: Halt the computer; manually insert the desired word in the accumulator; similarly insert an instruction in the instruction register that will transfer the contents of the accumulator into the desired memory location. Set the mode switch for a single instruction, and push the run button. The inserted instruction will then be executed, and the word will be inserted into the desired memory location. Other methods might make use of the external input equipment to read a single word into the desired memory address.

There is usually a set of switches associated with the input-output equipment. Although, as we have seen, the selection of input-output equipment can be made by the program itself, frequently provision is made on the control panel for the manual selection of external equipment. Other switches present a means for manually winding, rewinding, and positioning magnetic-tape units. More complicated external equipment such as cathode-ray or television displays of course require still additional controls.

Let us now illustrate a possible procedure for initiating a computation by means of our control panel. First the input medium is properly set in the input device. Next the input device itself is turned on, and the external selector is set to choose this input equipment. An instruction is inserted into address 000 that will read in the whole program. The accumulator, instruction register, and current-address counter are all cleared (why?). The mode is set at high speed, and then the start button is pushed. The computer will then execute the instruction found in 000 first.

EXERCISES

(*a*) Suppose that a characteristic "click" can be heard on the radio monitor each time the computer goes through a loop. Suppose also (for simplicity in this exercise) that any instruction takes precisely 100 μsec to execute on our one-address computer. Write a code that will play the first few notes of the song "Dixie." (HINT: Look up the audio frequency of the first few notes of "Dixie," and code successive loops that will repeat at these frequencies.)

(*b*) Describe a procedure for finding out, by means of the control panel, what word is in a particular memory address.

(*c*) Suppose that a subroutine were necessary in order to read in a program. What would be a procedure for initiating a program by means of the control panel?

(*d*) If a computer needs a read-in subroutine, how is the read-in subroutine itself read in (for the very first time)?

4-8. Additional Topics

a. Survey of Instruction Systems. Make a survey and analysis of instruction definitions used in many different presently available computers. Obtain the information from the programming manuals of computers such as the Sperry Rand UNIVAC 1103 and 1105; the IBM 650, 704, 705, and 709; the Bendix G15; the Datamatic 1000; the Datatron; and the Electrodata. Classify the instructions by the categories given in this chapter.

b. Survey of Control Consoles. Make a survey of control consoles and the function of the different switches, buttons, and lights on each. Use instruction and maintenance manuals for the computers mentioned above to obtain the information.

c. Indirect Addressing. If there are n bits in an address, then only 2^n memory locations can be addressed directly. Indirect addressing must be used when it is desired to address more than 2^n locations. For example, suppose that we desire to address $w \times 2^n$ memory locations. The entire memory would be divided into w parts, each with 2^n memory locations. For each of these w parts there will be a special register or memory location reserved, the contents of which will be the address desired within this part. Then, to address one of the $w \times 2^n$ memory locations, an instruction would simply address one of the w special memory registers, and the contents of this address would itself be interpreted as the address desired in the corresponding part of the memory. Variations of indirect addressing are used in the NBS Pilot computer, the Philco TRANSAC 2000 computer, and the IBM 709. Study these uses of indirect addressing.

d. Microprogramming. R. J. Mercer† defines microprogramming as "the technique of designing the control circuits of an electronic computer to formally interpret and execute a given set of machine operations as an equivalent set of micro-operations, elementary operations that can be executed in one pulse time." There are two logically closely related types of microprogramming, one analogous to indirect addressing, and the other that is concerned mainly with the logical design of the control circuits of the computer. The analogy to indirect addressing occurs when the instruction operation is an address, rather than an operation, and the contents of this address is interpreted as a sequence of micro-operations. The micro-operations might each consist of 4 bits, and there might be 10 of these in a single 40-bit word. The micro-operations themselves would indicate transfers of words between various registers, setting of switches to be sensed at a later time, various types of shifts, arithmetic operations, logical operations, and so forth. The same effect may be obtained by sequencing micro-operations by means of coding and decoding matrices. See, for example, R. J. Mercer's cited article; M. V. Wilkes and J. B. Stringer, Microprograming and the Design of the Control Circuits in an Electronic Digital Computer, *Proc. Cambridge Phil. Soc.*, April, 1953; H. T. Olantz, A Note on Microprograming, *J. Assoc. Computing Machinery*, April, 1956; S. V. Blankenbaker, Logically Micro-programmed Computers, *IRE Trans. on Electronic Computers*, vol. EC-7, no. 2, pp. 103–109, June, 1958.

† Robert J. Mercer, Micro-programming, *J. Assoc. Computing Machinery*, vol. 4, no. 2, pp. 157–171, April, 1957.

e. The Minimum Computer. It is clear that a general-purpose computer does not really "need" the multiplication or division instructions, for these operations can be formed by a combination of additions or subtractions. Similarly other operations can be omitted and replaced by simpler ones. The question arises: What are the minimum instructions necessary for a general-purpose computer—or, indeed, what criteria can be applied to determine whether or not a computer is actually a general-purpose computer? These questions have long been of interest to mathematicians and logicians. The so-called Turing machine, or "universal computer," was developed for this purpose (see A. M. Turing, On Computable Numbers, with an Application to the Entscheidungs Problem, *Proc. London Math. Soc.*, ser. 2, vol. 42, pp. 230–265; a more readable description is given in Martin Davis's "Computability and Unsolvability," chap. 1, McGraw-Hill Book Company, Inc., 1958). More recently S. P. Frankel has considered the minimum general-purpose computer; see his On the Minimum Logical Complexity Required for a General Purpose Computer, *IRE Trans. on Electronic Computers*, vol. EC-7, no. 4, December, 1958. Frankel's computer can only subtract. Can you program multiplication on Frankel's minimum computer (called the *Microcephal AC*)?

f. References on Programming Texts and Periodicals

Alt, Franz L.: "Electronic Digital Computers—Their Use in Science and Engineering," Academic Press, Inc., New York, 1958.

Jeenel, Joachim: "Programming for Digital Computers," McGraw-Hill Book Company, Inc., New York, 1959.

Ledley, R. S., and J. B. Wilson: "Programming and Utilizing Digital Computers," McGraw-Hill Book Company, Inc., New York, in press.

Levin, Howard S.: "Office Work and Automation," John Wiley & Sons, Inc., New York, 1956.

McCracken, D. D.: "Digital Computer Programming," John Wiley & Sons, Inc., New York, 1957.

Von Neumann, J., and H. H. Goldstine: "Planning and Coding of Problems for an Electronic Computing Instrument," Institute for Advanced Study reports written under U.S. Army Ordnance Contract W-36-034 ord 7481, Princeton, N.J., 1947–1948.

Wilkes, M. V., D. J. Wheeler, and S. Gill: "The Preparation of Programs for an Electronic Digital Computer," 2d ed., Addison-Wesley Publishing Company, Reading, Mass., 1957.

Wrubel, Marshal H.: "A Primer of Programming for Digital Computers," McGraw-Hill Book Company, Inc., New York, 1959.

Periodicals (in which special computer programming problems are discussed)

Automatic Control, monthly, New York.

Automation, monthly, Penton Publications, Cleveland, Ohio.

The Computer Journal, quarterly, British Computer Society, Ltd., London.

Computers and Automation, 10 times a year, Edmund C. Berkeley and Associates, New York.

Control Engineering, monthly, McGraw-Hill Publishing Company, New York.

Journal of the Association for Computing Machinery, quarterly, New York.

Journal of Operations Research Society of America, quarterly, Baltimore.

CHAPTER 5

ADVANCED PROGRAMMING

5-1. Introduction

The previous chapter was concerned with the details of programming and coding at the instruction level. The four-, three-, two-, and one-address instruction formats were described and compared. The octal-coded binary and decimal-coded binary systems were compared. The steps in preparing a flow chart, symbolic code, and finally the code itself written in the actual machine language were described. Decision branches, instruction modification, and recursion codes were discussed and examples given. Coding was found to be detailed and laborious, and the one-address system particularly aggravated these qualities.

Advanced programming techniques attempt to lighten the load of the programmer and coder. Its purpose is to try to make the computer itself help prepare the program or code, minimizing the amount of writing a programmer need do. The means used to accomplish this is to prepare or precode the computer with an intermediate, or *automatic, program* which is always available in the computer memory. This intermediate or automatic code encompasses certain aspects of coding that are common to many programs and of general applicability. Then, when the programmer is coding any specific problem, his work can be shortened by taking advantage of already precoded items of the automatic program that are already in the computer. Such automatic programming can greatly simplify the coding of computers otherwise difficult to code. Hence, in the design of some computers, little attention is paid to ease of coding, and such advanced programming techniques are heavily relied upon to compensate for this. Unfortunately, however, the coder must then learn how to use not only the computer but also the automatic programming schemes. Evidently a balance must be sought in computer design between the amount of automatic programming envisioned and the difficulty of coding. It is clear that automatic-programming concepts are of great importance to the computer engineer.

Rarely is a program written that is found to be absolutely correct the first time it is run on the computer. Hence in this chapter we first consider automatic aids for analyzing an already coded program for errors. Having discussed automatic programs to aid the coder to correct his code, we turn to such programs that aid the computer engineer to maintain the computer. Next automatic programs designed to lighten the load of the programmer are considered. These can be roughly classed in two categories—the interpretive routines and the compiling routines. Various examples are discussed as illustrative of the methods and techniques of formulating and using such automatic programs. The specific examples given are intended to illustrate the principles involved, and their particular form as chosen for this chapter was dictated by pedagogical considerations only. As will become clear during the course of the chapter, there are many other ways to solve the problems posed, and in actual practice the final choice depends on the primary purpose of the code and the detailed characteristics of the computer to be used.

5-2. Program Debugging Methods and Routines

The Problem of Program Debugging. The first time a program is tried on a computer, it usually does not run correctly. If the start button is pushed and the program allowed to run, one of four results can occur: First, the computer might run up to a point and then just stop, or "hang up." Second, the computer could continue to run for an excessively long time, neither stopping nor producing any results. Third, the computer could run and, as expected, print out results, but wrong results. And finally the program could run correctly. In the first three cases an error is indicated in the program, and the problem is to detect where—at what instruction or instructions—the error occurs and then to correct it. This procedure is often referred to as "debugging" a program.

It is hard to describe in general how to debug a particular program; ingenuity on the part of the programmer, systematic and neat flow charts, coding sheets with detailed remarks, and a thorough knowledge of the code are, of course, all essential. When a program just stops, this usually indicates that the computer has been asked to do an illegal instruction; for instance, the instruction might have an operation code not corresponding to any operation or might contain an address that the computer does not have, etc. Illegal instructions are often the result of an error in instruction modification. When a program never stops, the program usually has a loop with a faulty exit. The exit may not have been properly set up, or else a nonconverging iteration may be in process, etc. The printing out of wrong results requires examination of these results in the light of the problem: Is the scaling wrong? Are the constants correct? Has any phase of the calculation been skipped, etc.?

However, certain techniques can be discussed that are often used to aid the debugging process. These can be generally classified into *dynamic* debugging and *post-mortem* debugging. Dynamic techniques try to check the program while it is actually running on the computer; post-mortem techniques examine the computer memory after the program has stopped, or "hung up."

Dynamic Program Debugging. Let us first consider dynamic program debugging. There are two generally used methods: the first is called the *break-point* (or *check-point*) method; the second is called the *automonitor* method. The break-point method consists in writing into the program, at critical intermediate points, instructions that either will stop the computer so that important intermediate results can be observed or will make the computer print out such intermediate results automatically. The sign bit on an instruction word is often used to facilitate such a process. For example, some computers are designed so that after executing an instruction with a negative sign the computer will stop. Usually such computers also have the following additional feature: A manual switch on the control panel can be thrown so that in one position the computer will *not* stop after a negative instruction—this being called the normal switch position—and so that in the other position the computer will stop—this being called the break-point switch position. The advantage of this control-panel switch is clear: in debugging a program, the negative instruction stops are used, and the switch is set accordingly; when the program has been corrected, these negative stops are no longer desired and hence the switch is thrown to normal.

Another break-point procedure depends on the programmer inserting into the program at intermediate critical points manual-switch *jump* instructions. In debugging the program, the switches are positioned so that the computer will jump when it encounters such an instruction; usually special print-out routines are included in the program, and the *jump* instructions transfer the computer to these routines. When the program has been corrected, the switches are turned off and the computer simply ignores the *jump* instructions.

Use of the Automonitor Program. The automonitor method for dynamic debugging does not require any additions or special considerations in the program itself, as does the check-point method. Instead the automonitor method requires the use of a special automonitor program which is read into the computer in addition to the program to be debugged. The automonitor program is general and once written can be used for every program to be debugged. Occasionally the computer has the properties of the automatic automonitor program already wired in, so that the automonitor process can be carried out simply by throwing a switch on the computer console. An automonitoring program makes the computer *print out—while each instruction of the program being debugged is being executed—the instruction itself, the contents of each of the operand addresses, and the computed result.* For example, suppose the program to be debugged is as follows:

Address	Instruction				
δ_0	Operation	α_0	β_0	γ_0	δ_1
δ_1	Operation	α_1	β_1	γ_1	δ_2
δ_2	Operation	α_2	β_2	γ_2	δ_3

Then the automonitor program would have the computer print out the following:

Address	Instruction	1st operand	2d operand	Result
δ_0	(δ_0)	(α_0)	(β_0)	(γ_0)
δ_1	(δ_1)	(α_1)	(β_1)	(γ_1)
δ_2	(δ_2)	(α_2)	(β_2)	(γ_2)

In other words, the automonitor program gives a *complete record* for the program being debugged of just which instructions are being done by the computer and what the results were. With this information it becomes easy to spot any error in the program being debugged. Note that the automonitor program is a program that operates on another program—more about this point will be said in the succeeding sections.

Of course certain instructions of the program being debugged are handled somewhat differently. For example, in a *compare* instruction, (γ) need not be printed, but just δ, (δ), (α), (β), for the address of the next instruction tells which way the comparison went. Common sense clearly dictates what an automonitor program should print out.

An automonitor program for a three-address computer would be similar to that just described for a four-address system. For a two- or one-address system the contents of the accumulator or other registers becomes of importance, and hence the contents of the accumulator is printed out. For instance, in a one-address system, the print-out would look like this:

```
δ₀  (δ₀)  (α₀)  (acc)
δ₁  (δ₁)  (α₁)  (acc)
δ₂  (δ₂)  (α₂)  (acc)
```

The Automonitor Program Itself. In order to grasp more fully the general concept behind an automonitor program, let us examine in a cursory way how such an automonitor program itself might be written for a one-address system. The flow chart is given in Fig. 5-1. This automonitor program is in one part of the memory, and the program to be debugged is in another part of the memory. The automonitor program must be given the address of the first instruction of the program to be debugged. Then the automonitor program prints out the desired record and also executes the instructions of the program to be debugged. First the automonitor program prints the address of this first instruction, δ_0; then it copies (δ_0) into a special temporary located in the automonitor program's part of the memory. This instruction is printed, and (α_0) is printed. Next the automonitor program directs the computer to execute the first instruction *in* its location *within* the automonitor program. Then (acc) is printed, and the automonitor program goes on to the next instruction, etc. There are a few important details that must be attended to. When the program to be debugged is being executed, the contents of the accumulator from instruction to instruction is, of course, important. But when the automonitor program executes one instruction at a time and inserts print routines, etc., between these instructions, the contents of the accumulator becomes lost. In order to correct this situation, the contents of the accumulator (and other registers) must be recorded after the execution of each instruction of the program being debugged. Then, before the next such instruction is to be executed, the registers are properly preset.† In this way the two programs are properly interleaved.

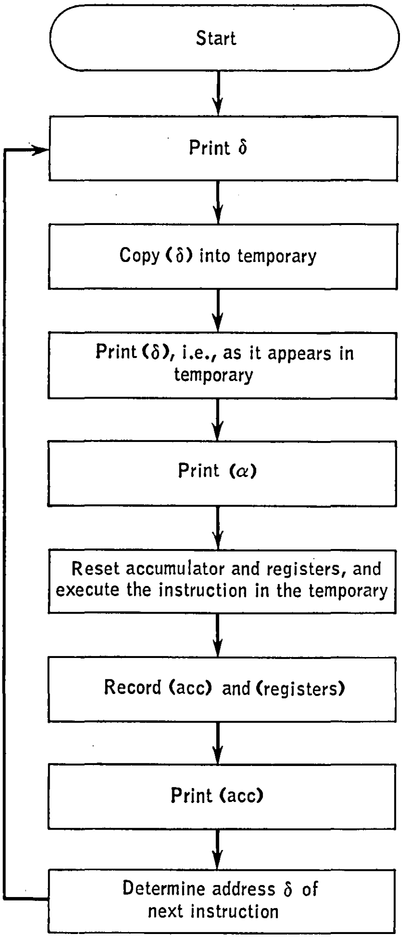

Start
Print δ
Copy (δ) into temporary
Print (δ), i.e., as it appears in temporary
Print (α)
Reset accumulator and registers, and execute the instruction in the temporary
Record (acc) and (registers)
Print (acc)
Determine address δ of next instruction

FIG. 5-1. Flow chart of automonitor program.

Post-mortem Program Debugging. Consider next post-mortem program debugging. The first method is to run the program through and observe the results. From a knowledge of the program and the erroneous results an error can often be located. For example, if the magnitude of the results is off, the scaling is checked for error, etc. The second method is to print out the entire memory of the computer after the program has been run. If the program got into a loop with no exit, then the tallies can be checked on this memory printout. All instructions that were to be modified by the program can be checked to see whether or not they were correctly modified, etc. A third method is to have another program that examines the program to be debugged after it has run. This program compares the program to be debugged with itself before and after it has been run and tells the programmer where they differ. In this way the programmer can determine if any instructions were inadvertently destroyed, if all differences that should appear actually do appear, etc.

Of course the programmer uses combinations of the techniques that are at his disposal. What we have briefly outlined here is in no way com-

† Also, provision is made in the automonitor code so that *jump* instructions are not performed directly but rather operate on the automonitor δ-address reading instruction.

plete but rather indicates some of the directions from which aids to programming may arise.

EXERCISES

(*a*) Write out precisely what an automonitor program would print if applied to the code of Example 3-1*c*.

(*b*) Suppose that a code contains a negative *jump* instruction. How can the automonitor program handle such an instruction? After the automonitor program executes the negative jump, how can it regain control again?

(*c*) Write a simple post-mortem debugging program as required for the third method.

5-3. Computer Maintenance Routines

Maintenance. If a computer circuit is malfunctioning, then the computer will not always produce the correct results. Evidently these incorrect results must in some way reflect the course of the malfunctioning. Hence it seems possible that specific routines can be designed especially for determining a malfunctioning circuit. Such programs are called *maintenance routines.* Of course one must be able to read into the computer, read out of the computer, and perform some operations in order that a maintenance routine be capable of application at all. Then there is the fact that even though a programmer using a computer may swear that it is malfunctioning, it may often be in perfect operating condition. Hence a maintenance routine must check to see whether or not the computer is indeed malfunctioning. However, if the user of the computer can demonstrate that it gives two different results to the same program, then the computer must be malfunctioning no matter what the maintenance routines determine. Finally observe that a maintenance routine often cannot detect a particular malfunctioning circuit but can determine only that one of several circuits is not operating correctly. Then, of course, the computer maintenance engineer must make electronic tests on these circuits themselves inside the computer.

Actually maintenance routines are composed after a detailed knowledge of the computer logical circuitry has been mastered. However, we have placed this discussion in Part 1, rather than at the end of Part 4, to illustrate for the prospective computer engineer the necessity of a thorough understanding of coding.

Locating Malfunctioning Parts. Maintenance routines usually are designed to determine whether or not some particular part of the computer is working correctly. For example, a routine may test the input-output equipment, another may test different parts of the memory, another may test the add operation, another the multiply operation, etc. The logic most often used in interpreting the meaning of a maintenance routine test is: if the part of the computer tested by the routine is operating correctly, then the malfunctioning circuit must be located in some *other* part of the computer. In this way the possible malfunctioning circuit is located by the elimination of correctly operating circuits. The fact that a maintenance routine test *fails* does *not* necessarily mean that the malfunctioning circuit is located in the part tested by this routine, whereas the fact that a maintenance routine test passes *does* mean that the circuits in the part tested are in operating order. The reason for this is that it is extremely difficult to distinguish by means of a maintenance routine between a malfunctioning circuit in the control unit and some malfunctioning circuit in the arithmetic unit.

As an example of a maintenance routine, suppose it has already been determined that the first sixteen memory cells are operating correctly, in addition to the accumulator and the *add, subtract, transfer,* and *jump* instructions. The problem is to test the rest of the memory. To see whether or not a memory location is dropping a bit, a word of all units is read into an address and, after a time, read out again. If the word read out is not all units, i.e., has at least one zero, then it can be concluded that this address is malfunctioning. Hence the entire memory from 020 on is filled with words of all units; and the program of Table 5-1 will test whether or not any bits are being dropped, where it is assumed that the computer is designed so that an overflow will cause a stop and that an illegal α will also cause a stop. For this code (accumulator) must initially be a word of all units—i.e., initially (acc) = 77777. Now, if a memory address being tested does *not* drop a bit, then the result of the subtraction will be all zeros and hence, after (012) is added to the accumulator, the initial conditions are once more set up and the next memory address can be tested. On the other hand, if a memory address *did* drop a bit, then some unit would be a zero and the result of the subtraction

TABLE 5-1. MAINTENANCE ROUTINE

Address	Instruction		Remarks
	Operation	α	(preload accumulator with 77 777)
000			
001	Subtract	020	Subtract contents of a memory address from accumulator, and put result into accumulator
002	Add	012	Add a word of all units to accumulator; if there is an overflow, we are assuming that computer will stop
003	Subtract	012	Clear accumulator
004	Add	001	Modify *subtract* instruction found as (001) to consider the next successive memory address
005	Add	013	
006	Transfer	001	
007	Subtract	001	Clear accumulator
010	Add	012	Set up initial conditions for next loop; take next instruction from 001
011	Jump	001	
012	77	777	Constant: a word of all units
013	00	001	Constant: 00 001

would contain a unit in the position where the memory address had a zero. Hence, when this result is added to a word of all units, overflow will occur and the computer will stop. (It was, of course, assumed that the computer in question had this property.) Then the α address of the instruction stored in address 001 is the malfunctioning address; and the position where the units of the accumulator turn to zero is the position where the bit is being dropped, i.e.:

$$(\cdot\cdot\cdot\ 111\quad 111) - (\cdot\cdot\cdot\ 11\overset{\substack{\text{Dropped bit}\\ \downarrow}}{0}\quad 111) = (\cdot\cdot\cdot\ 001\quad 000)$$

whence

$$(\cdot\cdot\cdot\ 001\quad 000) + (\cdot\cdot\cdot\ 111\quad 111) = (\cdot\cdot\cdot\ 00\underset{\substack{\uparrow\\ \text{Position of dropped bit}}}{0}\quad 111)$$

When a bit is dropped, this can often be due to either the control circuitry or the memory itself. Then these must be checked electronically.

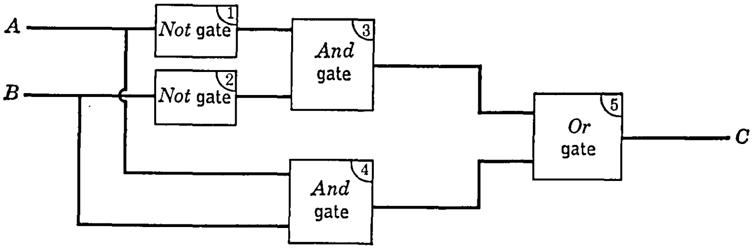


FIG. 5-2. Comparison (or equalizer) circuit.

As another example, suppose that the *add*, *compare*, and *transfer* instructions and the memory were operating; the problem is to check the *shift-left* instruction. Since shifting (α) left n positions is the same as forming $2^n \cdot x$ [where $(\alpha) = x$], we can perform the equivalent of shifting by adding a number to itself, and the result to itself, etc., repeated n times. Hence we perform the shift and also perform the *add* interpretation of a shift, and then see if we have gotten the same answer. Note that two things need to be checked: first that the bits do shift from each position to the next successive position to the left; also that the number of positions shifted is the number called for in the instruction.

Locating Malfunctioning Gates. Up to now we have discussed maintenance routines that attempt to locate the malfunctioning circuit down to some rather gross part of the computer. On the other hand it is sometimes possible to locate the actual gate or gates that may be causing the trouble. For example, consider the comparison (or equalizer) circuit described in Sec. 2-5, which is redrawn in Fig. 5-2.

Suppose that the gates were the relay gates described in Sec. 2-5. Let us assume that, if a normally open relay is malfunctioning, its contact is dirty and it does not close properly; also that, if a normally closed relay is malfunctioning, its relay arm is stuck and it will not open. Under these assumptions, let us try to write a routine for determining which gates are not operating correctly, if any at all. To do this, note that there are only four possible different input conditions for this circuit, which are summarized by the four columns in the following table. If the circuit is operating correctly, then the outputs for each gate in the circuit corresponding to each input are also given in the table.

Inputs:	
A	0101
B	0011
Outputs:	
Gate 1	1010
Gate 2	1100
Gate 3	1000
Gate 4	0001
Gate 5	1001

If any other output appears, then the circuit is malfunctioning and the problem is: How can such an output be interpreted in terms of a specific malfunctioning gate? We shall assume that only *one* of the gates is ever malfunctioning at a time. Hence suppose that the *not* gate for A is malfunctioning. Since it is made of a single, normally closed relay, according to our above discussion, instead of putting out *not* A, it will put out all units, no matter what A is. Similarly for the *not* gate of B.

Now consider an *and* gate. If either of the two relays cannot close, the output of the gate will be all zeros. For the *or* gate, if either relay will not close, then the input of this relay will not get through the gate but the other input will get through. If both relays will not close, then the output will be all zeros. Hence three possible outputs can occur if the *or* gate is malfunctioning.

We can summarize the above discussion by writing the four possible input conditions together with the outputs of each gate of the circuit, assuming that one at a time of the gates is malfunctioning:

Input:						
A	0101	0101	0101	0101	0101	0101
B	0011	0011	0011	0011	0011	0011
Output:						
Gate 1	1010	1111†	1010	1010	1010	1010
Gate 2	1100	1100	1111†	1100	1100	1100
Gate 3	1000	1100	1010	0000†	1000	1000
Gate 4	0001	0001	0001	0001	0000†	0001
Gate 5	1001	1101	1011	0001	1000	1000 0001 0000 †

† Malfunction.

Only the output of gate 5 goes to the memory and can be observed by a computer program. Now suppose that we choose for the input to A a word such that $P4$, $P3$, $P2$, and $P1$ are the bits 0101 and for B a word such that $P4$, $P3$, $P2$, and $P1$ are the bits 0011. Then the first four bits of the output word will correspond to the four different possible input conditions. Hence we can say that if $P4$, $P3$, $P2$, and $P1$ of the output word (i.e., the output of gate 5) are

1101 Then gate 1 is malfunctioning
1011 Then gate 2 is malfunctioning
0001 Then either gate 3 or gate 5 is malfunctioning
1000 Then either gate 4 or gate 5 is malfunctioning
0000 Then gate 5 is malfunctioning

Hence it becomes clear how a code can be written to determine which gate or gates are malfunctioning. Note that in some cases it cannot be determined which of two gates is malfunctioning, but only that one or the other is. Of course we can again observe that the detailed circuit design must be known in order to write such a code.

EXERCISES

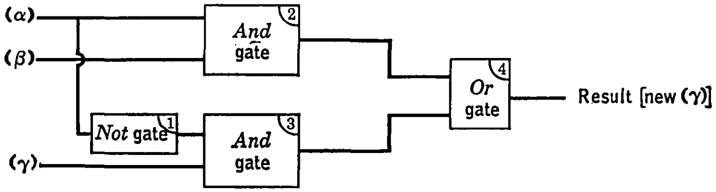
(*a*) Write a three-address code to check the *shift-left* instruction, assuming that the *add*, *compare*, and *transfer* instructions are operating, in addition to the memory.

(*b*) If *add*, *shift-left*, *compare*, and *transfer* instructions are operating, how would the multiplication instructions—both major and minor—be checked? (HINT: Be careful about not allowing any overflow when doing the additions.)

(*c*) Write the actual maintenance code for the gating circuit used in the above illustration. Use a one-address system.

If gate 1 is malfunctioning, put a 1 into 001
If gate 2 is malfunctioning, put a 2 into 001
If gate 3 or gate 5 is malfunctioning, put a 3 into 001
If gate 4 or gate 5 is malfunctioning, put a 4 into 001
If gate 5 is malfunctioning, put a 5 into 001

(*d*) Analyze the given circuit which does an *extract* instruction (in serial—see Chap. 2) for a maintenance code (where it is again assumed that the gates are relays that can malfunction as described above).



EXERCISE *d*

5-4. Interpretive Routines: Mathematical

Automatic Programming. There are two general kinds of *automatic programming*, called *interpretive* routines and *compiling* routines. The purpose of automatic programming techniques is to reduce the number of instructions actually written by the coder. The method for accomplishing this is previously to place in the computer a library of subroutines that can be used by every coder, so that these routines will not have to be rewritten each time they are needed. A set, or library, of particular subroutines, together with a method for using them, is called an automatic program. The difference in the two kinds of automatic programs lies in their use: In the interpretive program the subroutines are to be used by the coder's program while it is running on the computer, whereas in the compiling program the subroutines are used by the coder's program only at the time the coder's program is read into the computer.

We have arbitrarily classified interpretive routines into two types: (1) *mathematical,* and (2) *simulational.* The subroutines of the mathematical type of interpretive routines compute values of functions, while those of the simulational type are usually designed to make one computer look like some other kind of computer.

Use of Mathematical Interpretive Routines. Let us first consider an example of a mathematical interpretive routine. The purpose of such a routine is to enable a coder to form such functions as sin x, cos x, e^x, $x!$, $\sqrt[n]{x}$, etc., merely by writing a couple of instructions. Of course these functions would appear as subroutines already in the computer, as part of the interpretive routine. However, the interpretive routine must also *contain a section that can interpret the instructions that the programmer writes.* When a function subroutine is to be used, the following information must be recorded: (1) the address that contains the value of the independent variable x; (2) the address into which the result $f(x)$ is to be put; (3) the address in the program from which the jump to the subroutine was made, so that after the subroutine has been completed the computer will know where to *jump back* to the original program; and (4) the address to which to jump in order to go to the subroutine itself. Consider a two-address system: two words are needed to go to a subroutine (since four addresses are required). We could arrange the words as follows:

Operation	α	β
Return jump	Standard address	Address of 1st instruction of function subroutine
. . . .	Address of x	Address of $f(x)$

The first word is a *return jump* instruction; the second word is not an instruction but is just to tell the subroutine where x is and where $f(x)$ is to be put. Recall that the *return jump* instruction jumps to β, and in addition the instruction memorizes as the contents of α its own address. Hence the return jump takes care of items 3 and 4 above and the second word of items 1 and 2. Thus the programmer, each time he wants to use a subroutine listed in the interpreter library to evaluate a function, need write only the following "instructions": (*a*) a *return jump* instruction, where α is a standard address (standard, that is, for this particular interpreter), and where β is looked up in a book of instructions for the interpreter which lists the proper β for each function; (*b*) the word following the *return jump*, as described above. This has illustrated the procedure for using a mathematical interpreter.

The Interpretive Routine Itself. Now let us consider the interpreter program itself. Besides consisting of the function subroutines, as we observed above, it must also consist of a part that interprets the two words that the programmer writes. This part looks at the standard address to find out where the address of the *return jump* is. Knowing that the next word gives the locations of x and where $f(x)$ should be put, this part of the code then relates that information to the appropriate subroutine and transfers control to this subroutine. This part can be thought of as a subroutine also; we shall call it subroutine Q. The flow diagram might appear as in Fig. 5-3, where, to be specific, suppose that it is desired to use a cos x subroutine. The long dashes indicate the path of the computer through the program and subroutines; the short dashes, the operating of one subroutine on another subroutine. We let Z be the address of the *return jump* instruction; then $Z + 1$ is the address of the data word, and the program should resume at address $Z + 2$.

As an example of the type of things that occur in the subroutine Q, consider how it inserts the value of x into the subroutine for $f(x)$. Subroutine $f(x)$ has a certain cell reserved for the value of x; call this cell T_x for "temporary for x"; within subroutine Q is an instruction that transfers the value of x to T_x from the address given by the data word of the program. Suppose that the instruction were "Transfer (α) into β." Then Q must have a previous instruction that puts into the α address of the transfer instruction the address of x as given by the data word, i.e., an *extract* instruction that takes the α address of the contents of $Z + 1$ and puts it into the α position of the *transfer* instruction. But this *extract* instruction must involve the address $Z + 1$, which is obtained from the standard address associated with the return jump. Hence there must be previous instructions that take Z as given in the standard address, add 1, and insert this into the α part of the *extract* instruction. Next consider the β part of the x-to-T_x *transfer* instruction: this must be some address specifically reserved in the subroutine of $f(x)$ for the value of x. Since subroutine Q is a general subroutine, it must be told this address each time a subroutine for $f(x)$ is to be used. This is done in the part of the subroutine for $f(x)$ denoted in Fig. 5-3 by "In subroutine Q set $R = R_2$, $f(x) = \cos x$." In this part there is an instruction which extracts the address reserved for x into the *transfer* instruction of subroutine Q. This process is valid since the computer will pass through this instruction before getting to subroutine Q. Summarizing, there is an instruction in the subroutine for $f(x)$ that sets β of the *transfer* instruction; and there is a series of instructions preceding the *transfer* instruction in subroutine Q that leads to the setting up of the α address of the *transfer* instruction.

Our example of a mathematical interpretive routine has several interesting features: First, one subroutine uses another subroutine—i.e., each subroutine for $f(x)$ uses the subroutine Q. Such a feature may occur in other ways also. For example, in evaluating cos x, suppose that the power series were used,

$$\cos x = \sum_{n=0}^{\infty} (-1)^n \frac{x^{2n}}{(2n)!}$$

Then the subroutine for cos x will use the subroutine for $a^x/x!$, where $x = 2n$ in this case. In fact, to use the subroutine for $a^x/x!$, the same technique of the return jump and data word is used (why?). Of course the use of subroutines by subroutines can be carried deeper than two or three. Second, note how many instructions are actually needed to perform the simple transfer described above. The moral of this is that coding a simple job can be complicated on a computer. Of course this is the reason that it pays to have a separate subroutine Q rather than to repeat the instructions of this subroutine for each $f(x)$ routine. Finally, the main purpose of this mathematical interpretive routine is to make the coding problem easier by allowing only two words to be written when a function is to be evaluated. Of course the process described here was greatly simplified, since many other complications may arise—telling the $f(x)$ subroutine to what accuracy the function is to be calculated, taking care of the proper scaling, etc.

EXERCISES

(*a*) Write in symbolic code the series of instructions that lead to the setting up of the α address of the *transfer* instruction.

(*b*) How would the evaluation of a function of two independent variables affect the discussions of this section?

5-5. Interpretive Routines: Simulational

Use of Simulational Interpretive Routines. Next let us consider simulational interpretive routines. The purpose of these routines is to make one computer appear as an entirely different computer. For example, suppose that we had a computer that did *not* have floating operations. By means of the proper routine we can make this computer *appear* to be a computer that has built-in floating-point operations. Or we can make a one-address computer appear to be a two- or three-address computer. Another example might be the making of an ordinary computer into a special-purpose war-gaming computer. Of course the coding for the simulated computer will be quite different from the coding of the actual computer. However, the purpose of the simulational interpretive routine is to enable the programmer to write his code in the language of the simulated computer; he need never know that this is not the actual

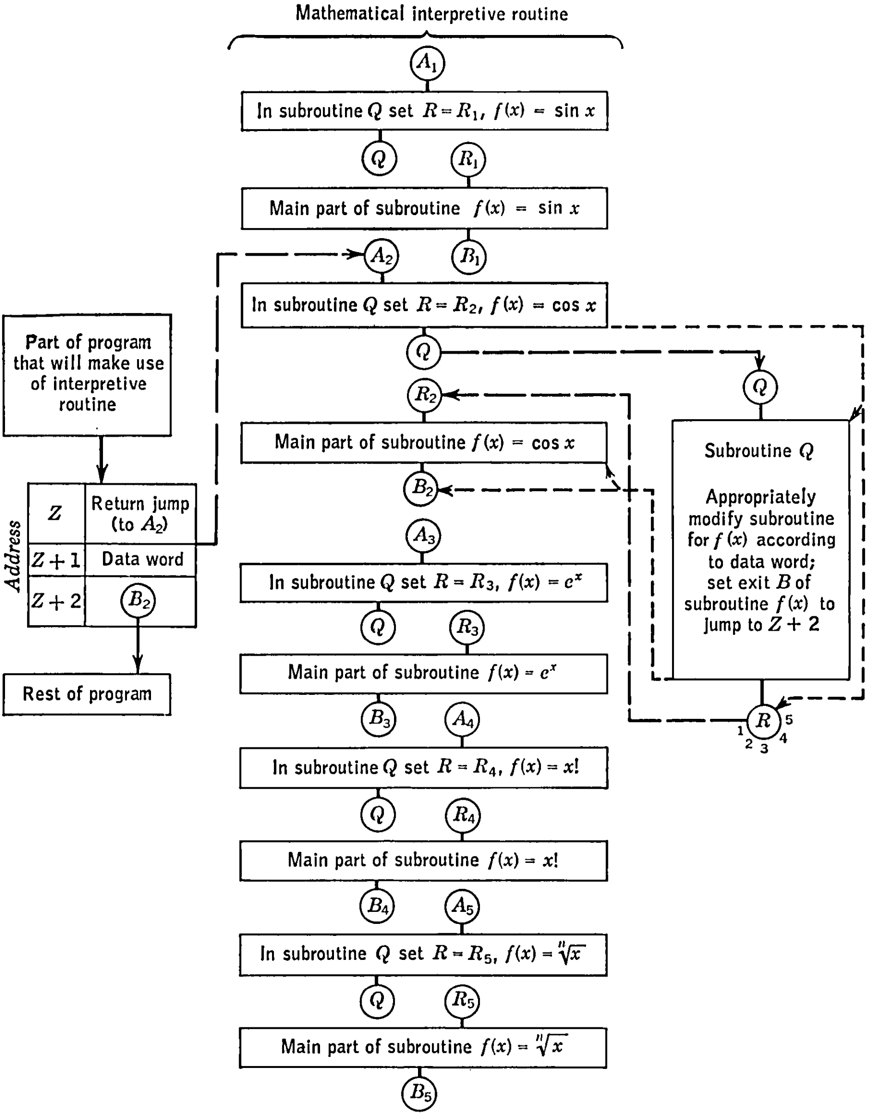
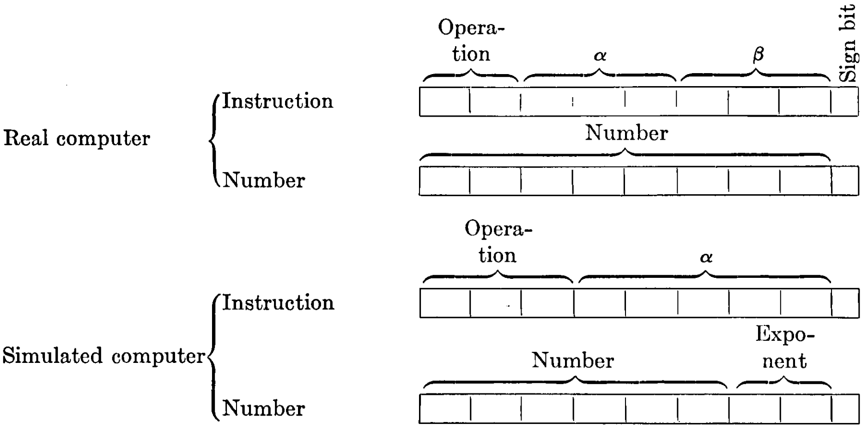

Mathematical interpretive routine
A_1
In subroutine Q set $R = R_1$, $f(x) = \sin x$
Q
R_1
Main part of subroutine $f(x) = \sin x$
A_2
B_1
In subroutine Q set $R = R_2$, $f(x) = \cos x$
Q
R_2
Main part of subroutine $f(x) = \cos x$
B_2
A_3
In subroutine Q set $R = R_3$, $f(x) = e^x$
Q
R_3
Main part of subroutine $f(x) = e^x$
B_3
A_4
In subroutine Q set $R = R_4$, $f(x) = x!$
Q
R_4
Main part of subroutine $f(x) = x!$
B_4
A_5
In subroutine Q set $R = R_5$, $f(x) = \sqrt[n]{x}$
Q
R_5
Main part of subroutine $f(x) = \sqrt[n]{x}$
B_5
Part of program that will make use of interpretive routine
Address
Z
Return jump (to A_2)
$Z+1$
Data word
$Z+2$
B_2
Rest of program
Q
Subroutine Q
Appropriately modify subroutine for $f(x)$ according to data word; set exit B of subroutine $f(x)$ to jump to $Z+2$
R
1
2
3
4
5

FIG. 5-3. Flow chart of interpretive routine.

method of coding for that computer. Obviously, then, the interpretive routine must interpret every instruction of the programmer's simulation-language code. Thus the interpretive routine looks at the first instruction of a simulation-language code, interprets it, then goes to the next instruction of the simulation-language code, interprets it, etc., for each instruction passed through in the simulation-language code. Immediately we see that the interpretive routine must also keep track of what instruction of the simulation-language code to take next; i.e., the interpretive routine must contain a simulated-instruction address counter, a simulated "simulation-language" instruction register, simulated decoder, simulated accumulator, etc. In other words, the simulation routine must perform all the functions that the electronic circuitry would perform if the simulated computer had been real.

The Interpretive Routine Itself. For the sake of illustration let us discuss a simulation interpretive routine that makes a *two*-address *non-floating* operation computer appear like a *one*-address *floating* operation computer. To be specific, suppose that there are 6 bits in the operation and 9 bits in each of the two addresses of the real computer. Suppose also that in the simulated computer there are to be three octal symbols for the operation and five octal symbols for the address. Of course, since the real machine can address only 2^9 memory cells, even though theoretically 2^{15} memory cells can be addressed by the simulated one-address instruction, only 2^9 memory cells are available. Hence three octal positions in the simulated-instruction format are meaningless. However, one could interpret these as addressing cells on magnetic tape, etc. In so far as the simulated numbers are concerned, the rightmost two octal positions of a word will represent the exponent, the rest of the positions the value of the number. These formats are then as follows, where each wide box represents an octal digit (i.e., three binary bits):



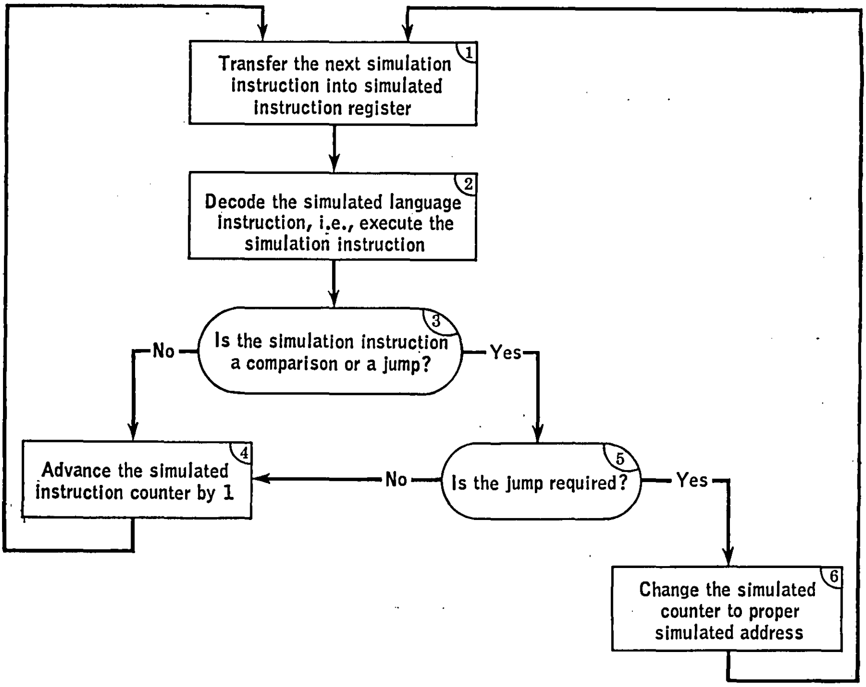
Let us suppose that the real computer has the two-address operations listed in Sec. 3-8, as well as the following additional instructions:

Code	*Description of operation*
30	Shift (acc) to left α positions; put result into β
31	Shift (acc) to right α positions; put result into β
20	Extract (α) into β, using (acc) as the mask; i.e., replace those bits of (β) with corresponding bits of (α) in those positions where (acc) has units, and put result into β

Suppose that the one-address instructions to be simulated are:

Code	*Description of operation*
001	Add (α) to (acc), put result in acc, floating unnormalized
002	Multiply (α) by (acc), put result in acc, floating unnormalized
003	Subtract (α) from (acc), put result in acc, floating unnormalized
004	Divide (acc) by (α), put result in acc, floating unnormalized
005	Conditional jump: if (acc) is negative, take (α) as next instruction
006	Transfer (acc) into α
007	Jump: take (α) as next instruction

Of course the accumulator referred to in these instructions is the simulated accumulator. To be more specific, let address 776 be the simulated-instruction address counter, 775 the simulated-instruction register, 774 the simulated accumulator.



1
Transfer the next simulation instruction into simulated instruction register
2
Decode the simulated language instruction, i.e., execute the simulation instruction
3
Is the simulation instruction a comparison or a jump?
No
Yes
4
Advance the simulated instruction counter by 1
5
Is the jump required?
No
Yes
6
Change the simulated counter to proper simulated address

FIG. 5-4. General flow diagram indicating how interpretive routine sequences the computer through the simulation-language code.

The programmer will now code in terms of these simulated instructions only (*never* writing down any of the real instructions). It is up to the interpretive routine to sequence through these simulation-language instructions, interpreting them as it goes along. Obviously the interpretive routine is written in the real machine language. Figure 5-4 is a general flow diagram showing how the interpretive routine sequences the computer through the simulation-language instruction code. The student should study it carefully. We shall first discuss some of the details of box 1 of this flow diagram.

The heart of box 1 is a pair of instructions that transfer the proper simulation-language instruction from its location in the code to the simulated-instruction register. Preceding this pair of instructions are instructions that set up this pair to operate on the proper addresses. In symbolic code, the pair might look like this:

Operation	α	β
Add, i.e., 53	Address of a constant of all zeros, i.e., ϕ	Address of proper simulation-language instruction
Transfer, i.e., 52	Simulated-instruction register, i.e., 775	Address of first instruction of box 2

The address of the proper simulation-language instruction is found in the simulated-instruction address counter. Hence the contents of the β part of this word (i.e., the contents of 776) must first be extracted into the β part of our add instructions:

Operation	α	β	Remarks
Add, i.e., 53	ϕ	Address of a constant that has all units in β position and zeros elsewhere	Set up mask
Extract, i.e., 20	776	Address of above *add* instruction	Do extraction

Suppose that we put the first instruction of box 1 into address 600; then we would have altogether for box 1:

Address	Operation	α	β	Remarks
600	53	604	605	Set up mask
601	20	776	602	Do extraction
602	53	604	. . .	Transfer
603	52	775	606	
604	00	000	000	Zero constant, i.e., ϕ
605	00	000	777	Mask

Each of the simulation-language operations is performed, of course, by a subroutine contained as part of the interpreter routine. Hence box 2 consists of these subroutines, in addition to a jumping table that looks at the simulation-language operation and jumps to the proper subroutine. The *shift* instruction would be used in this procedure. The simulation-language instruction is shifted to the right 17 (*octal*) positions [15 (*decimal*)], putting the operation code in the β position, leaving zeros elsewhere. Then this shifted word is added to the *transfer* instruction that will jump to the proper *jump* instruction of the jump table. In this way the simulated operation code is used to enter the proper subroutine from the jump table. Note that first the *transfer* instruction that jumps to the table must be reset, for it becomes modified each time it is used. Hence, in the first part of box 2, we have:

Address	Operation	α	β	Remarks
606	53	604	610	Put jump constant into accumulator Reset *jump* instruction
607	52	614	611	Put constant into 614
610	52	777	614	Constant used to reset *jump* instruction
611	53	604	775	Shift simulated-language instruction to right 17 (*octal*) positions in accumulator
612	31	017	777	
613	51	614	614	Add shifted instruction to *jump* instruction
614	52	777	614	*Jump* instruction
615	52	777	Add subroutine	Jump table that sends control to proper simulated operation subroutine
616	52	777	Multiply subroutine	
617	52	777	Subtract subroutine	
620	52	777	Divide subroutine	
621	52	777	Conditional jump subroutine	
622	52	777	Transfer subroutine	
623	52	777	Jump subroutine	

When a 52 or *transfer* instruction is used as a jump, the α address does not have much meaning. Hence we always put 777, the address of the real accumulator, for α to ensure no change due to this part of the *jump* instruction.

Let us examine briefly the subroutine for the simulation-language multiplication, floating unnormalized, as an example of our arithmetic subroutines. The purpose of this subroutine is to multiply (α) by (774), where α is the address found in the contents of 775, the simulated-instruction register (see the instruction and number formats for the simulated computer above). The result is to be put into 774, the simulated accumulator. In order to perform this multiplication, $P7$ through $P24$ of each word are to be multiplied, while $P1$ through $P6$ of each word are extracted into initially empty temporaries, the temporaries are added, and the result extracted into $P1$ through $P6$ of 774. In performing the addition, $P7$ of the sum must be checked for overflow: the exponent of the result cannot be larger than six positions. After performing such a subroutine the computer would go to box 3 of the general flow chart.

EXERCISES

(*a*) Write out the subroutine for the multiplication, floating unnormalized, in detail, letting 500 be the address of the first instruction.

(*b*) Write out a subroutine for the divide, floating unnormalized.

(*c*) Write out the codes for boxes 3 to 6 of the flow chart of Fig. 5-4.

5-6. Memory Space, Speed of Computation, and Automatic Programming

Sharing of the High-speed Memory. As we have seen above, a computer can have a high-speed memory (often magnetic cores), a medium-speed memory (magnetic drums), and finally, a slow-speed memory (magnetic tapes). As one might guess from our previous discussions, the high-speed memory may often be filled when an interpretive routine is stored in addition to a program. Hence careful attention must be paid to the location of the interpretive routine so that it will not interfere with memory space required for the program. It may often happen that the interpretive routine has to be stored on the drum. This may slow up the rate of computation because, as will be recalled, in order to use a word from the drum one must wait for the drum to turn to the proper address before a word can be read out.

This problem may be approached as follows: At the time a particular subroutine of the interpretive routine is to be used, this subroutine is transferred out of the drum into the high-speed memory, where the computations take place. In this way subroutines of the interpretive routines will be performed in the high-speed memory even though they are usually stored in the drum. However, this means that subroutines must be continually transferred out of the drum and into the high-speed memory and back into the drum, which takes precious time. Also, we have added a new requirement for the interpretive routine to perform. Now it must not only interpret the simulated instructions but must also *transfer* from the drum to the high-speed memory the subroutines of the interpretive routine as they are needed, and then transfer the subroutine back to the drum.

Complications. This problem becomes quite complicated when one is desirous of putting some flexibility into the procedure; for, in general, when transferring a subroutine from the drum to the high-speed memory it is best to be able to put the subroutine in that part of the high-speed memory which may not be in use at the moment. Hence it must be determined where in the high-speed memory to transfer the subroutine. This means that a subroutine may be located in different places when used at different times. Obviously the assignment of addresses often depends upon the exact location of the instructions being performed. Hence the interpretive routine must be able to translate this subroutine and fix up the addresses as required, depending upon the location of the subroutine. We shall discuss this problem in the next section in more detail.

The complexity of an interpretive routine can be increased still further. Observe, for example, that a function subroutine may be used repeatedly in the course of computation and hence it might be desired to leave it in the high-speed memory after it has been transferred there. Other subroutines that are transferred to the high-speed memory may have to be used only once in that program and hence after their use need not take up room in the high-speed memory. Therefore a distinction must be made as to subroutines that should be transferred temporarily. Further, during the computation for a problem, one subroutine may use another subroutine. Suppose that both these subroutines have already been transferred to the high-speed memory. Somehow the interpretive routine should remember not only that these subroutines have already been transferred to the high-speed memory so that the transfer will not have to be repeated, but also where they have been put. The solutions to these problems are too complex to be treated in this textbook. Our purpose has been to give the student an appreciation of the complexities that can develop in planning and writing interpretive routines.

Advantages of the Compiler Routine. Until now we have been considering the use of subroutines as the interpretation of simulation instructions *during* the running or computation of a program. An alternative method is initially to replace the simulation instructions with the proper subroutines before the program is run. That is, the program is written in terms of simulation instructions, but as the program is read into the computer, these simulation instructions are automatically replaced by the proper subroutines—the subroutines then becoming an integral part of the program. Of course, just as the interpretive routine handled the subroutines in our above discussions, so in this new case an automatic routine called a *compiler routine* performs the task of initially inserting the proper routines into the program. Examples of compiler routines will be described in the next section; in this section we wish only to discuss why such routines are used.

The first advantage of a compiler routine over the interpretive routine is that, since in the compiler routine the subroutines are initially amalgamated into the program before the problem is run, the necessity for transferring subroutines from drum to high-speed memory is eliminated. Hence the problem of *time* lost in transferring subroutines during computation no longer appears, because the subroutines will already have been made part of the main program. Another important advantage of the compiler routine over the interpreter routine is that, since the compiler routine is only used initially, it need not be stored in the computer when the compiled program is running. The interpreter routine must always remain in the computer memory since it operates during the running of the program. Hence, when a compiled routine is run, more *space* is available for the main program.

On the other hand, as can immediately be seen, after all the required subroutines have been incorporated into the main program, this program can be exceedingly long. Some of the *space* advantage of a compiler routine is thereby lost. The usual method for handling such long programs is to divide them up into sections, each of which can fit entirely into the high-speed memory. Then one section at a time is put into the high-speed memory and computed. Meanwhile the other sections are stored in the drum and on the magnetic tapes. Of course a relatively great amount of time is spent in changing sections from drum and magnetic tape to high-speed memory, and hence some of the *timesaving* advantages of the compiler routine are lost. The relative advantages of a compiler routine over an interpretive routine, if any, depend on the particular characteristics of the routines in question as well as on those of the program to which the routines may be applied. Further, compiler routines can become very complicated, for the compiler routine must break the program into the proper sections and assign the sections to parts of the drum or magnetic tapes. It must also form instructions additional to the main routine which will change sections at the proper time. Another problem arises when a subroutine appears in a program many times. For example, suppose that the program uses the sine routine often. It would be wasting space to compile the sin x routine each time it is required, having duplicates of this sin x routine incorporated into the program. It would be much wiser to have only one sine routine in the memory and to use it in the manner of a usual subroutine. If the program is compiled as it is read in, the compiling routine must sense that the sin x subroutine will be used many times and so must arrange to have it as a subroutine. Of course, if we have too many subroutines, so many that we have to put some in the drum, then we are in the dilemma described for the interpretive routine.

In the previous section we have discussed interpretive routines that enable a program to utilize subroutines while it is being computed. We have seen that owing to lack of high-speed-memory space this can be a slow process. The alternative, namely, that of compiling a program as it is read in, is considered in the following section. We must always keep in mind that the major object in using either interpretive or compiling routines is to reduce the amount of coding that the programmer must do.

5-7. Compiling Routines: Translator; Address Assigner

Kinds of Compiler Routines. The purpose of a compiling routine is to reduce the amount of necessary coding by allowing the programmer to write simulated-language instructions which are interpreted by the compiling routine. The difference between interpretive and compiling routines is that a compiling routine can be erased from the high-speed memory after it has completed its task. Evidently a compiling routine can compile mathematical subroutines as well as interpret and compile simulation subroutines of the kind described in Sec. 5-5. However, there are other uses of compiling routines that lend themselves more to the technique, and this section and the one following are devoted to these. We shall consider four such uses: (1) as a *translator*, (2) as an *address assigner*, (3) as a *decoder of algebraic symbols*, and (4) as a *decoder of recursion, or loop, symbols.*

Translator. Suppose that we are considering a decimal-coded binary computer for which we have written a symbolic code. Ordinarily specific addresses would have to be assigned before the symbolic code were put into the computer. However, it is possible to construct a compiling routine that enables the symbolic code itself to be read into the computer, the addresses being assigned automatically by the compiler. Such a compiler is called a translator, since it translates from symbolic language into computer language. In these discussions the student should keep in mind that two codes are under consideration: the compiler routine itself and some program upon which the compiler operates.

To be specific, let us consider a one-address decimal-coded binary instruction format having two characters for the operation and three for the address. Suppose, for simplicity in illustration, that the program is to add two numbers, 8,818 and 7,080. The symbolic code for this might be:

Address	Instruction		Remarks
	Operation	α	
CLR	MT	ZRO	Clear accumulator
A1C	AD	1ST	Add 1st number to accumulator
A2C	AD	2ND	Add 2d number to accumulator
TRR	TR	SUM	Transfer result into sum temporary
1ST	08	818	1st number
2ND	07	080	2d number
ZRO	00	000	0 constant
SUM	. . .	. . .	Result: sum temporary

If the compiler routine is to assign addresses to these instructions and numbers, then, in addition to the instructions themselves, in symbolic form, the *symbolic address* corresponding to each instruction, constant, or temporary must also be read into the computer. We might make the convention that the symbolic address will be read in immediately preceding its associated symbolic contents. For example, for our code we would initially read the following into the computer:

```
     CLR ⎫
MT   ZRO ⎪
     A1C ⎪
AD   1ST ⎪
         ⎬ Instructions
     A2C ⎪
AD   2ND ⎪
     TRR ⎪
TR   SUM ⎭

     1ST ⎫
08   818 ⎪
     2ND ⎪
07   080 ⎪
         ⎬ Constants and temporaries
     ZRO ⎪
00   000 ⎪
     SUM ⎪
00   000 ⎭
```

As yet there is no way for the compiler to distinguish between instructions and constants or temporaries. If the convention were made that only instructions will appear first and that all the constants and temporaries are to be put at the end, and if, in addition, the number of instructions in the program were to be given to the compiler, then the compiler would be able to distinguish the instructions from the constants and temporaries. Of course the desired first address of the program must also be given to the compiler. Hence one word would precede the symbolic code, with the convention that the number of instructions appears in the operation positions and the first address appears in the α positions. For our example this first word would be

04 600

if we desired the code to begin at address 600. Of course the symbolic code with its first word must be read into a standard position in the computer memory so that the compiler routine will know where to work. Suppose that the convention is made that the symbolic code is to be read into the memory starting with the first word in address 001.

Writing the Translator Compiler Routine. It now remains to discuss how to form a compiler routine that will assign specific addresses to a code when all the above-described conventions are adhered to. A general flow diagram for such a compiler appears in Fig. 5-5.

Consider box 1 of Fig. 5-5. First the real addresses would be assigned to the symbolic addresses of the instructions. By our conventions it is known that these symbolic addresses are found in every other word of the symbolic code, starting with the second word. The first symbolic address is assigned to the address found in the α position of the first word of the code. Next the assignment of real addresses for the constants and temporaries is made. This is done by looking at the contents of each instruction: if the symbolic address (in its α position) is not that of another instruction, then it must be that of a constant or temporary. Such addresses are then assigned in sequence. (Note that we cannot have the compiler assign addresses to the constants and temporaries in the same manner as it did for the instructions since we have not informed the compiler of how many constants and temporaries there are.) A detailed flow diagram of box 1 appears in Fig. 5-6.

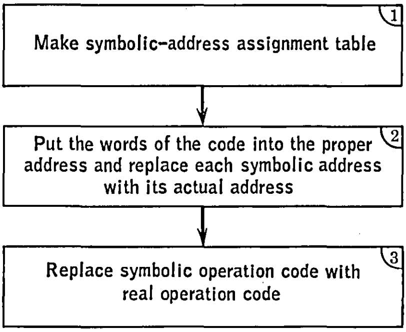

1
Make symbolic-address assignment table
2
Put the words of the code into the proper address and replace each symbolic address with its actual address
3
Replace symbolic operation code with real operation code

FIG. 5-5. General flow chart of a compiler routine.

Now that a table has been computed assigning real addresses for the symbolic addresses, the instructions and constants can be transferred into these locations, with the α positions of the instructions changed from the symbolic to the real address. Our convention of preceding each instruction and constant by its symbolic address again plays an important role. When the code is compiled into its real position, the instructions and constants will appear in the usual consecutive sequence.

Finally we must replace the symbolic operation code with the real operation code. For this the compiler routine must be given a table relating these operation codes.

Several further details must be observed. There are some instructions in which *no* address appears in the α positions, e.g., *shift* instructions where the number of positions to be shifted appears instead. Care must be taken that this is not interpreted as a symbolic address to be assigned a real corresponding address. Hence the instruction itself must be examined by the compiler to ensure that it is not one of these exceptional cases. Also, some constants may involve a symbolic address. Care must be taken to recognize these and replace them with the proper real address.

Summarizing, we can draw the following conclusions from our example: First, the writing of a compiling routine can be "tricky" if we are to be certain that all possibilities are taken care of. Second, in order to use a compiler routine, the programmer must adhere carefully to all conventions required by the routine.

Address Assigner. Suppose that it is desired to write many subroutines and it is not known beforehand where they are to be located in the memory. One procedure might be to write each subroutine as if it were to be located with its first instruction in some standard address, say 003, and then later to use a compiler to assign new addresses to each subroutine when the final decision is made as to where it shall be located. (Recall that a similar requirement arose in connection with the transferring of subroutines from the drum to various locations in the high-speed memory.) To perform this task, the compiler routine merely has to add *to each address* appearing in the code the difference between the desired address of the first instruction and 003. The only problem is to determine which are the addresses. For some instructions such as *shift* instructions do not contain addresses in all the address positions, and these positions should not be changed. On the other hand some constants may contain addresses, and these must be recognized and changed. One procedure is to require the programmer to list all the instructions first, then any constants that contain an address, finally any other

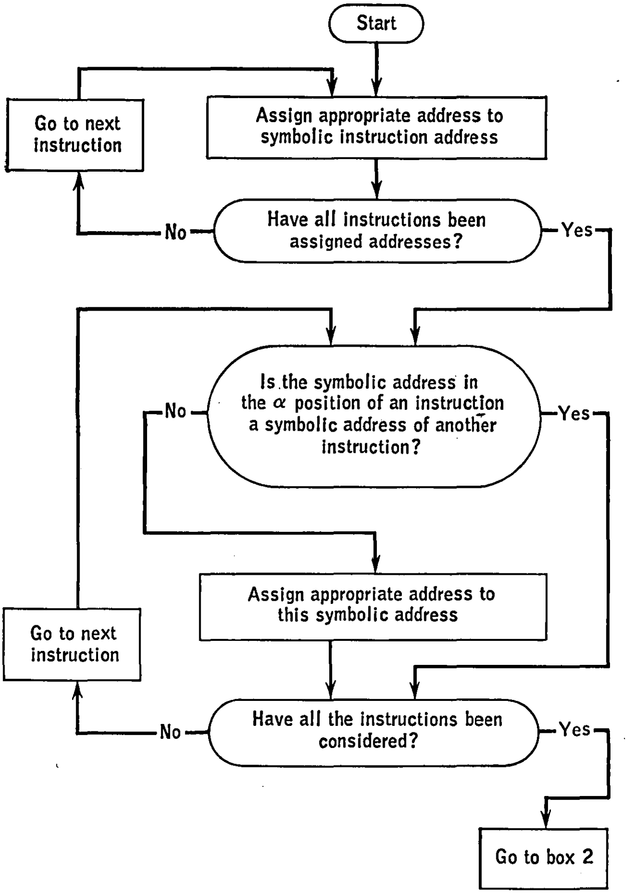

Start
Go to next instruction
Assign appropriate address to symbolic instruction address
No
Have all instructions been assigned addresses?
Yes
No
Is the symbolic address in the α position of an instruction a symbolic address of another instruction?
Yes
Assign appropriate address to this symbolic address
Go to next instruction
No
Have all the instructions been considered?
Yes
Go to box 2

FIG. 5-6. Detailed flow chart of box 1 of Fig. 5-5.

constants. In addition the programmer is required to insert words in front of the program that list (1) the desired address of the first instruction, (2) the total number of words in the code, (3) the total number of instructions in the code, and (4) the total number of words other than instructions that contain addresses. Suppose that we were considering a two-address system; we could then put these numbers, respectively, into the α and β positions of the word in 001 and the α and β positions of the word in 002.

The compiler routine will first determine which instructions contain addresses and will add to these addresses appropriately. Next the program will look in the α and β positions of the constants that have addresses—recognizing these addresses as numbers between 003 and 003 + the total number of words in the code—and will add to these addresses appropriately. Finally the compiler routine will transfer the code from the standard position to the required position.

EXERCISES

(*a*) Write the specific code for the flow diagram of Fig. 5-6.

(*b*) Write the code for box 2 of Fig. 5-5.

(*c*) Draw a detailed flow diagram for the address-assigner compiler routine described in this section.

5-8. Compiling Routines: Algebraic

Decoder of Algebraic Symbols. The purpose of this compiling routine is to enable the programmer merely to type into the computer an algebraic formula, the compiling routine automatically writing the detailed machine language code. Of course the programmer must also tell the compiling routine in what addresses the values of the independent variables will be found and into which address the result is to be placed. Let us consider first a function involving only the operations addition and multiplication, such as $((((a + b) \cdot c) + d) \cdot e)$. In order to avoid ambiguity, certain conventions must be observed: A dot is to be used for multiplication; i.e., we must write $a \cdot b$ and not ab. Also the parentheses must be placed wherever necessary and never left understood. For example, $a + b \cdot c$ is never written, but rather $(a + (b \cdot c))$. There is to be a parenthesis at the beginning and at the end of the function. Also the parentheses are to indicate binary operations only—i.e., operations involving two operands and no more. Thus $a + b + c$ would never be written, but rather $((a + b) + c)$—or $(a + (b + c))$; it makes no difference. Similarly, $a \cdot b \cdot c$ must be written as $((a \cdot b) \cdot c)$.

Evidently the parentheses play an important role in such a formula—for they tell in what order operations are to be performed. For example, in $((((a + b) \cdot c) + d) \cdot e)$ the order in which the operations must be performed is $a + b = w$; $w \cdot c = x$; $x + d = y$; $y \cdot e = z$, where z is the final result. A study of the proper use of parentheses shows that they are never ambiguous.† To see which operations should be done first, number the parentheses, say from left to right. Successive opening parentheses get successively larger numbers from left to right. A closing parenthesis directly facing an opening parenthesis is given the same number as that opening parenthesis. Other closing parentheses are given the next successive lower numbers that have not previously been used on a closing parenthesis.

Consider the following illustrations:

$$(1)\qquad \underset{1}{(}\underset{2}{(}\underset{3}{(}\underset{4}{(}a + b\underset{4}{)} \cdot c\underset{3}{)} + d\underset{2}{)} \cdot e\underset{1}{)}$$

$$(2)\qquad \underset{1}{(}\underset{2}{(}a \cdot \underset{3}{(}b + c\underset{3}{)}\underset{2}{)} + \underset{4}{(}d \cdot \underset{5}{(}e + f\underset{5}{)}\underset{4}{)}\underset{1}{)}$$

$$(3)\qquad \underset{1}{(}\underset{2}{(}\underset{3}{(}\underset{4}{(}a \cdot b\underset{4}{)} + \underset{5}{(}c \cdot d\underset{5}{)}\underset{3}{)} \cdot e\underset{2}{)} + \underset{6}{(}\underset{7}{(}\underset{8}{(}g + h\underset{8}{)} + \underset{9}{(}k + l\underset{9}{)}\underset{7}{)} \cdot m\underset{6}{)}\underset{1}{)}$$

Notice that a pair of opening and closing parentheses turns out to have the same number, which was of course the purpose for the above numbering rules. Also there are the same number of pairs of parentheses as there are + and · operations in the expression.

Now we have gained some insight into how a compiling routine might work to code an algebraic function automatically. The compiling routine would number the parentheses according to the above rule and would then proceed to write instructions to perform the operations between directly facing opening and closing parentheses which will have the same number: i.e., in illustration (1), to compute $\underset{4}{(}a + b\underset{4}{)}$; in illustration (2), to compute $\underset{3}{(}b + c\underset{3}{)}$ and $\underset{5}{(}e + f\underset{5}{)}$; in illustration (3), to compute $\underset{4}{(}a \cdot b\underset{4}{)}$, $\underset{5}{(}c \cdot d\underset{5}{)}$, $\underset{8}{(}g + h\underset{8}{)}$, and $\underset{9}{(}k + l\underset{9}{)}$. Each of these sets of symbols would then be replaced by the result of the operation: i.e., in illustration (1) let $s = a + b$; then (1) would read $\underset{1}{(}\underset{2}{(}\underset{3}{(}s \cdot c\underset{3}{)} + d\underset{2}{)} \cdot e\underset{1}{)}$. In (2) let $t = b + c$ and $u = e + f$; then (2) would read $\underset{1}{(}\underset{2}{(}a \cdot t\underset{2}{)} + \underset{4}{(}d \cdot u\underset{4}{)}\underset{1}{)}$. In (3) let $v = a \cdot b$, $w = c \cdot d$, $x = g + h$, and $y = k + l$; then (3) would read $\underset{1}{(}\underset{2}{(}\underset{3}{(}v + w\underset{3}{)} \cdot e\underset{2}{)} + \underset{6}{(}\underset{7}{(}x + y\underset{7}{)} \cdot m\underset{6}{)}\underset{1}{)}$. The computations between the new facing pairs is carried out. Thus in (1) instructions to compute $\underset{3}{(}s \cdot c\underset{3}{)}$ would be written; in (2) instructions to compute $\underset{2}{(}a \cdot t\underset{2}{)}$ and $\underset{4}{(}d \cdot u\underset{4}{)}$ would be written; and in (3) instructions to compute $\underset{3}{(}v + w\underset{3}{)}$ and $\underset{7}{(}x + y\underset{7}{)}$ would be written. In this way the code for systematically evaluating the function would be constructed.

Writing the Algebraic Compiler. To describe possible additional details of such a compiling routine, suppose that we consider a three-address system with 12 bits in each address and 6 bits for the operation. Hence

† See S. C. Kleene, "Introduction to Metamathematics," p. 24, D. Van Nostrand Company, Inc., Princeton, N.J., 1952.

words will be 42 bits, plus a sign bit, long. Now suppose that we arrange for the paper-tape punch machine to punch a 7-bit code to distinguish among letters of the alphabet and the symbols (,), $\cdot$, and $+$. That is, when a key corresponding to a letter of the alphabet or a symbol is pushed, a unique 7-bit code will be punched on the paper tape. Then successive words will contain the formula; i.e., for illustration (1) we shall need $17 \times 7 = 119$ bits, or 2 and $\frac{5}{6}$ words. Let us suppose that the first two bits of the 7-bit code tell whether or not this is the code for a parenthesis, operation, or letter; for example, 10_____ is a parenthesis, 01_____ is an operation, 11_____ is a letter. In particular let (be 1000000 and) be 1010000. Then, when the compiling routine counts parentheses, the count can be put into the rightmost four position, for example, 1000101 would be $\underset{5}{(}$, and 1010101 would be $\underset{5}{)}$. (Of course this limits us to 16 pairs of parentheses, but these suffice for our example.) Also we could have up to 32 different kinds of operations. Let $+$ be 0100001 and $\cdot$ be 0100010. Let the letters of the alphabet be denoted by their position in the alphabet, that is, a is 1100001, b is 1100010, c is 1100011, . . . , z is 1111010. For example, illustration (1) would look as follows after being punched and read into the computer:

```
100 0000   100 0000   100 0000   100 0000   110 0001   010 0001
110 0010   101 0000   010 0010   110 0011   101 0000   010 0001
110 0100   101 0000   010 0010   110 0101   101 0000   000 0000
```

where the last seven bits of the third word are filled in as all zeros and it is to be understood that seven zeros have no meaning for a formula.

First the compiling routine will number the parentheses as described above. The result will be

```
100 0001   100 0010   100 0011   100 0100   110 0001   010 0001
110 0010   101 0100   010 0010   110 0011   101 0011   010 0001
110 0100   101 0010   010 0010   110 0101   101 0001   000 0000
```

Next the compiling routine will search these words to determine which instructions to write first, as described above. After doing this, it will assign another letter to the combination and put all zeros in the 7-bit groups no longer needed. In illustration (1), as we have seen, instructions to compute $\underset{4}{(}a + b\underset{4}{)}$ would be written, and then the substitution $s = \underset{4}{(}a + b\underset{4}{)}$ would be made, resulting in the following for our three words:

```
100 0001   100 0010   100 0011   000 0000   111 0011   000 0000
000 0000   000 0000   010 0010   110 0011   101 0011   010 0001
110 0100   101 0010   010 0010   110 0101   101 0001   000 0000
```

where it is to be recalled that seven zeros are to be ignored. With these new numbers the same thing is repeated, etc. The flow diagram therefore becomes as in Fig. 5-7.

It can easily be seen how other operations such as division and subtraction can be included in the repertoire of the compiling routine. Note that five groups of memory addresses are involved in this compiling routine:

1. The locations of the compiling routine itself
2. The locations of the words punched from the symbols
3. The location of the table relating the letters of the function to the addresses that will contain the values of these letters as independent variables
4. The location of the values of the independent variables
5. The location of the program written by the compiling routine

When the compiling routine is writing the function-evaluation program, (1) to (5) are needed; when the resulting program is being run, only (4) and (5) are needed.

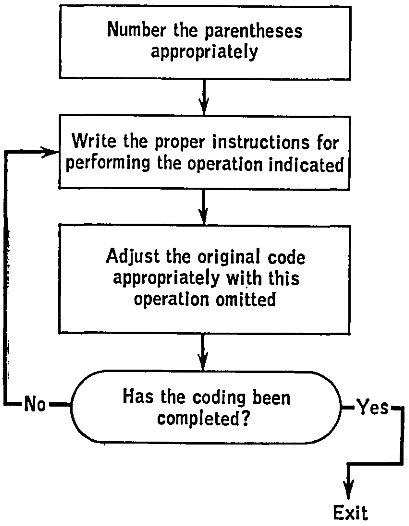

Number the parentheses appropriately
Write the proper instructions for performing the operation indicated
Adjust the original code appropriately with this operation omitted
Has the coding been completed?
No
Yes
Exit

FIG. 5-7. Flow chart of algebraic-compiler routine.

Recursive Compiler. Consider the problem of constructing a compiler program that will automatically code loops. This problem can become very complicated, depending on the generality with which it is desired to have the program perform.† However, to illustrate the concept, let us consider the following oversimplified example. Suppose that we desire to construct a compiler that will evaluate a single function by successive substitutions—in other words, one that will automatically construct a code to perform the flow diagram of Fig. 5-8. Here let us assume that the function $f(x)$ is given by a subroutine: then the only information the compiler needs is the number of the subroutine for $f(x)$, the initial value of x, namely, x_0, and the accuracy ϵ for which the function is to be evaluated. Of course in this oversimplified case we are tacitly assuming that $f(x)$ has certain properties. (Why?)

One very simple way of constructing such a compiler is to write a code as indicated by the flow diagram of Fig. 5-8. Then all the compiler need do is interpret the form in which a particular $f(x)$, x_0, and ϵ are introduced into the computer and put these values into the proper addresses of the code. Of course conventions must be made for the form of the particular $f(x)$, x_0, and ϵ, for the exit, and for the final location of the compiled code.

Simple extensions of such a compiler might enable it to compile loops

† For example, see Edward K. Blum, Automatic Digital Encoding System II, *NAVORD Repts.* 4209 and 4411.

within loops, where now some notation must be introduced to tell the compiler which loops are within which loops. We may make the compiler more versatile by enabling it to form partial products or partial sums as well as successive iterations. Again perhaps more general functions than these can be introduced. The whole thing could be integrated and amalgamated with the algebraic compiler to act as one large compiler, and so forth.

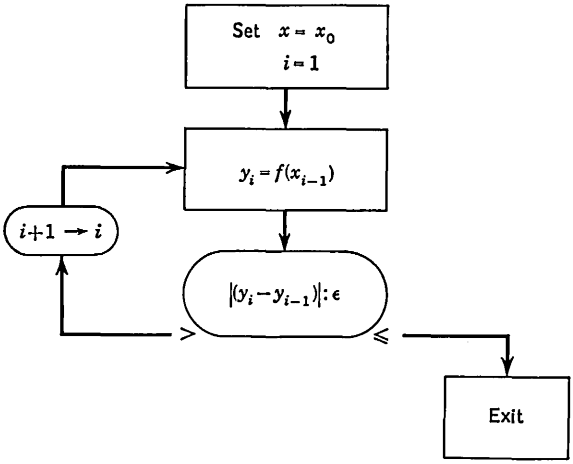


FIG. 5-8. Flow chart for the evaluation of a single function.

EXERCISES

(*a*) To use the algebraic-compiler routine, a word should precede the code for the symbols of the function. What information do you think the compiling routine should obtain from this word?

(*b*) Write a subroutine, using a three-address system, that would be contained in the algebraic compiler to number the parentheses.

(*c*) Write a program for the simplified recursive compiler.

5-9. The International Algebraic Language (ALGOL)

Languages. The purpose of automatic programming is to make the computer itself help prepare the program or code, thereby minimizing the amount of writing the programmer must do. In previous sections we have studied how the computer can be prepared or precoded with various automatic programs to aid the use of subroutine, to translate symbolic codes, to transfer parts of a code, to decode algebraic symbols, and to compile a loop, or recursion code. There were essentially two parts to each of these discussions: the part that told *how to use* the automatic program in writing a code, and the part that described how the automatic program itself could be constructed. We have been building up to the idea of a general *automatic language system,* where the procedures for *using* a combination of such automatic programs are unified and standardized (i.e., the *language*), and where the collection of such automatic programs themselves are made compatible with each other (i.e., the *system*).

Two processes are involved: First unified language itself (i.e., the procedures for using the automatic programs) must be established. Second, for the particular computer involved, the system of compatible automatic programs themselves must be constructed to enable the computer to interpret codes written in the language and compile the actual instructions to be executed. The principles involved in the latter process have already been considered in previous sections (see Exercises *a* and *b* below) and therefore are not considered here. We shall limit our discussion in this section to the former process and illustrate the principles involved in terms of a certain unified language.

With the combinations of methods presented in previous sections it is clear that *a language can be devised that is independent of particular computer instructions.* Thus a code can be written in terms of the *language alone*, and specific computer instructions need never be mentioned. If automatic programs were written for different computers to interpret the same language, then any code written in the language could be run equally well on any of the different computers. This fact indicates the feasibility of the important concept of establishing an *international automatic* language that may be universally accepted, analogous to the universal language of written music. Toward this goal the Association for Computing Machinery, representing American computer users, and the Association for Applied Mathematics and Mechanics, representing European computer users, sent representatives to Zurich, Switzerland, in May, 1958, to define a generally acceptable automatic language. The language agreed upon by this meeting has been named *ALGOL*, for "*algo*rithmic *l*anguage." In the remainder of this section we shall present a simplified version of ALGOL. (See also Sec. 5-10, Additional Topics.)

Anatomy of a Language. ALGOL and the simplification to be described are called *algebraic* languages. The term *algebraic* refers to the fact that they are intended primarily for algebraic computations. In general the requirements for a language are similar to the requirements for a specific instruction system: it must be able to tell a computer how to compute a function, to jump out of a normal sequence of computations, to decide between alternatives, to compute loops, or recursion processes, and to use subroutines.

Three syntactical entities are distinguished: *expressions, statements,* and *declarations. The statement is the operational unit* that tells the computer what to do and is analogous to a generalized instruction. Statements are composed of *expressions* combined with *delimiters* (i.e., punctuation marks, etc.), where expressions are numbers, names, labels (i.e., symbolic addresses), etc. Declarations are composed as are statements but are not operational, i.e., they tell the automatic programs certain facts about the entities referred to within the program, such as whether a variable is floating or a fixed point, the size of an array, the definition of a function, or what variables are required by a subroutine.

In order to illustrate how our language works, consider the following simple example of a code to compute the roots of $Ax^2 + Bx + C = 0$ for $A = +3, B = +1.7$, and $C = -0.31$. One of the two roots is given by

$$x = \frac{-B + \sqrt{B^2 - 4AC}}{2A}$$

Then the code becomes

```
; A:=3; B:=1.7; C:=−0.31;
                        root:=(−B+sqrt(B↑2↓−4×A×C))/(2×A); stop;
```

The meaning of this jumble of symbols becomes more or less obvious with the following explanation: First the symbols between semicolons (; . . . ;) *are statements*. This is analogous to putting instructions, constants, etc., on the separate lines of a coding sheet or to placing English sentences between periods. Statements are normally executed in sequence. The first three statements record the desired values of A, B, and C; the fourth statement computes the root; and the last statement stops the routine. In our example the expressions are **A**, **3**, **B**, **1.7**, **C**, **−0.31**, ***root***, **2**, **4**, **C**, ***sqrt*(B↑2↓−4×A×C)**, and ***stop***. The other symbols, namely, **:=**, **−**, **+**, **×**, **/**, **↑**, **↓**, **(**, **)**, and the semicolon **;**, are the delimiters. The automatic program that is to interpret this language compiles the appropriate instructions, and in case of the square root (***sqrt***) refers to the appropriate subroutine in the library.

In the following paragraphs we shall describe the detailed anatomy of our language in more formal terms.

Expressions. We shall distinguish five types of expressions: (1) *numbers*, (2) *simple variables*, (3) *subscripted variables*, (4) *functions*, and (5) *arithmetic expressions*.

1. If a represents a digit, that is, 0, 1, . . . , 9, then a *number* can be an integer $aa \cdots a$, a fraction $.aa \cdots a$, or a scale factor ${}_{10}\pm aa \cdots a$, or any combination of these. For example, **6**, **4,711**, **137.06**, **$2.9997_{10}12$** ($= 2.9997 \times 10^{12}$), **${}_{10}-12$**, and **$3_{10}-12$** are numbers. In this language a number can only be the contents of an address, and not the address itself.

2. *Simple variables* are designations for numbers or scalar quantities, as used in ordinary algebra. A little reflection will show that a *simple variable is essentially an address whose contents is the numerical value of the variable.* Simple variables may be represented by several adjacent characters *the first of which must be a letter, not a number* (so that numbers and simple variables may be distinguished). For example, **b**, **P63**, **QR82T**, **ALPHA** are all simple variables.

3. *Subscripted variables* designate the *addresses* of quantities that are components of multidimensional arrays, such as, for example, the com-

ponents of a list, a vector, or a matrix, etc. What would normally be the subscripts are placed in brackets. Thus a vector V_i would be written as **VECTOR[i]**, a matrix $M_{i,j}$ as **MATRIX[i,j]**, and so forth. The "subscripts" themselves may appear as integral numbers, simple variables, subscripted variables, or arithmetic expressions (see below). Note that subscripts are intrinsically integer-valued, no matter in what form they appear. The comma distinguishes the different subscripts; e.g., a three-subscripted variable is **VAR[6,ALPHA,3×i+k]**.

4. *Functions* are denoted by the usual algebraic functional notation: The independent variables, which can be simple variables, subscripted variables, or other functions, are enclosed in parentheses and separated by commas, for example, **FUNCT(x,y,z)**, where **FUNCT** (that is, the function name) can be thought of as essentially the first address of a subroutine that can evaluate the function. The **(x,y,z)** is the ordered set of independent variables, the numerical values of which must be ascertained before evaluating the function. Certain function names should be reserved for the standard functions, such as ***abs*(N)** for the absolute value of a variable N, ***sign*(N)** for the sign (i.e., + or −) of the value N, ***sqrt*(N)** for the square root of N, ***sin*(θ)** for the sine of **θ**, and so forth.

5. *Arithmetic expressions* are combinations of simple variables, subscripted variables, or functions with the operators **+**, **−**, **×**, and **/**, which have the usual meaning of addition, subtraction, multiplication, and division, respectively. A pair of arrows ↑ and ↓ are used for exponentiation, i.e., raising to a power. For example, **2↑3↓** means 2^3, **2↑3↑4↓↓** means $2^{(3^4)}$, **2↑3↓↑4↓** means $(2^3)^4$, and so forth. Ordinary parentheses are evaluated as described in the preceding section, except that the following rule of precedence can be used: **×** or **/** before **+** or **−**. Thus **a+b×c** means **a+(b×c)**. However, **(a×b)/c** should be used rather than **a×b/c**. An example of the use of all these symbols is the arithmetic expression

(−B+*sqrt*(B↑2↓−4×A×C))/(2×A) for $\dfrac{-B+\sqrt{B^2-4AC}}{2A}$

Finally it is important to remember that an expression which is not an actual number really (eventually) represents an address or location of the numerical value *of the expression* and in the case of a function refers (eventually) to a subroutine that can compute a value of the function. We use the word *eventually* to denote that an expression can be the address of another expression which itself is the address of a number, and so forth.

Statements. Now that we have described the ingredients from which statements are made (i.e., expressions), let us consider the statements themselves. Since the statement is the operational unit, we must describe what each statement does, analogously to describing the meaning of operations in considering instructions. But first let us recall that statements are set apart by semicolons. Thus the statement **; S;** means

"Execute the computations called for by S." It is often desirable to give a statement a label or name, L, analogous to the address of an instruction. The label **L** may be a simple or subscripted variable. If **L** is to be the name of statement S, we write **;L:S;**. Thus **;L:S;** tells the computer to assign label **L** to the statement S and perform statement S. Then some other statement may refer to the labeled statement simply by means of its label **L**. The analogy to addresses is evident.

Occasionally it is desired to consider a *compound statement*, i.e., a statement that itself consists of statements. Such compound statements are set between the words ***begin*** and ***end***, e.g., ***begin*** S; S; S; S ***end*** is a compound statement of four statements. A compound statement can also be labeled, but the label must appear after the word *end* as well as before the word *begin*, for example, **;L: *begin*** S; S; S; S ***end*** **L;**.

Five types of statements (of a total of seven) can now be described:

1. The first type, called the *assignment* or substitution statement, is probably the most important, although it is almost the simplest. The form of the assignment statement is

; VAR:=EXPR;

where **VAR** represents a variable (simple or subscripted) and **EXPR** can be *any type of expression.* This is a statement telling the computer to compute the value of the right-hand side and store that value (i.e., a number) in the storage location designated by the left-hand side, i.e., to assign the value of the expression to the variable. The coloned equal sign **:=** is used to indicate that the left-hand side is to be the name (or address) of the value of the right-hand side. If the expression is in terms of numbers, such as **1.7↑2↓−4×3×(−0.31)**, then the indicated operations are performed. Thus **Y:=1.7↑2↓−4×3×(−0.31)** tells the computer to assign 2.89 + 3.72, or 6.61, as the value of Y. A statement such as **A:=3** of course requires no computation and merely assigns the value of 3 to the variable A. *If the expression contains variables and functions, then the computer obtains the values of the variables, which must have been previously assigned, and evaluates the functions, by means of appropriate subroutines.* Thus, to compute **(−B+*sqrt*(Y))/(2×A)**, where Y represents **B↑2↓−4×A×C** and A is 3, B is 1.7, and C is -0.31, we would write the following sequence of statements, which would be executed from left to right:

;A:=3; B:=1.7; C:=−0.31; Y:=B↑2↓−4×A×C;
***root*:=(−B+*sqrt*(Y))/(2×A);**

When the new value of a variable is a function of the previous value of the variable, the expression symbols appear on both sides of the statement. For example, suppose that we desired to add 2 to J. We would write **;J:=J+2;**, which really means J (new value) = J (previous value) + 2.

2. The *stop* statement is used to stop the computer when a code has been completely executed. See below for an example.

3. Normally a sequence of statements is executed in the order in which they appear from left to right. This normal sequence of execution can be interrupted by the use of a *go to* statement that is analogous to an unconditional *jump* instruction. The form of the *go to* statement is

; *go to* **L**;

where **L** is an expression specifying the *label* of the statement to be executed.

4. Another type of statement that can change the normal sequence of executing instructions is the *if* statement, of the form

; *if* **REL**;

where **REL** must be a "relational" expression that can be either *true* or *false*. For example, **REL** could be **(A>N)**, or **(A=B)**, or **(A≤B)**, and so forth. For such a statement the computer would first determine whether **REL** is true or false (i.e., whether it holds or not). If **REL** is true, the statement following the *if* statement will be executed; otherwise it will be skipped, and the statement following it will be executed next. As an example of both the *if* and *go to* statements, consider the problem of finding the largest of three given numbers *U*, *V*, and *W*. The sequence of statements according to Fig. 3-3 will be

; *if* **(U>V)**; *go to* **UW**; *if* **(V>W)**; *go to* **VLN**; **WLN:LNC:=W**;
go to **T**; **VLN:LNC:=V**; *go to* **T**; **UW**: *if* **(U>W)**; *go to* **ULN**;
go to **WLN**; **ULN:LNC:=U**; **T**:*stop*;

Here the largest number is to be assigned as the value of **LNC**. Five statements were labeled to be referenced by the *go to* statements, namely, **WLN**, **VLN**, and **ULN** (which assign *W*, *V*, and *U* to **LNC**, respectively), **UW** [*if* **(U>W)**], and **T**, where **T** is the *stop* statement. This is somewhat reminiscent of a one-address code.

5. The automatic compilation of loops, or recursion codes, is accomplished by the *for* statement. The *for* statement causes the next statement (which may be a compound statement) to be executed repeatedly, where the value of some variable is changed for each repetition. The successive values to be assigned to this changing, or recursive, variable are given in the *for* statement itself by the initial value **NUI**, the final value **NUF**, and the incremental change in the value **ΔN**, which should be successively added to the initial value until the final value is attained or exceeded. The form of the *for* statement is

; *for* **VAR:=NUI(ΔN)NUF**;

where **VAR** is a simple variable and **NUI**, **NUF**, and **ΔN** are numbers or simple variables. As an example, consider again the problem of finding the largest of three given (positive) numbers **N[1]**, **N[2]**, and **N[3]**.

; **LNC:=0**; *for* **J:=1(1)3**; *begin if* **(LNC<N[J])**; **LNC:=N[J]** *end*; *stop*;

Here we initially set the contents of **LNC** at 0; then we compare successively each of the three numbers with the contents of **LNC**, assigning the number to **LNC** if it is larger than **LNC**; finally, then, the value of **LNC** will be the largest of the three numbers. Note that the *for* statement acts on the statement following, which in this case is a compound statement.

Frequently the number of iterations around the loop, namely, **(NUF−NUI)/ΔN,** is not known beforehand, as is the case in evaluating a function by means of a power series. Here the exit from the loop is usually determined by comparing some computed value with a small number ϵ. In such a case the compound statement being iterated contains an *if* and a *go to* statement that can jump out of the loop before the value of the recursive variable has reached the final value. Since it is not known how many iterations will be necessary, the final value **NUF** is made very large. For example, consider computing

$$\sin\theta = \theta - \frac{\theta^3}{3!} + \frac{\theta^5}{5!} - \frac{\theta^7}{7!} + \cdots$$

until $\theta^{2n+1}/(2n+1)! < \epsilon$. The code is as follows:

```
; TERM:=θ; SUM:=θ; for J:=1(1)50;
     begin TERM:=(−1×TERM×(θ↑2↓))/((2×J+1)×2×J);
          if (TERM<ε); go to R; SUM:=SUM+TERM end;
                                        R:VSIN:=SUM; stop;
```

Here we multiply the previous term by $-\theta^2/(2J+1)(2J)$ to obtain the present term. One rule, however, must be obeyed: An *if* statement can never immediately precede the word *end.*

The index of a *for* statement need not appear explicitly in the following statement. For example, consider the computation of $\sum_{i=1}^{5} n_i$, where $n_{j+1} = 3n_j{}^2 - 9n_{j-1}^2$, $n_0 = 0$, and $n_{-1} = 1$. The code is

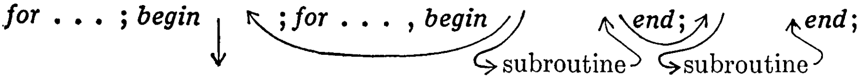
```
; SUM:=0; na:=1; nb:=0; for J:=1(1)5;
     begin nc:=3×(nb↑2↓)−9×(na↑2↓); SUM:=SUM+nc;
                                   na:=nb; nb:=nc end; stop;
```

Each time around the loop **J** is increased by 1; then the process ends after five iterations as desired, when **J** has become 5. Note that a, b, and c really correspond to $j-1$, j, and $j+1$, respectively.

Finally observe that loops can contain loops; or in other words *for* statements can act upon *for* statements. For example, consider the code for $\sum_{i=1}^{5}\left(\sum_{j=1}^{3}(i+1)(j+2)\right)$,

```
; SUM:=0; for i:=1(1)5; begin for j:=1(1)3;
                         SUM:=SUM+(i+1)×(j+2) end; stop;
```

A general rule to apply is that no jump may be made into the range of a *for* statement *except during the use of subroutines not affecting the for indices.* The following diagram indicates valid jumps (i.e., *if* followed by *go to* statements):



Declarations. Declarations serve to inform the computer of certain facts required by the code. They have no operational meaning as far as the code is concerned. Five declarations will now be described.

1. *Array* declarations tell the computer how many numbers there are in a vector or matrix array, so that the appropriate memory space may be reserved. The form of an *array* declaration is

; ***array*** **(VAR[L:U]);**

where **VAR** is the subscripted variable in question, **L** is the *list* of the lowest values the subscripts take, and **U** is the *list* of the largest values the subscripts take. Thus, if

$$\mathbf{MATRIX}_{ij} = \begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \\ a_{31} & a_{32} \end{pmatrix}$$

we would write ***array*** **(MATRIX[1,1:3,2])**. For a vector with five components V_1, V_2, V_3, V_4, and V_5 we would write ***array*** **(V[1:5])**, and so forth.

2. *Type* declarations classify variables or functions for the automatic program so that they may be treated accordingly. For example, arithmetic operations involving fixed-point variables differ from those involving floating-point variables. Labels of statements cannot have numerical values and therefore are treated differently from variables, and so forth. A *type* declaration would be as follows:

; TYPE (EXPA,EXPB, . . . ,EXPC);

where **TYPE** stands for some type, as *fixed point* (**FXD**), *floating point* (**FLT**), or *label* (**LAB**), that can be recognized by the program and **EXPA**, etc., represent the variables or functions of that type.

3. A ***switch***, or *variable-connector* declaration, specifies (by *their* labels) alternative statements to which a ***go to*** statement may jump. The form of a variable-connector declaration is

; ***switch*** **NAME:=(LA,LB, . . . ,LC);**

where **NAME** is the name of the connector and **LA**, **LB**, . . . , **LC** are the ordered set of labels of the possible statements that can be distinguished by the connector. The variable connector is used by a ***go to*** statement in the form

; ***go to*** **NAME[N];**

where **NAME[N]** is a subscript variable and **N** is an integer telling which label of the switch declaration to use. Thus if **LA**, the first label, is to be used, then **N** would be set to **1**; if the second label, **LB**, is to be used, then **N** would be set to **2**; and so forth. For example, consider the problem of forming $A = X^2 + Y$ if $N = 1$, $A = X + Y^2$ if $N = 2$, and $A = 0$ if $N = 3$. We form the required N and then jump by means of a variable connector for the proper arithmetic statement,

```
; switch S:=(P,Q,R); J:=N; go to S[J]; P:A:=(X↑2↓)+Y; go to T;
                          Q:A:=X+(Y↑2↓); go to T; R:A:=0; T: stop;
```

4. ***Comment*** *declarations* are used to add informal comments to a program; they have no effect on the program and are intended only as additional information for the reader. This declaration *must* start with the word ***comment***, and there *cannot* be a semicolon within the declaration. (Why?) For example,

```
; Comment This code will compute sin θ with an error no greater than ε;
```

5. A *function* declaration describes the computing rule to be used in evaluating the function in terms of its variables. The form of a function declaration is

```
; FUNCT(VARA,VARB, . . . ,VARC):=EXP;
```

where **FUNCT** is the name of the function, the independent variables are listed in the parentheses, and **EXP** is the computing rule. For example, if $f(u,v) = au^2 + buv + cv^2$, we would write

```
; QUADRATIC(u,v):=a×(u↑2↓)+b×u×v+c×(v↑2↓);
```

To use the function in a code, consider the evaluation of

$$Z = Af(r^2,s^2) + Bf(r^3,s^3)$$

After declaring the quadratic as above we would write

```
; Z:=A×QUADRATIC(r↑2↓,s↑2↓)
                        +B×QUADRATIC(r↑3↓,s↑3↓);
```

Note that it is the *order* of the variables appearing in the function declaration that is important. Thus, in **QUADRATIC(r↑2↓,s↑2↓)**, since r^2 occurs first it is substituted for u; s^2 occurring second is substituted for v. The computation actually performed is $a(r^2)^2 + b(r^2)(s^2) + c(s^2)^2$. The variable substituted may be a function itself, as we shall see in the example below.

The computer computes **QUADRATIC(u,v)** only when it is confronted with the computation of **Z**. The arguments u and v must first be computed by the program, and then, having u and v (namely, r^2 and s^2 in the first case and r^3 and s^3 in the second), the program can refer to the declaration to find out how to compute **QUADRATIC(u,v)**. How does the computer distinguish between **QUADRATIC(u,v)** when it is used as a declaration and when it is used as a function in a statement? If **QUADRATIC(u,v)** appears to the *left* of the := relation, it can only be a function declaration; if it appears to the right of the := relation, it must represent an actual function that refers to a declaration.

Consider a more complicated function that may be defined by a compound statement rather than a single arithmetic expression. The problem arises as to where the value of the function, as computed by the compound statement, will be found. The convention is used that the address of the final result should be called by the name of the function, e.g., **FUNCT**. For example, consider a series approximation to sin θ such that the last term is less than ϵ:

```
; sin(θ,ε):= begin TERM:=θ; SUM:=θ; for J:=1(1)50;
    begin TERM:=(−1×TERM×(θ↑2↓))/((2×J+1)×2×J);
if (TERM <ε); go to R; SUM:=SUM+TERM end; R:sin:=SUM end;
```

Note that this is all just a declaration of the function $sin(\theta,\epsilon)$, and the result is found in the address ***sin*** [*not* ***sin*(θ,ε)**, because that represents a function, not a variable or address]. This function declaration would be used as follows: Consider the computation of

$$z = 3 \sin^2 (u - v, \epsilon) + \sqrt{1 - \sin^2 (u^2 - v^2, \epsilon)}$$

for $\epsilon = 1/n$. After declaring ***sin*(θ,ε)** we would write

```
; z:=3×(sin(u−v,1/n))↑2↓
                    +sqrt(1−(sin(u↑2↓−v↑2↓,1/n))↑2↓);
```

Common functions, such as ***sqrt*(*x*)**, ***abs*(*x*)**, etc., need *not* be declared, because they are explicitly written into the automatic programming routine itself.

Use of Subroutines: Procedure Statements and Procedure Declarations. As previously discussed in Chap. 4, a subroutine must be "told" initial conditions and in what addresses to put the final results. (The automatic program takes care of the return jump back to the original code.) In our present code this is done by a *procedure statement* of the form

```
; SUBR(IA,IB, . . . ,IC)=:(FL,FM, . . . ,FN);
```

Here **SUBR** is the name of the subroutine, **IA**, **IB**, . . . , **IC** are the initial conditions (i.e., expressions that can be evaluated), and **FL**, **FM**, . . . , **FN** are the addresses where the final results are to be located (i.e., simple or subscripted variables). The subroutine itself is considered as a *procedure declaration.* Its form is

$$\underbrace{;\ \textit{procedure}\ \mathbf{SUBR(x,y,\ \ldots\ ,z)=:(u,v,\ \ldots\ ,w)};}_{\text{Procedure-declaration heading}}\ \underbrace{\mathbf{begin\ \ldots\ end};}_{\substack{\text{Declarations and}\\ \text{statements of}\\ \text{subroutine}\\ \text{proper}}}$$

A special statement, denoted simply by ***return***, is placed in the subroutine proper at that point from which the return jump back to the original code should occur. Note that, if a subroutine has a single value as its final result, then we could have as well used a *function declaration* referred to by a function in a statement. As in the case of the function the *order* of the variables is used to distinguish among them.

As an example, consider a subroutine that computes e^x, sin x, and cos x. Since

$$\begin{aligned} e^x &= 1 + x + x^2/2! + x^3/3! + x^4/4! + x^5/5! + \cdots \\ \sin x &= x - x^3/3! + x^5/5! - \cdots \\ \text{and} \qquad \cos x &= 1 - x^2/2! + x^4/4! - \cdots \end{aligned}$$

we can form $x^m/m!$ and then add this (with the appropriate sign) to the respective partial sums as called for by the series (i.e., alternating odd- and even-power terms). We use a switch declaration for the alteration. The procedure declaration (and the included subroutine) would be

```
; procedure TRIG(θ,ε)= :(EXPO,SIN,COS); begin switch SAC:=(S,C);
    TERM:=θ; SUME:=1+θ; SUMS:=θ; SUMC:=1; J:=2; I:=2;
B: TERM:=(TERM×θ)/J; if (TERM<ε); go to R;
                                    SUME:=SUME+TERM; go to SAC[I];
C: SUMC:=SUMC+((−1)↑J/2↓)×TERM; I:=1; go to T;
S: SUMS:=SUMS+((−1)↑(J−1)/2↓)×TERM; I:=2; go to T;
T: J:=J+1; go to B;
R: EXPO:=SUME; SIN:=SUMS; COS:=SUMC; return end;
```

Note that the inputs to the subroutine are **θ** and **ε** in that order; the outputs are **EXPO**, **SIN**, and **COS** in that order. The switch is reset to go through the alternate sin (or cos) loop at the end of each loop. Finally the computed results **SUME**, **SUMS**, and **SUMC** are assigned to their corresponding variables. Next suppose that we wanted to *use this subroutine* to compute $e^{-\omega t}(A \sin(\omega t) + B \cos(\omega t))$. The procedure statement with the appropriate code would be simply

```
; TRIG(ω×t,ε)= :(U,V,W); Z:=(A×V+B×W)/U;
```

As another example of a procedure declaration and its included subroutine consider matrix multiplication where it is desired to compute $C_{ik} = \sum_j A_{ij}B_{jk}$. If $i = 1, \ldots, I$; $j = 1, \ldots, J$; and $k = 1, \ldots, K$, the procedure declaration would be

```
; procedure MATMULT(A[i,j],B[j,k],I,J,K)= :(C[i,k]); begin array
(A[1,1:I,J]); array (B[1,1:J,K]); array (C[1,1:I,K]); for i:=1(1)I; begin
for k:=1(1)K; C[i,k]:=0; begin for j:=1(1)J; C[i,k]:=C[i,k]+A[i,j]
×B[j,k] end end; return end;
```

Finally consider an example of integration by means of the trapezoid rule (see Sec. 6-7, for example), namely,

$$A = \int_a^b f(x)\,dx = \frac{h}{2}\,(f(a) + 2f(a + h) + 2f(a + 2h) + \cdots + f(b))$$

Here we shall include a *comment* declaration as well:

; procedure TRAP(F(x),a,b,ε,V):=(A); *begin comment* a and b are the min and max, resp., of the points defining the interval of integration. F(x) is the function to be integrated, ε is the permissible difference between two successive trapezoidal sums, and V is greater than the maximum absolute value of F(x) on (a,b); Ibar:=V×(b−a); h:=(b−a); J:=(1/2)×(F(a)+F(b)); S:=0; n:=1; *for* i:=1(1)50; begin *for* k:= 1(1)n; S:=S+F(a+h×(k−1/2)); I:=h×(J+S); if (ε>abs(I−Ibar)); *go to* T; n:=2↑i↓; h:=h/2; Ibar:=I *end*; T:A:=I; *return end*;

Note that for the ith iteration we add 2^i more points to S, namely, the points dividing the previous points. Then h×((1/2)×(F(a)+F(b))+S) becomes the new approximation to the integral.

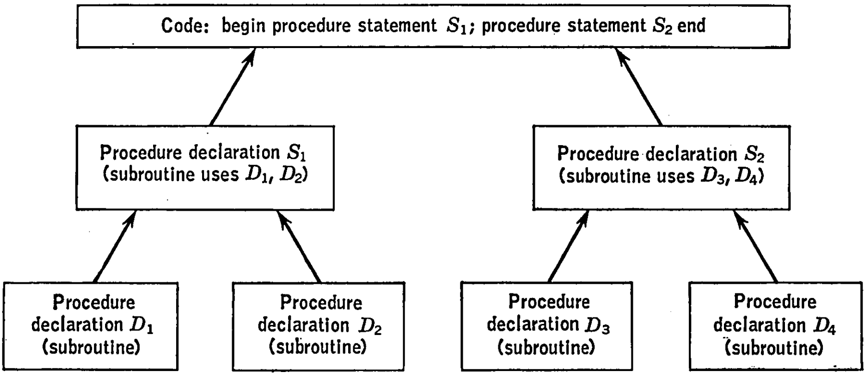

Code: begin procedure statement S_1; procedure statement S_2 end
Procedure declaration S_1 (subroutine uses D_1, D_2)
Procedure declaration S_2 (subroutine uses D_3, D_4)
Procedure declaration D_1 (subroutine)
Procedure declaration D_2 (subroutine)
Procedure declaration D_3 (subroutine)
Procedure declaration D_4 (subroutine)

FIG. 5-9. Use of procedures as building blocks.

The building-block method of using procedures (or subroutines) is illustrated in Fig. 5-9. The final code can be just a sequence of *procedure statements;* similarly each of these procedure declarations referred to by the statements can itself be a sequence of procedure statements, and so forth. The advantage of this building-block technique is that a large code can easily be subdivided into more elementary codes, and several coders and programmers at once can work on the subdivisions. Then the parts can be compiled into the large program, in as many levels as required.

EXERCISES

(*a*) At the beginning of this section we remarked that by the " . . . previous sections it is clear that a language can be devised that is independent of particular computer instructions." What previous sections do we have in mind, and how can these be used for such a language? (HINT: For example, Sec. 5-8 is concerned with an automatic program that interprets algebraic symbols, but the procedure for using this program given in that section depends on the word format of the computer. However, Sec. 5-7 describes a symbolic address translator, the use of which can easily be made independent of the word format. Thus the methods of the two automatic programs can be combined to result in a process for using algebraic-symbol interpretation that is independent of computer instructions.)

(*b*) Make a flow chart for the automatic program that can interpret each of the seven statements given in the above language.

Write codes in the symbolic algebraic language to compute the following functions:

(*c*) $n!/x!(n - x)!$.

(*d*) $u = \sum_{x=0}^{n} x \frac{n!}{x!(n - x)!} p^x q^{n-x}$.

(*e*) $S = \frac{30 I_0^2}{2\pi R^2} \left[\frac{\cos\left(\frac{\pi}{2}\cos\theta\right)}{\sin\theta} \frac{\sin\left(\frac{n\pi}{2}\sin\theta\cos\varphi\right)}{\sin\left(\frac{\pi}{2}\sin\theta\cos\varphi\right)} \right]^2$.

(*f*) Write a subroutine in the algebraic language to put the numbers V_i $(i = 1, \ldots, n)$ in numerical order (see Exercise *h* of Sec. 4-2).

(*g*) Write a subroutine in the language for Simpson's integration rule (see Sec. 6-7).

(*h*) Write a subroutine in the language to evaluate a determinant.

(*i*) Write a subroutine in the language to form the inverse of a matrix.

5-10. Additional Topics

a. References to Computer Maintenance and Program Debugging

Electronic Computers, Session III, *IRE Natl. Conv. Record,* 1954.

Ledley, R. S., and J. B. Wilson: "Programming and Utilizing Digital Computers," McGraw-Hill Book Company, Inc., New York, in press.

McCracken, D. D.: "Digital Computer Programming," John Wiley & Sons, Inc., New York, 1957.

Symposium on Diagnostic Programs and Marginal Checking for Large Scale Digital Computers, *IRE Conv. Record,* pt. 7, 1953.

Von Neumann, J.: Probabilistic Logics and the Synthesis of Reliable Organisms from Unreliable Components, "Automata Studies," Princeton University Press, Princeton, N.J., 1956.

Wheeler, D. J., and J. E. Robertson: Diagnostic Programs for the Illiac, *Proc. IRE,* vol. 41, pp. 1332–1340, October, 1953.

b. The International Algebraic Language. The automatic language described in Sec. 5-9 was a simplification of the proposed International Algebraic Language, now called *ALGOL* (see J. Perlis and K. Samelson, for the Association for Computing Machinery–GAMM Committee, Preliminary Report—International Algebraic Language, *Communs. Assoc. Computing Machinery,* vol. 1, no. 12, December, 1958). In our discussions several statements were omitted and others severely simplified, and the Boolean expression was omitted. By referring to the above-mentioned article, complete the study of ALGOL.

c. FORTRAN I and II. The most well-known algebraic language presently in common use is the IBM FORTRAN I and II (FORTRAN means "*for*mula *trans*la-tion"). Many aspects of this language are similar to the International language. By means of the FORTRAN I and II manuals, compare and contrast FORTRAN I and II and the International Language. (See "Programmer's Primer for FORTRAN," "Programmer's Reference Manual, FORTRAN," and "Reference Manual, FORTRAN II," all published by IBM for the 704 computer.)

d. Automatic Coding Systems. In order to give the reader an idea of how extensive has been work on automatic programming systems, we present in Table 5-2 a selected list of automatic programs, chosen from a more detailed compilation of over 90 that can be found in *Proc. Fourth Ann. Computer Appls. Symposium,* Oct. 24–25, 1957, Armour Research Foundation. (See especially R. W. Bemer, The Status of Automatic Programming for Scientific Problems.)

TABLE 5-2. LIST OF SELECTED AUTOMATIC PROGRAMMING SYSTEMS†

Computer	Name or acronym of automatic coding system	Developed by	Interpreter	Compiler	Algebraic
IBM 704, 705, and 709	AFAC	Allison GM		x	x
	FORTRAN I and II	IBM		x	x
IBM 650.......	Bell Li	Bell Tel. Labs.			
	IT	Carnegie Tech.	x	x	x
	FORTRANSIT	IBM–Carnegie Tech.		x	x
	APT	Applied Physics Lab.			
	SOAP I and II	IBM		x	
Sperry Rand 1103A	COMPILER I	Boeing, Seattle		x	x
	USE	Ramo-Wooldridge		x	
	QUICKTRICK	Operations Research Office	x	x	
	APT	Applied Physics Lab.			
	IT	Carnegie Tech., Ramo-Wooldridge		x	x
Sperry Rand UNIVAC I and II.......	AT3	Sperry Rand		x	x
Datatron 205...	IT	Purdue University		x	x
Whirlwind......	Algebraic	MIT		x	x
	Summer Session	MIT	x		
Proposed for all computers	ALGOL	Association for Computing Machinery–GAMM	x	x	x

† Information concerning these and many other automatic programs is being made available by the Association for Computing Machinery, New York.

PART 2

FUNCTIONAL APPROACH TO SYSTEMS DESIGN

CHAPTER 6

FUNDAMENTALS OF NUMERICAL ANALYSIS

6-1. Introduction

The systems design of a digital computer or control is the over-all block diagram and description of its parts, including its operational characteristics, input-output facilities, and coding or information-handling structure. The purpose of this part of the book is to discuss the functional approach to such systems design.

Consider first the so-called *general-purpose digital computer*, which is designed to perform the processes of numerical analysis. Actually we have already described its systems design in Part 1. It is important for the engineer to understand the basic concepts of numerical analysis as a foundation for appreciating criteria for the systems design of such a computer. In the decision on the instruction system of a proposed computer the engineer must comprehend the problems involved and the techniques used in reducing a mathematical problem to sequences of additions, subtractions, multiplications, and divisions. In addition the engineer will often find use for these methods in his own analytical problems.

Numerical analysis is a large and growing subject in itself, and our discussion must be limited to those aspects which are considered of importance to the computer engineer. These questions naturally arise in the study of digital computers: If a computer can do little more than the elementary arithmetic operations, how can it be used to perform function evaluation, integration, or differentiation, or to solve ordinary and partial differential equations? Also, in performing long sequences of arithmetic operations, how is the accuracy of the computing results analyzed and controlled? We can give in this chapter no more than a brief introduction to the answers to these questions, but it is felt that such an elementary survey can well serve to orient the student in this field.

The chapter starts with a technique for solving linear simultaneous equations (Sec. 6-2), which arise frequently in all fields of science and engineering. Next solution of algebraic and transcendental equations by means of successive approximations is considered (Sec. 6-3). Then the following three sections introduce the fundamental concept of polynomial approximations for the evaluation of functions. In passing, continued fractions are considered, so that the reader will not think there are no other methods for evaluating functions. The best-fit polynomial approximations are emphasized because of their extreme importance in digital-computer work. In order that the reader may develop a "numerical feeling" for such approximations, graphs are given of actually computed examples. The next four sections show how the same fundamental idea of polynomial approximation is used in numerical integration and differentiation and in the solution of differential equations. To complete the picture, the method of undetermined coefficients is given, so that the student can experiment and derive formulas for himself.

Up to this point discussion of error was omitted to allow full concentration on the concepts involved: the final section of this chapter is concerned with accuracy and error. Our discussion will be limited to fundamental definitions and to consideration of the arithmetic operations. In performing a computation on a digital computer, the significance and accuracy associated with arithmetic operations are usually uppermost in the mind of the computer user. Such considerations can greatly influence the systems design of new computers through the over-all word format, word length, and operations designs. Hence an appreciation of the problems involved becomes of prime importance to the computer engineer.

6-2. Simultaneous Linear Equations

The Problem. Consider a set of n simultaneous linear equations in n unknowns $x_1, \ldots, x_n$,

$$a_{11}x_1 + a_{12}x_2 + \cdots + a_{1n}x_n = y_1 \tag{1}$$
$$a_{21}x_1 + a_{22}x_2 + \cdots + a_{2n}x_n = y_2 \tag{2}$$
$$\cdots\cdots\cdots\cdots\cdots\cdots \qquad \cdot$$
$$a_{n1}x_1 + a_{n2}x_2 + \cdots + a_{nn}x_n = y_n \tag{n}$$

These equations admit of a unique set of solutions,

$$x_i = x_i(a_{11}, \ldots, a_{nn}, y_1, \ldots, y_n)$$

if at least one y_i is not zero, and if the determinant det $(a_{ij}) \neq 0$.† In order to describe a method for solving the equations, let us first consider a special set of equations with coefficients $s_{ii} = 1$ and $s_{jk} = 0$ if $j > k$:

$$\begin{aligned} x_1 + s_{12}x_2 + s_{13}x_3 + s_{14}x_4 &= y_1 \\ x_2 + s_{23}x_3 + s_{24}x_4 &= y_2 \\ x_3 + s_{34}x_4 &= y_3 \\ x_4 &= y_4 \end{aligned}$$

† See, for example, E. A. Guillemin, "The Mathematics of Circuit Analysis," pp. 13–18, John Wiley & Sons, Inc., New York, 1951.

These four equations can easily be solved for the unknowns x_1, x_2, x_3, and x_4, for $x_4 = y_4$ is already given, and it follows that

$$\begin{aligned} x_3 &= y_3 - s_{34}x_4 \\ x_2 &= y_2 - s_{24}x_4 - s_{23}x_3 \\ x_1 &= y_1 - s_{14}x_4 - s_{13}x_3 - s_{12}x_2 \end{aligned}$$

Hence it becomes clear that, if we can somehow manipulate Eqs. (1), (2), . . . , (n) so that the diagonal coefficients become 1 and the coefficients below the diagonal become zero, then the solutions can be rapidly obtained.

The Method. This process can be accomplished as follows: First divide both sides of Eq. (1) by a_{11}. If $a_{11} = 0$, then rearrange the order of the equations so that $a_{11} \neq 0$. (This, of course, can always be done. Why?) This results in

$$1 \cdot x_1 + \frac{a_{12}}{a_{11}} x_2 + \cdots + \frac{a_{1n}}{a_{11}} x_n = \frac{y_1}{a_{11}} \qquad (1)^1$$

Multiply Eq. $(1)^1$ by a_{21}, and subtract the result from (2), obtaining

$$0 \cdot x_1 + b_{22}x_2 + \cdots + b_{2n}x_n = y_2^{(1)} \qquad (2)^1$$

where $$b_{22} = a_{22} - \frac{a_{21}a_{12}}{a_{11}} \quad \cdots \quad b_{2n} = a_{2n} - \frac{a_{21}a_{1n}}{a_{11}}$$

and $$y_2^{(1)} = y_2 - \frac{a_{21}y_1}{a_{11}}$$

Similarly, multiply $(1)^1$ by a_{31}, and subtract the result from (3), obtaining

$$0 \cdot x_1 + b_{32}x_2 + \cdots + b_{3n}x_n = y_3^{(1)} \qquad (3)^1$$

where $$b_{32} = a_{32} - \frac{a_{31}a_{12}}{a_{11}} \quad \cdots \quad b_{3n} = a_{3n} - \frac{a_{31}a_{1n}}{a_{11}}$$

and $$y_3^{(1)} = y_3 - \frac{a_{31}y_1}{a_{11}}$$

Continuing in this, we obtain a new set of Eqs. $(1)^1$, $(2)^1$, . . . , $(n)^1$ such that $b_{11} = 1$ and $b_{i1} = 0$ for $i > 1$. Of course, it is clear that our new set of equations has the same solution as the original set.

Next we start with $(2)^1$ and divide both sides by b_{22}. [If necessary, a rearrangement of the order of Eqs. $(2)^1$, . . . , $(n)^1$ can be made to ensure that $b_{22} \neq 0$.] We obtain

$$1 \cdot x_2 + \frac{b_{23}}{b_{22}} x_3 + \cdots + \frac{b_{2n}}{b_{22}} x_n = \frac{y_2^{(1)}}{b_{22}} \qquad (2)^2$$

Multiply $(2)^1$ by b_{32}, and subtract the result from $(3)^1$, obtaining

$$0 \cdot x_2 + c_{33}x_3 + \cdot \cdot \cdot + c_{3n}x_n = y_3^{(2)} \qquad (3)^2$$

where $$c_{33} = b_{33} - \frac{b_{32}b_{23}}{b_{22}} \quad \cdot \cdot \cdot \quad c_{2n} = b_{3n} - \frac{b_{32}b_{2n}}{b_{22}}$$

and $$y_3^{(2)} = y_3^{(1)} - \frac{b_{32}y_2^{(1)}}{b^{22}}$$

Similarly, we obtain

$$c_{43}x_3 + \cdot \cdot \cdot + c_{4n}x_n = y_4^{(2)} \qquad (4)^2$$

and the same for $(5)^2, \ldots, (n)^2$. In this step we have made $c_{33} = 1$ and $c_{i2} = 0$ for $i > 2$.

Continuing in this way, we finally obtain, for example in the case of $n = 4$, the following equations, in the desired form for easy solution:

$$\begin{aligned} x_1 + \frac{a_{12}}{a_{11}} x_2 + \frac{a_{13}}{a_{11}} x_3 + \frac{a_{14}}{a_{11}} x_4 &= \frac{y_1}{a_{11}} \\ x_2 + \frac{b_{23}}{b_{22}} x_3 + \frac{b_{24}}{b_{22}} x_4 &= \frac{y_2^{(1)}}{b_{22}} \\ x_3 + \frac{c_{34}}{c_{33}} x_4 &= \frac{y_3^{(2)}}{c_{33}} \\ x_4 &= \frac{y_4^{(3)}}{d_{44}} \end{aligned}$$

Example. Let us carry this process through for a specific set of four equations.

$$\begin{aligned} 3x_1 + 9x_2 + 6x_3 - 12x_4 &= 9 && (1) \\ 2x_1 + 8x_2 + 12x_3 - 2x_4 &= 2 && (2) \\ 3x_1 + 12x_2 + 21x_3 - 9x_4 &= 15 && (3) \\ -x_1 - x_2 + 8x_3 + 8x_4 &= 5 && (4) \end{aligned}$$

Dividing both sides of (1) by 3, we find

$$x_1 + 3x_2 + 2x_3 - 4x_4 = 3 \qquad (1)^1$$

Now multiply Eq. $(1)^1$ by 2, and subtract from Eq. (2), obtaining

$$2x_2 + 8x_3 + 6x_4 = -4 \qquad (2)^1$$

Similarly, we multiply Eq. $(1)^1$ by 3 and subtract from Eq. (3), obtaining

$$3x_2 + 15x_3 + 3x_4 = 6 \qquad (3)^1$$

And finally $(4)^1$ is obtained by multiplying $(1)^1$ by -1 and subtracting from (4),

$$2x_2 + 10x_3 + 4x_4 = 8 \qquad (4)^1$$

Continuing, we form Eqs. $(2)^2$, $(3)^2$, and $(4)^2$,

$$\begin{aligned} x_2 + 4x_3 + 3x_4 &= -2 && (2)^2 \\ 3x_3 - 6x_4 &= 12 && (3)^2 \\ 2x_3 - 2x_4 &= 12 && (4)^2 \end{aligned}$$

Finally we have

$$x_3 - 2x_4 = 4 \qquad (3)^3$$
$$2x_4 = 4 \qquad (4)^3$$

and

$$x_4 = 2 \qquad (4)^4$$

Thus the set of equations from which the solution is obtained is

$$\begin{aligned} x_1 + 3x_2 + 2x_3 - 4x_4 &= 3 & (1)^1 \\ x_2 + 4x_3 + 3x_4 &= -2 & (2)^2 \\ x_3 - 2x_4 &= 4 & (3)^3 \\ x_4 &= 2 & (4)^4 \end{aligned}$$

whence

$$\begin{aligned} x_4 &= 2 &&= 2 \\ x_3 &= 4 + 2 \times 2 &&= 8 \\ x_2 &= -2 - 3 \times 2 - 4 \times 8 &&= -40 \\ x_1 &= 3 + 4 \times 2 - 2 \times 8 - 3 \times (-40) &&= 115 \end{aligned}$$

Figure 6-1 is the flow diagram for this method of solving simultaneous linear equations.

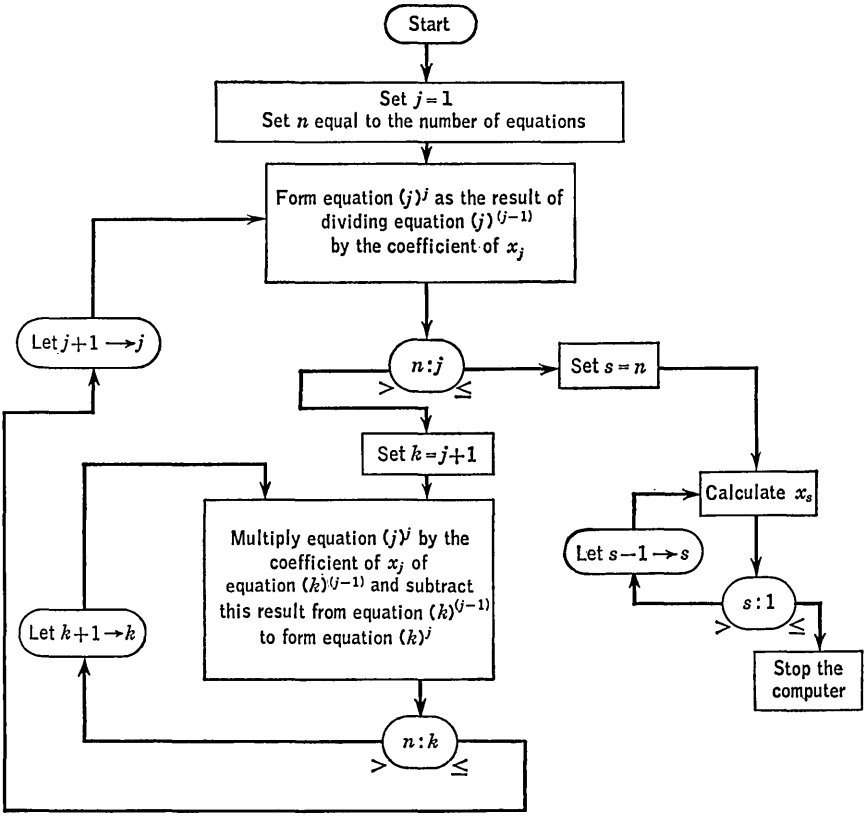

Start
Set $j = 1$
Set n equal to the number of equations
Form equation $(j)^j$ as the result of dividing equation $(j)^{(j-1)}$ by the coefficient of x_j
Let $j+1 \to j$
$n:j$
>
≤
Set $s = n$
Set $k = j+1$
Multiply equation $(j)^j$ by the coefficient of x_j of equation $(k)^{(j-1)}$ and subtract this result from equation $(k)^{(j-1)}$ to form equation $(k)^j$
Let $k+1 \to k$
$n:k$
>
≤
Calculate x_s
Let $s-1 \to s$
$s:1$
>
≤
Stop the computer

FIG. 6-1. Flow chart for solving simultaneous linear equations.

EXERCISES

(*a*) Solve

$$\begin{aligned} 5x_1 - 2x_2 &= 3 \\ 4x_1 - 5x_2 &= -18 \end{aligned}$$

by following in detail the steps of the flow diagram, keeping track of all the tallies.

(*b*) Solve

$$\begin{aligned} 0 \cdot x_1 + 4x_2 + x_3 - 3x_4 &= 1 \\ 4x_1 + 0 \cdot x_2 - 2x_3 + 2x_4 &= 14 \\ 8x_1 - 7x_2 + x_3 - 3x_4 &= -12 \\ -5x_1 + 8x_2 + 0 \cdot x_3 - x_4 &= -2 \end{aligned}$$

Solution. $x_1 = 8$, $x_2 = 7$, $x_3 = 27$, $x_4 = 18$.

(*c*) The flow chart of Fig. 6-1 does not account for zero coefficient in the leading term of Eq. $(j)^i$. How can the flow chart be adjusted to account for this?

(*d*) Using the three-address instructions of Chap. 3, write a code that solves simultaneous linear equations.

6-3. Algebraic and Transcendental Equations

Successive Approximations. The method of successive approximations can be used to evaluate the solutions of algebraic and transcendental equations. Although the best method for making the successive approximations in any case depends on the nature of the functions involved, we shall try by means of specific examples to illustrate the general principles that apply. Often a graph of the functions involved in the equations will help in the determination of the best method of approximation.

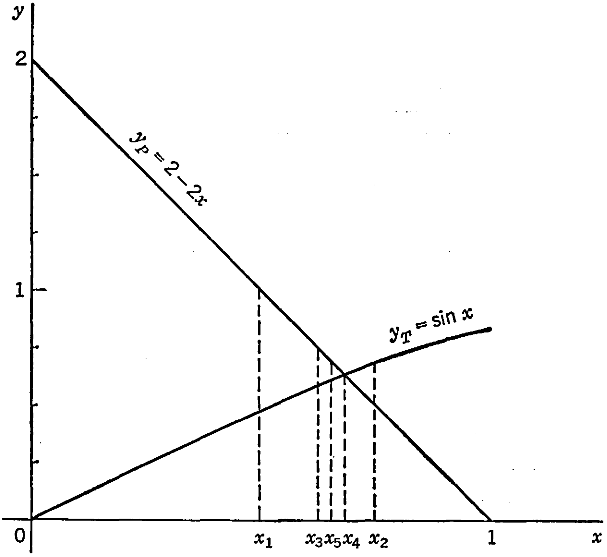


FIG. 6-2. Finding solution to sin $x + 2x - 2 = 0$ by the method of successive approximations.

Consider, for example, the equation

$$\sin x + 2x - 2 = 0$$

to be solved for x. We can write this as $\sin x = 2 - 2x$ and graph $y_T = \sin x$ and $y_P = 2 - 2x$ (see Fig. 6-2). The desired value of x is the x coordinate of the intersection of the two curves, where $\sin x$ will be equal to $2 - 2x$. We know that the value of x must be between 0 and 1. Hence we try x_1 such that $0 < x_1 < 1$. As can be seen from Fig. 6-2, if $\sin x_1 > 2 - 2x_1$, then the solution must lie to the left of x_1, that is, $x < x_1$; if $\sin x_1 < 2 - 2x_1$, then the solution must be to the right of x_1, that is, $x > x_1$. Suppose that we try $x_1 = 0.500$ (radians of course); then $\sin x_1 = 0.479$, and $2 - 2x_1 = 1$. Hence $0.500 < x < 1$, and so we try $x_2 = 0.750$ (that is, half the distance between 0.500 and 1). Then $\sin x_2 = 0.682 > 2 - 2x_2 = 0.5$, whence $0.500 < x < 0.750$. We let $x_3 = 0.625$ (half the distance between 0.500 and 0.750); then

$$\sin x_3 = 0.585 < 2 - 2x_3 = 0.75$$

and so we try next 0.687, etc. In each step we find in which half of the previous interval the solution lies; Fig. 6-3 shows these calculations. The fact that we are halving each time means that we are developing the solution as a binary number. If the solution lies in the right half of an interval in a step, then a unit corresponds to this step in the solution; otherwise a zero corresponds. Hence the solution $x = .10101111$ according to Fig. 6-3, column y_P.

(i)	x_i	$y_T = \sin x$	$y_P = 2 - 2x$	y_T greater	y_P greater
(1)	0.500	0.479	1		✓
(2)	0.750	0.682	0.5	✓	
(3)	0.625	0.585	0.75		✓
(4)	0.687	0.634	0.626	✓	
(5)	0.656	0.609	0.688		✓
(6)	0.672	0.621	0.656		✓
(7)	0.680	0.629	0.640		✓
(8)	0.684	0.631	0.632		✓

FIG. 6-3. Calculations for solving the equation $\sin x = 2 - 2x$.

The flow diagram (see Fig. 6-4) is relatively simple: depending on whether $\sin x_i < 2 - 2x_i$ or $\sin x_i > 2 - 2x_i$, we determine the next trial solution x_{i+1} by adding to or subtracting from the previous trial solution x_i one-half of the previously used increment Δx_i. Since each trial determines an additional bit of the solution, the procedure is repeated the same number of times as the number of bits of accuracy desired in the solution.

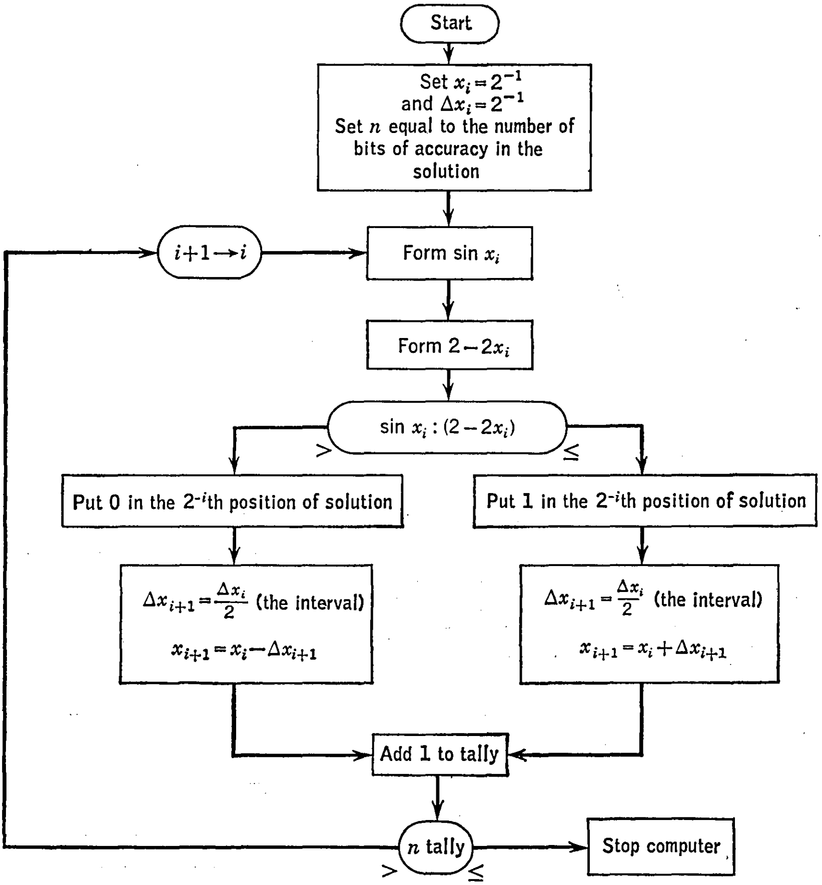

Start
Set $x_i = 2^{-1}$ and $\Delta x_i = 2^{-1}$ Set n equal to the number of bits of accuracy in the solution
$i+1 \rightarrow i$
Form sin x_i
Form $2-2x_i$
sin x_i : $(2-2x_i)$
>
≤
Put 0 in the 2^{-i}th position of solution
Put 1 in the 2^{-i}th position of solution
$\Delta x_{i+1} = \frac{\Delta x_i}{2}$ (the interval)
$x_{i+1} = x_i - \Delta x_{i+1}$
$\Delta x_{i+1} = \frac{\Delta x_i}{2}$ (the interval)
$x_{i+1} = x_i + \Delta x_{i+1}$
Add 1 to tally
n tally
>
≤
Stop computer

FIG. 6-4. Flow chart for solving the equation sin $x = 2 - 2x$.

Accelerating the Convergence of Iterative Processes.† The problem of solving an equation $F(x) = 0$ can be stated in general as that of finding a value (root) of x, say x_r, such that $F(x_r) = 0$. The general *iteration algorithm* involves successive approximations $x_0, x_1, x_2, \ldots$, which are determined as follows:

1. If $F(x) = 0$ can be expressed in the form $x = f(x)$, then we let $x_{n+1} = f(x_n)$. For some initial guess x_0, we find $x_1 = f(x_0)$, $x_2 = f(x_1)$, $x_3 = f(x_2), \ldots$, continuing until, for some n, $x_n - x_{n+1}$ is less than some preassigned small number, the allowable error.

2. If $F(x) = 0$ cannot be written in the form of $x = f(x)$, then we let $x_{n+1} = x_n + \theta F(x_n)$, where θ is some suitably chosen nonzero constant, and proceed as in case 1 (see Exercise h).

† This process was worked out by J. H. Wegstein and P. Henrici; see J. H. Wegstein, Accelerating Convergence of Iterative Processes, *Communs. Assoc. Computing Machinery,* vol. 1, no. 6, June, 1958.

Let us consider case 1, for what we shall say also holds for case 2; Fig. 6-5 describes the process. Since $x = f(x)$, the object is to find the intersection of the two curves $y = x$ and $y = f(x)$. From $y = f(x_n)$ we find y_n; from $y = x$ we find $y_n = x_{n+1}$; from $y = f(x_{n+1})$ we find y_{n+1}; from $y = x$ we find $y_{n+1} = x_{n+2}$; and so forth.

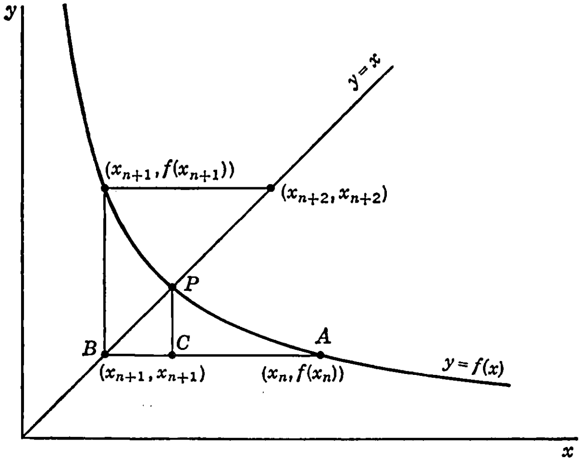


FIG. 6-5. The iterative processes.

Now observe from Fig. 6-5 that, instead of taking x_{n+1} for the substitution into $f(x)$, we would do better to choose a value x_{n+1}^* for the substitution, where

$$x_{n+1}^* = qx_n + (1 - q)x_{n+1}$$

i.e., some value between x_n and x_{n+1}. For instance, in the case illustrated in Fig. 6-5, q should be chosen so that

$$\frac{q}{1-q} = \frac{\overline{BC}}{\overline{AC}}$$

Of course we do not know the ratio $\overline{BC}/\overline{AC}$, but we can determine it approximately. For $\overline{PC} = \overline{BC}$, and thus

$$\frac{\overline{BC}}{\overline{AC}} = \frac{\overline{PC}}{\overline{AC}} = -m$$

where m is some value of $f'(x)$ between A and P (applying Rolle's theorem from the calculus). From the definition of the derivative, m can be approximated by

$$m \cong \frac{f(x_n) - f(x_{n-1})}{x_n - x_{n-1}} = \frac{x_{n+1} - x_n}{x_n - x_{n-1}}$$

Hence we can take

$$\frac{q}{1-q} = -m \qquad \text{or} \qquad q = \frac{m}{m-1}$$

The iterative process can then be described by the flow chart of Fig. 6-6, wherein we must observe carefully when x_n^* or x_n is used.

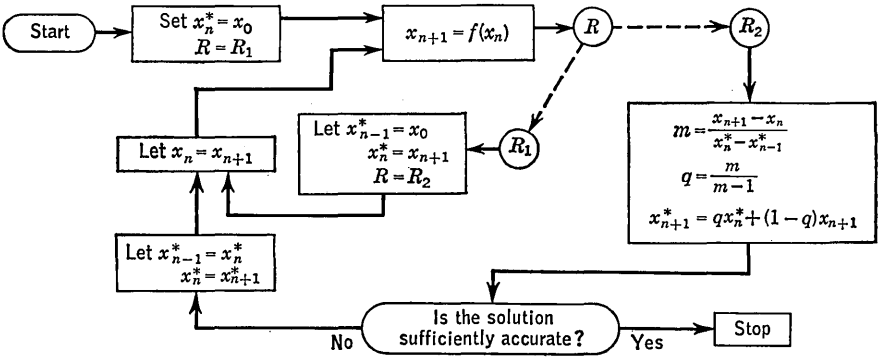


FIG. 6-6. Accelerating the convergence of the iterative processes.

As an example, consider the equation

$$x = \tfrac{1}{2}(e^{\alpha x} - e^{-\alpha x})$$

and with $\alpha = -0.5$ let us solve for the root $x = 0$, using both the ordinary and the accelerated iterative processes. Starting with $x_0 = 1$, we have Table 6-1 and Fig. 6-7.

TABLE 6-1. EXAMPLE OF BOTH THE ORDINARY AND ACCELERATED ITERATIVE PROCESSES FOR THE ROOT $x = 0$ OF $x = \tfrac{1}{2}(e^{-0.5x} - e^{0.5x})$

n	x_n by ordinary iterative substitution	x_n by accelerating iterative substitution	Corresponding value of m
0	1.000	1.000	
1	−0.521	−0.521	
2	0.263	−0.00348	0.340
3	−0.132	-1.32×10^{-5}	0.335
4	0.066	-1.65×10^{-11}	0.333
5	−0.033		
6	0.017		
7	−0.008		
8	0.004		

The ordinary iterative process may result in four types of behavior. The successive values of x_n can (1) oscillate and converge, (2) oscillate and diverge, (3) converge monotonically, or (4) diverge monotonically. The accelerating method just presented will make the nonconverging

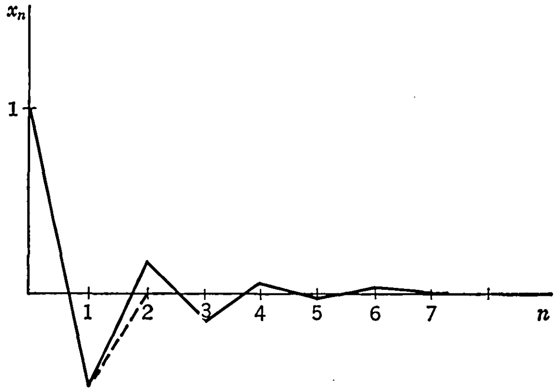


FIG. 6-7. Comparison of ordinary and accelerated iterative processes for the $x = 0$ root of $x = \frac{1}{2}(e^{-0.5x} - e^{0.5x})$.

types 2 and 4 converge and will accelerate the convergence of types 1 and 3. We have illustrated the situation for type 1. Exercises *e* to *g* will illustrate types 2 to 4.

EXERCISES

Solve by successive approximations, to within 0.001 (*decimal*):

(*a*) $1.5 \cos x + 2x - 2 = 0$.

(*b*) $\sin x - \tan^2 x = 0$.

(*c*) $\dfrac{1 - x^2}{x} - \dfrac{x}{1 - x^2} = 0$.

(HINT: There are four solutions.)

(*d*) $\dfrac{1 - x^2}{x} - \tan^2 x = 0$.

Consider the equation $x = \frac{1}{2}(e^{\alpha x} - e^{-\alpha x}) = \sinh \alpha x$. Take $x_0 = 1$, and show by direct computation:

(*e*) If $\alpha = -1.2$, the ordinary iterative process will be of type 2 but the accelerating process will converge.

(*f*) If $\alpha = 0.5$, the ordinary iterative process will be of type 3 but the accelerating process will make the convergence more rapid.

(*g*) If $\alpha = 1.2$, the ordinary iterative process will be of type 4 but the accelerating process will converge.

(*h*) Explain why we let $x_{n+1} = x_n + \theta F(x_n)$ in case 2 above. (HINT: First let $\theta = 1$.)

6-4. Function Evaluation: Series and Continued Fractions

Series Approximations. In the previous section we have tacitly assumed that the values of the functions entering into the equations could be somehow obtained by the computer for any given value of x. For polynomials no problem arises, since the evaluation of a polynomial is easily coded. On the other hand how were the values for sin x computed for a particular x? We shall discuss in this and the next few sections *three* ways in which such functions may be evaluated by the computer. These are (1) by power-series approximations, (2) by interpolation in a table, (3) by means of "best-fit" polynomial approximations. Of course our primary purpose is to discuss functions that are not themselves polynomials or fractions of polynomials, since the methods apply trivially for polynomials.

Many functions can be easily expanded in power series, and then the functions can be evaluated by means of these series for values of x within the region of convergence of the series. For example,

$$\sin x = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \cdots$$

Then, if sin 0.5 were required, the computer would be coded to compute

$$0.5 - \frac{0.5^3}{3!} + \frac{0.5^5}{5!} - \frac{0.5^7}{7!} + \cdots$$

If it were desired to know sin 0.5 to within ±0.0001, then the series would be carried out until some term were less than 0.0001. For example, $\sin 0.5 = 0.5 - 0.02083333 + 0.00026042 - 0.00000155$, where we stop. Hence, to the desired accuracy, $\sin 0.5 = 0.4794$. Another way of interpreting this result is that the *polynomial approximation*

$$\sin^* x = x - \frac{x^3}{3!} + \frac{x^5}{5!}$$

is sufficiently accurate for our purposes when $x = 0.5$. We write $\sin^* x$ instead of $\sin x$ to indicate that this is just an *approximation* to $\sin x$.

On the other hand, some series do not converge quite so rapidly. For example,

$$\arctan x = x - \frac{x^3}{3} + \frac{x^5}{5} - \frac{x^7}{7} + \cdots$$

For $x = 0.5$ we find

$$\arctan^* 0.5 = 0.5 - 0.041666 + 0.006250 - 0.001116$$

which obviously is not accurate to ±0.0001.

Often other expansions of a function can be found that might converge more rapidly than the power series; e.g., for arctan x we have the series

$$\arctan x = \frac{x}{1 + x^2}\left[1 + \frac{2}{3}\frac{x^2}{1 + x^2} + \frac{2 \cdot 4}{3 \cdot 5}\frac{x^4}{(1 + x^2)^2} + \cdots\right]$$

Continued-fraction Approximations. There are other expansions of functions that often aid in their evaluation, for example, continued fractions. For instance,

$$\arctan x = \cfrac{x}{1 + \cfrac{x^2}{3 + \cfrac{4x^2}{5 + \cfrac{9x^2}{7 + \cfrac{16x^2}{9 + \cdots}}}}}$$

It can be shown† that a continued fraction can be easily evaluated by means of 2 × 2 matrices as follows: Given the continued fraction

$$\frac{p}{q} = a_1 + \cfrac{b_2}{a_2 + \cfrac{b_3}{a_3 + \cfrac{b_4}{a_4 + \cfrac{b_5}{a_5 + \cdots}}}}$$

the *nth convergent* p_n/q_n is the result obtained when all terms from $n + 1$ on are neglected; e.g., the fourth convergent is

$$\frac{p_4}{q_4} = a_1 + \cfrac{b_2}{a_2 + \cfrac{b_3}{a_3 + \cfrac{b_4}{a_4}}}$$

The nth convergent may be evaluated from the following matrix equation:

$$\begin{pmatrix} p_n & p_{n-1} \\ q_n & q_{n-1} \end{pmatrix} = \begin{pmatrix} a_1 & 1 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} a_2 & 1 \\ b_2 & 0 \end{pmatrix} \begin{pmatrix} a_3 & 1 \\ b_3 & 0 \end{pmatrix} \cdots \begin{pmatrix} a_n & 1 \\ b_n & 0 \end{pmatrix}$$

From this we can find p_n and q_n, and also p_{n-1} and q_{n-1}. For example, for arctan 0.5 we have

$$\begin{pmatrix} p_5 & p_4 \\ q_5 & q_4 \end{pmatrix} = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} 1 & 1 \\ 0.5 & 0 \end{pmatrix} \begin{pmatrix} 3 & 1 \\ 0.25 & 0 \end{pmatrix} \begin{pmatrix} 5 & 1 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} 7 & 1 \\ 2.25 & 0 \end{pmatrix}$$
$$= \begin{pmatrix} 59.3750 & 8.00 \\ 128.0625 & 17.25 \end{pmatrix}$$

Hence

$$\frac{p_5}{q_5} = \frac{59.3750}{128.0625} = 0.4636 \qquad \text{and} \qquad \frac{p_4}{q_4} = \frac{8.00}{17.25} = 0.4637$$

Thus, arctan 0.5 = 0.4636 to within ±0.0001 (since the difference between the fourth and fifth convergents was 0.0001). We see that the infinite-fraction expansion of arctan x converged faster than its power-series expansion.

† See L. M. Milne-Thomson, "The Calculus of Finite Differences, p. 108, The Macmillan Company, New York, 1951.

There are of course many other kinds of expansions of functions—for instance, engineers will be familiar with Fourier-series expansions. However, the evaluation of a function by means of an expansion is not always the preferred method. Hence other means are often resorted to, and we shall consider a few of them briefly in the following sections.

EXERCISES

(*a*) Of the second series expansion given above for arctan x, how many terms are needed to determine arctan 0.5 to within ± 0.001?

(*b*) Using the continued-fraction expansion of arctan x, compute the value of π to seven significant figures. (HINT: arctan $1 = \pi/4$.)

(*c*) Draw a flow chart for evaluating the nth convergent of a continued fraction.

6-5. Function Evaluation: Interpolation

Polynomial Interpolation. Until now we have been considering functions that are memorized in the computer as subroutines. For instance, if $y = f(x)$, then for a given x_1 the computer will determine $f(x_1)$ by means of a program, which might be based on some expansion of $f(x)$. On the other hand it is often impossible to write a program for a certain function: e.g., the function might be empirical data, or perhaps the subroutine might require too much time or too much memory space. In these and other instances it is often convenient to record values of $f(x)$ in the computer memory in the form of a function table which gives the $f(x_i)$ corresponding to a set of specific values of x_i. It appears at first glance that this method has great limitations, since only those values of $f(x)$ which correspond to a relatively few values of x_i are in the memory of the computer. However, by means of *interpolation*, values of $f(x)$ corresponding to an x not listed in the table can be approximated.

Suppose that it is desired to find the value of $f(x)$ for a value of x not listed in the table. It may happen that there are listed in the table some values of $f(x_i)$ for x_i near x. Then, in order to find the value of $f(x)$, it seems plausible to draw a smooth curve, $y = f^*(x)$, through all the points listed in the table and to say that $y = f^*(x)$ as read from this curve is an approximation to $y = f(x)$ (see Fig. 6-8). Such a process is called *interpolation* and is most often accomplished by means of a *polynomial approximation.*

If $n + 1$ points (x_0,y_0), (x_1,y_1), (x_2,y_2), . . . , (x_n,y_n) of $f(x)$ are listed in the table, then a *unique* polynomial of the nth degree, $f^*(x)$, can be derived that passes through *all* these $n + 1$ distinct points.

For let

$$y = f^*(x) = A_0 + A_1x + A_2x^2 + \cdots + A_nx^n$$

If $f^*(x_0) = y_0$, $f^*(x_1) = y_1$, . . . , $f^*(x_n) = y_n$, then these are $n + 1$ equations which can be solved for $A_0, A_1, \ldots, A_n$. Once these values have been obtained, we can find for any $x' \neq x_i$ the value of $f^*(x')$. The more points considered, i.e., the higher the degree of $f^*(x)$, the closer will be the value of $f^*(x')$ to $f(x')$. Let $f^*_{0,1}(x)$ represent the polynomial, of first degree that goes through the points (x_0,y_0) and (x_1,y_1); let $f^*_{0,1,2}(x)$ represent the polynomial of second degree that goes through the points (x_0,y_0), (x_1,y_1), and (x_2,y_2); etc. Suppose that we want the interpolation $f^*(x')$ to within an accuracy of 0.001. To obtain this, we would calculate

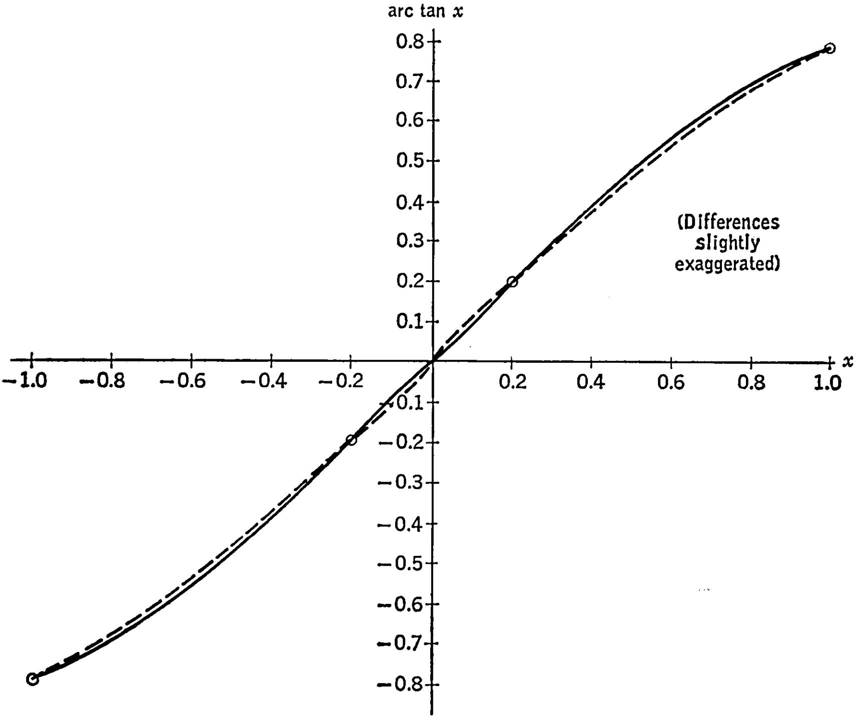


FIG. 6-8. A third-degree polynomial drawn through the four points $(-1,-0.785)$, $(-0.2,-0.197)$, $(+0.2,+0.197)$, $(+1,+0.785)$, as an approximation to arctan x, is shown by the solid line. Here $f^*(x) = \arctan^* x = 0.995x - 0.210x^3$. The dashed line is the actual arctan.

successively $f^*_{0,1}(x')$, $f^*_{0,1,2}(x')$, $f^*_{0,1,2,3}(x')$, etc., until

$$|f^*_{0,1,\ldots,r-1}(x') - f^*_{0,1,\ldots,r}(x')| < 0.001$$

and then we would take $f^*_{0,1,\ldots,r}(x')$ as our result.

Usually, the more known points used in an interpolation, the better becomes the accuracy of the interpolation. However, if the additional points are far from the point of interpolation, they may actually make the approximation worse. Similarly, with the same number of given points, the smaller the interval in which they are chosen, the more accurate the polynomial approximation in that interval. To illustrate this latter point, in Fig. 6-9 we have drawn the *error curves* for two polynomial approximations to arctan x. Each error curve has x as abscissa and the difference between the actual value of arctan x and its polynomial approximation as the ordinate. The first approximation is based on the six points -1, -0.6, -0.2, $+0.2$, $+0.6$, and 1; the second approximation, on the six points 0, 0.2, 0.4, 0.6, 0.8, and 1. Figure 6-9 illustrates the error curves for $0 \leq x \leq 1$. Note that in this interval the maximum error in the former case is over forty times the maximum error in the latter case.

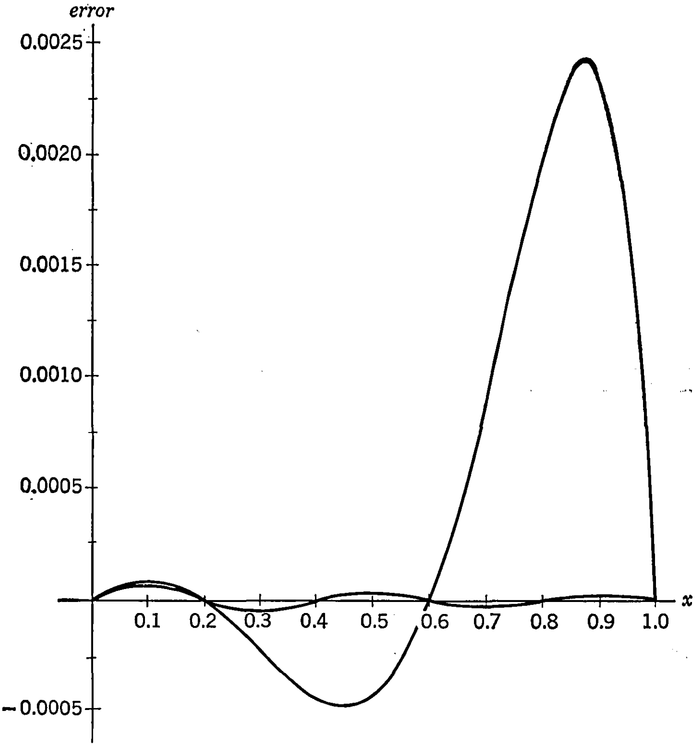


FIG. 6-9. Comparison of the error curves for the interpolation polynomial of arctan* x based on the six points -1, -0.6, -0.2, $+0.2$, $+0.6$, $+1$ (the curve with the big hump) and based on the six points 0, 0.2, 0.4, 0.6, 0.8, 1 (the curve that hugs the x axis). These error curves indicate just how much the interpolation polynomials differ from the true value of arctan x.

Obviously the sequence of calculations for a polynomial interpolation would be quite lengthy for each desired value if all the coefficients had to be calculated first, and then $f^*(x')$. The process can be shortened by calculating $f^*(x')$ directly, at essentially the same time as the coefficients are calculated. That is, the coefficients are never explicitly given, but $f^*(x')$ results directly. It should be remembered, however, that the meaning of $f^*(x')$ is the value at x' of the approximating polynomial that goes through the specified points.

Aitken's Method. One method for shortening calculations, called Aitken's method, follows.† First note that

$$f^*_{0,1}(x) = \frac{\begin{vmatrix} y_0 & x_0 - x \\ y_1 & x_1 - x \end{vmatrix}}{x_1 - x_0} \tag{a}$$

where $\begin{vmatrix} y_0 & x_0 - x \\ y_1 & x_1 - x \end{vmatrix}$ is the determinant with value $y_0(x_1 - x) - y_1(x_0 - x)$. For

$$f^*_{0,1}(x_0) = \frac{\begin{vmatrix} y_0 & 0 \\ y_1 & x_1 - x_0 \end{vmatrix}}{x_1 - x_0} = \frac{y_0(x_1 - x_0)}{x_1 - x_0} = y_0$$

and
$$f^*_{0,1}(x_1) = \frac{\begin{vmatrix} y_0 & x_0 - x_1 \\ y_1 & 0 \end{vmatrix}}{x_1 - x_0} = y_1$$

Since $f^*_{0,1}(x)$ goes through the points (x_0,y_0) and (x_1,y_1) and is of the first degree, it is the required (unique) polynomial. Hence, for a given number x', to find $f^*_{0,1}(x')$ we just calculate Eq. (a) with this number x' substituted for the symbol x.

The polynomial that goes through (x_0,y_0), (x_1,y_1), and (x_2,y_2) is given by

$$f^*_{0,1,2}(x) = \frac{\begin{vmatrix} f^*_{0,1}(x) & x_1 - x \\ f^*_{0,2}(x) & x_2 - x \end{vmatrix}}{x_2 - x_1}$$

where $f^*_{0,2}(x)$ is the polynomial that goes through (x_0,y_0) and (x_2,y_2).

For $f^*_{0,1,2,3}(x)$ we have similarly

$$f^*_{0,1,2,3}(x) = \frac{\begin{vmatrix} f^*_{0,1,2}(x) & x_2 - x \\ f^*_{0,1,3}(x) & x_3 - x \end{vmatrix}}{x_3 - x_2}$$

The process can be extended in this way. For example,

$$f^*_{0,1,2,3,4,5,6}(x) = \frac{\begin{vmatrix} f^*_{0,1,2,3,4,5}(x) & x_5 - x \\ f^*_{0,1,2,3,4,6}(x) & x_6 - x \end{vmatrix}}{x_6 - x_5}$$

Hence the process can be systematized, with $f^*(x)$ being recomputed, including more and more points until the desired accuracy is obtained (assuming that there are enough points in the table). The following scheme can be used for performing the calculations, the terms in each

† See W. E. Milne, "Numerical Calculus," pp. 68–72, Princeton University Press, Princeton, N.J., 1949.

row being determined in order:

$$\begin{tikzcd}
x_0,\, y_0 & (x_0 - x) \arrow[dr] \arrow[ddr] \arrow[dddr] \arrow[ddddr] & & & & \\
x_1,\, y_1 & (x_1 - x) \arrow[r] & f^*_{0,1}(x) \arrow[dr] \arrow[ddr] \arrow[dddr] & & & \\
x_2,\, y_2 & (x_2 - x) \arrow[r] & f^*_{0,2}(x) \arrow[r] & f^*_{0,1,2}(x) \arrow[dr] \arrow[ddr] & & \\
x_3,\, y_3 & (x_3 - x) \arrow[r] & f^*_{0,3}(x) \arrow[r] & f^*_{0,1,3}(x) \arrow[r] & f^*_{0,1,2,3}(x) \arrow[dr] & \\
x_4,\, y_4 & (x_4 - x) \arrow[r] & f^*_{0,4}(x) \arrow[r] & f^*_{0,1,4}(x) \arrow[r] & f^*_{0,1,2,4}(x) \arrow[r] & f^*_{0,1,2,3,4}(x)
\end{tikzcd}$$

EXERCISES

(*a*) Show that $f^*_{0,1,2,3}(x)$ actually does pass through all the points (x_0,y_0), (x_1,y_1), (x_2,y_2), (x_3,y_3) and is of degree 3.

(*b*) Show that

$$f^*_{0,1,2}(x) = \frac{\begin{vmatrix} f^*_{0,1}(x) & x_0 - x \\ f^*_{1,2}(x) & x_2 - x \end{vmatrix}}{x_2 - x_0} = \frac{\begin{vmatrix} f^*_{0,2}(x) & x_0 - x \\ f^*_{1,2}(x) & x_1 - x \end{vmatrix}}{x_1 - x_0}$$

(*c*) Compute arctan 0.9, where arctan x is given by a table for values of x differing by 0.4, as shown:

x_i	arctan x_i			*Solution*			
$x_0 = -1.0$	-0.7854	-1.9					
$x_1 = -0.6$	-0.5404	-1.5	0.3783				
$x_2 = -0.2$	-0.1974	-1.1	0.6111	1.2515			
$x_3 = +0.2$	$+0.1974$	-0.7	0.7707	1.1141	0.8736		
$x_4 = +0.6$	$+0.5404$	-0.3	0.7890	0.8917	0.7568	0.6691	
$x_5 = +1.0$	$+0.7854$	$+0.1$	0.7068	0.6863	0.7334	0.7509	0.7305

(*d*) Compute arctan 0.75 from the table values given in the previous exercise.

6-6. Function Evaluation: Best-fit Polynomial Approximation

Polynomial Approximations. As we have noted in Sec. 6-4, the evaluation of $f(x)$ by means of terms up to the x^n term of its power-series expansion can be considered as approximating $f(x)$ by a polynomial of degree n, where the coefficients are given by Taylor's formula. Similarly the method of interpolation can also be interpreted as the approximation of $f(x)$ by a polynomial, even though the polynomial may never be explicitly determined by the method given; the coefficients of the interpolation polynomial of degree n are determined from $n + 1$ known values of $f(x)$. For arctan x we have the power-series approximation

$$\arctan^* x = 1.000000x - 0.333333x^3 + 0.200000x^5$$

For the interpolation approximation (for the points -1, -0.6, -0.2, $+0.2$, $+0.6$, and 1 given in Sec. 6-5) we have

$$\arctan^* x = 0.999105x - 0.306897x^3 + 0.093190x^5$$

In order to show how good these approximations are, in Fig. 6-10 have been drawn the error functions

$$\epsilon(x) = \text{approximation} - \text{function} = \arctan^* x - \arctan x$$

for the range $0 \leq x \leq 1$, for both cases.

Best-fit Polynomials. The question immediately arises: Does there exist another fifth-degree polynomial that is a "better fit" to arctan x

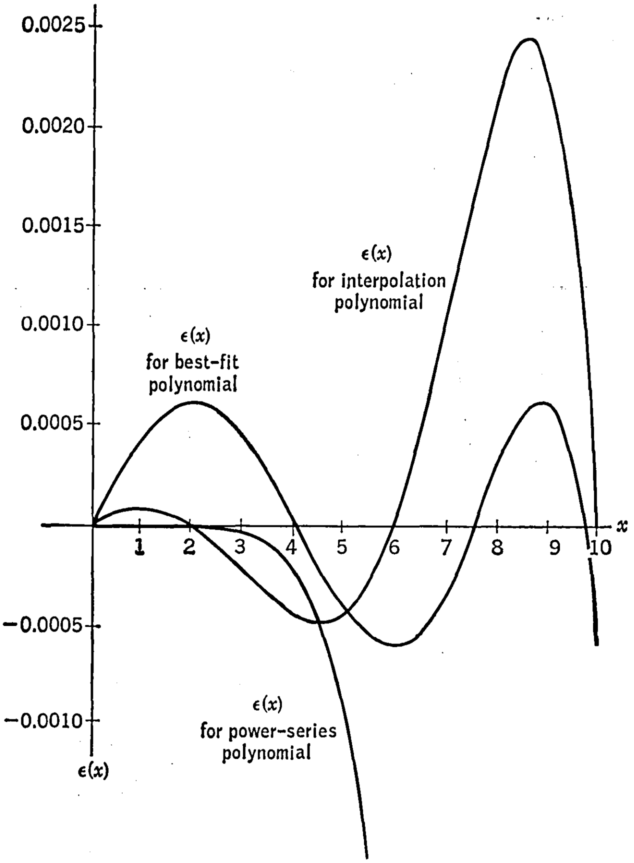


FIG. 6-10. A comparison, by means of error curves, of three fifth-degree polynomial approximations to arctan x for $0 \leq x \leq 1$.

1. Power-series approximation:

$$\arctan^* x = 1.000000x - 0.333333x^3 + 0.200000x^5$$

2. Interpolation approximation on the points −1, −0.6, −0.2, +0.2, +0.6, and 1:

$$\arctan^* x = 0.999105x - 0.306897x^3 + 0.093190x^5$$

3. Best-fit approximation:

$$\arctan^* x = 0.995354x - 0.288679x^3 + 0.079331x^5$$

for $0 \leq x \leq 1$? We say that $f_1^*(x)$ is a *better fit* to $f(x)$ than $f_2^*(x)$ for a given range of x if within that range max $|\epsilon_1(x)| <$ max $|\epsilon_2(x)|$. In general the answer to the above question is "yes." For arctan x the best-fit polynomial for $0 \leq x \leq 1$ is given by

$$\arctan^* x = 0.995354x - 0.288679x^3 + 0.079331x^5$$

Figure 6-10 also shows $\epsilon(x)$ for this approximation. How one arrives at such best-fit approximations is beyond the scope of this book.†

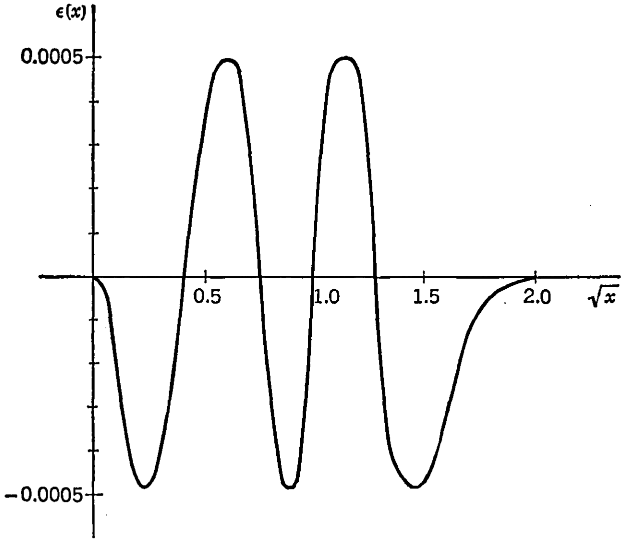


FIG. 6-11. Error curve for $f(x) = \frac{2}{\sqrt{\pi}} \int_0^x e^{-t^2}\,dt$.

The advantage of using a best-fit approximation over a Taylor-series approximation or interpolation is easily seen: fewer computations are required to obtain the same accuracy than for a Taylor series; and no tables need be memorized in the computer, as is necessary for interpolation. Hence time and space are saved in the computation. Although we have illustrated a best-fit polynomial that is good to within ±0.0005, best-fit polynomials can be formed that will give any desired accuracy. For example, a best-fit polynomial for arctan x, for $0 \leq x \leq 1$, which is good to within ±0.0000 01, is given by

$$\arctan^* x = 0.9999\ 7726\ x - 0.3326\ 2347\ x^3 + 0.1935\ 4346\ x^5 \\ - 0.1164\ 3287\ x^7 + 0.0526\ 5332\ x^9 - 0.0117\ 2120\ x^{11}$$

Best-fit approximations need not be of the forms shown for arctan x but should be of some relatively simple algebraic form. For example,

† For such a discussion see Cecil Hastings, Jr., "Approximations for Digital Computers," Princeton University Press, Princeton, N.J., 1955.

for $0 \leq x \leq \infty$, a best-fit approximation for

$$f(x) = \frac{2}{\sqrt{\pi}} \int_0^x e^{-t^2}\, dt$$

is given by

$$f^*(x) = 1 - \frac{1}{(1 - a_1x + a_2x^2 + a_3x^3 + a_4x^4)^4}$$

where $a_1 = 0.278393$, $a_2 = 0.230389$, $a_3 = 0.000972$, and $a_4 = 0.078108$. In this case $\epsilon(x)$ is given in Fig. 6-11, plotted against $\sqrt{x}$. This $f^*(x)$ is easily coded for a computer.

EXERCISES

(*a*, *b*). Write a code to evaluate each of the best-fit approximations to the arctan x given in this section.

6-7. Integration

By Definition. In this section we shall see how a computer can integrate. In his previous studies the student has been taught to integrate by means of the manipulation of abstract symbols. However, a little reflection will show that it is really numbers being handled, these numbers being represented for convenience by symbols. Of course a computer cannot directly manipulate symbols in the ordinary sense, but it can handle numbers. As we shall see, the simplest method of handling numbers to obtain an integral is to follow the very definition of the meaning of integration.

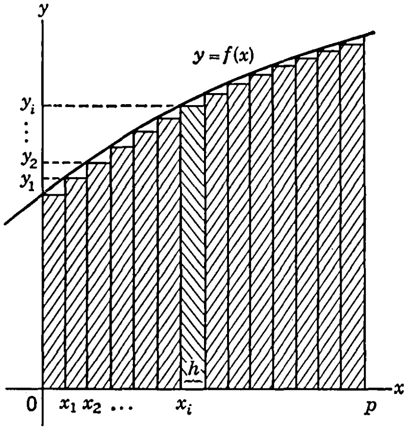


FIG. 6-12. Numerical integration.

Recall now the definition of integration (see Fig. 6-12). The integral of $f(x)$ from $x = 0$ to $x = p$, written $\int_0^p f(x)\, dx$, is the area between the curve $y = f(x)$ and the x axis, bounded by $x = 0$ and $x = p$. It is found by considering the rectangles formed by partitioning the x axis between 0 and p into n intervals each of the same length h, where $h = (p - 0)/n$. If there are n intervals each of length h, the area of the ith rectangle will be hy_i and the sum of the areas of all the rectangles will be

$$A = \sum_{i=1}^{n} hy_i$$

In another partition that has more intervals the length of each will be less. Hence, as the number of intervals becomes infinite, the length of each approaches 0. The integral $\int_0^p y\,dx$ is defined as the limit of the sum of the areas of the rectangles as the number of intervals increases and the length of each approaches zero,

$$\int_0^p y\,dx = \lim_{\substack{n\to\infty \\ h\to 0}} \sum_{i=1}^{n} hy_i$$

To integrate on a computer, we could just form $\sum_{i=1}^{n} hy_i$. Then we might take a smaller interval, say $h/2$, and form $\sum_{i=1}^{2n} (h/2)y_i$. If

$$\left| \sum_{i=1}^{2n} \frac{h}{2} y_i - \sum_{i=1}^{n} hy_i \right|$$

is less than the allowable error desired for this integration, then the integration is completed. If not, we could try intervals of $h/4$, and so on. This process naturally assumes that we have a subroutine to evaluate $y_1 = f(x_i)$ for each x_i under consideration.

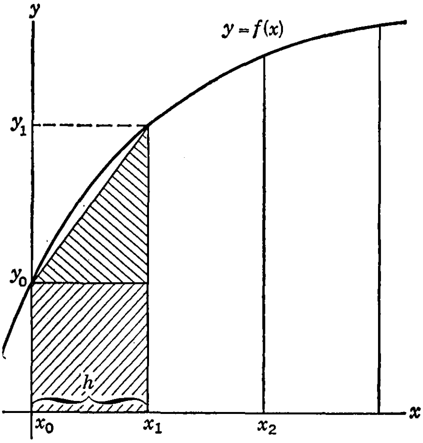


FIG. 6-13. Trapezoidal approximation to integration.

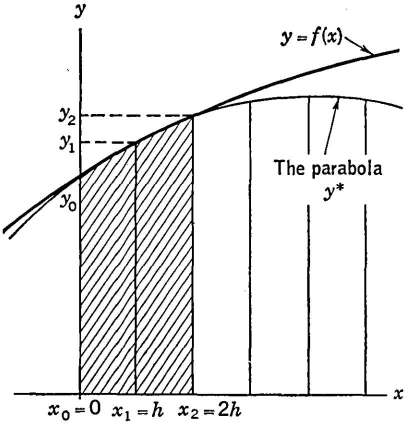


FIG. 6-14. Parabolic approximation to integration.

By the Trapezoid Rule. Although the above method is a good way for evaluating an integral, there are methods that may require less computation. Suppose that we considered the trapezoids formed by adding little triangles to the rectangles, as shown in Fig. 6-13. Here the area of the trapezoid is $h[y_0 + \frac{1}{2}(y_1 - y_0)] = (h/2)(y_0 + y_1)$. For n trapezoids we have $A = (h/2)(y_0 + 2y_1 + 2y_2 + \cdots + y_n)$. Certainly this will be a better initial approximation to the area under the curve than was $A = h(y_0 + y_1 + \cdots + y_n)$. Hence, using this method, we shall not need to go to as small intervals to obtain the desired accuracy as we did using the previous, cruder method. The method given in this paragraph is called the *trapezoid rule.*

By Polynomial Approximations. An even better initial approximation can be obtained by considering three points at a time instead of two. Suppose that we consider the area under the parabola $y^* = a + bx + cx^2$ that passes through three points, as shown in Fig. 6-14. Then

$$\begin{aligned} A_{0,2} &= \int_0^{2h} (a + bx + cx^2)\,dx = ax + \frac{bx^2}{2} + \frac{cx^3}{3}\bigg]_0^{2h} \\ &= 2h \cdot a + 2h^2 \cdot b + \tfrac{8}{3}h^3 \cdot c \end{aligned}$$

To determine the coefficients a, b, and c, we note that

$$\begin{array}{ll} \text{If } x = 0, & a + 0 \cdot b + 0 \cdot c = y_0 \\ \text{If } x = h, & a + h \cdot b + h^2 \cdot c = y_1 \\ \text{If } x = 2h, & a + 2h \cdot b + 4h^2 \cdot c = y_2 \end{array}$$

These, when solved for a, b, and c, yield

$$a = y_0 \qquad b = \frac{-y_2 + 4y_1 - 3y_0}{2h} \qquad c = \frac{y_2 - 2y_1 + y_0}{2h^2}$$

whence
$$A_{0,2} = \frac{h}{3}(y_0 + 4y_1 + y_2)$$

Similarly $A_{2,4} = (h/3)(y_2 + 4y_3 + y_4)$, and so forth, whence, if n is even, we have

$$A = \frac{h}{3}(y_0 + 4y_1 + 2y_2 + 4y_3 + 2y_4 + 4y_5 + \cdots + 4y_{n-1} + y_n)$$

This is known as *Simpson's rule.*†

Similarly we could try to make even better initial approximations to the area under the curve. Once the method of approximation is decided upon, more accurate results can be obtained only by increasing n, the number of intervals—as can be seen from the original definition of the integral.

We have been assuming until now that the function $y(x)$ can be evaluated at any point x by a subroutine. Suppose, on the other hand, that the function $f(x)$ were given by a table of points, spaced at equal intervals x_i. Then the above procedures would of course still hold. But these methods will evaluate integrals only between points that are given in the table, and not to other points. However, one might choose simply to integrate the approximating polynomial (determined as in the

† See, for example, Milne, *op. cit.*, pp. 116, 120.

previous sections) and then to evaluate this function between any two points, whether they are or are not given in the table.

EXERCISES

The following table gives the values of arctan x for equally spaced points:

x_i	*arctan* x_i
$x_0 = 0$	0.00000
$x_1 = 0.2$	0.19739
$x_2 = 0.4$	0.38051
$x_3 = 0.6$	0.54042
$x_4 = 0.8$	0.67474
$x_5 = 1.0$	0.78534

(*a*) Evaluate $\int_0^{0.8}$ arctan $x\,dx$ by the rectangular rule, first for partition at x_0, x_2, and x_4, and then for partition at x_0, x_1, x_2, x_3, and x_4; compare the results.

(*b*) Evaluate $\int_0^{0.8}$ arctan $x\,dx$ by the trapezoidal rule, first for partition at x_0, x_2, and x_4, and then for partition at x_0, x_1, x_2, x_3, and x_4; compare the results.

(*c*) Evaluate $\int_0^{0.8}$ arctan $x\,dx$ by Simpson's rule, first for partition at x_0, x_2, and x_4, and then for partition at x_0, x_1, x_2, x_3, and x_4; compare the results.

(*d*) Integrate each of the three polynomial approximations for the arctan x given in the previous section; evaluate each of these integrals from 0 to 0.8, and compare the results of each with the results of Exercises *a* to *c*.

(*e*) Observing that (the indefinite integral)

$$\int \arctan x\,dx = x \arctan x - \tfrac{1}{2} \ln (1 + x^2)$$

find $\int_0^{0.8}$ arctan $x\,dx$; compare this result with those of Exercises *a* to *d*.

(*f*) Write a code for evaluating Simpson's rule, assuming that $f(x)$ is evaluated by a subroutine.

6-8. Differentiation

By Definition. The derivative of $y = f(x)$ with respect to x, taken at $x = x_0$, is defined by

$$\left(\frac{dy}{dx}\right)_{x=x_0} = \lim_{\Delta x \to 0} \frac{f(x_0 + \Delta x) - f(x_0)}{\Delta x}$$

and represents the slope of the tangent drawn to the curve $y = f(x)$ at x_0. Hence, if $f(x)$ is given at $x = x_0$ and $x = x_1$, an approximation of the derivative might be

$$\frac{f(x_1) - f(x_0)}{x_1 - x_0}$$

Unfortunately it is not always feasible with computers to take x_1 successively closer to x_0 and evaluate the fraction each time, for $x_1 - x_0$

becomes very small, as does $f(x_1) - f(x_0)$, and often for a very small $x_1 - x_0$ the loss of significant figures throws the calculation far off.

By Polynomial Approximation. Suppose that $f(x)$ is given at $x_0 = 0$, $x_1 = h$, and $x_2 = 2h$. Analogous to the process described for integration, we might determine as an approximation the derivative of the parabola that goes through these three points. Let $y = a + bx + cx^2$, where a, b, and c are as in Sec. 6-7. Hence

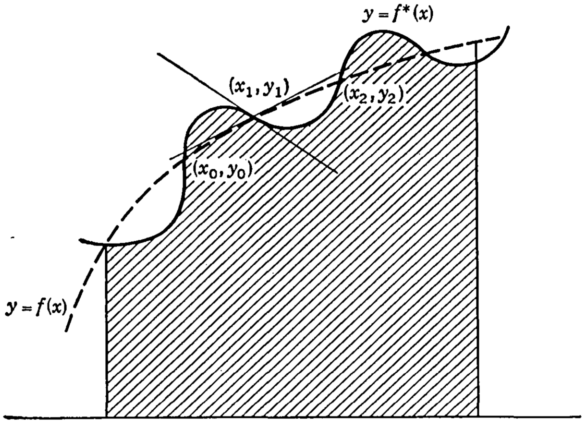


FIG. 6-15. Approximating the derivative from an oscillating polynomial.

$$\frac{dy}{dx} = b + 2cx = \frac{-y_2 + 4y_1 - 3y_0}{2h} + \frac{y_2 - 2y_1 + y_0}{h^2}\,x$$

Then for the derivatives taken at $x = x_0$, $x = x_1$, and $x = x_2$, respectively, we have

$$\left(\frac{dy}{dx}\right)_{\substack{x=0\\y=y_0}} = \frac{1}{2h}(-3y_0 + 4y_1 - y_2)$$

$$\left(\frac{dy}{dx}\right)_{\substack{x=h\\y=y_1}} = \frac{1}{2h}(-y_0 + y_2)$$

$$\left(\frac{dy}{dx}\right)_{\substack{x=2h\\y=y_2}} = \frac{1}{2h}(y_0 - 4y_1 + 3y_2)$$

The formulas are equally valid for any three points (x_0,y_0), (x_1,y_1), and (x_2,y_2) equally spaced on the x axis, since translating the curve so that $x_0 = 0$ does not affect its slope.† If the function is given by means of a table, then these rules can be used for evaluating derivatives at points given in the table.

Of course there are always the approximating polynomials of $f(x)$, which

† See, for example, *ibid.*, p. 96.

can easily be differentiated, the resulting derivative being evaluated at the requisite points.

However, numerical differentiation at best is often quite inaccurate. For these methods must be based on approximating polynomials that depend on values of the function at a few specific points. There is no practical way for the computer to tell how the approximating polynomial oscillates around the given function. Hence the slope of the polynomial at a point may be entirely different from the slope of the function, even though the curves lie very close, or even touch (see Fig. 6-15). On the other hand the value of the area under the curve is seldom so sensitive to these oscillations.

EXERCISES

Use the table of values of arctan x as given for the Exercises of Sec. 6-7.

(*a*) Determine the derivative of arctan x at $x = 0.6$, using first Δx between the two points at $x = 0.4$ and $x = 0.6$, and then between the points at $x = 0.6$ and $x = 0.8$.

(*b*) Determine the derivative of arctan x at $x = 0.6$ by using the three points first at $x_0 = 0.6$, $x_1 = 0.8$, and $x_2 = 1.0$, then at $x_0 = 0.4$, $x_1 = 0.6$, and $x_2 = 0.8$, and finally at $x_0 = 0.2$, $x_1 = 0.4$, and $x_2 = 0.6$.

(*c*) Differentiate each of the three approximating polynomials for arctan x given in Sec. 6-6, and in each case evaluate the derivative at $x = 0.6$.

(*d*) Using the form

$$\frac{d}{dx} \arctan x = \frac{1}{1 + x^2}$$

evaluate the derivative of arctan x at $x = 0.6$, and compare this with the approximations of (*a*), (*b*), and (*c*).

6-9. Undetermined Coefficients

The Method. As was shown in the previous two sections, formulas for the integral or derivative of a function can be obtained by integrating or differentiating the polynomial approximation functions that pass through some given points. However, in these sections formulas were specifically derived only where three points were fitted. Formulas based on more points can be derived by the method of undetermined coefficients,† which follows. We shall develop the method by means of illustrations.

Suppose that we wish to rederive the formula of Sec. 6-7, Simpson's rule, and find an approximation of $\int_{x_0}^{x_2} y\,dx$ based on the polynomial $y = f^*(x)$ of the second degree passing through the three points (x_0,y_0), (x_1,y_1), (x_2,y_2) with equally spaced abscissas. Since the value of an integral does not change if it is translated along the x axis, it will prove helpful to take advantage of symmetry, by letting the x coordinates of these points be $x_0 = -h$, $x_1 = 0$, $x_2 = +h$. Now let

$$A_0y_0 + A_1y_1 + A_2y_2 = \int_{x_0}^{x_2} y\,dx$$

† See *ibid.*, p. 104.

where A_0, A_1, and A_2 are the as yet undetermined coefficients. This formula must be *exact* (i.e., not an approximation) for $y = f(x)$, when $f(x)$ is any *polynomial of degree* 2 *or less*, because there is a uniquely determined polynomial of degree not more than n that passes through $n + 1$ points. Hence this formula must be true for the functions $y = 1$, $y = x$, and $y = x^2$. Substituting these three functions successively in the equation, we find

$$\begin{aligned} A_0 + \quad A_1 + \quad A_2 &= 2h \\ -hA_0 + 0 \cdot A_1 + hA_2 &= 0 \\ h^2A_0 + 0 \cdot A_1 + h^2A_2 &= \tfrac{2}{3}h^3 \end{aligned}$$

Solving for A_0, A_1, and A_2, we find

$$A_0 = A_2 = \tfrac{1}{3}h \qquad \text{and} \qquad A_1 = \tfrac{4}{3}h$$

Hence
$$\int_{x_0}^{x_2} y\,dx = \frac{h}{3}(y_0 + 4y_1 + y_2)$$

as desired.

Suppose that we now desire to find a formula for $(dy/dx)_{x=x_1}$ based on four points with equally spaced abscissas, (x_0,y_0), (x_1,y_1), (x_2,y_2), and (x_3,y_3). Let $x_0 = -h$, $x_1 = 0$, $x_2 = h$, $x_3 = 2h$,

and
$$B_0y_0 + B_1y_1 + B_2y_2 + B_3y_3 = \left(\frac{dy}{dx}\right)_{x=x_1}$$

where B_0, B_1, B_2, and B_3 are the undetermined coefficients. This formula must be *exact* when y is any polynomial of degree not more than 3, and hence for the polynomials $y = 1$, $y = x$, $y = x^2$, and $y = x^3$. Substituting these successively into the equation, we find

$$\begin{aligned} B_0 + \quad B_1 + \quad B_2 + \quad B_3 &= 0 \\ -hB_0 + 0 \cdot B_1 + hB_2 + 2hB_3 &= 1 \\ h^2B_0 + 0 \cdot B_1 + h^2B_2 + 4h^2B_3 &= 0 \\ -h^3B_0 + 0 \cdot B_1 + h^3B_2 + 8h^3B_3 &= 0 \end{aligned}$$

These, when solved for B_0, B_1, B_2, and B_3, give

$$B_0 = -\tfrac{1}{3}h \qquad B_1 = -\tfrac{1}{2}h \qquad B_2 = \frac{1}{h} \qquad \text{and} \qquad B_3 = -\tfrac{1}{6}h$$

whence
$$\left(\frac{dy}{dx}\right)_{x=x_1} = \tfrac{1}{6}h(-2y_0 - 3y_1 + 6y_2 - y_3)$$

EXERCISES

(*a*) Show that, if (x_0,y_0), (x_1,y_1), (x_2,y_2), and (x_3,y_3) are four points with equally spaced abscissas, then

$$\int_{x_1}^{x_2} y\,dx = \frac{h}{24}(-y_0 + 13y_1 + 13y_2 - y_3)$$

(HINT: Let $x_0 = -\frac{3}{2}h$, $x_1 = -h/2$, $x_2 = h/2$, and $x_3 = \frac{3}{2}h$; let the three functions be $y = 1$, $y = x$, $y = x^2$, and $y = x^3$.)

(*b*) If five points with equally spaced abscissas are given, (x_0,y_0), (x_1,y_1), (x_2,y_2), (x_3,y_3), and (x_4,y_4), then find $\int_{x_1}^{x_3} y\,dx$.

(*c*) Derive the formulas given in Sec. 6-8 by the method of undetermined coefficients.

(*d*) If four points with equally spaced abscissas, (x_0,y_0), (x_1,y_1), (x_2,y_2), and (x_3,y_3), are given, find formulas for

$$\left(\frac{dy}{dx}\right)_{x=x_0} \quad \left(\frac{dy}{dx}\right)_{x=x_1} \quad \left(\frac{dy}{dx}\right)_{x=x_2} \quad \left(\frac{dy}{dx}\right)_{x=x_3}$$

(*e*) Find the formula for $\int_{x_0}^{x_1} y\,dx$ in terms of the two points (x_0,y_0) and (x_1,y_1), and the derivatives at these points,

$$y_0' = \left(\frac{dy}{dx}\right)_{x=x_0} \quad \text{and} \quad y_1' = \left(\frac{dy}{dx}\right)_{x=x_1}$$

(HINT: Let $x_0 = 0$, $x_1 = h$; let

$$\int_{x_0}^{x_1} y\,dx = A_0y_0 + A_1y_1 + B_0y_0' + B_1y_1'$$

where A_0, A_1, B_0, and B_1 are undetermined coefficients. Now, when the functions $y = 1$, $y = x$, $y = x^2$, and $y = x^3$ are substituted successively into the equation, there result four simultaneous equations that determine A_0, A_1, B_0, and B_1.)

6-10. Differential Equations

Meaning of Solution. A differential equation describes how a system goes from a known point to the next infinitesimally close point. For example, the well-known equation $dv/dt = F/m$ describes the relationship of the velocity v of the mass m to the time t, when a force F is applied. Given a point (t_0,v_0), the equation says that at an infinitesimally short time later, say at t_1, the velocity will be $v_1 = (F/m)(t_1 - t_0) + v_0$. Such equations arise because in nature relationships between physical quantities often reduce to linear functions at the infinitesimal level.

Consider, for example, a differential equation of the first order in the form

$$\frac{dy}{dx} = f(x,y)$$

The solution is to be determined as a function of x only, that is, $y = g(x)$. Of course $y = g(x)$ can be represented as a curve, and the differential equation tells us how to draw this curve. One point on the curve, say (x_0,y_0), must be known or given. Then the differential equation says that

$$\left(\frac{dy}{dx}\right)_{x=x_0} = f(x_0,y_0)$$

and hence we know the slope at (x_0,y_0). A linear approximation to the next point might be obtained (see Fig. 6-16) by continuing along the tangent line, i.e., by choosing an x_1 and determining y_1 from

$$y_1 = \left(\frac{dy}{dx}\right)_{x=x_0} (x_1 - x_0) + y_0$$

Using (x_1,y_1), we find that $(dy/dx)_{x=x_1} = f(x_1,y_1)$ and then, choosing an x_2, determine y_2 from

$$y_2 = \left(\frac{dy}{dx}\right)_{x=x_1} (x_2 - x_1) + y_1$$

and in this manner (x_3,y_3), . . . are determined and the function $y = g(x)$ is approximated. Of course this is just an approximation, even if we make $x_1 - x_0$, $x_2 - x_1$, $x_3 - x_2$, . . . small, for the differential equation taken literally implies that $x_1 - x_0$, $x_2 - x_1$, . . . should be infinitesimal. What we are essentially saying by this method is that a straight line through (x_0,y_0) is a good approximation to $g(x)$ near (x_0,y_0) and therefore the next point (x_1,y_1) ought to lie very close to it.

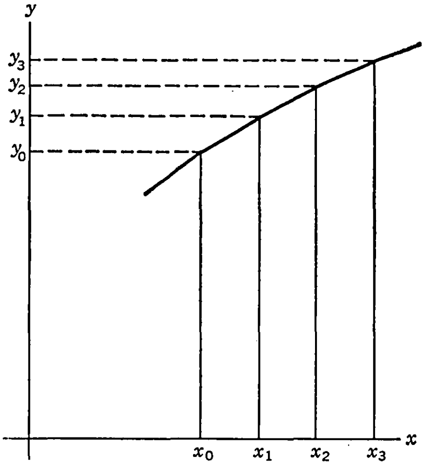


FIG. 6-16. Linear approximation from point to point along a curve.

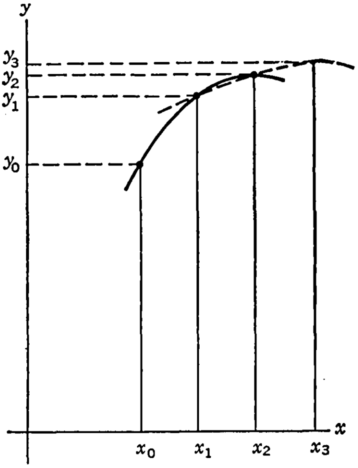


FIG. 6-17. Polynomial approximation from point to point along a curve.

By Polynomial Approximation. Consider again a first-order differential equation $dy/dx = f(x,y)$. A better approximation to $y = g(x)$ might be obtained if we started with two points, (x_0,y_0) and (x_1,y_1). Then we could use a parabola as the approximating curve of $g(x)$ near (x_0,y_0) and (x_1,y_1), to determine (x_2,y_2) (see Fig. 6-17). Take, for example, the second formula for the parabolic approximation of Sec. 6-8,

$$\left(\frac{dy}{dx}\right)_{x=x_1} = \frac{1}{2h}(-y_0 + y_2)$$

From this

$$y_2 = 2h\left(\frac{dy}{dx}\right)_{x=x_1} + y_0$$

Hence if we take $x_1 = x_0 + h$, $x_2 = x_0 + 2h$, etc., and if we know (x_0,y_0) and (x_1,y_1), then we can determine $(dy/dx)_{x=x_1} = f(x_1,y_1)$, and from this y_2. Our equation from Sec. 6-8 can be written in general as

$$y_{n+1} = 2h\left(\frac{dy}{dx}\right)_{x=x_n} + y_{n-1}$$

Thence we find y_3 for $x_3 = x_0 + 3h$, using $(dy/dx)_{x=2} = f(x_2,y_2)$, and so forth. This process is clear, provided that initially we know (x_1,y_1) as well as (x_0,y_0). Given (x_0,y_0), we can find (x_1,y_1) by forming a Taylor-series expansion around (x_0,y_0). Since we already know $dy/dx = f(x,y)$, then

$$\frac{d^2y}{dx^2} = \frac{d}{dx}f(x,y) \qquad \frac{d^3y}{dx^3} = \frac{d^2}{dx^2}f(x,y) \qquad \text{etc.}$$

for the coefficients, and hence (x_1,y_1) is easily found,

$$y_1 = y_0 + (x_1 - x_0)\left(\frac{dy}{dx}\right)_{x=x_0} + \frac{(x_1 - x_0)^2}{2!}\left(\frac{d^2y}{dx^2}\right)_{x=x_0} + \frac{(x_1 - x_0)^3}{3!}\left(\frac{d^3y}{dx^3}\right)_{x=x_0} + \cdots$$

where the first few terms are usually sufficient.

Similarly higher-degree approximation polynomials can be used to estimate successive points. For example, a cubic might be used that is based on

$$(x_0,y_0) \qquad (x_1,y_1) \qquad \left(\frac{dy}{dx}\right)_{x=x_0} \qquad \text{and} \qquad \left(\frac{dy}{dx}\right)_{x=x_1}$$

Such formulas can be derived by means of the method of undetermined coefficients.

Example. As an example, consider the differential equation

$$\frac{dy}{dx} = 1 + y^2$$

Suppose that, at $x_0 = 0$, $y_0 = 0$. Choose $x_1 = 0.1$, $x_2 = 0.2$, $x_3 = 0.3$, etc. Then, by our first, linear method,

$$y_1 = \left(\frac{dy}{dx}\right)_{x=x_0}(x_1 - x_0) + y_0 = (1 + y_0^2)(x_1 - x_0) + y_0$$
$$= (1 + 0)(0.1 - 0) + 0 = 0.1$$

$$y_2 = \left(\frac{dy}{dx}\right)_{x=x_1}(x_2 - x_1) + y_1 = (1 + y_1^2)(x_2 - x_1) + y_1$$
$$= 1.01 \times 0.1 + 0.1 = 0.201$$

$$y_3 = \left(\frac{dy}{dx}\right)_{x=x_2}(x_3 - x_2) + y_2 = (1 + y_2^2)(x_3 - x_2) + y_2 = 0.3050$$

and so forth.

Next consider the quadratic method. We must first find y_1 for $x_1 = 0.1$. Taking the first three terms of the Taylor expansion,

$$\begin{aligned} y_1 &= y_0 + (x_1 - x_0)(1 + y_0^2) + \frac{(x_1 - x_0)^2}{2!} 2y_0(1 + y_0^2) \\ &\qquad + \frac{(x_1 - x_0)^3}{3!} [2(1 + y_0^2)^2 + 4y_0^2(1 + y_0^2)] \\ &= 0 + 0.1 + 0 + 0.0003 \\ &= 0.1003 \end{aligned}$$

Then

$$y_2 = 0.2\left(\frac{dy}{dx}\right)_{x=x_1} + y_0 = 0.2 \times (1 + y_1^2) + y_0 = 0.2 \times 1.0101 + 0$$
$$= 0.2020$$

$$y_3 = 0.2\left(\frac{dy}{dx}\right)_{x=x_2} + y_1 = 0.2 \times (1 + y_2^2) + y_1 = 0.2 \times 1.0408$$
$$+ 0.1003 = 0.3085$$

and so forth. The exact solution to $dy/dx = 1 + y^2$ is $y = \tan x$. With greater accuracy, we have $y_0 = \tan 0 = 0$, $y_1 = \tan 0.1 = 0.1021$, $y_2 = \tan 0.2 = 0.2035$, and $y_3 = \tan 0.3 = 0.312$.

Since in the numerical solution of a differential equation each point depends on previous points, errors are cumulative. Hence the only way to get a better solution is to take the interval h smaller for a particular method of approximation.

EXERCISES

(*a*) Solve $dy/dx = 1 + y^2$ for $h = 0.01$ and $h = 0.001$ by both methods, and compare with a table of tan x.

(*b*) Solve $dy/dx = \sqrt{1 - y^2}$ by both methods, with intervals $h = 0.1$ and 0.01, where $x_0 = 0$, $y_0 = 0$. Compare the solutions with a table of sin x.

6-11. Accuracy and Error

Sources of Error. We can distinguish among three contributions to the error of a numerical computation:

1. First is the error due to the approximate nature of the numerical analysis of the computations. As we have seen in previous sections, frequently the value of a function or the solution to an equation can theoretically be obtained only by performing an infinite sequence of iterations. Of course the infinite process cannot be completed, and so we must stop with some iteration in the sequence, accepting this as an adequate approximation to the desired results. The resulting error is called a *truncated error* and often can be made as small as we please. *The problem here then is to estimate the accuracy of the result after an iteration to determine whether or not it is as precise as required, i.e., whether or not more iterations should be made.*

2. Second is the error due to the approximate nature of experimental data used in the calculation; this is called the *experimental error. Here the problem is so to analyze the experimental data as to obtain the best possible estimate of the measurement involved,* i.e., to determine the accuracy of the experimentally obtained numbers. The accuracy of such numbers depends on aspects other than mathematical considerations, and therefore from a computational point of view these errors are treated as initial constraints on the problem.

3. Third is the error due to the use of approximate numbers in performing the numerical calculations; such errors are called *round-off errors.* Any number that we write is necessarily in finite decimal (or binary) form, which is only an approximation to its infinite decimal conception.† For example, π might be approximated by 3.1415927. Numbers can be approximated to any degree of accuracy: e.g., a better approximation to π is 3.14159 26535 89793. Experimental numbers may also be considered to be finite approximations of some (unknown) infinite decimal. Here of course we cannot get any better approximation than those determined by methods associated with the above-discussed second source of error. *Thus the problem here is to perform the four arithmetic operations in a manner that retains the maximum desired accuracy under the circumstances.*

Consideration of truncated errors forms a large subject in itself. For the various iterative formulas given in the previous sections, formulas can be obtained for the errors, or for bounds on the errors, after any particular iteration. For example, the remainder formula for a Taylor series is familiar to the student as a measure of the error of approximation of a finite number of terms of a power series. Textbooks on numerical analysis consider these formulas in detail; this subject is, however, beyond the scope of the present brief treatment. Consideration of experimental errors is also beyond the scope of this treatment. (See Sec. 6-12, Additional Topics *b* and *c*.)

This section then considers the last of these three problems, the round-off errors. The importance of this subject to numerical calculators cannot be overestimated. During a computation it is not an overstatement to say that consideration of errors that may arise from arithmetic operations should be uppermost in the mind of the user of the computer. The discussions of this section merely form the basis for such error considerations in a particular computation. Very often it is necessary for the user of the computer to determine experimentally the behavior of his numbers during the computations, by having the computer perform series of trial computations. These experimental computations can be used to determine the correct operational procedures.

Significant Figures and Round Numbers. The accuracy of a number is measured in terms of the number of significant figures it contains. Any one of the digits 1, 2, 3, 4, 5, 6, 7, 8, 9 is a significant figure; 0 is some-

† Here we consider, for example, the number 0.5000 · · · to be an infinite decimal.

times but not always a significant figure. To determine when 0 is a significant figure, we distinguish three cases:

1. The zero lies to the left of all other nonzero digits of the number, e.g., as in 0.00123. Here the zero is used simply to fix the decimal point, and therefore it is *not* considered to be significant. None of the three zeros in 0.00123 is significant.

2. The zero or zeros lie between nonzero digits, e.g., as in 3,002. Here the zeros are considered to be significant.

3. The zero lies to the right of all other nonzero digits, e.g., 123,000. In this case it cannot be determined from the number itself whether or not the zero is significant; it must be determined from the context in which it was written. Often, if the zeros in 123,000 are not significant, we would write 123 × 10^3, while on the other hand if the first zero were significant we would write 1,230 × 10^2 to indicate this.

For example, the *number of significant figures* in each of the following numbers is four: 1,234, 9,072, 5,006, 0.008379, 0.01024, 6,080 × 10^3, 7,000 × 10^3. The leftmost significant figure in a number is called its *most significant figure;* in our examples the most significant figures are, respectively, 1, 9, 5, 8, 1, 6, and 7. The *least significant figure* is the rightmost significant figure of a number; in our examples the least significant figures are, respectively, 4, 2, 6, 9, 4, 0, and 0.

A number composed of n significant figures is said to be *correct to n significant figures* if its value is correct to within $\frac{1}{2}$ unit in the least significant position. For example, if 9,072 is correct to four significant figures, then it is understood that the number lies between 9,072.5 and 9,071.5 (that is, 9,072 ± 0.5); if 0.01024 is correct to four significant figures, then it lies between 0.010245 and 0.010235; if 6,080 × 10^3 is correct to four significant figures, it lies between 6,080,500 and 6,079,500.

Another term used to describe a number is the *number of decimal places* of the number, being the number of digits to the right of the decimal point. Thus 0.008379 has six decimal places, and 0.37 × 10^{-5} has seven decimal places.

To be used during a computation, numbers that are infinite decimals must be converted to finite-decimal approximations. Similarly numbers that contain more significant figures than are desirable for a particular computation must be approximated by numbers of less significant figures. The process of reducing the number of significant figures of a number is called *rounding.* Numbers that are rounded so that they contain n significant figures are formed so that they will be *correct to n significant figures;* they are then called *round numbers.* Rounding can be accomplished by means of the following rule: To round a number so that it is correct to n significant figures, retain the n leftmost significant figures, and discard the rest; if the discarded number is greater than $\frac{1}{2}$ unit in the least retained significant figure, then add 1 to this last retained figure; if the discarded number is less than $\frac{1}{2}$ unit in the least retained significant figure, leave the latter unchanged. The case where the discarded number is exactly ½ unit in the least retained significant figure is often handled thus: If the least significant figure is even, leave it unchanged; if odd, then add 1 to it. By means of this latter rule the errors due to rounding will tend to cancel, since even and odd digits occur on the average with equal frequency. Also, by leaving the rounded number in this case as an even number we increase our chances that the result of a division might be exact. As examples, the following numbers have been rounded to four significant figures:

Number	*Rounded number*
3.141|5926	3.142
0.1428|57142	0.1429
0.005329|601	0.005330
421.2|4899	421.2
58.76|5000	58.76
93.29|5000	93.30

Relative Accuracy and Absolute Accuracy. The error in a number, as compared with its true value, is stated in various ways. The *absolute error* of an approximate number N^* from its true value N is defined by

$$\epsilon_a = |N^* - N|$$

The relative error ϵ_r is defined by

$$\epsilon_r = \frac{|N^* - N|}{N^*} = \frac{\epsilon_a}{N^*}$$

The percentage error ϵ_p is defined by $\epsilon_r \times 100$ per cent. The relative- and percentage-error concepts have the advantage that they are independent of the unit of measurement, whereas the absolute error is expressed in terms of the unit used. For example, 3.14 is an approximate value of the infinite decimal π; the absolute error of this approximation is $|3.14 - \pi| < 0.002$ $(\pi - 3.14 = 0.00159 \cdots)$; the relative error is $|3.14 - \pi|/3.14 < 0.002/3.14 < 0.0006$; the percentage error $\epsilon_p < 0.06$ per cent.

The accuracy of an approximate number can be considered from two points of view. The first, and that most often considered in using digital computers, is the concept of *relative accuracy.* In this case the accuracy is measured by the *relative* error and not the absolute error. As will be shown two paragraphs below, the number of *significant figures* of an approximate number is a measure of the relative error and hence of the relative accuracy of the number. It is for this reason that significant figures are so important. To see the importance of relative error as an index of the accuracy of a number, consider the measurement of the 0.002-in. diameter of a human hair to within 1/100,000 in., as compared with the measurement of the 240,000-mile distance from the earth to the moon, to within 2 miles. The relative error of the measurement of the hair is 0.00001/0.002 = 1/200, while the relative error of the measurement of the distance to the moon is 2/240,000 = 1/120,000, while the absolute errors are 0.00001 and 126,720 in., respectively. Clearly the measurement of the distance to the moon is 600 times more accurate than that of the diameter of the human hair, even though the absolute error of the former is over 12×10^9 times that of the latter.

On the other hand the *absolute accuracy* of a measurement is often important from another point of view. For example, in the tolerance of machined parts it is the absolute error that counts as the measure of accuracy; hence the allowed absolute error is frequently stated explicitly by writing the desired sizes as finite decimals followed by ± and the tolerance which is the maximum allowable absolute error. In this case it is the number of decimal places that plays the important role.

The relation between the number of significant figures and the relative error inherent in an approximate number is given by the following theorem, in which ϵ_r is the relative error, K is the most significant figure of the number, and n is a positive integer:

If a given number is correct to n significant figures, then

$$\epsilon_r \leq \frac{1}{2K \times 10^{n-1}}$$

When the most significant figure is not explicitly known, we can say that the relative error $\epsilon_r \leq 1/(2 \times 10^{n-1})$. For any radix q the 10^{n-1} in these formulas is replaced by q^{n-1}. If $q = 2$ (*binary*), then the inequalities are read $\epsilon_r \leq 1/2^n$.

To see why this theorem is true, note that $N^* \geq K \times 10^{p-1}$ (for example, $521 \geq 500 = 5 \times 10^2$), where p locates the decimal point with respect to the most significant figure K.† Now since N^* represents an approximate number, correct to n significant figures, i.e., with an error of ½ unit in the last (nth) significant figure, then $\epsilon_a \leq \frac{1}{2} \times 10^{p-n}$ and

$$\epsilon_r \leq \frac{\frac{1}{2} \times 10^{p-n}}{N^*} = \frac{1}{2N^* \times 10^{n-p}} \leq \frac{1}{2K \times 10^{n-1}}$$

If K is not known, then to ensure that the inequality will hold for every K we choose the smallest, namely, 1.

This theorem is of important practical value in using a digital computer having floating-point arithmetic operations. It enables the evaluation of the number of significant figures that must be carried to ensure that the relative error be kept below a given amount; and conversely it enables the evaluation of the relative error for a specified number of significant figures.

Significant Figures in Arithmetic Operations. Since numerical computation as performed by a digital computer reduces to addition, subtraction, multiplication, and division, the problem of retaining significance during these operations becomes fundamental. We shall here give rules of thumb commonly used by mathematicians regarding the control of

† For example, 3,000 would be written $3 \times 10^3 = 3 \times 10^{4-1}$, whence $p = 4$. On the other hand, 0.003 would be written $3 \times 10^{-3} = 3 \times 10^{-2-1}$, whence $p = -2$.

significance during arithmetic operations. In any particular situation the rules serve merely as a guide; the details of the process usually depend on the particular numbers and errors involved and hence must often be carefully examined. *In many situations it is just as bad to overestimate an error resulting from a computation as it is to underestimate one.*

Consider addition first. The rule of thumb is that in adding two (positive) numbers there is usually no loss of significant figures; in fact the result sometimes has one more significant figure than either of the operands.

For example, suppose that 52,761.2 and 71,436.7 were rounded to 52,761 and 71,437 and added:

Rounded	*Unrounded*
52,761	52,761.2
+ 71,437	+ 71,436.7
124,198	124,197.9

The rounded numbers are both correct to five significant figures, and the result is correct to six, as is seen on comparison with the addition of the unrounded numbers. Here there was no loss of significance, because the rounding errors canceled out, and the added significant figure resulted from the carry. However, consider 12,761.5 and 11,435.5 rounded to 12,762 and 11,436 and added:

Rounded	*Unrounded*
12,762	12,761.5
+11,436	+11,435.5
24,198	24,197.0

Here the result is correct only to four significant figures since, as can be seen from the unrounded addition, the rightmost 7 is off by 1 unit. This is of course an exception to our rule, chosen so that the errors in the least significant figures of the operands were maximum and were added together. It is clear that, if one always includes one more significant figure than is necessary, he will be more sure of having at least the required significance in his result.

Subtraction, however, is another story. The rule of thumb is simply a warning that complete loss of significance is possible and that each case must be examined on its own merits. For example, suppose that we are forming the difference of 52,763.5 and 52,752.5 as rounded to 52,764 and 52,762:

Rounded	*Unrounded*
52,764	52,763.5
−52,762	−52,762.5
00,002	00,001.0

Although the subtrahend and the minuend were correct to five significant figures, the difference has no significant figures; i.e., the 2 is off by a full unit. More generally, there is not a complete loss of significance, but very frequently severe loss of significance occurs. There are two remedies that can be tried. The first is to use many more significant figures in the subtrahend and minuend to begin with and hope that enough significance will result for the remainder. The other is to try to transform the expression to something else and thereby avoid the subtraction altogether. For example, if one had to evaluate $1 - \cos x$ for small x, it might be wiser to write $1 - \cos x = 2 \sin^2 (x/2)$ and evaluate the latter expression.

For multiplication the rule of thumb is that up to two significant figures can be lost, though not necessarily. This same rule applies to division. Consider, for example, the product of 811.2 and 112.5, each rounded to three significant figures:

```
 Rounded       Unrounded

    811           811.2
    112           112.5
   1622           40560
   811           16224
  811            8112
  90832         8112
                91260.00
```

The third figure of the product is off by 4 units and therefore is not significant; the second figure from the left is, however, only off by 0.4 unit and therefore is significant. The product is thus correct to two significant figures, while the multiplicand and multiplier were each correct to three.

As an example of the loss of a significant figure during division, consider 1,761.5 and 1,763.4, rounded to 1,762 and 1,763:

```
       Rounded                  Unrounded

            1.0005                       1.001
  1,762|1,763.0000         17,615|17,634.000
        1,762                     17,615.000
            1.0000                    19.000
```

Here the rounded dividend and divisor are correct to four significant figures, but the quotient is correct only to three.

One further rule of thumb might be helpful. In digital computers, where it is in general no more difficult to handle numbers with many significant figures, keep as many significant figures as possible during the computations, and beware of loss of significance through subtraction.

Pitfalls. A person using an electronic computer always worries about whether the results he gets are or are not accurate. For usually only the final results of a computation are read out from the computer's memory, and intermediate results are neither retained nor observed. The results of computer programs are often checked by comparing a few values with some previously hand-computed values. However, the fact that these few isolated results check is no guarantee that all the results computed by the program will be equally correct. Rarely is every difficulty fully anticipated. Automatic programming routines aggravate this situation, increasing the possibility of overlooking pitfalls. Hence, when writing his program the programmer must plan against the loss of too many significant figures in the computation, as well as choose the computational method that can best result in the necessary accuracy. During the running of the program on the computer the programmer must be acutely aware of such pitfalls as overflows, loss of significance, difficulties arising in intermediate forms, and other, unanticipated, singular situations.

For example, consider a pitfall that occurs in evaluating $\sqrt{x}$. If $x = 0.0000\,3259$, where x is good to seven decimal places and uncertain in the last (i.e., the 9), then to eight decimal places we get

$$\begin{aligned}\sqrt{0.0000\,3258} &= 0.0057\,0789\\ \sqrt{0.0000\,3259} &= 0.0057\,0877\\ \sqrt{0.0000\,3260} &= 0.0057\,0964\end{aligned}$$

Hence $\sqrt{x}$ is good only to five places, is uncertain in the sixth place, and is erroneous in the last two places. However, the computer will carry these last two places, and the error may be magnified in subsequent calculations.

Another example occurs in computing, say, sin 314,159.3. For if we reduce the angle to one whose magnitude is less than $\pi/2$, we find

$$\sin 314{,}159.3 = \sin (314{,}159.3 - 314{,}159.27) = \sin 0.03 = 0.03$$

If the last figure of the original angle were 2 instead of 3, we would find $\sin 314{,}159.2 = -0.07$. If 4, we would find $\sin 314{,}159.4 = +0.13$. Hence, even though the original angle is known to within seven significant figures with an uncertainty of 1 in the last figure, the result is not good to within one decimal place.

The following quotation† summarizes the general attitude that should be maintained in performing numerical computations on a digital computer:

> Despite the availability of high speed computers, a successful computation depends on a judicious combination of mathematical analysis and numerical experimentation. In the actual programming of his problem, the numerical analyst should not use subroutines and automatic codes blindly, but should ascertain that the logical decisions made in them will be consistent with the required round-off error and tolerance limits. The possibility of pitfalls will always be present in spite of all efforts to avoid them. It is therefore important that the numerical analyst interpret his answers according to their true meaning. Although the tools of the numerical analysis are all based on rigorous mathematical theorems the application of these theorems requires discretion. In carrying

† Irene A. Stegun and Milton Abramowitz, Pitfalls in Computation, *J. Soc. Ind. Appl. Math.*, vol. 4, no. 4, December, 1956.

out a numerical computation program it is often necessary to use experimental techniques to establish the correct operational procedures.

EXERCISES

(*a*) Tell how many significant figures are in each of the following numbers, and round each to be correct to five significant figures:

2.7182818
7.3890568
0.3183099
0.3678794
484,813,681,110 × 10^{-6}
154.1500 × 10^{3}

1.11001110 (*binary*)
.00110010 × 2^{-3} (*binary*)
101011.00 (*binary*)
0011001100 × 2^{3} (*binary*)
1010101010 (*binary*)
11.001100 (*binary*)

(*b*) Calculate the maximum relative error of each of the above numbers, rounded to three significant figures (or bits in the case of binary numbers).

(*c*) If all the numbers in a computation were kept to five significant figures, what is a bound on the relative error of each number? If these numbers were binary numbers, what is the bound on the relative error?

(*d*) If it is desired that all numbers involved in a computation have a relative error of less than $\frac{1}{600}$, how many significant figures should each number contain?

(*e*) In a computation involving only binary numbers, if it is desired that all numbers should have a relative error of less than $\frac{1}{600}$, how many significant bits should each number contain?

6-12. Additional Topics

a. Finite Differences for Polynomials. Consider a polynomial whose values have been computed at equal intervals of $x_{i+1} - x_i = h$, that is, $y(x_0)$, $y(x_0 + h)$, $y(x_0 + 2h)$, $y(x_0 + 3h)$, ... The differences of successive values of y, namely, $y[x_0 + (i + 1)h] - y(x_0 + h)$, are called the *first differences* of the polynomial. The differences of successive first differences are called *second differences,* and so forth. An interesting result (see the references below) is that *the nth differences of a polynomial of the nth degree are constant when the values of the independent variable are taken at equal intervals.* Thus the accuracy of computations based on a polynomial approximation can be checked. For example, consider $y = 2x^2 - x + 1$, as tabulated in Table 6-2. The

TABLE 6-2. DIFFERENCES FOR $y = 2x^2 - x + 1$

x	y	First differences $\Delta_1 y$	Second differences $\Delta_2 y$
0	1		
		1	
1	2		4
		5	
2	7		4
		9	
3	16		4
		13	
4	29		4
		17	
5	46		

constant second differences are 4. Suppose that an error was made in computation and $y(3)$ was evaluated as 10; how could this be detected? The study of finite differences has many interesting and important aspects (see the references below, and also C. Jordan, "Calculus of Finite Differences," Chelsea Publishing Company, New York, 1947).

b. General Formula for Measurement Errors. Suppose that $N = f(x_1,x_2, \ldots ,x_n)$ denotes any function of n independent variables $x_1, x_2, \ldots , x_n$ that are subject to the experimental errors $\Delta x_1, \Delta x_2, \ldots , \Delta x_n$. Then a formula that gives the resulting error in the function N is given by

$$\Delta N = \frac{\partial N}{\partial x_1}\Delta x_1 + \frac{\partial N}{\partial x_2}\Delta x_2 + \cdots + \frac{\partial N}{\partial x_3}\Delta x_3$$

(See, for example, the references below.) Suppose that the error of the independent variables is not actually known, but it is known that the probability distribution of the errors is *normal* (Gaussian). Then we can define the probable error of a single measurement as a value such that half the errors of a series of measurements will be greater, the other half less. If $R, r_1, r_2, \ldots , r_n$ are the probable errors, respectively, of $N, x_1, x_2, \ldots , x_n$, then

$$R = \sqrt{\left(\frac{\partial N}{\partial x_1}\right)^2 r_1{}^2 + \left(\frac{\partial N}{\partial x_2}\right)^2 r_2{}^2 + \cdots + \left(\frac{\partial N}{\partial x_n}\right)^2 r_n{}^2}$$

For further discussions see Scarborough (see the references below), chaps. XIV–XV, and E. Frank, "Electrical Measurement Analysis," chaps. 5–7, McGraw-Hill Book Company, Inc., New York, 1959.

c. Error Formulas for Differentiation and Integration. Milne ("Numerical Calculus," pp. 108–114; see the references below) gives a method for developing the error formula for any differentiation-integration method that may be developed by the method of undetermined coefficients. If $Q(f)$ is the integration or differentiation to be performed on the function f and $P(f)$ is the formula developed by the method of undetermined coefficients for its approximation, then the error $R(f) = Q(f) - P(f)$. $R(f)$ is said to be of *degree n* when $R(x^m) = 0$ for $m \leq n$, but $R(x^{n+1}) \neq 0$. Also let $\overline{(x - s)}^n$ be defined by $\overline{(x - s)}^n = (x - s)^n$ if $x > s$ and $\overline{(x - s)}^n = 0$ if $x < s$. In terms of this definition let

$$G(s) = \frac{1}{n!} R_x[\overline{(x - s)}^n]$$

in which $R_x[\overline{(x - s)}^n]$ means $R[\overline{(x - s)}^n]$ regarded as a function of the variable x. Then it can be shown that

$$R(f) = \int_{-\infty}^{\infty} f^{(n+1)}(s)G(s)\, ds$$

where $f^{(n+1)}$ is the $(n + 1)$st derivative of f (see Milne for an example). Using this technique, derive error formulas for the integration and differentiation formulas of Secs. 6-7 to 6-9 and their exercises.

d. Runge-Kutta Method. There are formulas for the integration of differential equations other than those which can be derived by the method of undetermined coefficients. One well-known example is the so-called "Runge-Kutta method." This has the advantage that the error is of the order of h^5 and no special methods are needed to start the solution. Consider $y' = f(x,y)$; then the increment for advancing the dependent variable is given by

$$\begin{aligned}\Delta y &= \tfrac{1}{6}(k_1 + 2k_2 + 2k_3 + k_4)\\ k_1 &= hf(x_0,y_0)\\ k_2 &= hf(x_0 + \tfrac{1}{2}h,\ y_0 + \tfrac{1}{2}k_1)\\ k_3 &= hf(x_0 + \tfrac{1}{2}h,\ y_0 + \tfrac{1}{2}k_2)\\ k_4 &= hf(x_0 + h,\ y_0 + k_3)\end{aligned}$$

Then $x_1 = x_0 + h$, and $y_1 = y_0 + \Delta y$, and the process is continued. (For further details see the references below.)

e. Best Fit. Cecil Hastings's book "Approximations for Digital Computers" (see the references below) is an elementary discussion of the method of best-fit approximations written in a quite unique and entertaining style. It also includes an interesting source of best-fit approximations.

f. Monte Carlo Method and References. Computations that involve successive generation of *random numbers* are often called Monte Carlo calculations. (For definition of random numbers see W. Feller, "Probability Theory and Its Applications," John Wiley & Sons, Inc., New York, 1950.) Monte Carlo computations usually are associated with computer simulations of real processes, such as military combat, growth of cancer cells, business transactions, mortality studies, etc., where some phase of the simulation involves a probability distribution. For example, if a missile were 80 per cent effective against a certain type of target, then in simulating each missile a random number between 0 and 9 would be chosen; only if it lay between 0 and 7 would a kill be recorded. Similarly the size of a cancer growth is very sensitive to the initial rate of mitosis of the generating cells, which is randomly determined. Monte Carlo processes have also been used in evaluating integrals. To evaluate $\int_0^1 f(x)\,dx$ on the interval $0 \le x \le 1$, we choose N arguments x at random and evaluate $f(x)$ for each; it can be shown that, as N becomes larger, the average of all the $f(x)$ so determined converges to $\int_0^1 f(x)\,dx$.

The generation of random numbers with a computer is simple. Given a number R_i, find a new random number R_{i+1} as the *minor product* of ρR_i, where ρ is the largest power of 5 that can fit into a computer word. Start with $R_0 = \rho$. For $\rho = 5^{17}$ the sequence has a period of about 10^{12}.

References

Curtiss, J. H.: "Monte Carlo" Methods for the Iteration of Linear Operators, *J. Math. and Phys.*, vol. 32, pp. 209–232, 1953.

Davis, P. J., and P. Rabinowitz: Some Monte Carlo Experiments in Computing Multiple Integrals, *Mathematical Tables and Other Aids to Computations*, vol. 10, pp. 1–8, 206, 1954.

Kahn, H.: Modification of the Monte Carlo Method, *Proc. Seminar Sci. Computations*, pp. 20–27, International Business Machines Corporation, New York, 1949.

Metropolis, N., and S. Ulan: The Monte Carlo Method, *J. Am. Statist. Assoc.*, vol. 44, pp. 335–341, 206, 290, 1949.

Meyer, H. A. (ed.): "Symposium on Monte Carlo Methods, University of Florida. 1954," John Wiley & Sons, Inc., New York, 1956.

Todd, J., et al.: Monte Carlo Method, *NBS Appl. Math. Ser.*, vol. 12, 1951.

g. Texts on Numerical Analysis

Alt, Franz L.: "Electronic Digital Computers," Academic Press, Inc., New York, 1958.

Engineering Research Associates, Inc., Staff of: "High-speed Computing Devices," McGraw-Hill Book Company, Inc., New York, 1950.

Grabbe, E. M., S. Ramo, and D. E. Wooldridge: "Handbook of Automation, Computation, and Control," vol. 1, John Wiley & Sons, Inc., New York, 1958.

Hastings, Cecil, Jr.: "Approximations for Digital Computers," Princeton University Press, Princeton, N.J., 1955.

Hildebrand, F. B.: "Introduction to Numerical Analysis," McGraw-Hill Book Company, Inc., New York, 1956.

Householder, A. S.: "Principles of Numerical Analysis," McGraw-Hill Book Company, Inc., New York, 1953.

Lanczos, C.: "Applied Analysis," Prentice-Hall, Inc., Englewood Cliffs, N.J., 1956.

Levy, C., and B. A. Baggott: "Numerical Solutions of Differential Equations," Dover Publications, New York, 1950.

Milne, W. E.: "Numerical Calculus," Princeton University Press, Princeton, N.J., 1949.

———: "Numerical Solution of Differential Equations," John Wiley & Sons, Inc., New York, 1953.

Nielsen, Kaj L.: "Methods in Numerical Analysis," The Macmillan Company, New York, 1956.

Scarborough, J. B.: "Numerical Mathematical Analysis," 2d ed., The Johns Hopkins Press, Baltimore, 1950.

CHAPTER 7

SEARCHING, SORTING, ORDERING, AND CODIFYING

7-1. Introduction

Ingredients. A great proportion of computer and control applications involve searching, sorting, ordering, and codifying. The general-purpose computer considered in Part 1 is not by itself best suited for these nonarithmetic operations. To understand what special systems-design features are needed for such operations, we must first study the operations themselves.

The ingredients of searching are *items* and *characteristics* and *associations of characteristics with the items.* An item is a package of information; the characteristics associated with this item are attributes that are related to the item. Problems of searching all involve locating an item, or items, associated with a given set of characteristics. For example, each article reference included in a bibliography is an item; the subjects relating to the contents of the article are the characteristics. A bill of sale is an item; the name and address of the buyer may be considered as the characteristics. A research grant is an item; the name of the principal investigator, the amount of the grant, the subject under investigation, the name of the school or organization receiving the grant, etc., are all characteristics. The inventories of particular replacement parts are the items; the serial numbers, costs, acquisition record, etc., are the characteristics, and so forth.

Searching, Sorting, Ordering, and Codifying. Given a set of characteristics, the process of obtaining *every item associated with all the given characteristics* is called *searching.* In a bibliographical list of research reports one might search for all articles concerning the application of nuclear theory. The words *application, nuclear,* and *theory* are the characteristics, and the search should result in a collection of references, each concerned with the application of nuclear theory. As a specific illustration, consider the *item list* of article references and the *characteristic list* of subjects of a sample bibliography given in Fig. 7-1. We have given each item a boldface number and each characteristic an italicized number. The *item-characteristic associations* are given in Table 7-1, where a unit indicates that the characteristic of the row of the unit is associated with the item of its column. Consider the rows associated with the given characteristics. Form a new row that has a unit corresponding only to those (column) positions where all the given characteristic rows also have units. (This is called *logically multiplying* these rows; cf. Sec. 4-5.)

PART I. THE ARTICLE LIST (BIBLIOGRAPHY PROPER)

Article No.

1.1 Abrahams, A. P. Autoradiographic determination of radioactivity in rocks. *Nucleonics* 15:85–86 Mar 1957

1.2 Aravindakshan, C. A simple arrangement for obtaining optical transforms of crystal structures. *J. Sci. Instr.* 34:250 Jn 1957

1.3 Gasstrom, R. V. A very fast pulse-height analyzer with independent uptake, sorting, and storage of information. *Nuclear Instruments* 1:75–79 Mar 1957

1.4 NBS Circular 850. Bibliography on ignition and spark ignition systems. Nov 1 1956

Article No.

2.1 Nicholls, J. Alpha-scintillation monitor for hands and clothing. *Nucleonics* 15:80, 81, 83, 84 Mar 1957

2.2 Pope, M. I. An automatically recording vacuum balance. *J. Sci. Instr.* 34:229–232 Jn 1957

2.3 Powell, D. A. An apparatus giving thermogravimetric and differential thermal curves simultaneously from one sample. *J. Sci. Instr.* 34:225–227 Jn 1957

2.4 Seidle, F. G. P., et al. Modification of the Brookhaven fast chopper. *Nuclear Instruments* 1:92–93 Mar 1957

Article No.

3.1 Senior, D. A. The Kerr cell, a high speed electro-shutter, Pt. II. *Instr. Pract.* 11:471–476 May 1957

3.2 Smith, B. O., and Grimshaw, A. G. A pneumatic level indicator. *Instr. Pract.* 11:469–470 May 1957

3.3 Stockendal, R., and Bergkvist, K. E. Evaporation device for beta-spectrometer samples. *Nuclear Instruments* 1:53–54 Jan 1957

3.4 Tove, Per-Arne. Electronic time analyzer applied to the measurement of the half-lives of metastable nuclear states. *Nuclear Instruments* 1:95–100 Mar 1957

PART II. THE WORD LIST (CODE BOOK AND THESAURUS)

	Word No.		*Word No.*		*Word No.*
adaptability	*3.2*	evaluation	*6.1*	Netherlands	*4.2*
analysis	*4.1*	gas	*2.2*	nuclear	*4.3*
application	*5.1*	heat	*2.3*	plan	*6.2*
concept	*6.2*	hysteresis	*1.1*	theory	*4.4*
counting	*2.1*	implementation	*5.3*	thermal	*2.3*
design	*7.1*	instrumentation	*5.3*	use	*5.1*
differentiation	*3.1*	mass	*1.2*	versatility	*3.2*
England	*5.2*				

FIG. 7-1. Illustrative bibliography example.

TABLE 7-1. ASSOCIATION OF ITEMS WITH CHARACTERISTICS

Characteristic	Item											
	1.1	**1.2**	**1.3**	**1.4**	**2.1**	**2.2**	**2.3**	**2.4**	**3.1**	**3.2**	**3.3**	**3.4**
1.1	0	0	0	0	0	0	0	0	0	0	0	1
1.2	0	0	0	0	0	1	0	0	0	0	0	0
2.1	1	0	0	0	0	0	0	1	0	0	0	0
2.2	0	0	0	1	0	1	0	0	0	0	0	0
2.3	0	0	0	1	0	0	1	0	0	0	0	0
3.1	0	0	0	0	0	1	1	0	0	0	0	1
3.2	1	0	0	0	0	0	1	0	0	0	0	1
4.1	0	0	1	0	0	1	0	0	0	0	1	1
4.2	0	0	1	0	0	0	0	1	0	0	1	1
4.3	0	0	0	0	1	0	0	1	0	0	1	1
4.4	0	0	1	0	0	0	0	1	0	0	1	1
5.1	0	0	0	0	0	1	1	1	0	1	0	1
5.2	0	1	0	0	0	1	1	0	1	1	0	0
5.3	0	0	0	1	0	0	1	1	1	0	1	0
6.1	1	1	1	0	0	1	0	0	1	0	1	0
6.2	0	1	0	0	1	0	0	1	1	1	1	0
7.1	0	1	0	1	1	1	1	0	1	1	0	0

The units of the resulting row correspond to the columns of the desired items. For example, given *application* (*5.1*), *nuclear* (*4.3*), *theory* (*4.4*), we logically multiply rows,

$$\begin{array}{lllll} & \mathit{5.1} & 0000 & 0111 & 0101 \\ & \mathit{4.3} & 0000 & 1001 & 0011 \\ & \mathit{4.4} & \underline{0010} & \underline{0001} & \underline{0011} \\ \text{obtaining} & & 0000 & 0001 & 0001 \end{array}$$

with units in columns **3.4** and **2.4** corresponding to items by Tove and Seidle. However, it is rarely feasible to display a matrix such as in Table 7-1; we have used it simply as an illustration to aid the visualization of the searching methods to be described.

It is difficult to discuss searching without using the terms *sorting*, *ordering*, and *codifying*. It is often helpful for temporary use to substitute a symbolic abbreviation to represent an item or characteristic. For example, the use of boldface and italicized numbers in our illustrative bibliography made the table of associations easier to display. *A code is a list of such abbreviations; the making of a code is codifying.*† Fre-

† We use the term *codifying* here in preference to *coding* to distinguish the process being discussed in this chapter from the programming of a computer.

quently the code is given in the form of a "dictionary," or "thesaurus." In our illustrative bibliography the code for the item (article) list is presented in dictionary form, and the code numbers are listed in numerical order. The characteristic (word) list is in thesaurus form, where the word list is in alphabetical order; here synonyms are given the same code number. Codes are used in searching as follows: First the thesaurus of characteristics is compared with the list of desired characteristics, and the corresponding code numbers are found; next the search is made to find the code numbers of the resulting items; finally the actual items are determined from the dictionary. In computer searching the use of codes in this manner is almost always indicated. Further discussion of codes appears in Sec. 7-7.

By *ordering a list* is meant the process of arranging the list according to some linear ordering system, such as alphabetically or numerically, so that, if element Q of the list is greater than element P, then element Q lies further from the beginning of the list than P. By *sorting a list* is meant the process of separating the list into partitions, i.e., *mutually exclusive classes.* In sorting items the *equivalence relation* by means of which the classes, or partitions, are defined is often given in terms of associated characteristics. For example, research grants may be sorted according to the university that received the grant, etc. Searching may be defined as the process of sorting a list of items into two categories, those items which are associated at least with all of certain given characteristics, and those which are not. Frequently the classes, or partitions, of a sort are ordered. For example, collections of research grants sorted by university can be ordered alphabetically by the university's names.

Searching and sorting are both closely associated with ordering. For instance, to sort research grants by university, all that need be done is to order the grants alphabetically by university, and then the result will present all grants from the same university juxtaposed in the list; hence the grants will automatically be sorted as the partitions of the sort are ordered.

Summarizing, to *sort* a list of elements, some equivalence relation must be defined that tells when two elements are in the same equivalence class or not. To *order* a list linearly, an ordering relation must be defined. Sorting can often be accomplished by means of ordering, when the ordering relation is so chosen that the identities of some type of characteristic represent the desired equivalence relation. As we shall see in the next section, searching is often carried out by sorting ordered lists.

7-2. Methods of Searching

The First Two Methods. For concreteness we assume in what follows that the items and characteristics are listed on magnetic tape. Based on this assumption, we shall describe three methods for searching with a computer, using our illustrative bibliography as an example. Each of the searching techniques is, as will be shown, intimately associated with the listing arrangement of items and characteristics on the magnetic tape; this is because the *associations are indicated by means of the item-characteristic arrangement* on the tape.

First, consider the case where all characteristics associated with each item of the item list appear under that item (see Figs. 7-2*a* and 7-3*a*). Given a set of desired characteristics, the computer reads an item and its associated characteristics from the magnetic tape into the high-speed memory and compares to determine whether or not the given characteristics are included in the characteristics associated with the item; if so, the item would be suitably recorded. This process is accomplished for each item and its associated characteristics on the tape, one item at a time successively. In this way, after a single pass through the tape, every item associated with all the given characteristics is retrieved. The advantage of this system is that the items need not be ordered on the tape, since every item is searched. The disadvantage is that *every section* of the magnetic tape must be searched in detail, consuming much time in the computer.

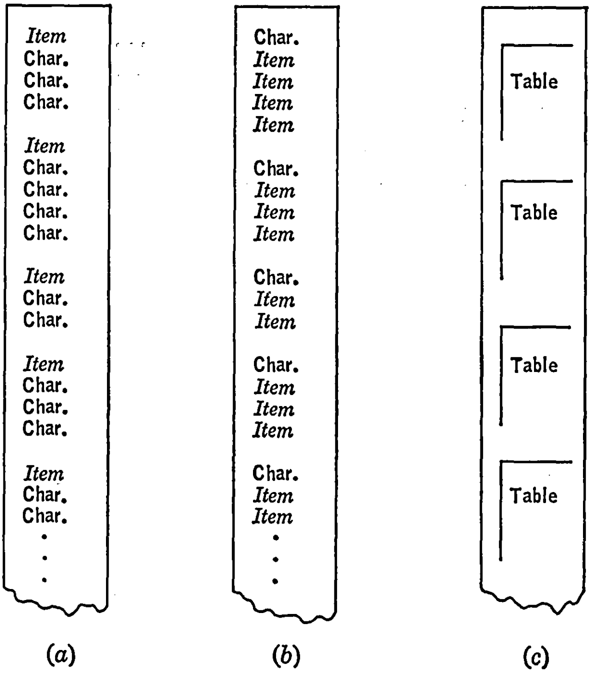


FIG. 7-2. Methods of listing on magnetic tape.

Second, consider the case where the characteristic list is recorded on the tape and under each characteristic is a list of items associated with that characteristic (see Figs. 7-2*b* and 7-3*b*). The characteristics must appear in some order on the tape—e.g., alphabetically or numerically.

Given a set of characteristics, the computer would first order them and then would go through the magnetic tape looking for the first, or smallest, characteristic; having found this, it would read the characteristic and its associated items into the high-speed memory and record all items associated with it. Continuing the search on the tape, the computer would look for the second, or next smallest, characteristic. Having found that, the computer would see which items associated with the second characteristic were *also* associated with the first and record them. Then the computer would look for the third characteristic and record only those items associated with it which were also associated with both the first two characteristics, and so on. The computer need not examine the whole tape in detail, but only those sections associated with each of the given characteristics; however, both the list of characteristics and the set of given characteristics must be preordered.

(a)	(b)	(c)	
. . .	. . .	. . .	
1.2	3.2	4.4	
5.2, 6.1, 6.2, 7.1	*1.1, 2.3, 3.4*	2.4	*5.1, 5.3, 6.2*
1.3	4.1	3.4	*5.1*
4.1, 4.2, 4.4, 6.1	*1.3, 2.2, 3.3, 3.4*	3.3	*5.3, 6.1*
1.4	4.2	1.3	*6.1*
2.2, 2.3, 5.3, 7.1	*1.3, 2.4, 3.3, 3.4*	5.1	
2.1	4.3	2.3	*5.2, 5.3, 7.1*
4.3, 6.2, 7.1	*2.1, 2.4, 3.3, 3.4*	2.2	*5.2, 6.1, 7.1*
2.2	4.4	3.2	*5.2, 6.2, 7.1*
1.2, 2.2, 3.1, 4.1, 5.1, 5.2, 6.1, 7.1	*1.3, 2.4, 3.3, 3.4*	2.4	*5.3, 6.2*
2.3	5.1	3.4	
2.3, 3.1, 3.2, 5.1, 5.2, 5.3, 7.1	*2.2, 2.3, 2.4, 3.2, 3.4*		
. . .	. . .	. . .	

FIG. 7-3. Sample segments in the lists for each of three methods.

Note that, in terms of our association matrix of Table 7-1, the first and second methods described above correspond, respectively, to recording the successive *columns* of the matrix and the successive *rows* of the matrix on the magnetic tape.

The Tabledex Method. Finally consider a third method for recording the items and characteristics on the magnetic tape. The method involves the recording of successive *tables* on the magnetic tape and is therefore called the *Tabledex* method. There is one table for each characteristic. A table is formed as follows: All items associated with this characteristic are listed in the table; adjacent to each item are *sublisted all those characteristics greater in order than the characteristic of the table* which are associated with that item (see Figs. 7-2*c* and 7-3*c*). For example, in Fig. 7-3*c*, consider the Tabledex table headed 4.4. Observe from Table 7-1 that the characteristic *4.4* is associated with items **1.3**, **2.4**, **3.3**, and **3.4**. Now observe that item **2.4** is associated with *2.1*, *4.2*, *4.3*, *4.4*, *5.1*, *5.3*, and *6.2*, but *only 5.1*, *5.3*, and *6.2* appear in the sublist, for only these are greater than *4.4*, the number of the table (see Fig. 7-3*c*). Similarly the sublists for **3.4**, **3.3**, and **1.3** were formed for this table. (Note that these sublists are ordered in the table on their first *characteristics.*)

The tables are listed in the order of their associated characteristic, and the items and the sublist of characteristics for each item are also listed in the proper order. Given a set of characteristics, the computer first puts them in the proper order, then searches the tape for the table associated with the lowest ordered characteristic, reading this into the high-speed memory, and finally searches the sublists, recording those items associated with a sublist containing all the other given characteristics. For example, if the three given characteristics are *7.1*, *5.1*, and *5.2*, these would first be written in order as *5.1*, *5.2*, and *7.1*; then the table for *5.1* would be searched; and finally, if a sublist were found to contain both *5.2* and *7.1*, its item would be recorded. As can be seen in Fig. 7-3*c*, the items whose sublists contained *5.2* and *7.1* are **2.3**, **2.2**, and **3.2**; hence these are all items each of which is associated with all the given characteristics.

Using the Tabledex method, only the single section of the tape that contains the appropriate table need be searched in detail.

Population vs. Substantive Ordering of the Lists. The reader may have wondered how we arrived at the code number for the characteristics and items given in Fig. 7-1. The choice of numbering is related to the concepts of *population-based ordering* and *substantive-based ordering.* Substantive ordering refers to an ordering arrangement based on some attribute of the individual terms in the list. Thus an alphabetical ordering of items by author is a substantive-based ordering, an ordering of characteristics by research-field classification is substantive-based, etc. In Fig. 7-1 the *item*-list code numbers were assigned substantively to the articles, in alphabetical order; in addition the number of an item locates its column and line, e.g., item **2.3** is in column 2, line 3.

Population-based ordering is quite different: it refers to an ordering based on the *quantities of items or characteristics in the associations.* For example, population-based code numbers for each characteristic might be assigned as the number of items associated with it. To distinguish among characteristics that have the same number of associated items, a decimal number can be appended. The characteristics of our example are numbered in this way. Thus in Fig. 7-1 it is seen that *analysis*, *Netherlands*, *nuclear*, and *theory* each have four associated items; hence their population-based numbers are *4.1*, *4.2*, *4.3*, and *4.4*, respectively. Ordering the characteristics by population-based code numbers can save searching time. In the second method given above, it is most efficient to begin searching in the section of the smallest given characteristic when population-based numbers are used. (Why?) Similarly, in the Tabledex method when population-based numbers are used, tables that include *many* items have *short* sublists, and tables that have long sublists include *few* items. (Why?)

Programming a Computer for Searching. The writing of codes for the techniques discussed in this section is in general reasonably straightforward. However, consideration of some factors concerning the use of magnetic tapes might prove helpful.

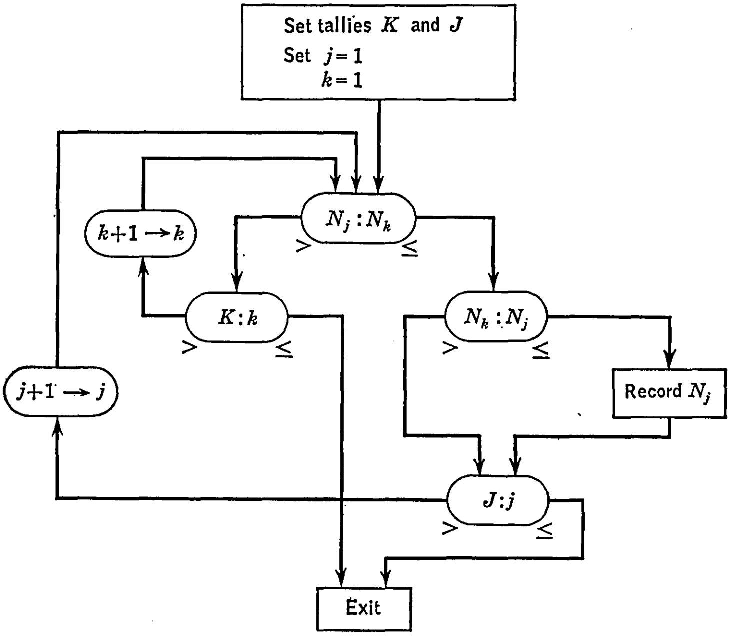


FIG. 7-4. Flow chart to record common numbers on two ordered lists.

Recall from Chap. 4 that magnetic-tape read-in, read-out, and tape-moving instructions usually specify the number of *blocks* to be read in, read out, or moved. Hence only multiples of blocks can be handled. Therefore, in recording a list on a section of magnetic tape the list is usually padded at the end with words of all zeros so that the entire list will make up an integral number of blocks. The first word of the first block of the list is often used as a *labeling word:* it contains information about the contents of the list, and, most important for our discussion, it tells how many blocks long the list is. The magnetic tape will then consist of a sequence of such lists, each labeled with the number of blocks in its length as the first word of the list.

There are two advantages to labeling the lists: (1) The computer will at first observe only one label word of a list. If this list is not the one desired, the computer can then move the magnetic tape by the number of blocks indicated in the label, past this list to the next list; the label of the next list will be observed, and so forth. In this way the computer may jump over parts of the magnetic tape not of interest during the particular search. (2) When a list is of interest, then each word of the list must be examined; this is usually accomplished by some kind of iteration or loop. The tally of this loop can be set according to the length of the list to be examined, as given in the label word. For example, in the second method for searching given above, each characteristic is followed by a list of its associated items. Hence the first, or label, word of each item list will denote the characteristic and its length, say in the α and β address positions for a two-address system. Then the computer will look for the first of the given (ordered) characteristics by jumping over lists until the appropriate list label is obtained.

When the appropriate section of tape is found, the problem basically reduces to the comparison of two ordered lists for common members. Suppose that we have two ordered lists of numbers, with K numbers in the first list, indexed by k ($k = 1, 2, \ldots, K$), and J numbers in the second list, indexed by j ($j = 1, 2, \ldots, J$). A flow diagram of a code to record every number that appears in both lists is shown in Fig. 7-4. If the number in the j list is greater than the number being compared with it in the k list, then the code moves along the k list until a common number is obtained, or until a number in the k list is reached that is greater than the one in the j list. Then the code moves along the j list in a similar way, until the k number is equaled or exceeded. This process depends basically on the fact that the j and k lists are ordered.

EXERCISES

(*a*) By means of the information given in Table 7-1 complete the lists and tables partially shown in Fig. 7-3*a* and *b*.

(*b*) Using the information given in Fig. 7-1 and Table 7-1, compile Tabledex tables using as the characteristics the words themselves, in alphabetical order.

(*c*) Using each of the three lists completed in (*a*) and (*b*), search for all items associated with each of the following sets of given characteristics: evaluation, concept, design; and differentiation, application, England, gas.

(*d*) Assign population-based numbers to the items of Fig. 7-1, and compile the lists of the first searching method from the information of Table 7-1.

(*e*) Draw a flow chart for each of the searching methods given in this section.

7-3. Manual Searching Methods

For each of the three searching methods given above there is an analogous manual searching method. Besides being of considerable importance in themselves, these manual methods can serve as a concrete visualization of the more abstract operations that take place in a computer.

Marginal-punch Cards. The first method given above corresponds to the use of marginal-punch cards, where each card represents an item and a marginal position on a card represents a characteristic. Different cards represent different items, but the same respective marginal position represents the same characteristic on each card. The association of characteristics with an item is made on its card by punching a notch in the marginal position of those characteristics which correspond to the particular item of the card. One type of card commonly used has holes corresponding to each marginal position. To sort a deck of the cards according to a particular characteristic, a needle is placed through the holes in the cards corresponding to this characteristic; then, when the needle is lifted, those cards which have been punched in this position will fall, the other cards will be lifted by the needle, and the deck is thereby separated into two parts (see Fig. 7-5). A search is performed by repeating this process successively on the dropped cards (which have the desired characteristic) until those cards punched in at least all the desired characteristic positions are obtained. Note that the initial deck of cards need not be kept in any particular order whatsoever.

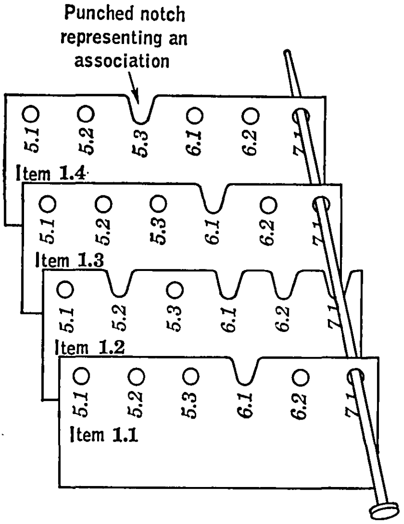


FIG. 7-5. Marginal-punch cards.

Peek-a-boo Cards. The second method given above corresponds to the use of the so-called "Peek-a-boo cards." Here each card corresponds to a characteristic; the surface of a card is laid off in an

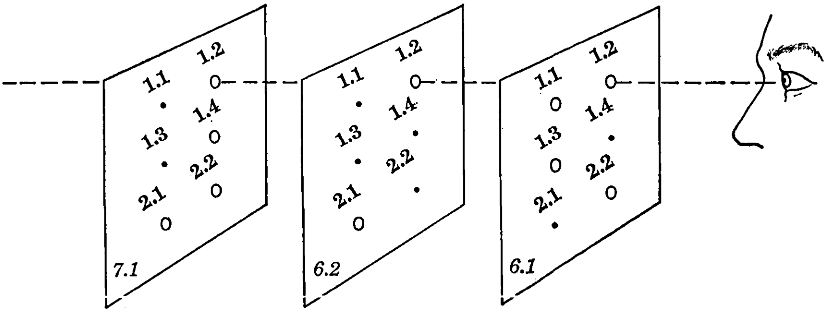


FIG. 7-6. Peek-a-boo cards.

X, Y coordinate system with each particular pair of coordinates representing an item. Different cards represent different characteristics, but the same respective X, Y coordinates represent the same item on each card. The association of characteristics with items is made by punching a hole at the X, Y coordinate positions that represent each item associated with the characteristic of the card. A search is made by first selecting from a deck of cards those cards which correspond to the given desired characteristics. The selected cards are placed together to form a pack; if an item is associated with all the given characteristics, then there will be a hole passing completely through the pack at the coordinate point corresponding to that item (see Fig. 7-6). Note that the initial deck of cards must be kept in some definite order so that the cards can be easily selected for a particular search, and note that the pack of selected cards must be carefully aligned.

Programming a Computer to Generate a Manual Tabledex Collection. The third method given above corresponds to the manual use of the Tabledex tables themselves. Here the tables are printed in conventional bound-book form. The use of a collection of tables is directly analogous to their use on magnetic tapes. For example, suppose that we have a collection of tables corresponding to our bibliography illustrated above and that we desire to find all articles on the *application* of *nuclear theory.* From the alphabetical word list the code numbers are found: *application 5.1*, *nuclear 4.3*, *theory 4.4*. These are placed in numerical order: *4.3*, *4.4*, and *5.1*. One then turns in the collection of tables to Table *4.3*, which would appear as shown:

TABLE *4.3*

✓	**2.4**	*4.4*	*5.1*	*5.3*	*6.2*
✓	**3.4**	*4.4*	*5.1*		
	3.3	*4.4*	*5.3*	*6.1*	
	2.1	*6.2*	*7.1*		

The first two rows of Table *4.3* contain both *4.4* and *5.1* and are checked. Hence articles **2.4** and **3.4** by Seidle and by Tove are associated with the given words. The advantage of the manual use of Tabledex lies in its being in bound-book form, without the necessity for the manipulation of cards.

A computer can be programmed to automatically compile and print a Tabledex collection for manual use. The method used to compile the Tabledex collection is of particular interest as an application of the techniques of this chapter. First the characteristics of each article-characteristic association are ordered in themselves. For example, we would have the article-characteristic association: item **2.4**, characteristics *2.1*, *4.2*, *4.3*, *4.4*, *5.1*, *5.3*, and *6.2* (see Table 7-1). Second, from each such article-characteristic association, a list of all possible rows of the tables can be generated. For example, from the associations just illustrated the following seven rows can be generated:

2.4	*2.1*	*4.2*	*4.3*	*4.4*	*5.1*	*5.3*	*6.2*
2.4	*4.2*	*4.3*	*4.4*	*5.1*	*5.3*	*6.2*	
2.4	*4.3*	*4.4*	*5.1*	*5.3*	*6.2*		
2.4	*4.4*	*5.1*	*5.3*	*6.2*			
2.4	*5.1*	*5.3*	*6.2*				
2.4	*5.3*	*6.2*					
2.4	*6.2*						

Third, the list of all such rows generated from every article-characteristic association is then sorted by ordering on the *first* (*leftmost*) *characteristic code number.* After the ordering, all rows with the same first characteristic appear adjacent. For example, the following rows appear adjacent:

2.4	*4.3*	*4.4*	*5.1*	*5.3*	*6.2*
3.4	*4.3*	*4.4*	*5.1*		
3.3	*4.3*	*4.4*	*5.3*	*6.1*	
2.1	*4.3*	*6.2*	*7.1*		

When the number *4.3* has been removed, the adjacent rows are just Table *4.3*. Thus from this ordered list of rows the tables can be printed out by the computer.

7-4. Searching with Relaxed Conditions

The Problem of Relaxed Conditions. It frequently happens that no item is associated with all the characteristics given. In such a case it is usually advisable to *search with relaxed conditions.*

For example, suppose that there are given 10 equally important characteristics and that no item is associated with all of them. Then the natural question to ask is: Are there any items associated with all but one of the characteristics, i.e., with any 9 of the characteristics? If not, then are there any items associated with all except 2 of the characteristics, i.e., with any 8 of the 10, and so forth? *We shall describe a procedure for accomplishing this important kind of search which does not involve trying all possible combinations of* 9 *of the* 10 *characteristics, then all possible combinations of* 8 *of the* 10, *etc.* This procedure can be extended to include a second type of relaxed-condition problem which may arise as follows: Suppose that 3 of the 10 given characteristics are most important and that it is desired to find all items associated with these 3 characteristics and also any 6, or any 5, etc., of the remaining 7 characteristics. The procedure can be easily generalized to include almost all relaxed-condition situations that occur in searching.

The idea of a *simple sort* can be used to compare searching techniques. A simple sort is a separation of a collection of items into two parts by means of a *single* characteristic C, that is, into a part whose items are each associated with C and a part whose items are not. A simple sort is analogous to the manual separation of a deck of marginal-punch cards into two parts by means of a single needle.

To see the advantage of the searching technique to be described over the straightforward method of trying all possible combinations of leaving out one of the characteristics at a time, then two at a time, then three at a time, . . . , then $K = n - r$ at a time, note that the total number of simple sorts required for the straightforward method is

$$\sum_{k=0}^{K} \frac{n!}{(n-k-1)!}$$

However, the number of simple sorts required by the method to be presented is simply

$$\frac{(K + 1)(n - K)}{2}$$

Searching for Items Associated with Any r Out of n Characteristics. The method of procedure is illustrated by Fig. 7-7. Suppose that there are four given characteristics, represented by C_1, C_2, C_3, and C_4. We would first sort the collection of items for those associated with C_1. Next we would sort those which *have* C_1 for those which also have C_2. Then those having *both* C_1 and C_2 would be sorted for those also having C_3, and so forth, until we find all items that have C_1, C_2, C_3, and C_4. In Fig. 7-7 we indicate the collection of items that are associated with *at least* C_1 by C_1, those associated with at least C_1 *and* C_2 by $C_1 \cdot C_2$, etc. The collection of items that do *not* have C_1 is indicated by $\bar{C}_1$, etc. Then $C_1 \cdot C_2 \cdot \bar{C}_3$ would represent the collection of items that at least have C_1 and C_2 and not C_3. Thus the results of each successive sort of the *first pass* through the collection of items are indicated by the first row of Fig. 7-7. Here, in the first sort, the collection of items is divided into two parts, the part C_1 and the part $\bar{C}_1$. Then the collection with C_1 is sorted for C_2, resulting in a part $C_1 \cdot C_2$ and a part $C_1 \cdot \bar{C}_2$, and so forth. After the first pass there will result five collections $\bar{C}_1$, $C_1 \cdot \bar{C}_2$, $C_1 \cdot C_2 \cdot \bar{C}_3$, $C_1 \cdot C_2 \cdot C_3 \cdot \bar{C}_4$, and $C_1 \cdot C_2 \cdot C_3 \cdot C_4$; all but the last of these will be used if another pass is necessary.

If the collection $C_1 \cdot C_2 \cdot C_3 \cdot C_4$ is empty, i.e., if no items are associated with all of $C_1 \cdot C_2 \cdot C_3 \cdot C_4$, we proceed to the second pass (the second row of Fig. 7-7). This time we start by first sorting $\bar{C}_1$ for C_2, obtaining $\bar{C}_1 \cdot C_2$ and $\bar{C}_1 \cdot \bar{C}_2$. We put the collections $\bar{C}_1 \cdot C_2$ and $C_1 \cdot \bar{C}_2$ together; these are sorted for C_3, and two collections are obtained, one with $\bar{C}_1 \cdot C_2 \cdot \bar{C}_3$ and $C_1 \cdot \bar{C}_2 \cdot \bar{C}_3$, the other with $\bar{C}_1 \cdot C_2 \cdot C_3$ and $C_1 \cdot \bar{C}_2 \cdot C_3$. This latter collection is put with $C_1 \cdot C_2 \cdot \bar{C}_3$ and these sorted for C_4 to obtain two collections, one with $\bar{C}_1 \cdot C_2 \cdot C_3 \cdot \bar{C}_4$, $C_1 \cdot \bar{C}_2 \cdot C_3 \cdot \bar{C}_4$, and $C_1 \cdot C_2 \cdot \bar{C}_3 \cdot \bar{C}_4$, the other with $\bar{C}_1 \cdot C_2 \cdot C_3 \cdot C_4$, $C_1 \cdot \bar{C}_2 \cdot C_3 \cdot C_4$, and $C_1 \cdot C_2 \cdot \bar{C}_3 \cdot C_4$. This latter collection and collection $C_1 \cdot C_2 \cdot C_3 \cdot \bar{C}_4$ form the collection of all items having all except one of the given characteristics. If this collection is empty, then we proceed to a third pass, as shown in the third row of Fig. 7-7, to find the collection of items with all except two of the characteristics, and so forth.

Searching with Permanent Conditions. Suppose that there were seven characteristics given, $C_1, C_2, \ldots, C_7$, and that no item of the collection under consideration were associated with all these seven characteristics. The next step would be to relax the conditions, but suppose that we must have items associated with C_5, C_6, and C_7, and with as many of the others as possible. The characteristic combination $C_5 \cdot C_6 \cdot C_7$ is called a *permanent condition.* The searching technique would be first to sort the collection to obtain all items associated with $C_5 \cdot C_6 \cdot C_7$. Then we would sort these on C_1, C_2, C_3, and C_4, as described above, to obtain all items each associated with $C_5 \cdot C_6 \cdot C_7$ *and* three, or two, or one, of C_1, C_2, C_3, C_4, As another illustration of a permanent condition, suppose that we desired to obtain all items associated *with one or more* of C_5, C_6, and C_7, and with as many of C_1, C_2, C_3, and C_4 as possible. In such a case we would first sort for C_5, sort the remaining items for C_6, and then sort those still remaining for C_7. These three sets would be put together to form the collection of items having one or more of C_5, C_6, and C_7. We would next sort this collection on C_1, C_2, C_3, and C_4, as described above, to obtain all items each of which is associated with C_5, C_6, and/or C_7, and three, or two, or one, of C_1, C_2, C_3, and C_4. In either of these examples, if no items were to have the permanent conditions, no further sorting could be accomplished.

Searching with Preferential Conditions. Suppose that there were nine characteristics given, $C_1, C_2, \ldots, C_9$, and that no item of the collection were associated with all nine given characteristics. However, suppose that a preference were associated with the characteristics so that, say, C_1, C_2, and C_3 have the most preference, C_4, C_5, and C_6 have the next preference, and C_7, C_8, and C_9 have the least preference. The searching procedure would first try for all of C_1, C_2, and C_3, or at least two of C_1, C_2, and C_3, or at least one, etc., as described above. Having chosen the items associated with the greatest number of the most preferred characteristics, these would be considered a permanent condition and would be sorted for all of C_4, C_5, and C_6, or at least two of C_4, C_5, and C_6, or at least one, etc. From the items having the most preferred characteristics we would now have chosen the items with the most characteristics of the next preference. Considering these as representing a permanent condition, we would try next for the items with the most characteristics of the least preference, that is, C_7, C_8, and C_9.

Note that the process of this section would be the natural one to follow in the problems of the previous paragraph when no items satisfy the desired permanent condition. The initial permanent condition would then be stated in terms of preferential conditions, and the process of this section would be used to obtain an approximation to the initial permanent condition.

Relaxing Sets of Conditions. Suppose that it is desired to search for all items associated with at least one of C_1, C_2, and C_3, *and* with at least one of C_4, C_5, and C_6, *and* with at least one of C_7, C_8, and C_9, *and* with at least one of C_{10}, C_{11}, C_{12}. Suppose that no such item exists; then the problem is to find all items that have at least one characteristic from at least three of the four sets of characteristics, or from at least two of the four sets, and so forth. Here we wish to relax sets of conditions. In order to facilitate the discussion, let us represent condition C_1, C_2, and/or C_3 by C^1, condition C_4, C_5, and/or C_6 by C^2, condition C_7, C_8, and/or C_9 by C^3, and condition C_{10}, C_{11}, and/or C_{12} by C^4. Then the searching technique becomes as described and illustrated above, with superscripts replacing subscripts, and with the single sorts replaced by the combinations of sorts necessary to determine the C^i.

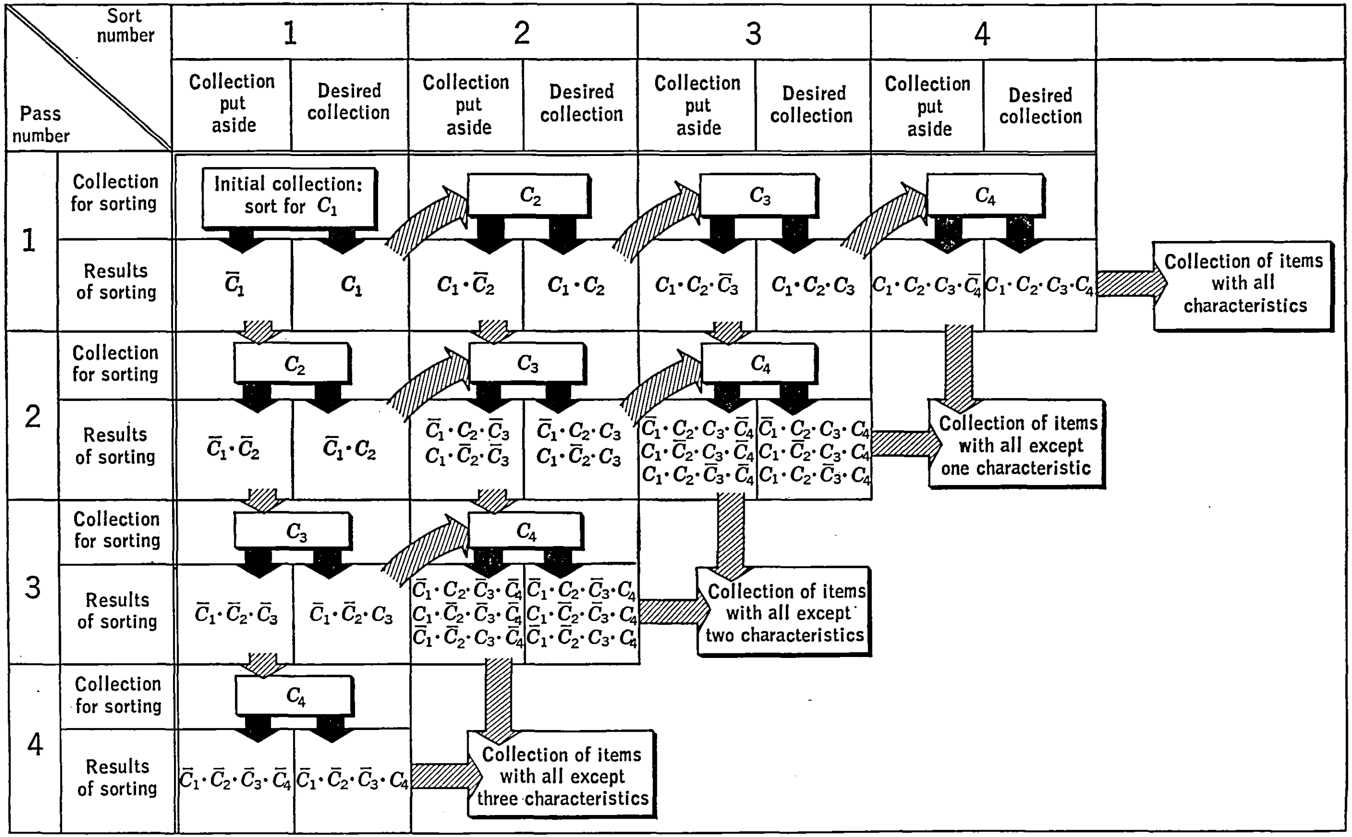

Sort number
Pass number
1
2
3
4
Collection put aside
Desired collection
Collection for sorting
Results of sorting
Initial collection: sort for C_1
$\bar{C}_1$
C_1
C_2
$C_1 \cdot \bar{C}_2$
$C_1 \cdot C_2$
C_3
$C_1 \cdot C_2 \cdot \bar{C}_3$
$C_1 \cdot C_2 \cdot C_3$
C_4
$C_1 \cdot C_2 \cdot C_3 \cdot \bar{C}_4$
$C_1 \cdot C_2 \cdot C_3 \cdot C_4$
Collection of items with all characteristics
$\bar{C}_1 \cdot \bar{C}_2$
$\bar{C}_1 \cdot C_2$
$\bar{C}_1 \cdot C_2 \cdot \bar{C}_3$
$C_1 \cdot \bar{C}_2 \cdot \bar{C}_3$
$\bar{C}_1 \cdot C_2 \cdot C_3$
$C_1 \cdot \bar{C}_2 \cdot C_3$
$\bar{C}_1 \cdot C_2 \cdot C_3 \cdot \bar{C}_4$
$C_1 \cdot \bar{C}_2 \cdot C_3 \cdot \bar{C}_4$
$C_1 \cdot C_2 \cdot \bar{C}_3 \cdot \bar{C}_4$
$\bar{C}_1 \cdot C_2 \cdot C_3 \cdot C_4$
$C_1 \cdot \bar{C}_2 \cdot C_3 \cdot C_4$
$C_1 \cdot C_2 \cdot \bar{C}_3 \cdot C_4$
Collection of items with all except one characteristic
$\bar{C}_1 \cdot \bar{C}_2$
$\bar{C}_1 \cdot \bar{C}_2 \cdot \bar{C}_3$
$\bar{C}_1 \cdot \bar{C}_2 \cdot C_3$
$\bar{C}_1 \cdot C_2 \cdot \bar{C}_3 \cdot \bar{C}_4$
$C_1 \cdot \bar{C}_2 \cdot \bar{C}_3 \cdot \bar{C}_4$
$\bar{C}_1 \cdot \bar{C}_2 \cdot C_3 \cdot \bar{C}_4$
$\bar{C}_1 \cdot C_2 \cdot \bar{C}_3 \cdot C_4$
$C_1 \cdot \bar{C}_2 \cdot \bar{C}_3 \cdot C_4$
$\bar{C}_1 \cdot \bar{C}_2 \cdot C_3 \cdot C_4$
Collection of items with all except two characteristics
$\bar{C}_1 \cdot \bar{C}_2 \cdot \bar{C}_3 \cdot \bar{C}_4$
$\bar{C}_1 \cdot \bar{C}_2 \cdot \bar{C}_3 \cdot C_4$
Collection of items with all except three characteristics

FIG. 7-7. Analysis of method of searching for all items each of which is associated with all, all except one, all except two, etc., of the given characteristics C_1, $\mathcal{E}_2$, C_3, and C_4.

EXERCISES

(*a*) Punch a hole in each corner of sixteen 3 × 5 cards. Then write the names of the authors (see Fig. 7-1) on these cards, one name on each card. Mark off eight equally spaced positions on one 5-in. edge, nine positions on the other 5-in. edge of each card. Assign one word for each position, the same on each of the 16 cards. Next notch the edge of each card corresponding to the positions of words characteristic of this author's article (see Table 7-1). The cards can now be searched by placing a knitting needle under the deck of cards at the position of the desired word (see Fig. 7-8); the notched cards will drop, and the unnotched ones can be separated by putting another knitting needle through a corner hole of the raised cards. By appropriately separating collections of these cards as in Fig. 7-7, find all items associated with at least six, at least five, at least four, and at least three of the following seven given characteristics: *differentiation, application, England, evaluation, design, gas,* and *concept.*

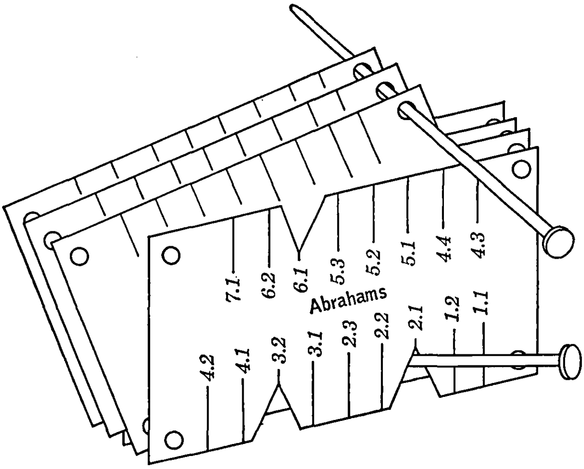


FIG. 7-8. Use of knitting needles to search marginal-punch cards.

(*b*) Using the cards made in (*a*), find all items that are associated with *design* and *England* and as many of the following as possible: *differentiation, application, evaluation, gas,* and *concept.*

(*c*) Using the cards made in (*a*), find all items that are associated with either *evaluation* or *design* or both and as many of the following as possible: *differentiation, application, England, gas,* and *concept.*

(*d*) How can the process of searching for all except one, except two, etc., be accomplished by means of the Tabledex tables? (HINT: A succession of tables must be used: one table for the first pass, two tables for the second pass, three tables for the third pass, etc.)

(*e*) Draw a flow chart for a search for items associated with all the given characteristics except one, except two, etc. First assume the data to be given as in Fig. 7-2*b*. Then assume the data to be given in terms of the Tabledex tables, as in Fig. 7-2*c*.

7-5. Sorting and Ordering

As discussed in Sec. 7-1, sorting can always be accomplished by ordering; in this section we shall turn our attention to ordering. Sorting and ordering are essentially the same process, except that ordering carries the process all the way, while sorting carries the process only part of the way. In our discussion of ordering we shall refer to the ordering of a given list of *numbers* in order to be concrete, but it is clear that the methods apply for any linear ordering rule.

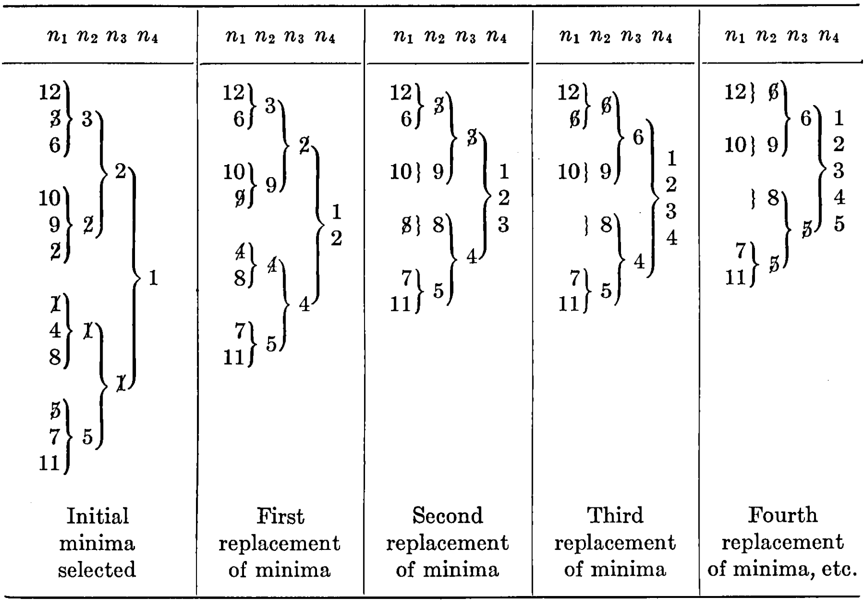
Minimum- or Maximum-in-pass. The *first* of the five methods of ordering to be presented is called the *minimum-in-pass* method. The description will be given in terms of the use of magnetic tape, but the principles involved are perfectly general. In this method, on the first pass over the tape the minimum number is recorded as follows: The first number is recorded in a temporary cell and is compared successively with the other numbers until a smaller number appears. The first number replaces this smaller number on the tape as this number is placed in the temporary cell. The successive comparisons are continued until another smaller number appears; again the two numbers are interchanged, and so forth. At the end of the first pass it is clear that the smallest number appears in the temporary cell. This is placed as the first number on the tape, and the *second pass* is started with the *second number* on the tape, proceeding as before. At the end of this pass the second smallest number appears in the temporary cell and is placed as the second number on the tape, and so forth. The ith pass starts with the ith item, determines the ith smallest number, and puts it in the ith position on the tape. For N numbers this method requires $N - 1$ passes through the tape; however, if the tape can be positioned, starting each pass with the next number in turn effectively reduces the number of passes by half, to approximately $N/2$.

It is obvious that the maximum instead of minimum number could have been recorded on each pass (*maximum-in-pass*) and placed at the end of the remaining list; this method would again require about $N/2$ passes. If both the minimum and a maximum are recorded in each pass, the latter being placed on the tape at the end of the pass, the former at the beginning of the next pass, the number of passes is again halved, to approximately $N/4$. Depending on the size of the high-speed memory, the least two, or least three, or least m numbers, and the greatest two, or greatest three, or greatest n numbers could be found on each pass (and ordered within the high-speed memory), further reducing the number of passes required.

If more than one tape is available, the minimum-in-pass method can be accomplished in parallel. The original list to be ordered is divided into sublists (say n_1 in number). A new list is made from the minima which are *removed* from each of the n_1 sublists. This new list is divided into n_2 sublists, and a third list is made from the minima which are *removed* from each of the n_2 sublists. As many new lists as desired can be made, and the first number of the ordered list will be the minimum of the last such list. As a number is removed from a list, it is replaced by the minimum from the sublist from which it was originally taken (until the sublist is exhausted). Table 7-2 illustrates the process applied to an original list of 12 numbers, divided into $n_1 = 4$ sublists. The minima from these are divided into $n_2 = 2$ sublists; their minima are then divided into $n_3 = 1$ sublist, whose minimum goes into the final ordered list, labeled n_4.

Transposition. The *second method* orders a list by *transposition.* The process is to transpose adjacent numbers on the list if the order of their characteristics is inverted. It starts by comparing the first two numbers and continues on *overlapping pairs* through the list. Pass after pass is made until no transpositions are required on an entire pass; then the list is in order. For example, consider Table 7-3 (where the caret represents comparison, the t transposition). Here three passes were necessary to order the numbers. The fourth pass will result in *no* further transpositions, indicating that the list is in order. A variation on this method is to order successive overlapping sets of three or more (say r) numbers; a single tape may still be used in this process, and the number of passes required is approximately divided by $r - 1$. Table 7-4 illustrates the procedure for overlapping sets of three numbers.

TABLE 7-2. MINIMUM-IN-PASS ORDERING USING MULTIPLE SUBLISTS



Ordering by Radix Coefficients. The *third method* is probably the most obvious of all the methods when a clearly defined radix occurs; in the present discussion we shall consider decimal numbers (i.e., radix ten). The method consists in sorting the numbers by the most significant figure, putting the sorts in order. The numbers within each sort are ordered on the second most significant figure; then *within each of these subsorts* the numbers are ordered on the third most significant figure; and so forth. For an example, see Table 7-5. This method has the disadvantage that after the first pass the computer must work on subregions of the tape, which can present difficulties. For sorting punch cards, for example, this means that individual sorts must be made on successively smaller and smaller pieces of the original deck; such handling becomes increasingly inconvenient.

TABLE 7-3. ORDERING BY TRANSPOSITION†

1st pass	5	4	4	4	4
	4	5	1	1	1
	1	1	5	3	3
	3	3	3	5	2
	2	2	2	2	5
2d pass	4	1	1	1	1
	1	4	3	3	3
	3	3	4	2	2
	2	2	2	4	4
	5	5	5	5	5
3d pass	1	1	1	1	1
	3	3	2	2	2
	2	2	3	3	3
	4	4	4	4	4
	5	5	5	5	5

† Successive overlapping pairs are scanned.

The *fourth method*, which is essentially the inverse of the third, eliminates this difficulty. Here the first sort is made on the least significant figure. Then the entire tape is sorted on the second least significant figure, and so forth. For an example, see Table 7-6. However, both these methods offer the difficulty that simultaneously used multiple tape units are indicated. Of course the subsorts might be made by one of the first two methods, whence the entire sorting procedure becomes a mixture of methods.

TABLE 7-4. ORDERING BY TRANSPOSITION WITHIN OVERLAPPING SETS OF THREE NUMBERS

12	3									3	3								3	1								1
3	6	6								6	6	2							2	2	2							2
6	12	10	9							9	9	6	1						1	3	3	3						3
10	10	12	10	2						2	2	9	6	4					4	4	4	4	4					4
9		9	12	10	1					1		1	9	6	6				6		6	6	5	5				5
2			2	12	10	4				4			4	9	8	5			5			5	6	6	6			6
1				1	12	10	8			8				8	9	8	7		7				7	7	7	7		7
4					4	12	10	5		5					5	9	8	8	8					8	8	8	8	8
8						8	12	10	7	7						7	9	9	9						9	9	9	9
5							5	12	10	10							10	10	10							10	10	10
7								7	12	11								11	11								11	11
11									11	12									12									12

Suppose that for the fourth method ten bins, or magnetic-tape units, in addition to the unit holding the original list, were not available for the separation by least significant decimal digit. For example, suppose that only 5 units were available, 1 of which contains the original list. Then we could convert the least significant decimal digit of each number to two radix-4 digits and in two passes order the list by the least significant decimal digit. Similarly two passes would be required for the next least significant decimal digit, and so forth.

TABLE 7-5. ORDERING BY THE THIRD METHOD

Original list	Sort by most significant figure	Sort by 2d most significant figure	Sort by 3d most significant figure
345	256	256	255
344	265	255	256
537	255	265	264
256	264	264	265
554	345	337	337
447	344	345	344
337	337	344	345
265	347	347	347
255	447	447	446
466	466	446	447
446	446	466	464
464	464	464	466
347	537	537	336
264	554	536	337
536	536	544	544
544	544	554	554

TABLE 7-6. ORDERING BY THE FOURTH METHOD

Original list	Sort by least significant figure	Sort by 2d least significant figure	Sort by 3d least significant figure
345	344	536	255
344	554	537	256
537	464	337	264
256	264	344	265
554	544	544	337
447	345	345	344
337	265	446	345
265	255	447	347
255	256	347	446
466	466	554	447
446	446	255	464
464	536	256	466
347	537	464	536
264	447	264	537
536	337	265	544
544	347	466	554

EXERCISES

(*a*) Draw a flow chart for coding each of the above four ordering methods.

(*b*) After shuffling a deck of playing cards, order them by each of the four methods without regard to suit (i.e., the four Queens, for example, will appear adjacent, after the Jacks and before the Kings, but not ordered by suit among themselves).

(*c*) Suppose that one were ordering alphabetically by the fourth method, using a computer with 10 tape units. What radix could be used, and how would the tape units be utilized?

7-6. Ordering by Merging

Merging. The *fifth method* is based on the simple idea of the *merging of two ordered lists.* For example, consider these two ordered lists of numbers:

265		255
337		264
446	and	464
466		536
		544

Forming a single ordered list containing all the numbers is accomplished by successively comparing the first numbers on each list and removing the smaller number to a third list. Thus, since 255 < 265, 255 becomes the first number of the new list; now 264 becomes the first number of the second list, and since 264 < 265, 264 becomes the second number of the new list; next, since 265 < 464, 265 becomes the third number of the new list; and so forth. In this manner the two ordered lists are *merged* to form a single new ordered list, namely,

255
264
265
337
446
464
466
536
544

In more mathematical notation, if the two ordered lists are $a_1 < a_2 < \cdots < a_I$ and $b_1 < b_2 < \cdots < b_J$, then the merged list, namely, $C_1 < C_2 < \cdots < C_{(I+J)}$, is formed as indicated in the flow diagram of Fig. 7-9.

The initial step of the general ordering by merging process is to separate the given list into *two* lists in a manner that preserves any ordering that might have originally appeared among adjacent numbers. The first number of the given list is used to start the first of two generated lists.

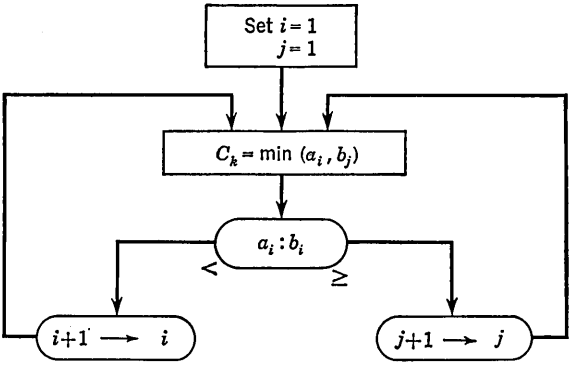


FIG. 7-9. Flow chart for merging two ordered lists.

Then successive numbers of the given list are added to this new list so long as they appear in the proper order. The first number encountered that appears out of order is used to start the second generated list. All successive numbers that are in the proper order are placed successively in this second list. When a number out of order again appears, it is placed in the *first* list, with those following numbers which are in the proper order, and so forth. Thus in forming the two new lists we switch output lists whenever a number appears out of order, so that any order within a string of adjacent numbers in the given list will be retained in one of the two lists. For the example below, strings of numbers in the original list that appear in the proper order are included in braces; these are alternately placed in the two generated lists.

Original list	*New list*	*New list*
{345	{345	⎧344
⎧344	⎧256	⎩537
⎩537	⎩554	{447
⎧256	{337	{265
⎩554	⎧255	⎧446
{447	⎩466	⎩464
{337	{347	⎧264
{265		⎨536
⎧255		⎩544
⎩466		
⎧446		
⎩464		
{347		
⎧264		
⎨536		
⎩544		

When this initial step is finished, the ordering proceeds as follows: Each of the two lists contains sublists that are ordered, and each sublist of one can be paired with a sublist of the other, except perhaps the last sublist of one of the lists (see the horizontal lines below). The pairs of ordered sublists are then merged. The merged results are again placed alternately† in two new lists as shown below:

345	merge	344 537
256 554	merge	447
337	merge	265 446 464
255 466	merge	264 536 544
347		

New list	*New list*
344 345 537	256 447 554
265 337 446 464	255 264 466 536 544
347	

The process is repeated, forming each time two lists of increasingly longer ordered sublists, until all the numbers appear in one ordered list.

† The alternation of the merged sublists in the new lists ensures the maximum number of pairings of sublists for the next merge; if the total number of sublists is known, the same result would be obtained by putting the first half on one tape, the second half on the other.

Second merge results		*Third merge results*		*Ordered list*
256	255	255	347	255
344	264	256		256
345	265	264		264
447	337	265		265
537	446	337		337
554	464	344		344
	466	345		345
	536	446		347
	544	447		446
347		464		447
		466		464
		536		466
		537		536
		544		537
		554		544
				554

Often four magnetic tape units are used in this ordering process. Two of the tape units hold the old lists, and the other two record the results of the merging. Then, when both old lists have been exhausted, the newly formed lists become the old lists and the pairs of tape units exchange functions. In merging two sublists, the end of a sublist is sensed when $a_i > a_{i+1}$; when both sublists are exhausted, i.e., when both $a_i > a_{i+1}$ and $b_j > b_{j+1}$, then the merge starts on the next pair of sublists, the results being put on the *other* receiving tape unit. Figure 7-10 gives a flow chart for this process.

Variations on Merging. The technique of ordering by merging two ordered lists is easily extended to ordering by merging more than two

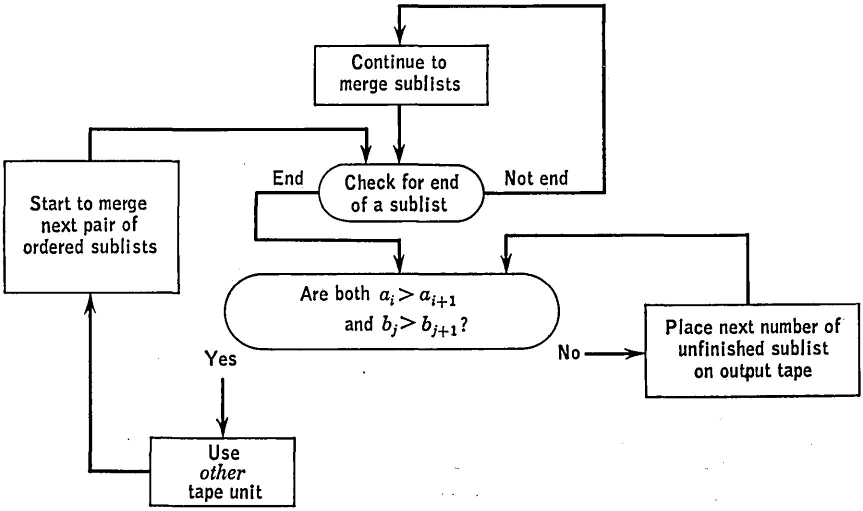

Continue to merge sublists
Start to merge next pair of ordered sublists
End
Check for end of a sublist
Not end
Are both $a_i > a_{i+1}$ and $b_j > b_{j+1}$?
Yes
No
Place next number of unfinished sublist on output tape
Use *other* tape unit

FIG. 7-10. Flow chart for ordering by merging.

ordered lists. The initial separation of a list to be ordered can involve more than two new lists, and then the merging of ordered sublists would be accomplished successively with one sublist from each. If the initial step formed m new lists and the merging of sublists from these formed n new lists, then $m + n$ magnetic tapes would be involved. Below we illustrate the case for three initial new lists (i.e., $m = n = 3$):

	Initial step			*First merge*			*Ordered list, second*
Original list	*New list*	*New list*	*New list*		*results*		*merge result*
345	345	344	256	256	255	264	255
344	447	537	554	344	265	536	256
537	255	337	265	345	337	544	264
256	466	446	347	447	347		265
554		464		537	446		337
447	264			554	464		344
337	536				466		345
265	544						347
255							446
466							447
446							464
464							466
347							536
264							537
536							544
544							554

Another variation takes further advantage of any intrinsic ordering in the original list. Here a tape is not closed off as a smaller item occurs, but rather is held open to receive any larger item later in the list. Each new item is compared with the final items on each tape, in order of largest to smallest, and is put on that tape whose last item is as large as possible without exceeding the new item. If the new item is smaller than all the final items, it is placed on the tape with the largest item to start a new sublist. Using this technique to sort the above list of numbers, merging from two lists, gives the following (with the sublists indicated):

Initial step		*First merge results*		*Second merge results*	
345	344	344	256	256	255
537	*554*	345	337	337	264
256	337	537	447	344	265
447	255	554	*464*	345	347
265	446	255	*466*	447	446
466	464	265	264	464	536
347	264	446	347	466	544
536		536		537	
544		544		554	

The third merge, instead of the fourth required above for two-tape merging, will give the ordered list. Note how changing tapes for the italicized numbers allows the lengthening of the sublists.

Comparison of Ordering Methods. The number of passes, i.e., the number of times one must look through all the numbers of the original list, is very large for the first two methods as compared with the merge method. The straight minimum-in-pass method requires $N/2$ (or $N/4$ or perhaps less depending on the equipment) passes for N numbers. The merge sort using $2T$ magnetic tape units requires no more than $K + 1$ passes, where K is the smallest integer such that $T^K \geq N$. The merge sort has the additional feature that full advantage is taken of any ordered sublists that may appear in the original list, and if the numbers are randomly distributed, there ought to be about $N/2$ ordered sublists. In such a case $1 + K'$ passes are required, where K' is the smallest integer such that $T^{K'} \geq N/2$. For the transposition method there can be as many as $n(n + 1)/2$ passes, but here again any ordered sublists that may appear in the original list can reduce this number of passes. The variation of the transposition method is of course more efficient. Ordering by the radix-coefficient methods can compete with the merge sort when all numbers from 1 to N are in the list, with practically no ordered sublists. But if the numbers have more than K digits, where K is the smallest integer such that $R^K > N$, where R is the radix being used, then the merge sort would require fewer passes. Since the radix-coefficient methods do not take advantage of any ordering that might already exist in the list, the merge sort can be still more advantageous. However, if only a single tape is available, only the minimum-in-pass or transposition can be done.

Of course the number of passes required for ordering a list is not necessarily the only criterion applied in deciding on a method for ordering. The nature of the equipment available, e.g., how many magnetic-tape units, or bins, are available and whether these can or cannot be used concurrently, how complex operations can be performed, etc., often limit the types of methods that can be used. The number of items or numbers to be sorted, the number of times such a process must be repeated, etc., all enter into the choice of the ordering system. In general a combination of ordering techniques can be used to take advantage of particular situations that may occur at the different levels of the ordering process. However, if we were asked to choose which system is of greatest importance, we would say the merge ordering method and its variations have the most advantages.

EXERCISES

(*a*) Draw a flow chart for coding a merge sort using four tapes, passing the numbers from two tapes to the other two.

(*b*) Draw a flow chart for coding a merge sort, passing the numbers from m tapes to n tapes, where m and n can be specified arbitrarily.

(*c*) What is the maximum number of passes required to put a deck of 52 cards in some order by suit as well as by denomination, merging from two bins to two bins? From three bins to one bin (counting the pass required to redistribute the cards into the three bins after each merge)? From four bins to four bins? From five bins to three bins? Considering these results, discuss the most efficient way to utilize a fixed number of tape units in a merge sort.

7-7. Codifying: Error Correction and Superimposition

Encoding, Decoding, and Codifications. By *codifying*† is meant the process of relating to an item or characteristic a *set of symbols*, called a *codification.* As we have seen (cf. Sec. 7-1), codifying is intimately associated with searching, sorting, and ordering. Codifying is used for three purposes: First just for convenience in handling the information in a shorter form rather than in raw form; second for making the information compatible with a particular mechanical or electronic processing device being used; and third to facilitate the particular method of searching, sorting, or ordering being used.

In general, codifying involves two processes, called *encoding* and *decoding.* Encoding is the process of generating the codification related to a given item or characteristic; decoding, conversely, is determining the items or characteristics related to a given codification. The processes of encoding and decoding are accomplished by means of either a *code book* or a *relation scheme.* For example, a telephone book is an encoding code book that gives the codification (i.e., the telephone number) related to an item (i.e., a person); the decoding is accomplished after dialing, by means of a scheme built into the telephone company's electrical dialing equipment. Another example is the biological codification of the genes, where the encoding is the evolutionary process and the decoding is the ontogenetic process that results in, say, an individual animal.

The codification of items or characteristics can be either *substantive* or *nonsubstantive* in character. A substantive codification is one in which the symbols have meaning in themselves. A substantive codification of names, for example, might comprise simply the first four consonants of the last names with the initials. A word in a dictionary is a substantive codification of its meaning. The genes, of course, represent an outstanding example of substantive codification.

For a nonsubstantive codification consider the example of the codification of research reports by accession numbers, or the codifying of an item in a bibliography by the page and line number on which it is found. Nonsubstantive codifications are frequently merely *addresses* that tell the locations of the related items or characteristics. An exception to this is the population-based (nonsubstantive) codification considered in Sec. 7-2, in which the symbol for a characteristic is the number of items associated with that characteristic.

The relationships between encoding, decoding, and information processing are summarized in Fig. 7-11. For example, consider our illus-

† See footnote on p. 217.

trative bibliography discussed above. Here the encoding was the relating of population-based numbers to the given characteristics (i.e., the given retrieval words) by means of a code book. The processing was searching, where, for instance, the Tabledex method might be used, which resulted in a collection of numbers related to the associated items (i.e., the page and line numbers of desired article references). Finally the decoding process was accomplished by locating the items at their address (i.e., the entries are retrieved from their page and line numbers). The information processing here consisted in transforming a set of characteristic codifications into a set of item codifications.

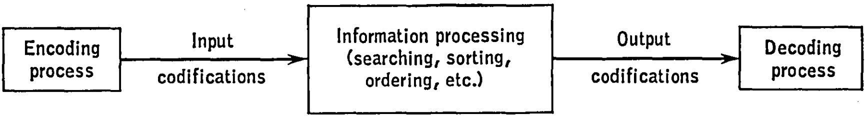


FIG. 7-11. Relation between encoding, decoding, and information processing.

Efficiency of Codifications. To determine how many symbol *positions* are required in an unambiguous codification using K different symbols, *we consider each symbol as a number in the base (radix) K and use enough symbol positions to count the number of items in the base K.* That is, with N items (or characteristics) and K symbols, *the number of symbol positions in a codification must be no smaller than the smallest integer n such that $K^n \geq N$ (or $n \geq \log_K N$) if the codification relating to an item is to be unique in the list.* For example, for 600 items and a binary codification (that is, $K = 2$), we have†

$$n \geq \log_2 600 = \frac{2.7782}{0.3} = 9.261$$

or $n = 10$. If the number of positions is greater than n, that is, if more positions are being used than are necessary, the codification is called *redundant.* If fewer than the required n positions are used, the codification is called *irredundant* and the code for an item is *not* unique in the list. A codification that is neither redundant nor irredundant is called *efficient.*

For example, an English word can be considered as a codification of an idea using the 26 alphabetical symbols. Suppose that we have a dictionary all the words of which are no longer than 10 letters. We can make all the words the same length by introducing a 27th letter, say α, and filling up with α's all positions to the left of a word to make 10 positions altogether, if necessary. It is then possible to have more than 145×10^{12} different words in our dictionary. However, a desk dictionary has only approximately 100,000, or 10^5, words, and so a dictionary is redundant. Then again, that's to be expected.

In the next paragraphs we shall consider a use for purposely redundant

† Recall that $\log_K N = \log_{10} N / \log_{10} K$. For $K = 2$, $\log_2 N = \log_{10} N / \log_{10} 2$, where $\log_{10} 2 \approx 0.3$.

codification; that discussion will be followed by consideration of a use for purposely irredundant codifications.

Self-correcting Redundant Codification. Frequently situations arise where errors may be introduced into a codification during storage or transmission. If errors cannot be tolerated, a redundant code can be devised with the extra positions of the redundancy used to correct such errors as may occur. However, the self-correcting scheme that makes use of redundant positions *must also be able to correct errors* that may occur in these extra *redundant positions* as well. The process can be outlined as follows: First from the given information-carrying codification there is generated the redundant codification. This redundant codification is stored or transmitted, etc., during which time some errors can enter into the codification. Finally, when the information is to be used, there is generated from the redundant codification (which may contain some errors) the *correct* information-carrying codification in spite of any errors in the redundant codification (see Fig. 7-12). In what follows we shall confine ourselves to *binary codification.*

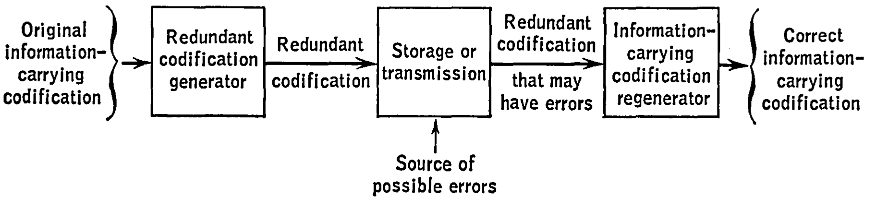

Original information-carrying codification
Redundant codification generator
Redundant codification
Storage or transmission
Redundant codification that may have errors
Information-carrying codification regenerator
Correct information-carrying codification
Source of possible errors

FIG. 7-12. The self-correcting redundant-codification process.

Let us first illustrate a self-correcting technique that will correct a single error. The first step is the formation of the redundant codification. Suppose that the information-carrying part of the codification has 4 bits, I_0, I_1, I_2, I_3—say, for example, 1101. We form an array as shown in Fig. 7-13 (with the top row all units and the columns otherwise having all possible combinations of zeros and units). The information-carrying codification is set down beside the array as a column. The desired redundant codification is formed bit by bit by comparing this column successively with each column of the array, counting the number of units in common (i.e., logically multiplying the columns and counting the number of units in common; see Sec. 4-5 and Part 3). If the number of units is even, a zero is placed under that column of the array; if odd,

		0123	4567
Information-carrying codification	$I_0 = 1$	1111	1111
	$I_1 = 1$	0101	0101
	$I_2 = 0$	0011	0011
	$I_3 = 1$	0000	1111
Redundant codification		1010	0101

FIG. 7-13. Work for forming redundant codification for single-bit self-correction.

a unit. For example, for the sixth bit, under column 6 we have

$$\begin{pmatrix}1\\1\\0\\1\end{pmatrix}\cdot\begin{pmatrix}1\\0\\1\\1\end{pmatrix}=\begin{pmatrix}1\\0\\0\\1\end{pmatrix}$$

$$\therefore r_6 = \overline{0}$$

for the seventh bit

$$\begin{pmatrix}1\\1\\0\\1\end{pmatrix}\cdot\begin{pmatrix}1\\1\\1\\1\end{pmatrix}=\begin{pmatrix}1\\1\\0\\1\end{pmatrix}$$

$$\therefore r_7 = \overline{1}$$

and so forth.

Suppose that this redundant codification 1010 0101 is stored, and during the storage the third bit r_3 becomes changed. Then the incorrect code is 1011 0101. Of course we assume that we *know nothing* about the error in the codification except that there may be at most a single error. The problem remains to determine the correct bits I_0, I_1, I_2, and I_3 of the original information-carrying codification. The correct value of the bits I_1, I_2, I_3, and I_0 will each be derived, in this order, in spite of the error in the code. First note that, if $I_1 = 0$, then because of the method of formation we would clearly have $r_0 = r_1$, $r_2 = r_3$, $r_4 = r_5$, and $r_6 = r_7$; that is, adjacent pairs of bits would be the *same*. If $I_1 = 1$, then *none* of these adjacent pairs would be the same. A single error can affect only a single pair of positions; hence, to find I_1, we simply check to see whether we have more "same" pairs, or more "different" pairs. More same pairs means $I_1 = 0$; more different pairs means $I_1 = 1$. Similarly I_2 is determined from pairs (r_0,r_2), (r_1,r_3), (r_4,r_6), and (r_5,r_7) and I_3 from pairs (r_0,r_4), (r_1,r_5), (r_2,r_6), and (r_3,r_7) (see Fig. 7-14 for the work). To find I_0, consider an array composed of the rows of the original array corresponding to the I_1, I_2, or I_3 which are found to be units and the redundant-code row. Under each column at this array place a zero if the number of units in the column is even, a unit if odd. If the number so formed is mostly zeros, $I_0 = 0$; if mostly units, $I_0 = 1$.

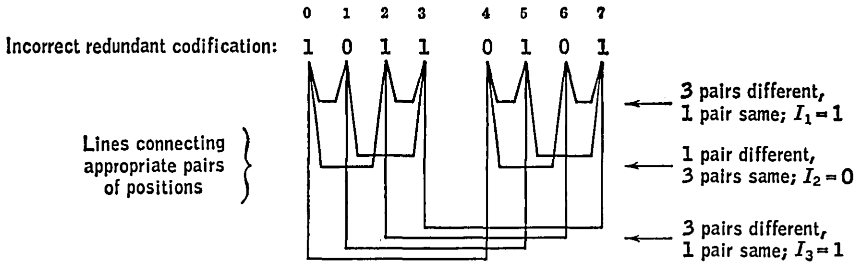

0 1 2 3 4 5 6 7
Incorrect redundant codification: 1 0 1 1 0 1 0 1
Lines connecting appropriate pairs of positions
3 pairs different, 1 pair same; $I_1 = 1$
1 pair different, 3 pairs same; $I_2 = 0$
3 pairs different, 1 pair same; $I_3 = 1$

FIG. 7-14. Work for finding I_1, I_2, and I_3.

Incidentally this new number clearly shows the position of the single error (see Fig. 7-15).

As another illustration, consider a 5-bit information-carrying codification and an extension of the above scheme to correct as many as 3 erroneous bits in the redundant codification. Let the original 5 bits be $I_0 = 0$, $I_1 = 0$, $I_2 = 1$, $I_3 = 1$, and $I_4 = 0$. Figure 7-16a represents the work required to generate the redundant codification which is to be transmitted or stored; the work proceeds just as in the example above. Suppose that erroneous codification shown in Fig. 7-16 results from the storage, then the work required to determine the correct 5 information-carrying codification bits appears in Fig. 7-16b.

	0123	4567	
$I_1 = 1$	0101	0101	
$I_3 = 1$	0000	1111	
Incorrect redundant codification	1011	0101	
New number	1110	1111	← more units; ∴ $I_0 = 1$

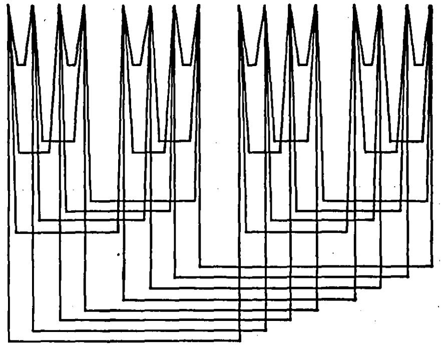
FIG. 7-15. Work for finding I_0. (Note that error occurred in position 3.)

(*a*) Generating the redundant codification

Information-carrying codification

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
$I_0 = 0$	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
$I_1 = 0$	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1
$I_2 = 1$	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1
$I_3 = 1$	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1
$I_4 = 0$	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1
Redundant codification:	0	0	1	1	1	1	0	0	0	0	1	1	1	1	0	0
Erroneous codification:	0	0	1	0	1	1	1	0	0	0	1	1	0	1	0	0

(*b*) Regenerating the original information



3 pairs different
5 pairs same →
∴ $I_1 = 0$

5 pairs different
3 pairs same →
∴ $I_2 = 1$

5 pairs different
3 pairs same →
∴ $I_3 = 1$

3 pairs different
5 pairs same →
∴ $I_4 = 0$

	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1
	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1
	0	0	1	0	1	1	1	0	0	0	1	1	0	1	0	0
New number: ∴ $I_0 = 0$	0	0	0	1	0	0	1	0	0	0	0	0	1	0	0	0

Note that errors occurred in positions 3, 6, and 12.

FIG. 7-16. Scheme illustrating three-error correcting process.

Observe that in the first example we used 8 bits to codify 4 information-carrying bits, or a redundancy of 4 bits; in the second example we used 16 bits to codify 5 bits, or a redundancy of 11 bits. In the first case we are trying to recognize 9 different possibilities, i.e., no error, or a single error in one of eight positions. This clearly takes 4 bits, the redundancy used. For the second case we are trying to recognize 697 different possibilities, i.e., no error, a single error in one of 16 positions, a double error that can occur in $\binom{16}{2} = 120$ different ways, or a triple error that can occur in $\binom{16}{3} = 560$ different ways.[†] To distinguish among 697 possibilities, we clearly need 10 bits; so the 11-bit redundancy of the second example is one more than the minimum required. However, note that the 0th column of the array could have been omitted, and the scheme would work just as well. This would give us our minimum 10-bit redundancy.

Superimposed Irredundant Codification. As we have seen, a binary codification that will distinguish uniquely between N characteristics requires at least $\log_2 N$ bit positions. Thus, if 4,500 characteristics are in the list, 13 bits are required. It frequently happens that, although N characteristics *could occur*, fewer than N of these *actually appear* in a particular situation. However, it is in general not known *beforehand* how many will appear. In such cases, particularly when a *substantive codification for the characteristics is used,* at least $\log_2 N$ bit positions are used for the codification anyway, to take account of all eventualities. Thus, even though not all 4,500 characteristics will occur, 13 bits might still be reserved for the codification.

Furthermore, suppose that we are dealing with items each of which is associated with X of these characteristics; then, according to our above discussion, at least $X \log_2 N$ bit positions must be used to identify an item. For example, if $X = 3$ and $N = 4{,}500$, then $3 \times 13 = 39$ bit positions would be necessary. Thus, in principle, N^X items can be distinguished this way; but as we observed above, we are here considering situations where they do *not* all actually appear. Evidently we are being wasteful of bit positions because fewer than N^X items require fewer than $X \log_2 N$ bit positions for a codification. The problem therefore arises: *How can we use fewer than $X \log_2 N$ bits for our codification,* say, *$H < X \log_2 N$, and yet maintain our ability to distinguish among all N characteristics?* The answer is found in the concept of superimposed codification.

By the *superimposed codification corresponding to X characteristics* we mean the result of forming the *logical sum* (see Sec. 4-5 and Part 3) of the individual codifications for these X characteristics. That is, if the units represent, for example, notches on the edge of a card, two

[†] Recall that the number of ways of taking X objects Y at a time is given by the binomial coefficient $\binom{X}{Y} = \frac{X!}{Y!(X-Y)!}$.

notches in the same position are equivalent to a single notch. Thus, if three characteristics have the following codes:

Characteristic	Position																			
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
1st..........	0	0	1	0	0	0	1	0	1	0	1	0	0	0	0	0	0	0	0	0
2d..........	0	0	0	1	0	0	1	0	1	0	0	0	0	1	0	0	0	0	0	0
3d..........	0	0	0	0	0	0	0	0	0	1	1	0	0	0	0	1	0	0	1	0

then the corresponding superimposed codification is

0 0 1 1 0 0 1 0 1 1 1 0 0 1 0 1 0 0 1 0

We shall illustrate by means of an example how superimposed codification can be made to satisfy the above criterion of using fewer than $X \log_2 N$ bits and yet maintain the ability to distinguish among all N characteristics. Suppose as above that $N = 4{,}500$. Let us codify the characteristics as follows: Use 20 positions, and let each codification contain precisely 4 units. This codification will allow us to code each characteristic uniquely, because 20 positions can be taken four at a time in $\binom{20}{4} = \frac{20!}{4!(20-4)!} = 4{,}845$ different ways, and $4{,}845 > 4{,}500$. Finally, suppose that each item is associated with three characteristics; that is, $X = 3$. Then *let the codification of the item be the superimposed codifications of its associated three characteristics.* Note that the codification for an item now comprises only 20 positions, rather than the 39 thought necessary above, saving about half the positions.

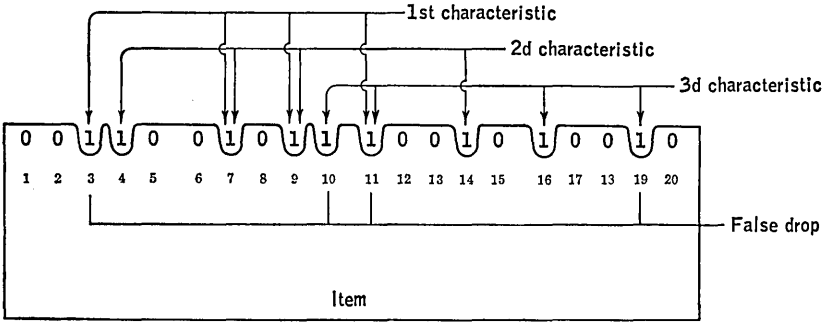


FIG. 7-17. Example of superimposed codification.

The advantages of this technique can most readily be visualized in terms of the marginal punched cards described above, where each card represents an item, and where the marginal positions are the bit positions with a punch representing a unit, no punch a zero (see Fig. 7-17). Suppose that an item is associated with the three characteristics whose codifications appear above and that the 20 marginal positions of the item card are punched with the superimposed codification. Then, if all items associated with the first characteristic are being searched for, the sorting needles are placed in positions 3, 7, 9, and 11, and clearly the proper item cards will drop. Similarly for the second or third characteristic. However, *we must evidently pay a price for this convenience and saving in item-codification positions.* For suppose that the code for some other characteristic is

$$00\overset{3}{1}00 \qquad 0000\overset{10}{1} \qquad \overset{11}{1}0000 \qquad 000\overset{19}{1}0$$

Our item card (see Fig. 7-17) is *not* associated with this characteristic but nonetheless will drop in the search for items associated with this characteristic. The reason for this is that our codifications overlapped, allowing for more than three combinations of 4 units. In fact, in our illustrative example, nine punches appear, allowing for

$$\binom{9}{4} = \frac{9!}{4!(9-4)!} = 126$$

such possibilities, and hence $126 - 3 = 123$ possible *false* combinations. However, the situation is not so bad as it seems, for remember that we supposed that not all the $\binom{20}{4} = 4{,}845$ possible characteristics actually appear, and hence not all the false combinations will ever be sensed.

The subject of superimposed codification therefore resolves itself into the problem of determining the probability p that an item (card) not *associated with a particular characteristic will be selected (dropped) during the search for that characteristic.* We shall refer to p as the probability that an item (card) will be a *false drop.* The formulas given below for this probability p are based on the assumption that the codes for the characteristics are randomly chosen and that the associations of items with characteristics are also randomly chosen.

Let H = number of positions in superimposed codification
Y = number of units in a characteristic codification
X = number of characteristics associated with an item

Then
$$p = \frac{\left[\binom{H}{Y} - 1\right]^X - \sum_{i=0}^{i=Y-1} E_{Y-i}}{\binom{H}{Y}^X}$$

where

$$E_Y = \binom{Y}{0}\binom{H-Y}{Y}^X$$

$$E_{Y-1} = \binom{Y}{1}\binom{H-Y+1}{Y}^X - \binom{Y}{1}E_Y$$

$$E_{Y-2} = \binom{Y}{2}\binom{H-Y+2}{Y}^X - \binom{Y}{2}E_Y - \binom{Y-1}{1}E_{Y-1}$$

$$E_{Y-3} = \binom{Y}{3}\binom{H-Y+3}{Y}^X - \binom{Y}{3}E_Y - \binom{Y-1}{2}E_{Y-1} - \binom{Y-2}{1}E_{Y-2}$$

$$E_{Y-4} = \binom{Y}{4}\binom{H-Y+4}{Y}^X - \binom{Y}{4}E_Y - \binom{Y-1}{3}E_{Y-1} - \binom{Y-2}{2}E_{Y-2} - \binom{Y-3}{1}E_{Y-3}$$

and so forth.

For our example above,† $H = 20$, $Y = 4$, $X = 3$; hence

$$p = \frac{\left[\binom{20}{4} - 1\right]^3 - (E_1 + E_2 + E_3 + E_4)}{\binom{20}{4}^3}$$

$$= \frac{0.1131692 \times 10^{12} - 0.10890324 \times 10^{12}}{0.11323941 \times 10^{12}} = 0.0377$$

From probability theory we know that, if there are M items, then the expected number of false drops will be Mp. For our example, if $M = 200$, we would expect $200 \times 0.0377 = 7.54$ false-drop cards during the search on a particular characteristic. To find the probability $P(m)$ of obtaining m false drops, we use the formula from probability theory,

$$P(m) = \binom{M}{m} p^m (1-p)^{M-m}$$

In Fig. 7-18 we have graphed some of these values for our example.

So far we have been considering false drops when searching on *one* characteristic only. Now we shall discuss the situation that arises when more than one characteristic is used in the search. Let n be the number of characteristics entering the search. The codifications of these particular characteristics may overlap. Let us suppose that their logical sum has precisely h units (punches). Then

$$p = \frac{\sum_{j=0}^{j=n-1} F_{n-j} - \sum_{i=0}^{i=h-1} E_{h-i}}{\binom{H}{Y}^X}$$

† For numerical calculations some helpful results are $\binom{U}{0} = 1$, $\binom{U}{1} = U$, $\binom{U}{V} = \frac{U(U-1)\cdots(U-V+1)}{1 \times 2 \times \cdots \times V}$, $0! = 1$, $1! = 1$.

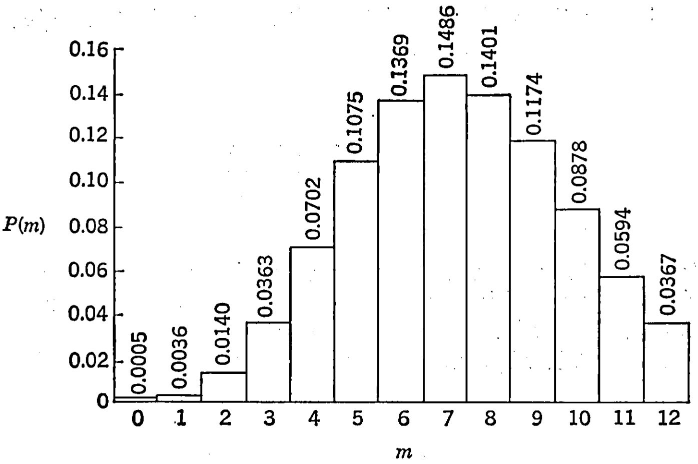


FIG. 7-18. Probability of obtaining exactly m false drops for $H = 20$, $Y = 4$, $X = 3$, and $M = 200$. (Here $h = 4$.)

where

$$F_n = \binom{n}{0}\left[\binom{H}{Y} - n\right]^X$$

$$F_{n-1} = \binom{n}{1}\left[\binom{H}{Y} - n + 1\right]^X - \binom{n}{1}F_n$$

$$F_{n-2} = \binom{n}{2}\left[\binom{H}{Y} - n + 2\right]^X - \binom{n}{2}F_n - \binom{n-1}{1}F_{n-1}$$

$$F_{n-3} = \binom{n}{3}\left[\binom{H}{Y} - n + 3\right]^X - \binom{n}{3}F_n - \binom{n-1}{2}F_{n-1} - \binom{n-2}{1}F_{n-2}$$

and so forth, and

$$E_h = \binom{h}{0}\binom{H-h}{Y}^X$$

$$E_{h-1} = \binom{h}{1}\binom{H-h+1}{Y}^X - \binom{h}{1}E_h$$

$$E_{h-2} = \binom{h}{2}\binom{H-h+2}{Y}^X - \binom{h}{2}E_h - \binom{h-1}{1}E_{h-1}$$

$$E_{h-3} = \binom{h}{3}\binom{H-h+3}{Y}^X - \binom{h}{3}E_h - \binom{h-1}{2}E_{h-1} - \binom{h-2}{1}E_{h-2}$$

and so forth.

Observe that, if $\binom{H}{Y} \gg n$, then $\sum\limits_{j=0}^{j=n-1} F_{n-j} \approx \binom{H}{Y}^X$. This is the case for our example of $H = 20$, $Y = 4$, and $X = 3$. Let us suppose that we are searching on three characteristics with the codifications given above (see Fig. 7-17); then $h = 9$. Thus we have

$$p = \frac{\binom{20}{4}^3 - \sum_{i=0}^{i=8} E_{9-i}}{\binom{20}{4}^3} = \frac{(0.113732 - 0.113716) \times 10^{12}}{0.113732 \times 10^{12}} = 0.00014$$

It is interesting to compare $P(m)$ when searching on a single characteristic with $P(m)$ when searching on three characteristics that overlap to present the nine units of our illustration (see Exercise *j*). Observe that h can range from 4 to 12; some values of p for various values of h are

h	4	6	9	12
p	0.0377	0.0061	0.00014	0.0000003

Note that, as long as $\binom{H}{Y} \gg n$, then the value of h is all-important and p is effectively independent of n. However, if, for example, $H = 6$, $Y = 2$, $X = 3$, then an n of 1, 2, or 3 can no longer be neglected.

EXERCISES

(*a*) How many codifications can be made of no more than 10 positions with 26 symbols? If $K = 26$, what is the most efficient number of positions to use for codifying 100,000 words?

(*b*) Given the information code 10101, find the redundant codification, as in Fig. 7-16.

(*c*) If a redundant codification was received as 0100 1000 0001 0011, what was the information code sent and in what positions did errors occur? Suppose that 0100 1000 0001 0001 was received; in what positions are the errors?

(*d*) Suppose that there were 6 bits in an information code; if the scheme given above were used, how many errors could be detected and how many bits would there be in the redundant codification?

(*e*) For $H = 4$, $Y = 2$, and $X = 2$, find p by means of the above formulas for $n = 1, h = 2$; for $n = 2, h = 2$; for $n = 2, h = 3$; and for $n = 2, h = 4$. Check your result by making an enumeration of the $\binom{H}{Y}^X = \binom{4}{2}^2 = 36$ equally probable conditions.

(*f*) Compute p for $H = 6$, $Y = 2$, $X = 3$ when searching on a single characteristic.

(*g*) Compute p for $H = 6$, $Y = 2$, $X = 3$ when searching on three characteristics (that is, $n = 3$) that overlap so that $h = 5$.

(*h*) Compare the results of Exercises *f* and *g* by graphing $P(m)$ for $M = 50$ in both cases.

(*i*) Find the expected number of false drops for $H = 100$, $Y = 6$, $X = 4$ when $h = 15$.

(*j*) Graph $P(m)$ for a search on three characteristics that overlap to present 9 units (i.e., $h = 9$, whence $p = 0.00014$—see above). Compare the results with Fig. 7-18.

7-8. Additional Topics

a. Equivalence Relation and Linear Ordering. To a mathematician the term *relation* means a rule that can be applied to two objects of a set to determine whether or not they are related in a fashion defined by the rule. A relation is usually symbolized, and if the relation holds for elements a and b of a set, then we write "*a symbol b*," read "a is in relation to b." An *equivalence relation*, symbolized by $\cong$, is a relation defined for a set such that, if a, b, and c are any elements of the set, then (1) $a \cong a$ (an element is in relation to itself, called the *reflexive* property), (2) if $a \cong b$, then $b \cong a$ (if a is in relation to b, then b is in relation to a, called the *symmetric* property), (3) if $a \cong b$ and $b \cong c$, then $a \cong c$ (called the *transitive* property). Can you give some examples of relations that are equivalence relations (i.e., that are reflexive, symmetric, and transitive)? For a particular set and equivalence relation the subset of objects that are all equivalent to each other is called an *equivalence class*. As an example, consider the set of all integers and the relation: Two integers are in relation to each other if their difference is divisible by 3. Show that this relation is an equivalence relation and that there are three equivalence classes: . . . -2, 1, 4, 7, . . . ; . . . -1, 2, 5, 8, . . . ; and . . . 0, 3, 6, 9, ...

The most important property of an equivalence relation is that *each object of a set is included in one and only one of the equivalence classes generated by an equivalence relation.* Why is this property important in considering searching, sorting, ordering, and codifying? (See R. L. Wilder, "Introduction to the Foundations of Mathematics," John Wiley & Sons, Inc., New York, 1952; or G. Birkhoff and S. MacLane, "A Survey of Modern Algebra," The Macmillan Company, New York, 1941.)

Another important relation is called the *linear-ordering* relation, denoted by $\geq$ and defined as follows: (1) for any two objects a and b of the set, either $a \geq b$ or $b \geq a$; (2) if $a \geq b$ and $b \geq c$, then $a \geq c$; (3) if $a \geq b$ and $b \geq a$, then a and b are identical (written $a = b$). A relation is called a *partial ordering* if (1) does not hold. Can you give examples of linear- and partial-ordering relations? In this chapter methods were given for ordering objects according to a linear-ordering relation. What could "ordering objects according to a partial-ordering relation" mean? How could this be accomplished?

b. References on Sorting and Ordering

Bell, D. A.: The Principles of Sorting, *Computer J.*, vol. 1, no. 2, pp. 71, July, 1958.

Davies, D. W.: Sorting of Data on an Electronic Computer, *Proc. IEE,* vol. 103B suppl., p. 87.

Friend, E. H.: Sorting on Electronic Computer Systems, *J. Assoc. Computing Machinery,* vol. 3, p. 134, July, 1956.

Hosken, J. E.: Evaluation of Sorting Methods, *Proc. Eastern Joint Computer Conf.*, 1955, p. 35.

Isaac, E. J., and R. C. Singleton: Sorting by Address Calculation, *J. Assoc. Computing Machinery,* vol. 3, p. 169, July, 1956.

Shannon, C. E.: A Mathematical Theory of Communication, *Bell System Tech. J.*, vol. 27, pp. 379–423, 623–656, 1948.

c. References on Information Retrieval

Bracken, R. H., and H. A. Tillitt: Information Searching with a 701 Calculator, *J· Assoc. Computing Machinery,* vol. 4, p. 131, 1957.

Dennis, Bernard K.: Rapid Retrieval of Information, *Computers and Automation,* October, 1958, p. 8.

Digital Computer Newsletter, vol. 10, no. 4, p. 4, October, 1958.

Fairthorne, R. A.: Automatic Retrieval of Recorded Information, *Computer J.*, vol. 1, no. 1, p. 36, April, 1958.

Ledley, R. S.: TABLEDEX, A New Coordinate Indexing Method for Bound Book Form Bibliographies, *Proc. Intern. Conf. Sci. Inform.*, Area V, 1958, Washington, D.C.

Luhn, H. P.: A Statistical Approach to Mechanized Encoding and Searching of Literary Information, *IBM J. Research Develop.*, vol. 1, p. 309, 1957.

Mooers, C.: Putting Probability to Work in Coding Punched Cards, Zatocoding, *Zator Tech. Bull.* 3, 1947, Zator Company, Boston, Mass.; also Zatocoding and Development in Information Retrieval, *Aslib. Proc.*, vol. 8, no. 1, pp. 3–22, February, 1956.

National Science Foundation Booklet, "Non-conventional Technical Information Systems in Current Use."

National Science Foundation Booklet, "Current Research and Development in Information Retrieval."

Office of Research and Development, U.S. Patent Office: "Patent Office Research and Development Reports," U.S. Department of Commerce.

Proc. Intern. Conf. Sci. Inform., Area V, 1958, p. 317.

Ray, L. C., and R. A. Kirsch: Finding Chemical Records by Digital Computers, *Science*, vol. 126, p. 814, 1957.

Shera, J. H., A. Hert, and J. W. Perry: "Information Systems in Documentation," Interscience Publishers, Inc., New York.

Waldo, W. H., and M. DeBacker: Printing Chemical Structures Electronically; Encoded Compounds Searched Generally with IBM 702, *Proc. Intern. Conf. Sci. Inform.*, Area IV, 1958.

Wildhack, W. A., and J. Stern: "The Peek-a-boo System in the Field of Instrumentation," p. 209, Interscience Publishers, Inc., New York.

d. References on Error-detecting and -correcting Codes. The method given in this chapter for constructing error-correcting codes is essentially that of Irving S. Reed (see A Class of Multiple-error-correcting Codes and the Decoding Scheme, *IRE Trans. on Inform. Theory*, vol. IT-4, pp. 34–49, September, 1954). The initial theory was first stated in a well-known paper of R. W. Hamming (see Error Detecting and Error Correcting Codes, *Bell System Tech. J.*, vol. 29, pp. 147–160, April, 1950). Many other papers have recently appeared in this field, and we list some of these below. It would be well to consider these papers after Part 3 of the text has been mastered.

Brown, A. B., and S. T. Myers: Evaluation of Some Error Correction Methods Applicable to Digital Data Transmission, *IRE Conv. Record*, vol. 6, pt. 4, pp. 37–55, 1958.

Elias, P.: Error Free Coding, *IRE Trans. on Inform. Theory*, vol. IT-4, pp. 29–38, September, 1954.

Golay, M.: Binary Coding, *IRE Trans. on Inform. Theory*, vol. IT-4, pp. 23–28, September, 1954.

Huffman, D.: A Linear Circuit Viewpoint on Error Correcting Codes, *IRE Trans. on Inform. Theory*, vol. IT-2, pp. 20–28, September, 1956.

Peterson, W.: An Experimental Study of a Binary Code, *Communs. and Electronics*, July, 1958, pp. 388–392.

Sacks, G.: Multiple Error Correction by Means of Parity Checks, *IRE Trans. on Inform. Theory*, vol. IT-4, pp. 145–147, December, 1958.

Shannon, C. E.: General Treatment of the Problem of Coding, *IRE Trans. on Inform. Theory*, vol. IT-1, pp. 102–104, February, 1953.

Shapiro, H. S., and D. L. Slotnik: On the Mathematical Theory of Error-correcting Codes, *IBM J. Research*, January, 1958, pp. 25–34.

Slepian, D.: A Class of Binary Signaling Alphabets, *Bell System Tech. J.*, January, 1956, pp. 203–234.

Ulrich, W.: Non-binary Error Correction Codes, *Bell System Tech. J.*, November, 1957, pp. 1341–1388.

e. Irredundant Coding. To our knowledge Mooers [see the references in (*c*)] first discussed irredundant coding. Formulas that differ from those given in this chapter are found in C. S. Wise, Mathematical Analysis of Coding Systems, in R. S. Carey and J. W. Perry (eds.), "Punched Cards," chap. 20, Reinhold Publishing Corporation, New York, 1951. The formulas given by Wise are not precise. (Why?) Prove the formulas given in this chapter of our text by means of an exact analysis of the problem.

CHAPTER 8

SPECIAL-PURPOSE DIGITAL-COMPUTER SYSTEMS DESIGN

8-1. Introduction

Design for a Purpose. In previous chapters we have seen several examples of digital-computer systems design. Chapters 2 to 4 were primarily concerned with the systems design of a general-purpose digital computer; here some of the fundamental ideas of computer engineering were introduced. The general-purpose computer might be classified basically as a "scientific" computer well adapted to the processes of numerical analysis considered in Chap. 6. It appeared from Chap. 7, however, that the processes of searching, sorting, ordering, etc., made extensive use of the data-handling components of a computer and that perhaps special features could be designed into the computer that might considerably enhance its effectiveness. Similarly other types of computational problems that occur frequently may indicate still further special computational facilities. Some of these were mentioned in the illustrations given in Chap. 1.

In the present chapter we shall consider some of these special-purpose systems designs. Our intention is to free the student from the rigors of the general-purpose-computer systems design, which heretofore has held our attention. Our method is to use specially selected examples of systems designs ranging from those which differ radically from the general-purpose computer to those built upon the extended general-purpose concept. It is our hope that such examples of computers—having no instruction systems at all, having temporary specialized memories, no arithmetic functions, great deviations from the conventional instruction format, and multiple code-handling capabilities—will present a broad view of the infinite possibilities that exist in digital-computer systems design.

This does not imply that there are no guiding signs to computer systems design. Quite the contrary, the unique features in a computer design should be determined solely by the intended purpose of the computer. Thus, in analyzing or describing a variety of different systems designs, we shall pay close attention to the purpose which indicated each such design.

Outline. We shall consider first the digital differential analyzer. This computer has no instruction system but rather consists of a collection of computer units that are wired together as required by each particular problem. Next we shall consider a logical-control computer, using the control of a chemical factory as an example. Here a special "external-control" instruction is introduced to facilitate the logical computations. A special design for a searching computer will then be considered. This special-purpose computer can be incorporated in a general-purpose computer as well. Business and logistics problems illustrate the need for special features enabling the handling of large arrays of numbers. Here a radically different instruction format is illustrated. Finally we consider the multiple-general-purpose computer, which is essentially an interlocked collection of computers, used in such a manner as will take greatest advantage of the special-purpose components of which it is comprised. In our illustration this multiple-purpose computer will be comprised of the computer components discussed in the previous sections.

8-2. The Digital Differential Analyzer

Purpose. Frequently it is necessary to compute the value of some function point by point. For example, in the automatic-milling-machine application described in Chap. 1, the computer attached to the milling machine must guide the tool from point to point along a predetermined path that was given only grossly at the input, e.g., a linear interpolation between two points or a circle of a certain radius. Similarly in many other control operations rapid calculations of sin θ and cos θ may be necessary, as in accurately sweeping an area with an antenna or in computing some functions. Not infrequently the familiar analog techniques cannot produce a sufficient number of significant figures for the accuracy required. In such cases one can use a digital differential analyzer as a small, relatively simple and inexpensive special-purpose digital computer.

The Concept of the Digital Differential Analyzer. The digital differential analyzer computes successive values of a function by means of successive *differential additions.* Consider the problem of computing a table of values for a function $y = f(x)$. If we start with a given $y(x_0)$, then $y(x_1) = y(x_0) + (y(x_1) - y(x_0)) = y(x_0) + (\Delta y)_{x_0}$. Similarly we write $y(x_2) = y(x_1) + (\Delta y)_{x_1}$, and so forth. In other words,

$$y(x_{i+1}) = y(x_i) + (\Delta y)_{x_i}$$

Now suppose that $y = e^x$. Then $dy = e^x\,dx = y\,dx$; or approximately

$$(\Delta y)_{x_i} = y(x_i)(x_{i+1} - x_i) = y(x_i)\,\Delta x$$

Thus we can compute successive approximate values of e^x by

$$y(x_{i+1}) = y(x_i) + y(x_i)\,\Delta x$$

The smaller we make Δx, the more accurate our results will be. However, there is an obstacle here. Remember that multiplication produces a *double-length result,* so that, if Δx is small enough, the major part of $y(x_i)\,\Delta x$ will be zero. Thus errors will result when the minor product, which contains the significant figures of $y(x_i)\,\Delta x$, is dropped. It would therefore be better to work with the double-length extension of $y(x_i)$. However, this is not particularly desirable for what is supposed to be a small special-purpose computer.

The use of a double-length accumulator can be avoided; to find out how, let us examine our procedure more carefully. As we have noted, each time we form $y(x_i)\ \Delta x$ the major product is zero; but certainly the major part of $y(x_i)$ must change eventually. This must occur during the addition $y(x_i) + y(x_i)\ \Delta x$ and will be the result of a *carry* from the minor part into the major part. In other words, as we accumulate in the minor part, we eventually propagate a carry, or overflow, from the minor part into the major part of $y(x_i)$. If we are working in *binary*, the carry can only be a 1 and the major part of $y(x_i)$ can change at most by 1 during any iteration. Hence we do not need a double-length accumulator at all—we need a *single-length accumulator* that simply accumulates successive minor parts of $y(x_i)\ \Delta x_i$ and a *counter* that holds the major part of $y(x_i)$ and *adds* 1 to the major part of $y(x_i)$ each time there is an *overflow* from the minor accumulator. We call the accumulated minor parts the *residual.*

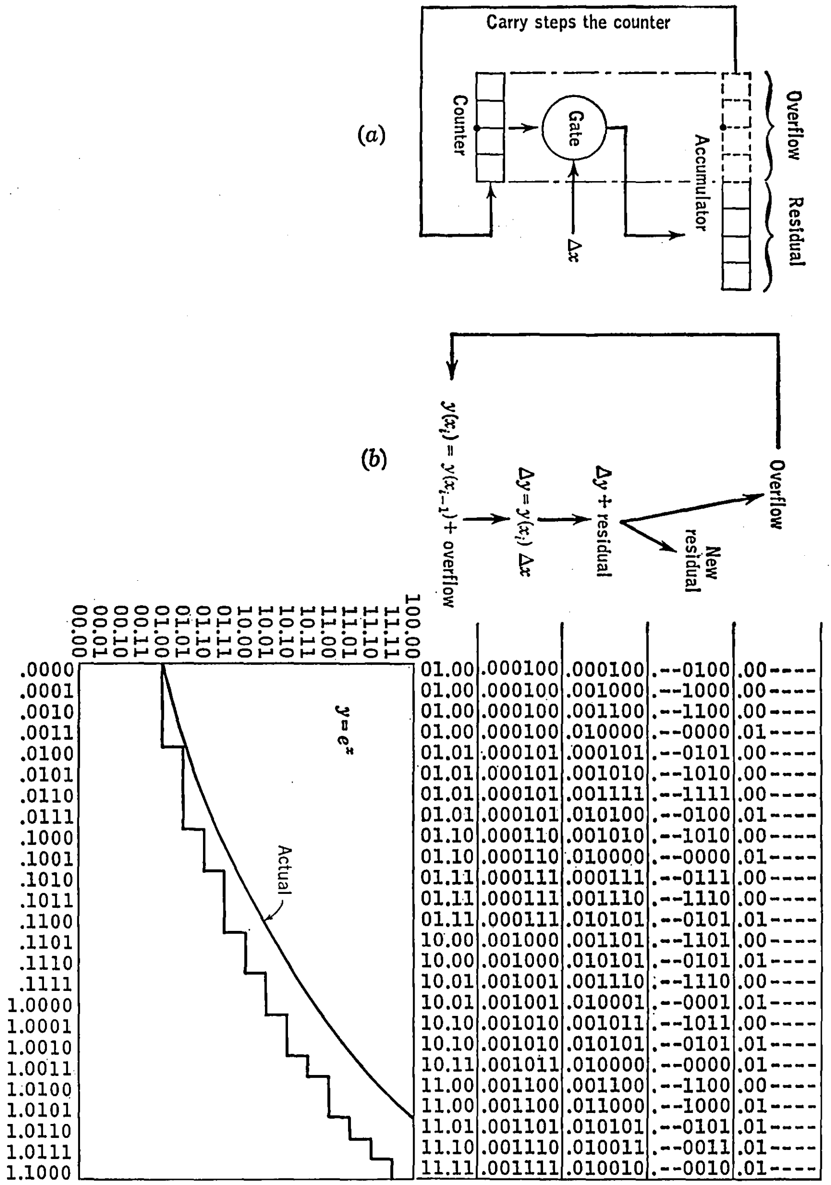
True multiplication is not essential in finding $y(x_i)\ \Delta x$, for we can always take $\Delta x = 1/2^q$, and we need only shift $y(x_i)$ by q positions to the right to form $y(x_i)\ \Delta x$. But since we do not have a major part to our accumulator, we can always choose Δx so that the shifting of $y(x_i)$ need not have to occur *actually* (although of course it occurs *virtually*).

To illustrate these points, consider a 4-bit binary word and let $\Delta x = .0001$ (see Fig. 8-1). In Fig. 8-1*a* we have illustrated the accumulator that is to hold the residual and the counter that is to hold the major part of the function value y. The circle represents the component wherein y is *virtually* but *not actually* shifted; this is simply a gate that at the proper time passes y to be accumulated with the previous residual to form the new residual. The dash-dot lines represent the "true" juxtaposition of the double-length value of y. For our illustration, start with $x_0 = 0$, and $y_0 = e^0 = 1$ preloaded into the counter. Then $y(x_0) = 01.00$ and $y(x_0)\ \Delta x = 00.00\ 0100$, where 0100 is the residual. Since

$$y(x_1) = y(x_0) + y(x_0)\ \Delta x = 01.00\ 0100$$

the 01.00 remains unchanged in the counter and 0100 is put into the accumulator (see Fig. 8-1*b*). With $y(x_1)\ \Delta x = 00.00\ 0100$ again, we have $0100 + 0100 = 1000$ as the residual, and 01.00 remaining still unchanged in the counter as $y(x_2)$. Then $y(x_3)\ \Delta x = 00.00\ 0100$, whence $1000 + 0100 = 1100$ is the new residual and 01.00 remains unchanged as $y(x_4)$ in the counter. Next $y(x_4)\ \Delta x = 00.00\ 0100$, whence $1100 + 0100 =$ *carry 1* $+ 0000$; now the counter is increased by 1, putting 01.01 in the counter as $y(x_5)$ and leaving 0000 as the new residual. We continue with $y(x_5)\ \Delta x = 00.00\ 0101$, and so forth. The result of each step and the graph of the function so calculated are shown in Fig. 8-1*b*.

Computing Units. The combination of accumulator, gate, and counter is called a computing unit of a digital differential analyzer. Many units



$y(x_i) = y(x_{i-1}) +$ overflow	$\Delta y = y(x_i)\,\Delta x$	$\Delta y +$ residual	New residual	Overflow
01.00	.000100	.000100	.--0100	.00----
01.00	.000100	.001000	.--1000	.00----
01.00	.000100	.001100	.--1100	.00----
01.00	.000100	.010000	.--0000	.01----
01.01	.000101	.000101	.--0101	.00----
01.01	.000101	.001010	.--1010	.00----
01.01	.000101	.001111	.--1111	.00----
01.01	.000101	.010100	.--0100	.01----
01.10	.000110	.001010	.--1010	.00----
01.10	.000110	.010000	.--0000	.01----
01.11	.000111	.000111	.--0111	.00----
01.11	.000111	.001110	.--1110	.00----
01.11	.000111	.010101	.--0101	.01----
10.00	.001000	.001101	.--1101	.00----
10.00	.001000	.010101	.--0101	.01----
10.01	.001001	.001110	.--1110	.00----
10.01	.001001	.010001	.--0001	.01----
10.10	.001010	.001011	.--1011	.00----
10.10	.001010	.010101	.--0101	.01----
10.11	.001011	.010000	.--0000	.01----
11.00	.001100	.001100	.--1100	.00----
11.00	.001100	.011000	.--1000	.01----
11.01	.001101	.010101	.--0101	.01----
11.10	.001110	.010011	.--0011	.01----
11.11	.001111	.010010	.--0010	.01----

FIG. 8-1. Concept of the digital differential analyzer.

can be connected together to compute various functions. The contents of the accumulator, the residual, is denoted by R; the contents of the counter, the functional value, is denoted by y; the overflow, or carry, from the accumulator that steps the counter, i.e., the differential value that is to be added to some functional value, is denoted by Δy (see Fig. 8-2). The gate may cause either addition or subtraction of the contents of the counter (virtually but not actually shifted) to the accumulator.

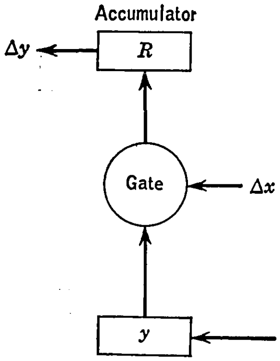


FIG. 8-2. Computing unit of digital differential analyzer.

As an example of how units can be combined, consider now the problem of computing sin x and cos x. Let

$$y_1 = \cos x \qquad \text{and} \qquad y_2 = \sin x$$

Then the differential equations are

$$dy_1 = -\sin x\,dx = -y_2\,dx$$

and $$dy_2 = \cos x\,dx = y_1\,dx$$

Thus we have

$$y_1(x_{i+1}) = y_1(x_i) + \Delta y_1 \qquad \Delta y_1 = -y_2\,\Delta x$$

and

$$y_2(x_{i+1}) = y_2(x_i) + \Delta y_2 \qquad \Delta y_2 = y_1\,\Delta x$$

Hence the arrangement shown in Fig. 8-3 will compute y_1 and y_2.

Programming. There are three steps to programming a problem: First, the proper differential equations must be worked out so that Δy_j appears in the form $y_i\,\Delta x$. Second, the connections between units must be diagramed. And finally the scaling must be worked out. Consider this latter step first. Note that, *since the counter can be increased only by 1 at a time*, then during any iteration y can increase only by 1. In other words, the graph of the function *cannot be steeper* than 45°. Thus in our example of Fig. 8-1 we could not compute e^x for any greater values of x. However, the slope of any function can be adjusted by scaling so that it does not exceed 45° within the range of computation.

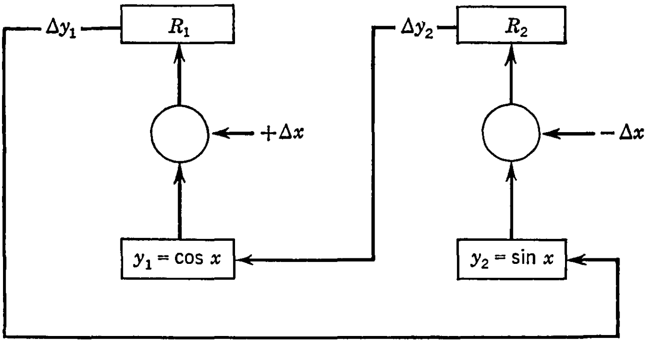


FIG. 8-3. Combinations of computing units of digital differential analyzer.

Consider as another example the problem of computing the product $y_3 = y_1y_2$. Since $dy_3 = y_1\,dy_2 + y_2\,dy_1$, we have

$$y_3(x_{i+1}) = y_3(x_i) + \Delta y_3 \qquad \Delta y_3 = y_1\,\Delta y_2 + y_2\,\Delta y_1$$
$$y_2(x_{i+1}) = y_2(x_i) + \Delta y_2$$

and $$y_1(x_{i+1}) = y_1(x_i) + \Delta y_1$$

Hence the configuration of Fig. 8-4 will compute $y_3 = y_1y_2$. Note that the counter for y_3 has two inputs and that these must be electronically arranged so that they do not step the counter at precisely the same time.

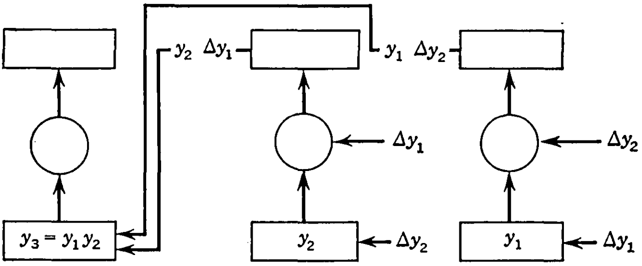


FIG. 8-4. Computing $y_3 = y_1y_2$.

Note also that not all of the computing unit for y_3 is used. Another feature of this arrangement is that the inputs to the gates are not a single Δx, but rather Δy_1 and Δy_2 themselves.

Next consider the computation of $y = x^2$. Here $dy = 2x\,dx$ and $y(x_{i+1}) = y(x_i) + \Delta y$, $\Delta y = 2x_i\,\Delta x$; Fig. 8-5 shows the arrangement. Here

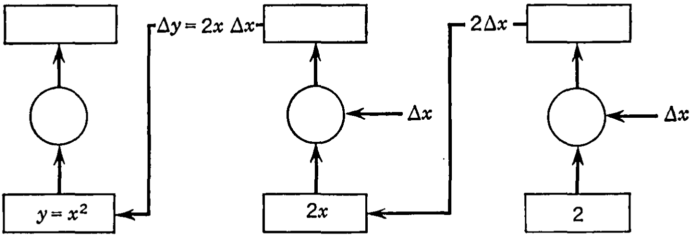


FIG. 8-5. Computing $y = x^2$.

note that one of the counters is loaded with the constant 2 and never changes.

Finally consider the computation of $y = 1/x$, that is, division. Here $dy = (-1/x^2)\,dx$; letting $w = -1/x^2$, we find $dy = w\,dx$ and

$$dw = \frac{2}{x^3}\,dx = -\frac{2}{x}\left(-\frac{1}{x^2}\,dx\right) = -2y\,dy$$

Thus $$y(x_{i+1}) = y(x_i) + \Delta y \qquad \Delta y = w\,\Delta x$$

and $$w(x_{i+1}) = w(x_i) + \Delta w \qquad \Delta w = -2y\,\Delta y$$

The arrangement of units for this computation is left as an exercise

Advantages. The digital differential analyzer does not require an instruction system or a stored program, for the "programming" is accomplished simply by appropriately wiring units together. Thus no control unit is necessary, and the computer is very inexpensive. Of course it is correspondingly quite limited in its capabilities.

The accumulator and the counter require storage, and for this a drum is frequently used. For example, each band around the drum might be divided into two words of storage, one for the accumulator and the other for the counter, of a single computing unit. The inputs and outputs from the heads of each band would be brought to a plugboard console. Here the gate and counter inputs and the accumulator (overflow) outputs for each of the units could be appropriately connected by "jumpers."

EXERCISES

(*a*) Compute sin x and cos x, and graph the results as they would be computed by a digital differential analyzer (see, for example, Fig. 8-1). If the word length were 4 bits, how would the problem be scaled?

(*b*) Draw the arrangement of units to compute $y = 1/x$.

(*c*) Draw the arrangement of units to compute $y = \ln x$. [HINT: $dy = (1/x)\,dx$. Let $w = 1/x$; then $dy = w\,dx$, and $dw = (-1/x^2)\,dx = -w\,dy$.]

(*d*) Draw the arrangement of units to compute $y = x^n$. {HINT: $dy = nx^{n-1}\,dx = x^n[(n/x)\,dx] = x^n\,d(n \ln x) = y\,d(n \ln x)$.}

(*e*) Draw the arrangement of units to solve the following simultaneous differential equations:

$$\frac{dy_1}{dx} = \lambda_{12}y_2 + \lambda_{13}y_3 \qquad \frac{dy_2}{dx} = \lambda_{21}y_1 + \lambda_{23}y_3 \qquad \frac{dy_3}{dx} = \lambda_{31}y_1 + \lambda_{32}y_2$$

(*f*) How can a digital differential analyzer be used in conjunction with machine-tool control (see Chap. 1)?

8-3. Real-time Logical Systems Control: A Real-time Control Computer

An Example. Very often the primary function of an on-line digital control system (i.e., a system connected to a dynamic process) is to make sequences of logical decisions that depend on the state of certain variables associated with the world external to the computer. Here we shall consider the special-purpose systems design of a real-time logical-control computer and show how it might be applied in a specific example of controlling a chemical factory. By discussing first the specific application, the reasons for the various aspects of the control computer will be better appreciated. Although the example is concerned with the control of a chemical factory performing a specific chemical process, the situation is conceptually similar in almost all such real-time control problems as may occur in other types of automatic factory control, in tactical gunnery control, or in distributing tracking data to command centers according to predetermined strategies. The example also contains a feedback loop by which the results of the input digital disposition influence the outside world, which in turn again influences the input to the computer.

Figure 1-10 on page 12 shows the flow chart of a chemical factory manufacturing Al_2O_3. Briefly, clay is mixed with water and a detergent to make a wash containing the raw material. A certain amount of this wash is tapped off into an evaporation tank, where it is concentrated by evaporation. Then in another tank HCl gas is mixed with the wash until it is saturated. The wash is centrifuged and a precipitate of $AlCl_3 \cdot 6H_2O$ is obtained. The precipitate is decomposed, by heating, into gaseous HCl, which is reused, and into solid Al_2O_3, the desired end product. It is of course understood that, even though this problem is based on an actual process, it has here been grossly oversimplified and actually stylized. It is not meant to be examined from a chemical-engineering point of view, since many details of both the necessary conditions and desired results are but casually indicated.

TABLE 8-1. SIGNALS SHOWING THE STATE OF THE CHEMICAL FACTORY

Signals generated by factory	*Symbolism*
Level of $L(1)$ reached	L(1)
Concentration test good	C
Time for concentration test	T(C)
Valve 4 open	V(4)
Level of $L(2)$ reached	L(2)
Tank 2 empty	E(2)
Tank 3 full	F(3)
Temperature $T(1)$	T(1)
Temperature $T(2)$	T(2)
Saturation test positive	S
Tank 3 empty	E(3)
Valve 7 open	V(7)
Centrifuging finished	C(F)
Centrifuge empty	E(C)
Pressure of HCl too high	P
Enough volume	W

The state of the chemical factory is given by 16 signals, listed in Table 8-1. The computer periodically senses these signals, generating from them 12 control signals that direct the future state of the factory. The future state of the factory results in changes of the input signals, whence new control signals are generated by the computer, and so forth. Table 8-2 gives the relation between the sensed-signal conditions and the necessary control signals to be generated. A symbolic notational form for each statement is also given, where · means *and* and ¯ means *not* (i.e., no signal).

Programming Logical Control. The computer will have one subroutine for each set of the control signals to be generated. Hence a jump table is required to initiate the proper subroutine for a given state of the sensed signals. However, we need an input to the jump table that tells which

TABLE 8-2. CONTROL SIGNALS

Sensed-signal conditions	Symbolic representation	Corresponding signals to be generated
1. Level is at $L(1)$, *and* concentration test is bad, *and* time for testing is due	$L(1) \cdot \bar{C} \cdot T(C)$	Throw away solution by opening valve 3, and add clay, detergent, and water in tank 1 by opening hatch 1, valves 1 and 2
2. Level is at $L(1)$, *and* concentration test is good, *and* time for testing is due, *and* tank 2 is empty	$L(1) \cdot C \cdot T(C) \cdot E(2)$	Tap off certain amount of fluid into tank 2 by means of valve 4
3. Level $L(1)$ has not been reached, *and* valve 4 is not open	$\bar{L}(1) \cdot \bar{V}(4)$	Add detergent, water to tank 1 by opening valves 1 and 2
4. Valve 4 is not open, *and* level is $L(2)$, *and* tank 3 is empty	$\bar{V}(4) \cdot L(2) \cdot E(3)$	Transfer fluid to tank 3 by opening valve 5
5. Temperature is $T(1)$, *and* tank 3 is full, and valve 7 is off	$T(1) \cdot F(3) \cdot \bar{V}(7)$	Cut rate of HCl flow to slow by closing valve 6
6. Temperature is $T(2)$, *and* tank 3 is full, *and* valve 7 is off	$T(2) \cdot F(3) \cdot \bar{V}(7)$	Increase rate of HCl flow to fast by opening valve 6
7. Saturation test is good, *and* valve 7 is closed, *and* centrifuge is empty	$S \cdot \bar{V}(7) \cdot E(C)$	Close flow of HCl by means of valve 6, transfer fluid to centrifuge by valve 7, and start centrifuge
8. Centrifuge finished.......	$C(F)$	Put precipitate in furnace by opening hatch 2, throw away wash by opening valve 8
9. Pressure of HCl high.....	P	Lower heat of furnace
10. Weight is enough........	W	Take out resultant Al_2O_3 by opening hatch 3

jump instruction to use, and we shall first describe how this input may be accomplished.

At any instant the bits in the signal-sensing word (Fig. 1-10) tell which signals are on and which are off. For instance, the word might be as in Table 8-3; in the table under each bit of the word is shown the state of the corresponding signal. We can similarly use a word for each of the significant sensed-signal conditions of Table 8-2, so that the conditions can

TABLE 8-3. EXAMPLE OF SIGNAL-SENSING WORD

1	0	1	0	0	0	1	1	0	0	0	0	1	0	1	0
$L(1) \cdot$	$\bar{C} \cdot$	$(TC) \cdot$	$\bar{V}(4) \cdot$	$\bar{L}(2) \cdot$	$\bar{E}(2) \cdot$	$F(3) \cdot$	$T(1) \cdot$	$\bar{T}(2) \cdot$	$\bar{S} \cdot$	$\bar{E}(3) \cdot$	$\bar{V}(7) \cdot$	$C(F) \cdot$	$\bar{E}(C) \cdot$	$P \cdot$	$\bar{W}$

be matched with the state of the sensed signals at any time. For example, the first condition, L(1)·$\bar{\text{C}}$·T(C) · · · , would be represented by 101 · · · . But what about the rest of the bits? Condition L(1)·$\bar{\text{C}}$·T(C) does not mention the rest of the signals because for that condition they make no difference. They are to be ignored. Thus we need another

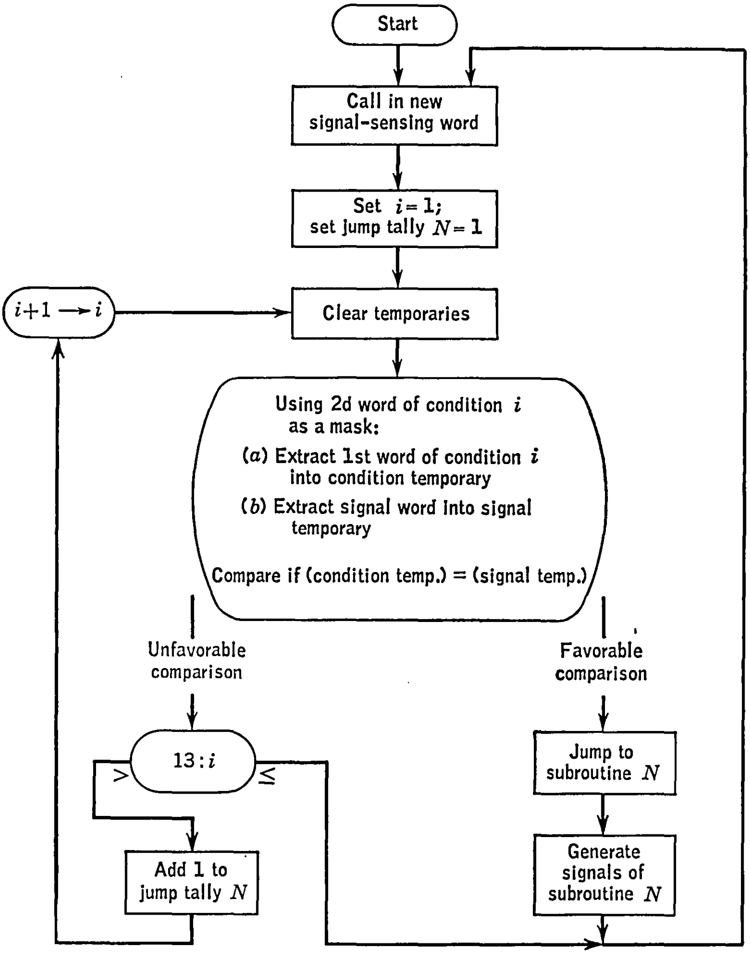

Start
Call in new signal-sensing word
Set $i = 1$; set jump tally $N = 1$
$i+1 \rightarrow i$
Clear temporaries
Using 2d word of condition i as a mask:
(a) Extract 1st word of condition i into condition temporary
(b) Extract signal word into signal temporary
Compare if (condition temp.) = (signal temp.)
Unfavorable comparison
Favorable comparison
$13 : i$
>
≤
Add 1 to jump tally N
Jump to subroutine N
Generate signals of subroutine N

FIG. 8-6. Flow chart of real-time operating program.

word that tells what positions are to be considered and what positions are to be ignored. This second word will have units corresponding to bits that are to be considered and zeros corresponding to bits to be ignored. For example, L(1)·$\bar{\text{C}}$·T(C) is denoted by the following two words:

1st word:	101x	$xxxx$	$xxxx$	$xxxx$
2d word:	1110	0000	0000	0000

where the x means that it makes no difference whether the bit is zero or unit. Similarly the fourth condition, $\bar{V}(4) \cdot L(2) \cdot E(3)$, is denoted by

1st word:	$xxx0$	$1xxx$	$xx1x$	$xxxx$
2d word:	0001	1000	0010	0000

There would then be two words representing each of the 10 conditions of Table 8-2.

Figure 8-6 is a flow chart of a real-time operating program that will compare each of the conditions with the sensed-signal word until a favorable comparison is obtained; then, depending on which condition compared favorably, a jump will be made to the proper signal-generating subroutine. After completing the subroutine, a new sensed-signal word will be called into the high-speed memory, and the process will start over. If no condition compared favorably, the program would call in a new sensed-signal word and repeat the comparisons, and so forth.

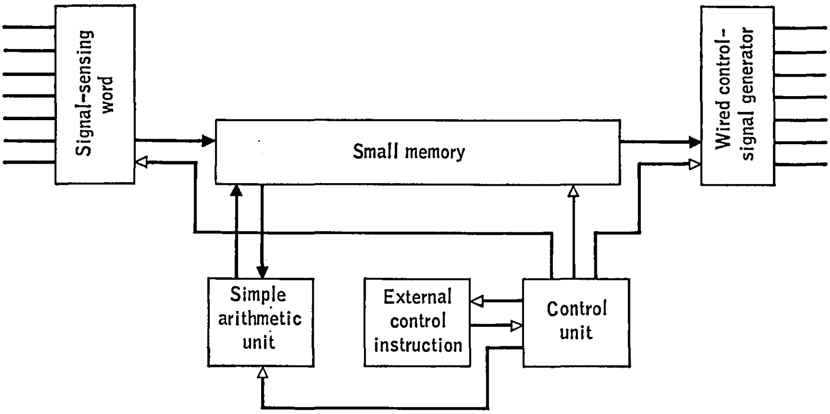


FIG. 8-7. Real-time logical-control computer.

Systems Design of a Special-purpose Logical-control Computer. When hundreds of signals must be sensed, as would occur in a more realistic case, it may be more efficient to have the two extractions and comparison made in one step by a special computer instruction, which we shall call an *external-control instruction.* Such an instruction would have three inputs: the sensed-signal word and the first and second condition words. If the comparison is unfavorable, the instruction would automatically increase a relative counter. This instruction would repeat itself as the next instruction. The addresses of the two condition words are taken relative to this counter so that they are automatically modified appropriately for the next condition when the external-control instruction repeats itself. If the comparison is favorable, the next instruction is taken from the address of the external-control instruction plus the counter reading; i.e., the external-control instruction would be followed by the jump table. The method by which the comparisons are repeated is left as an exercise (see Exercise *a*).

For a permanent installation the subroutines generating the control signals can be wired in. Hence the real-time logical-control computer need contain little more than the external-control instruction, an instruction to clear the relative counter, a *comparison* instruction, a *jump* instruction, and an *add* instruction (see Fig. 8-7).

Since the program will not often be changed, it can be inserted into the computer by means of the control panel; therefore no input-output equipment is necessary except the signal-sensing word and the wired control-signal generator. The memory of such a special-purpose computer need be only a little larger than necessary to accommodate the condition words. The speed of the memory need only be fast compared with the feedback reaction time of the system it is controlling. For the chemical factory here described this need not be particularly fast; for a tracking system it must be very high-speed.

EXERCISES

(*a*) How would the external-control instruction break the loop when *no* comparison is successful? (HINT: A dummy comparison would automatically compare favorably. How would this dummy comparison be arranged, i.e., what would be the first and second words of the dummy condition?)

(*b*) Suppose that the input word to a real-time logical-control computer has the following inputs (see Fig. 8-8): tank 1 full, **L(1)**; tank 2 full, **L(2)**; valve 1 open, **V(1)**; valve 2 open, **V(2)**; valve 3 open, **V(3)**. For the output or control-signal word let **O(1)** mean open valve 1; **O(2)**, open valve 2; **O(3)**, open valve (3); **C(1)**, close valve 1; **C(2)**, close valve 2; and **C(3)**, close valve 3. Define an appropriate instruction, including an external-control instruction, for the logical-control computer as discussed above, and code the process of filling the two tanks, one at a time.

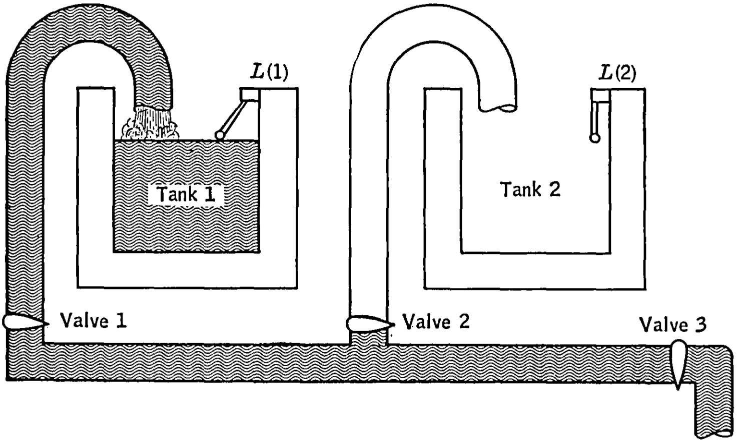


FIG. 8-8. Simple system controlled by real-time logical-control computer.

(*c*) Code the problem used in (*b*) without the use of an external-control instruction.

8-4. Systems Design for Special-purpose Information-retrieval Computer

An Example. The occasion may frequently arise where a special-purpose searching, sorting, and ordering computer becomes feasible. As a specific example of the use of such a computer, which will indicate the motivation for its systems design, consider a cost-accounting system in a factory. Each workman in the factory is given a job; he procures supplies and parts; he spends time using the supplies and assembling the parts to produce some final object; then he may be given another job (which may be a repetition of the first), etc. The executives and accountants of the factory would like to know how many objects of each type have been completed, how many supplies and how many parts were used during a particular period of time, how many hours a particular workman spent on a particular job, how many man-hours altogether were spent in making a certain number of these objects, and so forth. The description given here of a cost-accounting system is necessarily oversimplified, but the basic principles and concepts are the same.

Suppose that each procurement of supplies and parts, each object completed, the time consumed in each job, etc., were recorded by part number, workman's name, job number, date, and so forth, at the time the action was taken or the job completed; suppose that all this information for 1 week could be compiled on a single magnetic tape. The problem then reduces to searching, sorting, and ordering the items on the tape to answer the above questions. In general a computer that can perform these operations need be capable only of comparing, counting, and accumulating. Whether or not such a special-purpose computer is feasible depends on its cost and on the time it takes to process the necessary number of data.

A Small Retrieval Computer. Let us suppose that any item can be recorded in 50 characters or less (a character being 6 bits). Then the computer will first read a set of 50 characters into the temporary storage, a nonending loop of tape (see Fig. 8-9). Selected characters will be appropriately compared with characters from the previously loaded argument storage, a second loop of tape. Depending on the outcome of this comparison, the computer may then add into a partial sum in the accumulator some number represented by certain of the 50 characters, or may increase the count of the counter, and so forth. Actually several sets of 50 bits may be recorded in the temporary storage unit. When writing back onto the magnetic tape, the computer can transpose sets if it is ordering the information, or can insert new information, and so forth. In summary, successive items from the magnetic tape are stored temporarily in the computer; here information is extracted from them, or they are otherwise processed; then they are replaced on the magnetic tape, perhaps in some transposed order. The processing is done in conjunction with information preloaded on the argument storage unit.

The number of instructions to be written into a program will be limited to the number of instructions that can be executed from the time

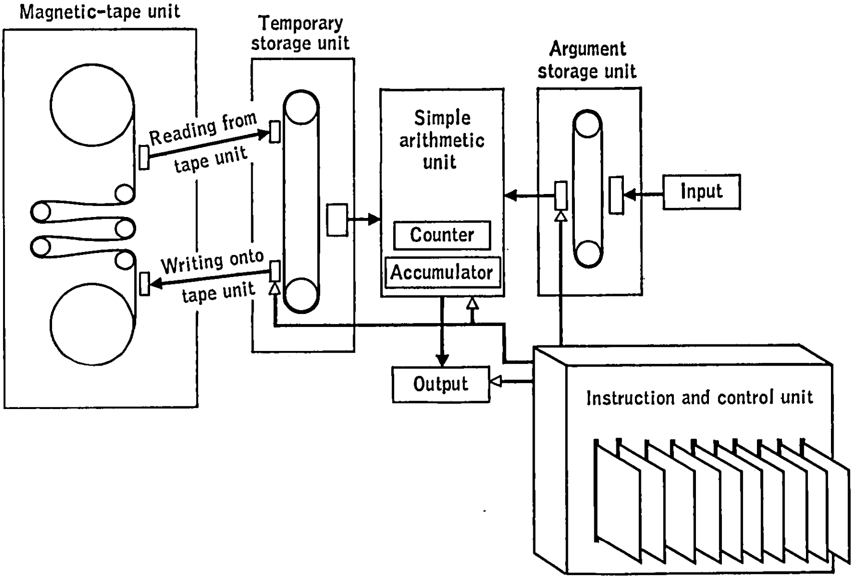


FIG. 8-9. Small retrieval computer.

an item is read from the tape until it is put back again. The instruction system can be very simple. Suppose that the instructions were punched on cards and the cards for one program put into slots in the control unit, where the holes on the cards can be sensed. The arguments on the argument-storage-unit tape loop, which are fixed during any program, can also be addressed. The instruction format could be as shown in Fig. 8-10. Each instruction is concerned with selected characters, as recorded on the cards, of the current item in the temporary storage unit. An instruction may involve two of the arguments of the argument storage unit if, for example, a search for characteristics less than α but greater than β is desired. A two-way exit, γ or δ depending on the outcome of the comparisons, is allowed for flexibility in coding. Consider as an illustration the computation of the number of hours in a week that workman 1224 required to make part 2746 and the number of parts he completed in that week. Suppose that 40 was the code for an item recording a finished job. The instructions would compare the properly selected characters of each item with 1224, and 2746, and 40. When an "equal" comparison was made, the computer would add 1 to the counter and add

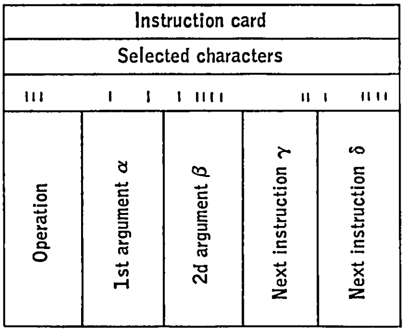


FIG. 8-10. Instruction format.

the number of hours recorded on this item to the accumulator. If an equal comparison was not made, the computer would just replace the item on the tape. By this means the total number of man-hours worked in making a particular object during any specified time period can be determined in one pass through the tape. Similarly the number of man-hours that were worked by a certain person, or in a certain shop, etc., or the number of a certain part that were consumed, or the amount of a certain material that was used, in any shop or by any person, in making a particular object, during a specified time, can be determined in a single pass, and so forth. Suppose that the magnetic tape runs at a rate of 100,000 bits/sec, i.e., about 17,000 characters/sec (6 bits = 1 character), or 350 items/sec (50 characters = 1 item), or 21,000 items/min. Suppose also that there are 500 workmen in the factory, each generating an average of 5 items/day, or a total of 2,500 items/day, or 50,000 items/month (20 working days = 1 month). One pass through the tape containing 1 month's items would take approximately 2 min and 22 sec. Comparison with more conventional bookkeeping methods is hardly necessary. On the other hand, if we were to run a standard payroll calculation on the man-hours worked for each of the 500 men, it would take (500 men) (2 min 22 sec) = 20 hr. Also we have not mentioned the input and output equipment problem, which in the case of payroll, for instance, is a subject in itself.

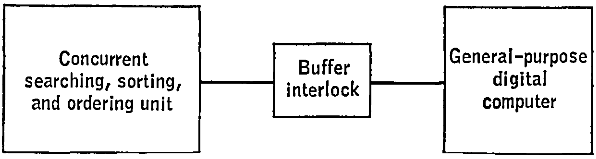

Concurrent searching, sorting, and ordering unit
Buffer interlock
General-purpose digital computer

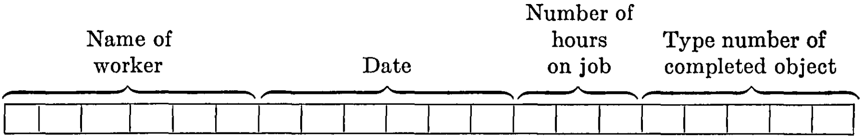
FIG. 8-11. Special-purpose computer attached to general-purpose computer.

Integration with a General-purpose Computer. In order to increase the working speed of this type of computer, we can enlarge the argument storage unit and include more accumulators and counters to accommodate more than one comparison, so that in one pass several pieces of information can be accumulated; or we can run several tapes in parallel, and so forth. The time per information pass can thus be decreased by a factor of 10 or 100, that is, to within 20, or even 2, sec. This would increase the complexity and hence the cost of the computer. Finally we can attach our special-purpose searching, sorting, and ordering computer to a general-purpose computer, enabling more complicated processing to be accomplished as well as more automatic operation (see Fig. 8-11). In such a case the tapes can be searched at the same time as the general-purpose computer computes other things. The fixed card memory of the special-purpose computer will be replaced by a part of the high-speed memory shared with the general-purpose computer. Two codes will be written, one for the computing unit of the general-purpose computer, the other for the simple arithmetic unit of the special-purpose computer.

An interlocking buffer must be designed so that the general-purpose computer will stop and wait if it needs some information not yet retrieved from the special-purpose auxiliary computer, and so that the searching computer will stop and wait if it in turn needs information not yet generated by the general-purpose computer. An auxiliary computer for this special purpose can be exceedingly effective and timesaving when large amounts of this kind of nonnumerical manipulative computing are necessary.

EXERCISE

(*a*) Suppose that a factory item is composed of the following word,



where each box can be filled with a digit or an alphabetic character. Define an instruction system for a special-purpose retrieval computer as shown in Fig. 8-9, and write a code that will determine how many objects of type 54321 were completed between Mar. 3, 1959, and Mar. 15, 1959.

8-5. Manipulations with Rectangular Arrays: A Business and Logistics Computer†

The Need. An important class of computations occurring frequently in business and in logistics (i.e., processes involved in supply) is concerned with rectangular, matrixlike arrays of variables. The operations to be performed on the variables are usually very simple, the main problem being the manipulations necessary for handling such large-scale arrays. In this section we shall first indicate some applications of these techniques; we shall then show how a few relatively simple special-purpose instructions can significantly aid these processes; and finally we shall discuss a special-purpose computer especially designed to perform manipulations with rectangular arrays.

As a first example, consider the keeping and updating of a spare-parts inventory. Suppose for simplicity that there are 10 parts under consideration and that the first row of the array in Table 8-4 represents the desired inventory for each of the 10 parts—i.e., there *should be* 100 each of the parts in stock. The second row represents the actual stock level of each of the parts at time *t*. Now let us suppose that the parts are consumed in the course of three kinds of overhauls and that the number of parts used for one of each kind of overhaul appears on lines 4, 5, and 6, respectively. Now suppose that since time *t* there were one overhaul of kind 1, two overhauls of kind 2, and three overhauls of kind 3. The

† The main concepts discussed in this section are based on the original research of Dr. W. H. Marlow, principal investigator of The George Washington University Naval Logistics Research Project.

problem is to determine the present stock levels, i.e., to adjust row 2, and to determine how many of each part should be ordered in order to have the desired number of spare parts on hand, i.e., to determine row 3. To take account in the inventory levels of the parts used for the overhaul of kind 1, subtract row 4 from row 2, column by column; for the two overhauls of kind 2 multiply each figure in row 5 by 2, and subtract the result from row 2, column by column; and similarly form

TABLE 8-4. DATA ARRAY FOR INVENTORY-UPDATING PROBLEM

	Part 1	Part 2	Part 3	Part 4	Part 5	Part 6	Part 7	Part 8	Part 9	Part 10
Row 1, desired inventory	100	100	100	100	100	100	100	100	100	100
Row 2, stock on hand (time *t*)	80	75	90	80	95	90	85	95	70	75
Row 3, parts to order										
Row 4, overhaul kind 1, parts used	5	15	5	10	5	0	0	5	10	5
Row 5, overhaul kind 2, parts used	10	0	0	10	10	20	15	5	10	0
Row 6, overhaul kind 3, parts used	5	0	10	10	5	10	15	20	0	5

$$\text{Row } 2 - 3 \times (\text{row } 6)$$

We thus obtain the present stock levels in row 2:

40 60 55 20 55 20 10 20 40 55

Subtracting row 2 from row 1 and putting the results in row 3, column by column, we obtain the number of parts to order of each type. Of course many other related problems can be similarly solved; e.g., we may record the usage of parts in each time period and then extrapolate column by column to predict usages in the future, and so forth. We have used only 10 different parts, but in realistic cases thousands of parts may be involved.

Another example is the calculation of building-material purchase schedules and requirements as functions of time, as would be needed in the construction of a ship or of a large office building. An array can be made of the materials needed at the construction site as a function of time. That is, successive rows of the array will represent successive times, and the columns of the array will represent the various building materials; then an element in the array will indicate how much of a certain building material is needed at the building site at a particular time. However, there is a certain time lag that must be allowed for the delivery of each of the materials from the time it is ordered. This time lag differs for different materials and can be represented as a row of the array. The problem here is to schedule the orders so that the materials will arrive at the site at the proper time. In addition the costs of the materials per unit can be represented in another row. If the materials are paid for at the time they are ordered, then a calculation should be made as to how much money will be spent during each time period. This calculation can be made after the ordering schedules are set by multiplying the number of units to be ordered in a time period (i.e., a row) by the cost of each per unit and summing over the elements of the resulting row, i.e., adding the columns of that row independently.

The keeping of bank accounts presents similar array-manipulation problems. Here suppose that each row contains the balance of each of the savings accounts for successive weeks and that interest is computed monthly, based on the minimum weekly balance. Here the minimum of the rows each month must be obtained and the interest rate applied to each account (i.e., each column of the minimum record row). If the bank desired to know the total deposits each week, the row for each week is totaled. In this way calculations of interest, principal, penalties, interest rates, and other banking problems can be made.

Military and nonmilitary dynamic-operations simulation studies can involve these processes. For example, consider air-traffic control (in three dimensions). Three rows can record the x and y coordinates and altitudes, respectively, of the aircraft represented by the successive columns. If the x and y velocity components are known, the coordinates of the aircraft can be predicted in time and aircraft that may be flying too close appropriately warned. When changes in course are made, recalculations of predicted positions of all the aircraft become easy.

Of course, we could describe above only the simplest of applications. Our intention is to stimulate the reader's awareness of the great variety of fields in which problems arise that involve the manipulations of large arrays. Although a general-purpose computer can handle such problems, special-purpose computers can fulfill the need much more effectively and efficiently. We shall now describe some special-purpose operations that would be incorporated into such a computer.

Basic Operations and Instruction Format. With the above examples in mind the reasons for the definitions of the following basic manipulatory operations are clear. In fact it is quite surprising that these few simple operations can accomplish the task so well. The instructions that we shall illustrate contain the necessary ingredients, although their form in actual practice may vary greatly. It is felt that a concrete example may best serve to demonstrate the concepts.

Since we are concerned primarily with arrays of numbers, i.e., with numbers arranged in rows and columns, suppose that we literally arrange these numbers in such an array in the computer's memory. Figure 8-12 represents such an arrangement on a drum memory. Then in order to "address" a particular number, all we need specify is its *row* and *column.* For reasons that will presently appear clear, we denote the number in row j and column x by the following functional notation: $f_j(x)$. See, for example, Table 8-5.

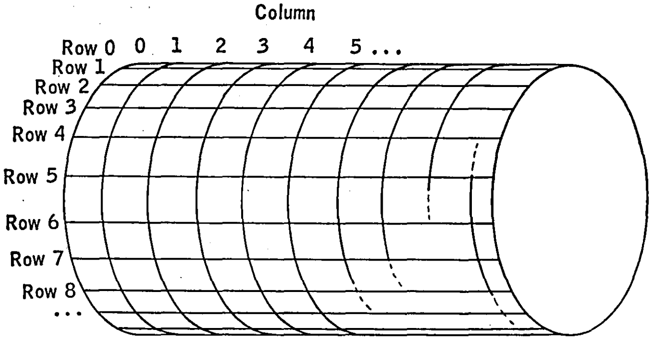


FIG. 8-12. Arrays of numbers stored on drum memory.

The address of $f_j(x)$ is thus specified by the *pair of indices: j, x.* We use the phrase *generalized address* because the instruction actually indicates a collection of addresses in the manner now to be described. Consider a multiplication instruction. Rather than form the product of just two numbers, $f_i(x) \cdot f_j(x)$, we wish now, *with a single instruction,* to form a whole row of successive column-by-column products, that is, $f_i(1) \cdot f_j(1)$, and $f_i(2) \cdot f_j(2)$, and $f_i(3) \cdot f_j(3)$, and so forth. We might represent this as the operation $f_i(x) \cdot f_j(x)$ for $x = 1, 2, 3, \ldots .$ To add a little more flexibility to the operation, we might form, with a single instruction, $f_i(x) \cdot f_j(x)$ for $p \leq x \leq q$, that is, the successive products for only the columns p through q. For example, if $i = 1, j = 2, p = 5$, and $q = 8$ we would form (see Table 8-6)

TABLE 8-5. NOTATION OF THE ARRAY

	Column x								
	0	1	2	...	p	...	q	...	n
0	$f_0(0)$	$f_0(1)$	$f_0(2)$	...	$f_0(p)$	...	$f_0(q)$	...	$f_0(n)$
1	$f_1(0)$	$f_1(1)$	$f_1(2)$	...	$f_1(p)$	...	$f_1(q)$	...	$f_1(n)$
2	$f_2(0)$	$f_2(1)$	$f_2(2)$	...	$f_2(p)$	...	$f_2(q)$	...	$f_2(n)$
...	...	...	...	...	...	...	...	...	...
i	$f_i(0)$	$f_i(1)$	$f_i(2)$	...	$f_i(p)$	. .	$f_i(q)$	...	$f_i(n)$
...	...	...	...	...	...	...	...	...	...
j	$f_j(0)$	$f_j(1)$	$f_j(2)$	...	$f_j(p)$	...	$f_j(q)$	...	$f_j(n)$
...	...	...	...	...	...	...	...	...	...
m	$f_m(0)$	$f_m(1)$	$f_m(2)$	...	$f_m(p)$	...	$f_m(q)$	...	$f_m(n)$

$$f_1(5) \cdot f_2(5) \qquad f_1(6) \cdot f_2(6) \qquad f_1(7) \cdot f_2(7) \qquad \text{and} \qquad f_1(8) \cdot f_2(8)$$

Thus the multiplication instruction must now contain the row indices i and j, and the column indices p and q; this collection of indices represents the so-called "generalized addresses." Suppose that we wish the results to be placed in row k of the corresponding columns p to q; then the index

k must also appear in the instruction. Thus i, j, and k, together with p and q, represent the generalized α, β, γ, and δ addresses, and all these indices must appear in the instruction format itself.

TABLE 8-6. ILLUSTRATION OF OPERATION $f_1(x)f_2(x)$, $5 \leq x \leq 8$

	Column									
	0	1	2	3	4	5	6	7	8	9
Row 1.....	9	7	8	6	1	5	2	4	3	1
Row 2.....	6	2	5	1	0	3	4	8	7	2
Row 3.....	...	...	...	...	...	15	8	32	21	...

A description of the fourth generalized address, that of the next instruction, requires reconsideration of the drum memory. An instruction would appear as a partial row of numbers and can be recorded along with the data on the drum. For greatest efficiency a row length on the drum should be a multiple of an instruction length. To address the part of a row that is an instruction, only the row and column of the first number of the instruction need be known, since the length of the instruction is fixed by the instruction format (see below). Let us denote the row and column number of the first characteristics of an instruction by δ_r and δ_c, respectively. These must also appear in each instruction word.

There are several other considerations that can lead to greater flexibility in our instruction format. First suppose that we desired to form $C \cdot f_i(x) \cdot f_j(x)$ (for $p \leq x \leq q$), where C is a constant. Since this is frequently desired (e.g., interest and discount rates), it is convenient to include the constant C in the instruction format itself, rather than address it separately. Second, observe that so far we can work on the different rows *within* the same column but we cannot have a number in one column affect a number in another column. Thus, for example, the product $f_i(x) \cdot f_j(x)$ becomes $f_k(x)$ in the same column. To circumvent such a restriction, it suffices to introduce into the instruction format the alternative of placing the result of the operation in corresponding columns in the kth row, *or of placing the result one column to the left in the kth row, i.e., displaced one column.* That is, one may choose to have the product $f_i \cdot (x) \cdot f_j(x)$ become $f_k(x)$ or $f_k(x - 1)$. Finally recall that (γ) is usually *replaced* by the result of the operation. However, great flexibility results if there is introduced into the instruction the alternative of either replacing (γ) *or else accumulating (adding) the results of the operation to the original contents of* γ. As an example, suppose that, given the initial rows 1, 2, and 3 of Table 8-7, we formed $f_1(x) \cdot f_2(x)$, for $5 \leq x \leq 8$ (that is, $i = 1, j = 2, p = 5, q = 8$), and *accumulated* the result in row 3, *displaced.* Then we would have

$$f_3(x - 1) \text{ final} = f_1(x) \cdot f_2(x) + f_3(x - 1) \text{ initial}$$

for $5 \leq x \leq 8$ (see Table 8-7).

TABLE 8-7. ILLUSTRATION OF OPERATION $f_1(x)f_2(x)$, $5 \leq x \leq 8$, PLACED IN ROW 3 DISPLACED BY ONE COLUMN, AND SO AS TO ACCUMULATE

	Column									
	0	1	2	3	4	5	6	7	8	9
Row 1...........	9	7	8	6	1	5	2	4	3	1
Row 2...........	6	2	5	1	0	3	4	8	7	2
Row 3, initial....	3	2	4	6	2	1	5	3	2	6
Row 3, final.....	3	2	4	6	17	9	37	24	2	6

With these considerations in mind we can represent an instruction format as:

Operation code	Row i	Row j	Row k	p	q	Displaced or not	Accumulated or not	C	δ_r	δ_c

We shall use the convention that 1 means *displace*, 0 means *do not*, and that 1 means *accumulate*, 0 means *do not*, in their respective columns. The instruction that will produce the final result shown in Table 8-7 is:

Operation	i	j	k	p	q	Displace	Accumulate	C	δ_r	δ_c
Multiply	1	2	3	5	8	1	1	1		

We are now ready to define seven basic operations, where the column-by-column disposition of each type of result is understood to be in accord with the above discussion, and where only the column-by-column operations on the arguments need be given.

1. Multiplication

$$Cf_i(x) \cdot f_j(x) \qquad p \leq x \leq q$$

2. Division

$$\frac{Cf_i(x)}{f_j(x)} \qquad p \leq x \leq q$$

3. Linear combination

$$Cf_i(x) + f_j(x) \qquad p \leq x \leq q$$

4. Minimum

$$\min_{i,j} [f_i(x), f_j(x)] \qquad p \leq x \leq q$$

The final three instructions require some additional explanation:

5. Displaced linear combination $Cf_i(x-1)+f_j(x)$. Here the addition is not performed column by column, but rather *displaced* column by column. Of course, it is always understood that $p \leq x \leq q$, but then, when $x = p$ and $p = 0$, the meaning of $f_i(0-1)$ is undetermined. We therefore define $f_i(0-1) = f_i(0)$.

6. Partial scalar product $\sum\limits_{\mu=p}^{x} C \cdot f_i(\mu) \cdot f_j(\mu)$. This defines *one result for each* x such that $p \leq x \leq q$. For example, if row i and row j were:

	Column					
	0	1	2	3	4	5
Row i.........	7	6	5	4	3	2
Row j.........	2	3	4	5	6	7

and $p = 2$ and $q = 5$, then the results would be

$5 \times 4 = 20 \qquad 5 \times 4 + 4 \times 5 = 40 \qquad 5 \times 4 + 4 \times 5 + 3 \times 6 = 58$

and $\qquad 5 \times 4 + 4 \times 5 + 3 \times 6 + 2 \times 7 = 72$

7. Compare $(\min\limits_{p \leq x \leq q} [f_i(x)]):C$. This operation does not yield a column-by-column result; rather it is a *two-way exit.* If the minimum number in row i, columns p to q, is *greater than* C, then the next instruction is to be taken as the instruction that has row j, column k as the address of its first number; otherwise, i.e., if $\min\limits_{p \leq x \leq q} [f_i(x)] \leq C$, the next instruction is to be chosen as usual, from δ_r, δ_c. Of course *displace* and *accumulate* have no meaning for a *compare* instruction.

Illustrations of Coding. As a first illustration of using these special-purpose instructions in writing codes, consider the problem of accumulating the amounts within each column, from row i_0 to row $i_0 + h$, putting the results into row $i_0 + h$. Suppose for simplicity that the drum has 22 columns (just twice the instruction length) and that the code is written in address 0, 0; 1, 0; 2, 0; etc. The code will make use of operation 3, with $C = 1$, *not* accumulating the results of summing i and $i + 1$. (Why?) Then a *compare* instruction will occur for the tally, and the iterations will proceed by adding 1 to the generalized addresses i, j, and k of the first instruction until the accumulation is completed. Since the first instruction is updated on each iteration, its i address can be used as the tally. The code appears in Table 8-8.

As a second example, consider a linear extrapolation. Suppose that each column represented an aircraft and that rows x_1, y_1, and t_1 represented the x and y coordinates of each aircraft at time t_1, and rows x_2,

y_2, and t_2 represented the x and y coordinates of each aircraft at time t_2. The problem is to make a linear extrapolation of the coordinates of each plane at time t. The formulas to be used are

$$x = (t - t_2)\frac{x_2 - x_1}{t_2 - t_1} + x_2 \qquad \text{and} \qquad y = (t - t_2)\frac{y_2 - y_1}{t_2 - t_1} + y_2$$

We first form $x_2 - x_1$ in row x_1, $y_2 - y_1$ in row y_1, $t_2 - t_1$ in row t_1, and finally $t - t_2$ in row t. Then we form the quotient of rows x_1 and t_1 in row x_1 and of rows y_1 and t_1 in row y_1. Next we form products of rows t and x_1 in row x_1 and of rows t and y_1 in row y_1. Finally row x_2 is added to row x_1 and put in row x_3; and row y_2 is added to row y_1 and put in row y_3; and the extrapolation is completed. The method of coding is straightforward, using $C = -1$ in operation 3 for subtraction.

TABLE 8-8. CODE TO ACCUMULATE THE ROWS i_0 TO $i_0 + h$, COLUMN BY COLUMN†

Instruction address	Operation	i	j	k	p	q	Displace	Accumulate	C	δ_r	δ_s	Remarks
0, 0	3	i_0	$i_0 + 1$	$i_0 + 1$	0	21	0	0	1	1	0	Add i to $i + 1$; put in $i + 1$
1, 0	7	0	3	0	1	1	. . .	. . .	$i_0 + h - 1$	2	0	$i:i_0 + h - 1$ $>$: Stop iteration $\leq$: Continue
2, 0	3	4	0	0	1	3	0	0	1	0	0	Update (0,0)
3, 0												Stop computer
4, 0	0	1	1	1	0	0	0	0	0	0	0	Constant

† Recall that the columns are numbered 0 through 21.

As a third example, consider a numerical integration according to the rectangular approximation. Suppose that the successive values of the function at points x are given by $f_i(x)$ and that the intervals are unequal and are given in row $i + 1$. Then for the rectangular approximation we want to form the product of rows i and $i + 1$, column by column, and then accumulate this product over the columns. Here we make direct use of operation 6 and form

$$\sum_{u=0}^{21} f_i(u)f_{i+1}(u)$$

in row $i + 2$. Then the value of the approximation is found in row $i + 2$, column 21.

Systems Design for a Special-purpose Business or Logistics Computer. One of the most important features that a special-purpose computer of this type should have is rapid, flexible input-output. A general characteristic of the problems under consideration in this section is the large mass of input data to fill the array and the large mass of data computed from the array. This is in fact the main reason for having instructions that operate with generalized addresses. The given data may be characterized as items and characteristics associated with the items. Here the characteristics are the numbers upon which it is necessary to perform operations, and the items are the specifications of these numbers. Hence on the drum the *location of a number* can represent the item or specification, while the *number itself*, i.e., the contents of this location, will be the *characteristics*.

For example, in a naval supply problem an item might consist of the set of numbers (x_0,x_1,x_2,x_3), where x_0 = ship number, x_1 = time period (e.g., month and year), x_2 = Navy material class (e.g., class 75, diesel engine parts), and x_3 = source of supply (e.g., the vessel itself, an afloat activity, a shore activity). The associated characteristics might consist of the set of numbers (y_0,y_1,y_2), where y_0 = total number of items consumed, y_1 = total weight in pounds, and y_2 = total volume in cubic feet. The source of this data may have covered 12 ships, during 72 time periods, for 50 Navy material classes, and from three sources of supply, or 129,600 items. For each of these items there appear three six-digit numbers representing y_0, y_1, and y_2. The data would then contain 2,332,800 digits.

Processing this data might consist in computing and printing tables with entries y_0, y_1, and y_2 corresponding to some specified set of the x's. For example, the set (x_0,x_1) would lead to a two-way table of vessel number vs. time period, with entries covering all classes of material from all sources. Or additional functions of the y's may be desired. Average values are about the simplest example: $y_3 = y_1/y_0$ = average weight in pounds; $y_4 = y_2/y_0$ = average volume in cubic feet; $y_5 = 2{,}240y_2/y_1$ = stowage factor in cubic feet per long ton.

It often occurs that a characteristic of one problem becomes an item specification for another problem, or the reverse. For instance it might be desired to compute a table giving the values of y_0, y_1, and y_2 for each x_0, x_1, and y_3, that is, for each ship, time period, and *average weight*. In such a case the numbers y_0, y_1, and y_2 will have to be relocated so that x_0, x_1, and y_3 will now specify their locations. Hence it becomes clear that in business and logistics computations searching, sorting, and ordering play an important role, in addition to the types of computations considered above.

In the previous section we assumed that the information was recorded on a magnetic tape; here, however, we shall consider searching, sorting, and ordering of items by characteristics accomplished on an auxiliary drum, different from the one with which the computations are performed. The input data, which might, for instance, be on punched cards, is read onto the searching drum, both the items and the characteristics being recorded. This drum is appropriately searched, and the characteristics determined by the search are recorded on the computational-array drum in positions (i.e., addresses) that specify the associated item. Any rearrangement or reordering is accomplished by use of the searching drum.

Our special-purpose business or logistics computer is thus seen to be *two interlocking computers,* an *array-manipulation computer* and a searching or *information-retrieval computer* (see Fig. 8-13). Each would have

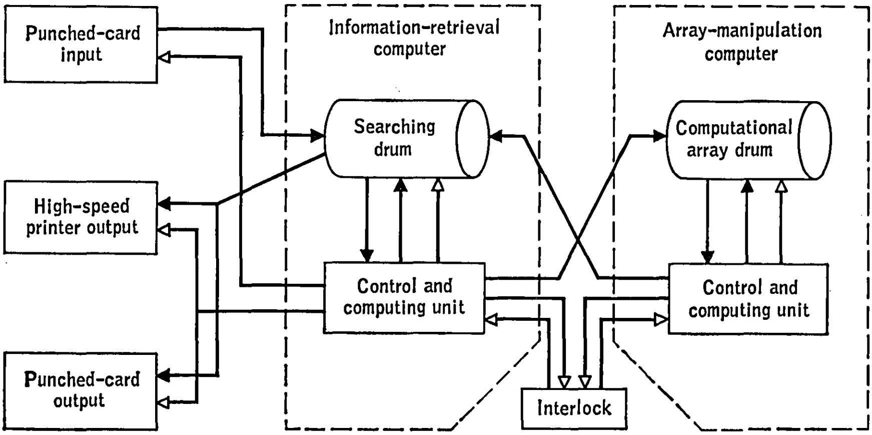

Punched-card input
High-speed printer output
Punched-card output
Information-retrieval computer
Searching drum
Control and computing unit
Array-manipulation computer
Computational array drum
Control and computing unit
Interlock

FIG. 8-13. Special-purpose business or logistics computer.

its own programming system. The array-manipulation computer would have special-purpose array-manipulation instructions as described above; the information-retrieval computer would have instructions enabling it to perform the operations described in the previous section. Ideally the two computers should be able to operate concurrently; appropriate interlocking conditions would have to be satisfied, so that the two computers and computer programs would be properly synchronized.

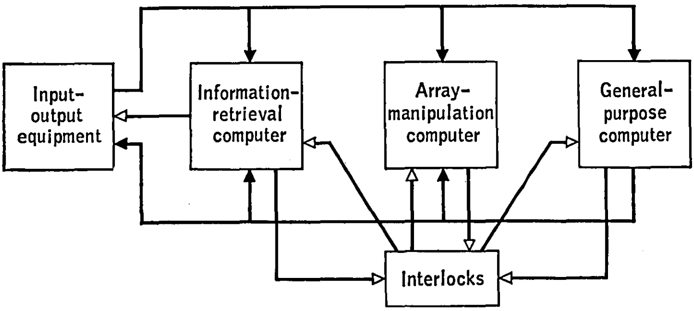


FIG. 8-14. Large-scale system consisting of information-retrieval, array-manipulation, and general-purpose computers.

In addition we could attach a general-purpose computer to this setup to have a large-scale system (Fig. 8-14). Again for concurrent operation interlocks must be maintained. More will be said about such an arrangement in the next section.

EXERCISES

(*a*) Consider the example of computing building-material purchase schedules and requirements. Suppose that the requirements at the site for each month during the construction are as follows:

Month	Type of material										
	0	1	2	3	4	5	6	7	8	9	10
March	100	75	60	54	0	0	20	60	10	20	30
April	50	0	40	60	0	0	20	90	10	20	0
May	25	0	30	50	0	0	30	150	10	30	0
June	30	115	20	70	0	0	50	50	50	0	20
July	0	0	10	30	10	20	90	20	70	0	10
August	0	100	5	50	50	30	100	10	90	50	5
September	0	0	0	90	100	100	150	0	100	90	0
October	0	0	0	20	0	150	200	0	150	100	0

Suppose that the delay time and the cost per unit are as follows (where the cost per unit of the material varies with the number of units ordered as shown):

	Type of material										
	0	1	2	3	4	5	6	7	8	9	10
Delay time, months	3	5	6	2	4	6	5	3	5	6	7
Units ordered:											
0–9	10	50	35	20	5	40	30	20	30	100	150
10–19	10	45	35	15	5	35	30	15	30	100	120
20–49	10	40	30	14	5	30	25	15	25	95	110
50–99	7	40	25	13	4	25	25	10	25	95	100
100 or over	7	38	20	12	3	20	20	10	20	90	100

Determine the schedule for the ordering, and determine how much money will be spent each month, assuming that the orders are paid for at the time they are ordered.

(*b*) Write a code that will compute Exercise *a*.

(*c*) Write a code for numerical integration by the trapezoidal rule.

(*d*) Write a code for the linear extrapolation.

(*e*) What situations might require interlocks between the information-retrieval computer and the array-manipulation computer?

8-6. The Large-scale Data Processor

Systems Design. Perhaps the most important observation about systems design is that whenever possible systems design should be accomplished for the *specific purpose* for which the computer is to be used. This does not mean that the computer should be of limited capabilities—rather, design for a purpose means that the computer's capabilities should be *extended* to *include* features that will enhance its effectiveness for the special purpose for which it is intended to be used. However, one pitfall must be avoided in the design of special-purpose computers. It is usually possible to utilize special-purpose computers for problems not directly involved with the special purpose for which the computer was designed. In such cases the coding may be much more difficult. This often leads to increased cost of coding or to making undesirable simplifications in a problem to have it more nearly conform to the type for which the special-purpose computer was designed. Such a situation tends to diminish the advantages to be derived from using a digital computer in the first place. Hence it is usually unwise to use a special-purpose computer for problems other than those for which it was specifically designed. These considerations emphasize the importance of *carefully evaluating the purpose* before embarking on a computer design for that purpose, to be certain that the problems under consideration are truly in line with the special features of the proposed design.

The systems design of a digital computer is characterized by the specifications for (1) its computing components, (2) its memory components, (3) its input-output components, and (4) its interlock control. The specifications of a computing component include the instruction and quantity format, the list of instructions, the speed of execution of instructions, the logical design technique, and the type of electronic circuitry used. These last two specifications will be covered in Parts 3 to 5. The specifications of a memory component include its type (i.e., magnetic core, magnetic drum, etc.), size (i.e., number of words), speed or access time, logic (e.g., for a drum whether the words are round the circumference or along the axis, etc.), and circuitry. The specifications of the input-output components include the kind, speed, and electromechanical design. The interlock controls, already mentioned in the preceding sections, will be discussed in greater detail below.

Special Input-Output Problems. When data is collected, the recording scheme used is ordinarily determined from criteria dictated by the particular problem, by convenience, etc. Rarely is the best form for computer utilization considered as a factor. Also, data produced by a computer will not in general be directly compatible for input into another type of computer. These problems become extremely serious in, for example, missile-testing programs, aircraft-flight evaluations, etc., where exceedingly large masses of data are produced very rapidly. The problem of conversion of this data to a form that can be read directly into the computer for analysis then becomes almost a larger task than collecting the data.

One method for solving this problem is to have a special input drum for the computer. Everything that is read into the computer is *read directly onto this drum just as it appears on the input medium.* If the input were punched cards with 80 columns and 10 rows, then the data of the cards would be read onto the drum just as it appears on the cards, with a unit corresponding to a hole, a zero to no hole. Then a special *translating* *computing unit* would be coded to interpret this information appropriately. The translating unit would interpret the raw information on the drum, change it into appropriately coded binary data, and put it into the computer's main memory. This technique is extremely flexible, since all that is needed is a method for getting the raw data onto the drum, in any digital fashion whatsoever.

This naturally brings up a second problem. Since reading in and translating the data will be time-consuming, it would be more efficient if the computer could be computing on other data, or even on another program, concurrently with the read-in and translation. Such a *concurrent-input feature* presents no problems other than logical and electronic design, provided that certain interlocks are observed. In reading data into a computer it is certainly known in which part of the main memory the information will eventually be stored. A simple type of interlock will halt a concurrent computer program that refers to a main memory address that has not yet been read into; when the input information is inserted into this address, the program automatically continues.

In so far as output from the computer is concerned, the translator acts in reverse, putting information on the special drum in a form which when read out from that drum will correspond to any desired external coding scheme. Similarly with proper interlocks concurrent output can be accomplished.

The Large-scale Data Processor. The large-scale data processor as described in this paragraph (see Fig. 8-15) is a fictitious computer; its description is used here as a vehicle for discussing the problems that arise in *multiple-unit systems design.* The purpose of this illustrative computer would be to perform general data processing. It encompasses each of the three features described in Secs. 8-3, 8-4, and 8-5, in addition to the translating component. Hence it is comprised of a general-purpose computing unit, a logical-control computing unit, a searching, sorting, and ordering computing unit, an array-manipulating computing unit, and an input-output translator computing unit. It contains a high-speed memory, a general-purpose magnetic drum, a translator input-output drum, a sorting drum, an array-manipulation drum, as well as magnetic-tape units. It has conventional input-output equipment and a signal-sensing input unit for real-time control.

The main point, however, to be illustrated by this example is that any or all of the computing units may be operating (i.e., computing) concurrently, using any or all of the memories or input-output units. The mechanisms that enable such concurrent operation are found in the in-out selector–concurrent input-output interlock control and in the computing-units interlock control.

Let us consider the computing-units interlock control first and describe two possible ways it can operate. Consider the case where our data processor is operating on multiple independent codes simultaneously. The instructions for each of these codes are found in the memory units. At any time, each code is being computed in a different computing unit.

Each computing unit has its own instruction register, and its own current-address register that sequences the program according to the instruction definitions of the particular unit in which it is located. There are times, however, when one code will want to operate in a different unit, will want to change computing units. These *change-unit* instructions are not handled in the computing units; instead they are sensed and sent to the computing-units interlock control. This interlock control keeps track of which units are being used. If a code wants to change to a computing unit that is being used, the code progress is stopped until the desired unit is free. A waiting line, or queue, is formed if necessary for each computing unit. The order, or priority, in which waiting codes are selected from the queues can be made as complicated as is considered necessary for the purposes of the computer design.

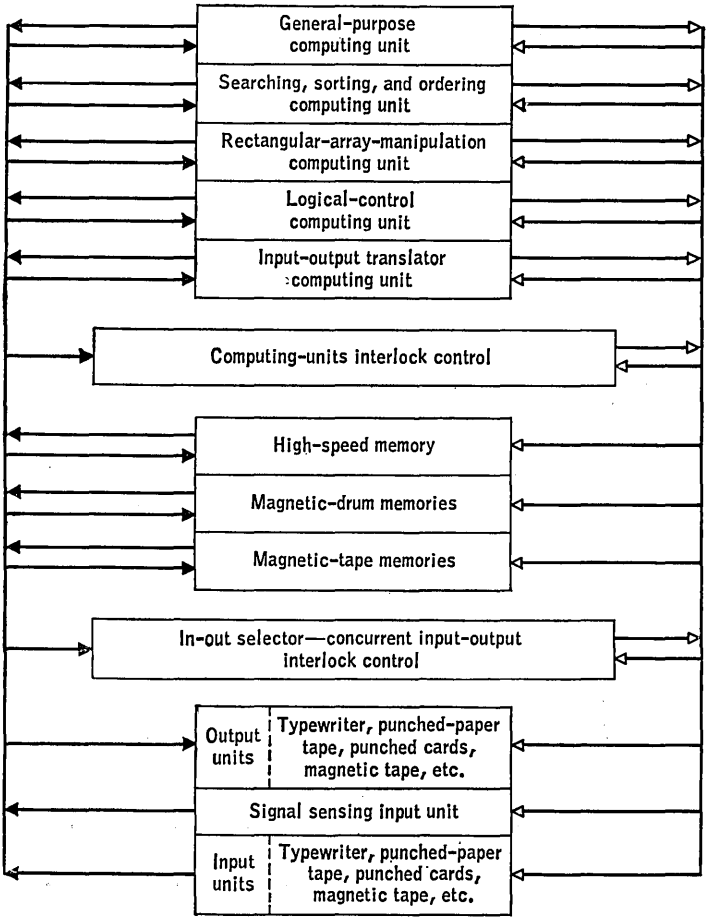

General-purpose computing unit
Searching, sorting, and ordering computing unit
Rectangular-array-manipulation computing unit
Logical-control computing unit
Input-output translator computing unit
Computing-units interlock control
High-speed memory
Magnetic-drum memories
Magnetic-tape memories
In-out selector—concurrent input-output interlock control
Output units
Typewriter, punched-paper tape, punched cards, magnetic tape, etc.
Signal sensing input unit
Input units
Typewriter, punched-paper tape, punched cards, magnetic tape, etc.

FIG. 8-15. Large-scale data processor.

When the desired computing unit becomes available, the contents of the instruction address register is transmitted to this computing unit and the code is continued in its new unit. In other words, a computing unit processes the instructions it retrieves from the memory as directed by its current-address register; when changing from one code to another, a computing unit merely needs to have its current-address register changed. The control mechanism for this change is located in the computing-units interlock control. By this means each of the computing units can act almost as a separate computer in its own right, except when required to idle by the computing-units interlock control. Even though this concept for handling multiple codes at the same time is reasonably simple, the detailed techniques in a specific case may present many special considerations and exceptions.

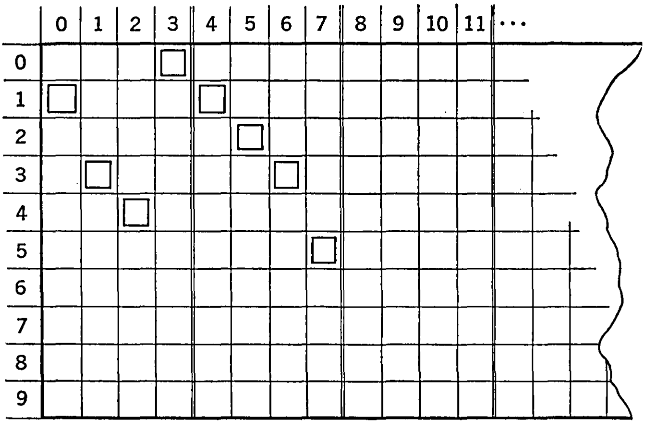
The second way in which a computing-unit interlock control might operate is to have only a single code being computed. However, it often happens that different parts of this code can be computed at the same time. That is, the logic or flow diagram of the code may not necessitate a completely serial sequencing of operations, but several parts of the code might be processable in parallel at the same time. In this mode the computing-units interlock control scans the entire code, recognizing parts of the code that can be executed independently. It does this by observing memory reference locations. If one part of a code never refers to a memory location used by another part of the code, then it can be accomplished at the same time as that other part of the code. The rules for observing this can become rather complicated. The computing-units interlock control must accomplish both functions; how thoroughly it accomplishes either function depends on the systems design and purpose of the computer.

Finally we consider the in-out selector–concurrent input-output interlock control. Besides in-out selections, this control unit must accomplish input-output interlock control similar to that described above. All input-output instructions are automatically sensed and sent to this interlock control. If free, the appropriate unit is selected; if not, the code will wait, as was the case in the computing-units interlock control. This interlock control must communicate with the computing-units control, so that a code may be stopped when it wants to use a memory location not yet loaded. Again the details can become exceedingly complicated.

EXERCISES

(*a*) Make a chart with columns labeled with the specification names of the computer components and rows labeled with the various purposes considered in this chapter. Fill in the table with component specifications, giving a reason for each decision. (For example, the entries in the table will state how many addresses the instruction system should have, how large the memory should be, the access speed, and so forth.)

(*b*) Draw a flow chart of a code that would interpret raw data from cards. Suppose that a computer word contains 15 bits, with the decimal point to the right. Assume that the raw data appear on cards with 80 columns and 10 rows, where each card is interpreted as containing 20 four-digit numbers and the digit associated with each column is the row location of a single punch in that column. For example, the first two numbers on the card illustrated in the figure are 1,340 and 1,235. Assume



EXERCISE *b*

that the translator drum records the *rows* of each card in sequence, from 0 to 9. The code is to compile the successive numbers (i.e., successive groups of four columns) *in binary* in the main memory of the computer.

8-7. Additional Topics

a. References on Digital Differential Analyzers. In general, digital differential analyzers are far more versatile and complicated than described in Sec. 8-2. For a more thorough understanding of their variations and potentialities see the following references:

Bush, Vannevar: Differential Analyzer, *J. Franklin Inst.*, vol. 212, no. 4, pp. 447–448, October, 1931.

Donan, J. F.: The Serial-memory Digital Differential Analyzer, *Mathematical Tables and Other Aids to Computation*, vol. 6, no. 38, pp. 102–112, April, 1952.

Hartree, Douglas R.: "Calculating Instruments and Machines," University of Illinois Press, Urbana, Ill., 1949.

Mendelson, Myron J.: The Decimal Digital Differential Analyzer, *Aeronaut. Eng. Rev.*, February, 1954, pp. 42–54.

Palevsky, M.: The Design of the Bendix Digital Differential Analyzer, *Proc. IRE*, vol. 41, pp. 1352–1356, October, 1953.

Sprague, R. E.: Fundamental Concepts of the Digital Differential Analyzer, *Mathematical Tables and Other Aids to Computation*, vol. 6, pp. 41–48, January, 1952.

Weiss, E.: Applications of the CRC 105 Decimal Digital Differential Analyzer, *IRE Trans. on Electronic Computers*, vol. EC-1, pp. 19–24, December, 1952.

Winslow, D. J.: Incremental Computers in Simulation, *Meeting of South East Simulation Council*, Oct. 30, 1958, Huntsville, Ala.

b. Multiple-component Computing Systems. In Sec. 8-6 the systems design of computers composed of several different interlocked computing components was described. The following references give further details about specific examples of systems designs illustrating this principle:

Dreyfus, Phillipe: France's Gamma 60—A Step Forward in Data Processing? *Datamation Research Eng. J.*, May–June, 1958, p. 34.

Dunwell, S. W.: Design Objectives for the IBM Stretch Computer, *Proc. Eastern Joint Computer Conf.*, December, 1956, p. 20.

Eckert, J. P.: Univac-Larc, the Next Step in Computeɪ Design, *Proc. Eastern Join Computer Conf.*, December, 1956, p. 20.

c. System Engineering. We have been considering the systems design of digital computers. However, more often than not the digital computer is itself a component of a larger organized system. The study of such large-scale systems engineering is considered in a text by Harry H. Goode and Robert E. Machol, "System Engineering," McGraw-Hill Book Company, Inc., New York, 1957. Chapter 2 of this text is devoted to examples of such large-scale systems. How can a digital computer fit into each of the systems described in that chapter? Other discussions of systems design can be found in M. M. Hunt, Bell Labs 230 Long Planners, *Fortune,* May, 1954, p. 120; and P. F. Drucker, The Promise of Automation, *Harper's Magazine,* April, 1955. The following periodicals frequently discuss large-scale systems: *Control Engineering,* a McGraw-Hill publication; and the *IRE Transactions on Industrial Electronics.*

CHAPTER 9

SYSTEMS DESIGN OF THE PEDAGAC

Purpose. Pedagac, which stands for "pedagogic automatic computer," is the name of a computer that will be completely designed in this text. The purpose of the Pedagac is to provide the necessary *thread of continuity* to the study of digital-computer engineering. The process of its design as presented in this book will relate the systems design to the logical design and to the electronic design of a computer. Its primary purpose is to teach; its systems design, logical design, and electronic design have been chosen to demonstrate the procedural chain involved in the engineering of a computer. In many ways the Pedagac may not be appropriate for construction for any purpose other than engineering teaching and demonstration; for the oversimplification often used for pedagogic purposes makes the Pedagac less efficient in computing speed and use of electronic hardware than is really necessary. However, the Pedagac can be built; it is a general-purpose digital computer.

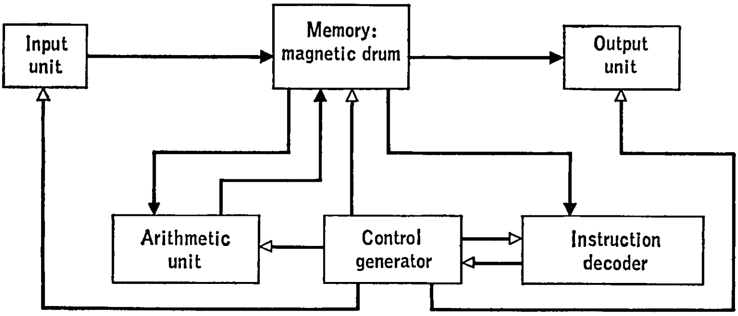


FIG. 9-1. Block diagram of the Pedagac.

Block Diagram. Figure 9-1 is a block diagram of the Pedagac. The Pedagac is a *binary* computer, whose code will be written in octal-coded binary notation. The instruction system is a *one-address* format; the binary point is assumed to be to the left of a quantity word (i.e., all numbers are less than 1). It has a *serial* arithmetic unit, its memory is a drum, and it has a single output unit and a single input unit.

Word Format. A Pedagac word contains 19 bits. The instruction and quantity formats are summarized in Fig. 9-2. The α address contains 12 bits; hence the addressable memory has $2^{12} = 4{,}096$ addresses, from 0000 to 7777 (*octal*).

The Pedagac Coding Manual. Table 9-1 summarizes the instructions that will be available on the Pedagac.

TABLE 9-1. INSTRUCTION LIST FOR THE PEDAGAC

Operation code	Operation name	Symbolic description	Meaning
53	Add	$(\text{acc}) + (\alpha) \rightarrow \text{acc}$	Add (α) to (acc); put result in acc
42	Major multiplication unrounded	$(\text{acc}) \times (\alpha) \rightarrow \text{acc}$	Multiply (α) by (acc); put major product into acc
32	Minor multiplication	$(\text{acc}) \times (\alpha) \rightarrow \text{acc}$	Multiply (α) by (acc); put minor product into acc
41	Division unrounded	$(\text{acc}) \div (\alpha) \rightarrow \text{acc}$	Divide (acc) by (α); put result into acc
54	Subtract	$(\text{acc}) - (\alpha) \rightarrow \text{acc}$	Subtract (α) from (acc); put result into acc
52	Transfer	$(\text{acc}) \rightarrow \alpha$	Transfer (acc) into α; leave (acc) unchanged
43	Conditional jump	$(\text{acc}) < 0$; take α	If (acc) is negative, take (α) as next instruction; otherwise take next instruction in sequence as usual; leave (acc) unchanged
44	Normal jump	Take α	Take (α) as next instruction; leave (acc) unchanged
50	Clear accumulator	$0 \rightarrow \text{acc}$	Make (acc) = 0
71	Logical multiplication	$(\text{acc}) \llcorner\!\cdot\, (\alpha) \rightarrow \text{acc}$	Logically multiply (acc) by (α); put result into acc
72	Logical addition	$(\text{acc}) \llcorner\!+\, (\alpha) \rightarrow \text{acc}$	Logically add (acc) to (α); put result into acc
73	Logical equivalence	$(\text{acc}) \llcorner\!=\, (\alpha) \rightarrow \text{acc}$	Equalize (acc) with (α); put result into acc
74	Logical nonequivalence	$(\text{acc}) \llcorner\!\neq\, (\alpha) \rightarrow \text{acc}$	Unequalize (acc) with (α); put result into acc
70	Circular shift	$(\text{acc}) \overset{\alpha}{\circlearrowright} (\text{acc})$	Circular shift (acc) clockwise by α positions
60	Stop	Stop	Stop computer
00	Read in	Input unit $\rightarrow \alpha$	Read in one word from input unit into α
21	Read out	$(\alpha) \rightarrow$ output unit	Read out (α) to output unit

The Control Panel. The control panel of the Pedagac is very simple. Besides the power on-off button there are only three other manual push buttons: the *start button*, the *stop button*, and the *run button*. Before we can describe the action of these buttons, we must first describe the *idling mode* and the *computing mode* of the computer's operation. The computer is said to be idling when the power is on but no instructions are being executed; it is said to be computing when instructions are being sequenced and executed. There are three ways in which the computer can enter the idling mode: (1) it will assume the idling mode just after the power is turned on; (2) it will be in the idling mode after a *stop* instruction has been executed; (3) it will be in the idling mode after the stop button has been pushed. There are two ways in which the computer can be made to change from the idling mode to the computing mode: (1) push the start button; (2) push the run button.

As we have observed, pushing the stop button puts the computer into the idling mode. The current-address counter will contain the address of the next instruction, and the computer will begin to idle after completing the execution of the present instruction. The stop button has no effect if pushed when the computer is idling. When the run button is pushed, the computer will proceed with the program, first executing the instruction whose address is in the current-address counter. The run button has no effect if pushed when the computer is in the computing mode.

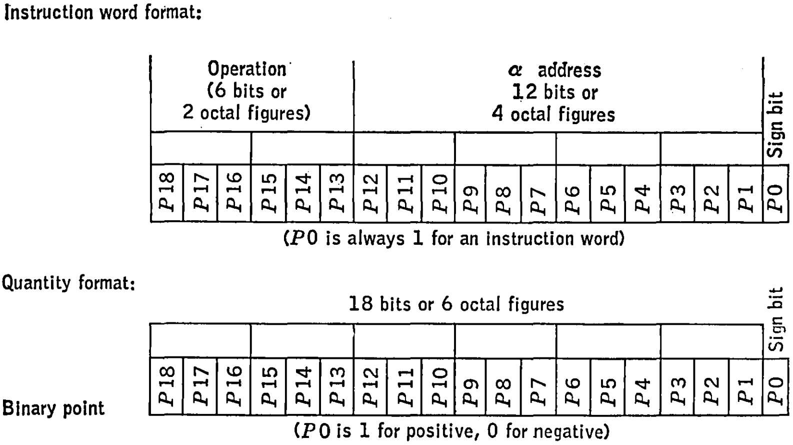

Instruction word format:
Operation (6 bits or 2 octal figures)
α address 12 bits or 4 octal figures
Sign bit
P18 P17 P16 P15 P14 P13 P12 P11 P10 P9 P8 P7 P6 P5 P4 P3 P2 P1 P0
(P0 is always 1 for an instruction word)
Quantity format:
18 bits or 6 octal figures
Sign bit
P18 P17 P16 P15 P14 P13 P12 P11 P10 P9 P8 P7 P6 P5 P4 P3 P2 P1 P0
Binary point
(P0 is 1 for positive, 0 for negative)

FIG. 9-2. The word format of the Pedagac.

The operation of the start button is more complicated. When pushed, first the current-address counter is cleared, second the instruction register is cleared, and third the computer changes to the computing mode and proceeds to execute the *instruction in the instruction register*. What instruction in the instruction register? Namely, 00 0000, which says, "Read one word from the input unit into address 0000 and take the next instruction from the address in the current address counter—namely, address 0000." In other words, *when the start button is pushed, the computer reads one word into the address* 0000 *and then proceeds to execute* (0000) *as an instruction.* The start button has no effect if pushed when the computer is in the computing mode. The start button has two uses: First, as we shall soon see, it is used to read in the *read-in routine.* Second, it is used in interrupting the program to jump out of the normal sequence of instructions; if the programmer desired to jump to a different part of the program, he would push the stop button, place an appropriate *jump* instruction in the input unit, and then push the start button.

A Read-in Subroutine for the Pedagac. The read-in instruction of the Pedagac reads in only one word at a time. Usually in reading in a program or data it is desired to read in many words at one time. Thus the need arises for a subroutine which will read n_0 words into adjacent addresses α_0, $\alpha_0 + 1$, $\alpha_0 + 2$, . . . , $\alpha_0 + (n_0 - 1)$. The read-in subroutine can be initiated by a program that is being computed, e.g., to call in additional data, or else it can be initiated by pushing the start button. Assuming that the read-in subroutine is already in the memory of the Pedagac, Fig. 9-3 represents a flow chart for it. However, for this flow chart to operate properly, the data or words to be read in must be preceded by a word with α_0, the first address to be loaded, and n_0, the number of words to be read in. If the read-in process is to be initiated by the start button, the first word of the data must be a jump to RR1, which is the first address of the read-in routine itself (see Fig. 9-4). The read-in code itself appears in Table 9-2. Here α_iRR represents the read-in instruction that is modified, n_iRR is the address of the tally word n_i, and ONE is the address of the constant 1.

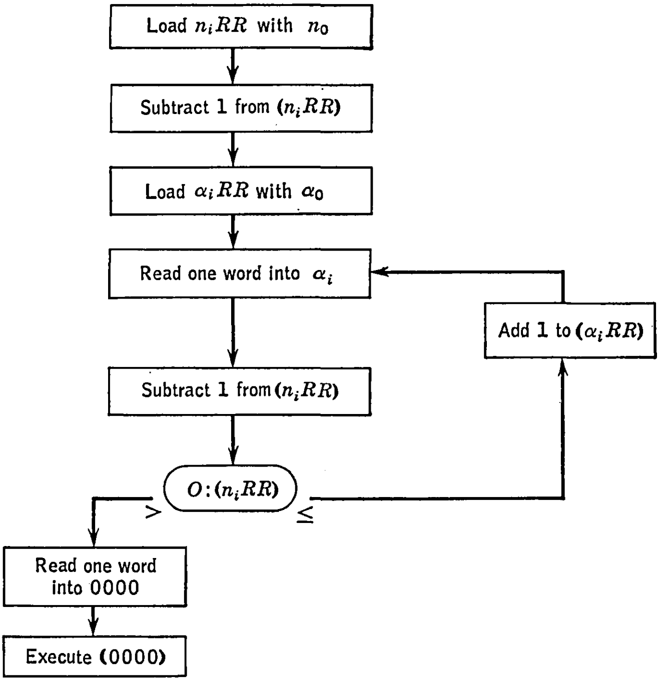

Load n_iRR with n_0
Subtract 1 from (n_iRR)
Load α_iRR with α_0
Read one word into α_i
Add 1 to (α_iRR)
Subtract 1 from (n_iRR)
$O:(n_iRR)$
$>$
$\leq$
Read one word into 0000
Execute (0000)

FIG. 9-3. Flow chart of read-in routine.

Initial Read-in Subroutine for the Pedagac. Suppose that the read-in subroutine was not in the memory. Then a special *initial read-in* sub-

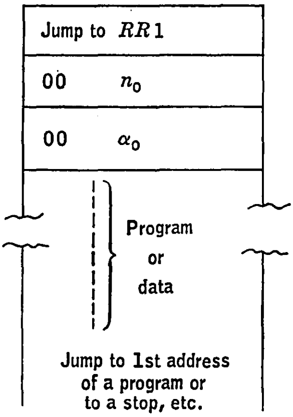


FIG. 9-4. Sequence of words on the input medium to read in a program or data.

routine must be used. The purpose of this subroutine is to load the computer with the normal read-in routine and then give this routine control. We assume that the computer and drum have been cleared and that the start button is pushed. This reads in one word from the input unit into 0000. This word will be an instruction, "Read one word into 0001." The computer then will execute this instruction, which is

TABLE 9-2. THE READ-IN ROUTINE

Address	Operation	α	Remarks
RR1	00	n_iRR	Load n_iRR with n_0
RR2	50	0000	Subtract 1 from n_0
RR3	53	n_iRR	
RR4	54	ONE	
RR5	52	n_iRR	
RR6	00	α_iRR	Load α_iRR with 00α_0
α_iRR	00	0000	Read one word into α_i
RR7	50	0000	Subtract 1 from n_i
RR10	53	n_iRR	
RR11	54	ONE	
RR12	43	RR21	0:n_iRR
RR13	52	n_iRR	
RR14	50	0000	Add 1 to α_i
RR15	53	α_iRR	
RR16	53	ONE	
RR17	52	α_iRR	
RR20	44	α_iRR	Jump to *read-in* instruction
RR21	00	0000	Read one word into 0000
RR22	44	0000	Execute (0000)
ONE	00	0001	
n_iRR			

now in 0000 (see description of the start, stop, and run buttons). This second word to be read in will be an instruction, "Read one word into 0002." The current-address counter will be stepped by 1 in its normal fashion, and the instruction, which is now in 0001, will be executed. A third word will then be read into 0002; this will be a *jump* instruction to jump unconditionally to 0000. Recall that (0000) is the instruction "Read one word into 0001." This fourth word will be the instruction "Read one word into address RR1," where RR1 is supposed to be the first address of the normal read-in routine. The computer will then execute this instruction, reading in the fifth word, which is the first instruction of the normal read-in routine. The computer will now take the next instruction from address 0002, which jumps back to address 0000. The instruction at 0000 will read in a sixth word into 0001; this will be an instruction to read in the second instruction of the normal read-in routine, into address RR1 + 1, of course. This instruction is executed; then the jump to 0000 is executed, and so forth. This process continues until the read-in subroutine has been completely read into the drum memory. Then, when the computer returns to address 0000, it will read into 0001 a *jump* instruction to RR1, that is, the first address of

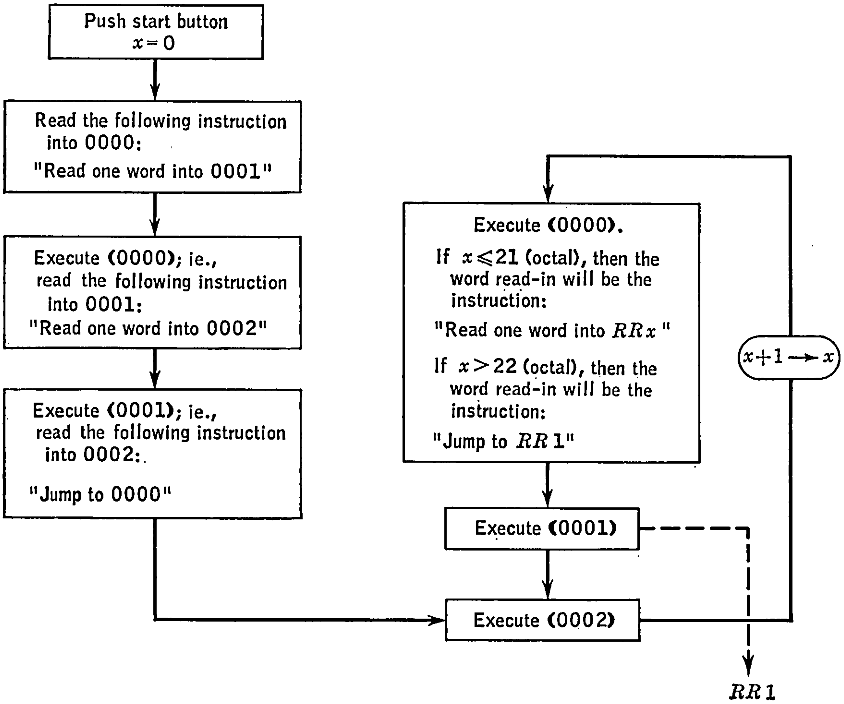

Push start button
$x=0$
Read the following instruction into 0000:
"Read one word into 0001"
Execute (0000); ie., read the following instruction into 0001:
"Read one word into 0002"
Execute (0001); ie., read the following instruction into 0002:
"Jump to 0000"
Execute (0000).
If $x \leqslant 21$ (octal), then the word read-in will be the instruction:
"Read one word into RRx"
If $x > 22$ (octal), then the word read-in will be the instruction:
"Jump to $RR1$"
$x+1 \rightarrow x$
Execute (0001)
Execute (0002)
$RR1$

FIG. 9-5. This is *not* a flow chart of a program. It is a flow diagram illustrating the sequence of operations actually performed during the *initial read-in* routine that reads in the *read-in* routine.

the normal read-in routine (see Fig. 9-5). Note that the sequence of words that will pass through the input unit is as follows:

First word	Read one word into 0001
Second word	Read one word into 0002
Third word	Jump to 0000
Fourth word	Read one word into address RR1
Fifth word	First instruction of normal read-in routine
Sixth word	Read one word into address RR1 + 1
Seventh word	Second instruction of normal read-in routine
Eighth word	Read one word into address RR1 + 2
Ninth word	Third instruction of normal read-in routine
Tenth word, etc	Read one word into address RR1 + 3, etc.
Last word of initial read-in word sequence	Jump to address RR1

Additional Topics

a. Write an automonitor routine for the Pedagac.

b. Compile a library of subroutines for the Pedagac.

c. Write an automatic program for the Pedagac so that it can be programmed in the simplified International Algebraic Language of Sec. 5-9.

PART 3

FOUNDATIONS FOR THE LOGICAL DESIGN OF DIGITAL CIRCUITRY

CHAPTER 10

INTRODUCTION TO BOOLEAN ALGEBRA AND DIGITAL-COMPUTER CIRCUITS

10-1. Introduction

Boolean Algebra. Boolean algebra is an algebraic discipline analogous to the ordinary algebra with which we are all familiar but which follows different rules. We could begin this chapter directly by presenting these new rules, but such an approach would not provide the rationale and background so necessary to a thorough understanding of the methods involved in Boolean algebra. For Boolean algebra was originally developed, not for designing computer circuits, but as the symbolic interpretation of some forms of logical reasoning and as the method for describing relations between classes and sets. A knowledge of such interpretations can aid significantly in the study of Boolean algebra as applied to computer circuits by suggesting analogies and alternative methods for thinking through circuit-design problems.

Therefore in this chapter we introduce Boolean algebra by means of logic, in accordance with its historical development. A discussion of Boolean algebra itself is then given, followed by its interpretation as the algebra of sets. Having dispensed with these preliminaries, we turn to the main purpose of this chapter, the interpretation of Boolean algebra in terms of digital-computer circuits.

Historical Background. It is of historical interest to note that this algebra was first investigated by George Boole (1815–1864) and the theory essentially completed by Ernst Schroeder in his monumental work on the algebra of logic before the turn of this century. Boole, Schroeder, and other logicians of that time had in mind the practical uses of the algebra of logic in the solution of everyday problems. However, they found that dealing with just four to five variables made practical computation unfeasible, requiring the employment of extremely ingenious methods for solution. Hence interest in solving large-scale logical problems of this kind was lost.

It was at that time, too, that Whitehead and Russell, Hilbert, and others realized that logic could be developed to encompass the foundations of mathematics, and after the turn of the century the attention of logicians became focused on the so-called *functional calculus of symbolic logic.* Problems of extreme metamathematical and philosophical importance were considered. Such problems are still today of primary interest to professional logicians. Their methods have necessarily been esoteric and have as yet found little everyday application.

On the other hand, the advent of digital-computer circuitry brought a widespread reawakening of interest in the algebra discovered by Boole. The first use of Boolean algebra in circuit design was made by Shannon† in 1938. Computational difficulties again limited the direct, practical application of Boolean algebra. To help offset some of these difficulties, new digital computational methods have recently been developed, and these are presented in the chapters of this part.

It is to be reemphasized that this chapter will proceed as follows: (1) Logic will be discussed in order to introduce Boolean algebra in a natural way. (2) Boolean algebra itself will be described. (3) The set interpretation of Boolean algebra will be given as an aid to visualizing some of the relationships. And (4) the circuit interpretation will be given, together with methods for going from the circuit diagram to the Boolean function and conversely from the Boolean function to the corresponding circuit diagram.

10-2. Definition of Logic and Propositions

Symbolic Logic. Logic is the science that teaches how thinking or reasoning should proceed; it produces the results of thinking; and it encompasses laws by means of which a judgment can be passed as to the correctness of the products of thought or reasoning. As H. Reichenbach points out, however, "When we call logic *analysis of thought,* the expression should be interpreted so as to leave no doubt that it is not actual thought which we pretend to analyze, it is rather a substitute for thinking processes, their *rational reconstruction,* which constitutes the basis of logical analysis." In other words, once a result of thinking or reasoning is obtained, a chain of thought can be constructed from the initial premises to the final conclusions and an analysis of this rational reconstruction of thinking reveals those rules called laws of logic. Logic presents a means by which thinking or reasoning may be simulated and the results of thinking obtained or checked.

Only when the results of reasoning have been stated in terms of sentences can they be tested or evaluated; logic deals with sentences (e.g., statements) and the rules correlating premise and conclusion sentences. The best method of handling these sentences in logic is a symbolic method, for while simple logical operations can be performed without the help of a symbolic notation, the structure of complicated relations cannot be seen without the use of symbols.

† C. E. Shannon, A Symbolic Analysis of Relay and Switching Circuits, *Trans. AIEE*, vol. 57, pp. 713–723, 1938.

The Proposition. The methods of symbolic logic that are concerned with deduction must lead from true sentences to true sentences. Hence the whole sentence becomes a fundamental unit for logic since only a whole sentence can be *true or false.* In fact, attention in this book will be limited only to whole sentences with this property, and such sentences will be called propositions. *A proposition is a whole sentence about which it is meaningful to say that its content is true or false.* Some authors put this somewhat differently, limiting their consideration only to meaningful sentences. In their system a sentence is meaningful only if it is *verifiable* as true or false. The part of symbolic logic that deals with propositions is called the *propositional calculus.*

Examples of propositions and their truth values are as follows:

Proposition	*Truth value*
a. 3 is an even number	False
b. The grass is green	True
c. Logic is the science that teaches how thinking or reasoning should proceed	True
d. The proposition on the line below is true	False
e. January is a summer month	False

In textbooks dedicated solely to symbolic logic a distinction in symbolism is made between propositions and propositions about propositions. However, since the purpose of this chapter is only to introduce computational methods in the propositional calculus of symbolic logic and not to expound the subject itself, no consistent attempt will be made to separate propositions of these two types. The distinction between propositions of these types is to be made by the reader from the context.

10-3. Definition of Propositional Operations

The Propositional Operations. It is to be reemphasized that this chapter deals with a logic of two truth values, viz., *true* and *false,* and that the primary interest is in determining the truth value of propositions. In considering combinations of propositions, the truth value of the combination is again the primary object.

Propositions can be combined or manipulated by the words *and, or,* and *not* (negation), these words being called the *propositional operations.* The operations *and* and *or* combine two propositions, whereas the operation *not* (or negation) is associated with only one proposition. Given the truth value of propositions, the truth value of their combinations with the propositional operations is defined as follows:

1. The *and* combination of two propositions is true if and only if both propositions are true. If either one or both of them are false, the combination is false. For example, "The grass is green, *and* logic is the science that teaches how thinking or reasoning should proceed" is true, but "The grass is green, *and* 3 is an even number" is false. The *and* combination of two propositions is often called their *conjunction,* or *logical product.*

2. The *or* combination of two propositions is true if and only if either one or both of the propositions are true. The *or* combination of two propositions is false only if both propositions are false. For example, "The grass is green, *or* 3 is an even number" is true; "3 is an even number, *or* January is a summer month" is false. The *or* combination of two propositions is often called their *disjunction*, or their *logical sum.*

3. The negation of a proposition is true if and only if the proposition is false. For example, "3 is *not* an even number" (which is the negation of "3 is an even number") is true.

Propositions may be symbolized by letters: A, B, C, . . . , W, X, Y, ... In a later chapter we shall symbolize propositions by letters with subscripts: A_1, A_2, . . . , B_1, B_2, ... We use the symbol $\cdot$ for the operation *and,* $+$ for *or,* and $^-$ (bar) over the letter to indicate negation. This symbolism is summarized in the following table:

Operational symbol	Interpretation	Terminology
$\bar{A}$	*Not A*	Negation
$A + B$	*A or B*	Disjunction, logical sum
$A \cdot B$	*A and B*	Conjunction, logical product

The use of $+$ for the *or* symbol arises from the parallelism between this logical operation and arithmetic addition. For example,

$$2 \text{ bananas} + 3 \text{ apples} = 5 \text{ pieces of fruit}$$

where pieces of fruit = bananas *or* apples. The parallelism between the logical product and the arithmetical product also holds; for example, 2 ft $\cdot$ 3 lb = 6 ft-lb since the unit of work, namely, foot-pounds, is a logical *and* combination.

Boolean Functions. Combinations of propositions joined by *or, not,* and *and* can be determined true or false and hence are propositions themselves. The uncombined propositions are called *elementary elements* and are usually denoted by letters at the beginning of the alphabet: A, B, ... Propositions combined by the propositional operations are called *combined elements,* or *Boolean functions,* and are usually denoted by letters at the end of the alphabet: X, Y, ... Combined elements are sometimes written in functional form, that is, $f(A,B, \ldots)$.

For example, consider the combined element $\bar{A} + B \cdot C$. Suppose that A is false, B is true, and C is false (that is, A might represent proposition *a* of the examples in Sec. 10-2, B might represent proposition *b*, and C might represent proposition *e*). In such a case $B \cdot C$ is false (since C is false), and $\bar{A}$ is true (since A is false); hence $\bar{A} + B \cdot C$ is true (since $\bar{A}$ is true). Note that $\bar{A} \cdot \bar{B}$ is distinct from $\overline{A \cdot B}$; the former is read (not A) and (not B), while the latter is read [not (A and B)] (the parentheses are for clarity). If A is true and B is false, then $\bar{A} \cdot \bar{B}$ is false and $\overline{A \cdot B}$ is true.

EXERCISES

Suppose that A is false, B is true, and C is false; determine the *truth value* of the following combined elements:

	Answer
(*a*) $\bar{A} \cdot B \cdot \bar{C}$	True
(*b*) $\bar{A} \cdot B + A \cdot \bar{B}$	True
(*c*) $(\bar{A} \cdot B + A \cdot \bar{B}) \cdot C$	False
(*d*) $\overline{A \cdot C}$	True
(*e*) $\overline{A + B}$	False
(*f*) $(A + B) \cdot C + (A + C) \cdot \bar{C} + \bar{A} \cdot \bar{B} \cdot \bar{C}$	False

10-4. Implication, Equivalence, and Tautology

There are two particularly important combined elements, namely, $\bar{A} + B$ and $A \cdot B + \bar{A} \cdot \bar{B}$. Their significance and meaning are discussed here.

Implication. First consider $\bar{A} + B$. This is usually abbreviated $A \rightarrow B$, called "*A implies B*," or "*If A, then B*," by logicians. In a sense this corresponds to the everyday linguistic use of the word *implies*, for if A is true and $A \rightarrow B$ is true, then B must be true; also, if $A \rightarrow B$ is true and B is true, A may or may not be true. For example, consider the proposition "if an airplane is a fighter plane, then it has light armor." Certainly, if the airplane is a fighter, it will have light armor, according to the proposition; but if the airplane has light armor, the proposition does not reveal whether it is a fighter or not, for other types of airplanes than fighters have light armor.

Equivalence. Consider now the important function $A \cdot B + \bar{A} \cdot \bar{B}$, called "*A is equivalent to B*" and abbreviated $A = B$. *If* $A \cdot B + \bar{A} \cdot \bar{B}$ *is true, then A and B have the same truth values*, for then either A and B are both true or they are both false. Also, the meaning of the equals (=) sign in logic parallels its meaning in arithmetic, in which any item can be substituted for an equivalent one. For it can be shown that, if $A = B$ is true, then some combination of the propositions U, V, and A, say, $f(U,V,A)$, is equivalent to that combination with B replacing A, that is, $f(U,V,A) = f(U,V,B)$. In other words, it is always true that $(A = B) \rightarrow [f(U,V,A) = f(U,V,B)]$ in the propositional calculus.

Tautology. This naturally brings up the topic of tautology. A *tautology* is a combined element that is always true, independent of the truth or falseness of its component propositions. For example, $A + \bar{A}$, $(\overline{A \cdot B}) + A$, $(\overline{A \cdot B}) + (A + \bar{B}) \cdot (\bar{A} + B)$, as well as

$$(A = B) \rightarrow [f(U,V,A) = f(U,V,B)]$$

are all tautologies, for they are true no matter what the truth values of A, B, U, and V are (or what the truth value of the combined element f is). Therefore there is no logical reason to distinguish between these combined elements, and any one of them is called the universally true element, or tautology, and denoted by I. The negation of I, namely, $\bar{I}$, is the universally false element, denoted by 0. A consequence of these concepts is that $0 \rightarrow X$ and $X \rightarrow I$ are always true; i.e., the universally false element implies everything, while everything implies the tautology.

To review, then, there are two meanings to truth. One is "tautologically true," and the other is "true because it is a physical fact or supposition." "The grass is green" is true in the latter sense; "Today is Tuesday, or today is not Tuesday" is tautologically true. Logicians often use a special symbol to distinguish these meanings, but for our limited purposes this would be an unnecessary complication.

EXERCISES

(*a*) Note that $A \rightarrow B$ is true if both A is false and B is false. Can you think of examples of this situation that might occur in real life?

(*b*) Consider the sentence "$A + \bar{A}$ is a tautology." Is this tautologically true, or is it factually true?

10-5. Truth Tables

The truth values of a combined element depend on the truth values of its component elementary elements, as dictated by the definitions of the propositional operations. A table can be made presenting all possible combinations of truth values; we call this a truth table. Consider, for example, $A + B$. The columns under A and B represent all possible truth-value combinations of A and B, and the column under $A + B$ represents the corresponding truth value of $A + B$, where T stands for *true* and F for *false*.

A	B	$A + B$
T	T	T
F	T	T
T	F	T
F	F	F

Similarly the truth tables for $\bar{A}$ and $A \cdot B$ are:

A	$\bar{A}$
T	F
F	T

A	B	$A \cdot B$
T	T	T
F	T	F
T	F	F
F	F	F

These truth tables follow from the definitions of the propositional operations and, conversely, could have been taken to be the definitions of these operations. With these basic truth tables in mind the truth table for any combined element can be constructed.

For example, recalling that $A \rightarrow B$ stands for $\bar{A} + B$ and that $A = B$ stands for $A \cdot B + \bar{A} \cdot \bar{B}$, the following table can be formed:

A	B	$A \rightarrow B$	$A = B$
T	T	T	T
F	T	T	F
T	F	F	F
F	F	T	T

Note that $A = B$ is true only when A and B have the same truth values. Similarly truth tables can be made for combinations of any number of elementary elements. For example, for three elementary elements A, B, and C, the truth table for the combined element $\bar{A} + B \cdot C$ becomes:

A	B	C	$\bar{A} + B \cdot C$
T	T	T	T
F	T	T	T
T	F	T	F
F	F	T	T
T	T	F	F
F	T	F	T
T	F	F	F
F	F	F	T

EXERCISES

Find the truth tables for:

(*a*) $(\overline{A \cdot B}) + A$. (*b*) $\bar{A} \cdot B \cdot \bar{C}$. (*c*) $(\bar{A} \cdot B + A \cdot \bar{B}) \cdot C$.

(*d*) $\overline{A \cdot C}$. (*e*) $\overline{A + B}$.

Solution

A	B	C	$(\overline{A \cdot B}) + A$	$\bar{A} \cdot B \cdot \bar{C}$	$(\bar{A} \cdot B + A \cdot \bar{B}) \cdot C$	$\overline{A \cdot C}$	$\overline{A + B}$
T	T	T	T	F	F	F	F
F	T	T	T	F	T	T	F
T	F	T	T	F	T	F	F
F	F	T	T	F	F	T	T
T	T	F	T	F	F	T	F
F	T	F	T	T	F	T	F
T	F	F	T	F	F	T	F
F	F	F	T	F	F	T	T

Note that $(\overline{A \cdot B}) + A$ is indeed a tautology since it is true for every combination of truth values of the elementary elements of which it is a combination.

(*f*) Show that $A + \bar{A}$ and $\overline{A \cdot B} + (A + \bar{B}) \cdot (\bar{A} + B)$ are tautologies.

10-6. Boolean Algebra

Equations of Boolean Algebra. With the foregoing interpretation the algebraic properties of the logical operations can be more clearly understood. The following equations characterize a Boolean algebra. It is to be noted that these equations are quite different from the familiar rules of ordinary algebra. The subject that comprises algebraic manipulations according to the equations given below is therefore called *Boolean* algebra to distinguish it from ordinary algebra. Some of the equations of Boolean algebra are particularly important, and mathematicians have given them special names as shown:

	Name of equation
1. $A + A = A$	Absorption rule for +
2. $A \cdot A = A$	Absorption rule for ·
3. $A + B = B + A$	Commutative rule for +
4. $A \cdot B = B \cdot A$	Commutative rule for ·
5. $(A + B) + C = A + (B + C) = A + B + C$	Associative rule for +
6. $(A \cdot B) \cdot C = A \cdot (B \cdot C) = A \cdot B \cdot C$	Associative rule for ·
7. $A \cdot (B + C) = A \cdot B + A \cdot C$	Distributive rule of · over +
8. $A + B \cdot C = (A + B) \cdot (A + C)$	Distributive rule of + over ·
9. $\overline{A \cdot B} = \bar{A} + \bar{B}$	De Morgan's rule for ·
10. $\overline{A + B} = \bar{A} \cdot \bar{B}$	De Morgan's rule for +
11. $A + I = I$	
12. $A \cdot I = A$	
13. $0 + A = A$	
14. $0 \cdot A = 0$	
15. $\bar{A} + A = I$	
16. $A \cdot \bar{A} = 0$	
17. $A + A \cdot B = A$	
18. $A + \bar{A} \cdot B = A + B$	
19. $A \cdot B + B \cdot C + C \cdot A = (A + B) \cdot (B + C) \cdot (C + A)$	
20. $A \cdot \bar{B} + A \cdot C + B \cdot C = A \cdot \bar{B} + B \cdot C$	
21. $\bar{\bar{A}} = A$	

These equations are to be interpreted as tautologies. The equals sign means that, for any given combination of truth values for A, B, and C, the combined element on the right-hand side of the equation has the same truth value as the combined element on the left-hand side. Hence these equations can be checked by writing the truth tables for the right-hand and left-hand combined elements.

Algebraic Proofs. Some of these equations can be "proved" algebraically by using others. For example, let us prove (20) by using (12), (15), (7), (6), (4), (3), and (17).†

† Note how difficult and rather "tricky" it is to prove (20). With experience and practice some facility can be gained in performing such manipulations. However, the

$$\begin{aligned}
&A \cdot \bar{B} + A \cdot C + B \cdot C \\
&\quad = A \cdot \bar{B} + A \cdot C \cdot I + B \cdot C && \text{by (12)} \\
&\quad = A \cdot \bar{B} + A \cdot C \cdot (\bar{B} + B) + B \cdot C && \text{by (15)} \\
&\quad = A \cdot \bar{B} + (A \cdot C) \cdot \bar{B} + (A \cdot C) \cdot B + B \cdot C && \text{by (7)} \\
&\quad = A \cdot \bar{B} + (A \cdot \bar{B}) \cdot C + B \cdot C + (B \cdot C) \cdot A && \text{by (6), (4), and (3)} \\
&\quad = A \cdot \bar{B} + B \cdot C && \text{by (17) applied twice}
\end{aligned}$$

New equations may also be proved by using those listed above. To show that $I \cdot A = A$, we have $I \cdot A = A \cdot I = A$ by (4) and (12).

EXERCISES

(*a*) Using $I \cdot A = A$, (8), and (15), prove (18).

Solution

$$\begin{aligned}
A + \bar{A} \cdot B &= (A + \bar{A}) \cdot (A + B) = I \cdot (A + B) \\
&= A + B
\end{aligned}$$

as desired.

(*b*) What other equations listed above would be used to prove (19)?

Solution. (1), (2), (3), (6), (7), (8). [HINT: Start with the right-hand side of (19) and apply (8), using (1), (2), (3), (6), and (7) to simplify the terms.]

Alternative Algebraic Proofs. There is another way of proving equations, namely, by making use of (15) and (11), for we want to show that an equation is a tautology (i.e., the meaning of "proving equations"). If we can show that the equation itself is equivalent to I (in truth value), then we have shown that it is a tautology, since I is the universally true element (i.e., always true for all truth-value combinations of the elementary elements).

For example, consider the proof of (17): $A + A \cdot B = A$ is an abbreviation for

$$(A + A \cdot B) \cdot A + (\overline{A + A \cdot B}) \cdot \bar{A}$$

but, on using (2), (3), (4), (6), (7), (9), (10), (11), and (15), we find that this expands to

$$\begin{aligned}
A + A \cdot B + \bar{A} \cdot (\bar{A} + \bar{B}) \cdot \bar{A} &= A + A \cdot B + \bar{A} + \bar{A} \cdot \bar{B} \\
&= A + \bar{A} + A \cdot B + \bar{A} \cdot \bar{B} \\
&= I + A \cdot B + \bar{A} \cdot \bar{B} \qquad \text{by (15)} \\
&= I \qquad \text{by (11), to complete the proof}
\end{aligned}$$

As another example, we can show that $A \cdot B \rightarrow A$ is a tautology, for $A \cdot B \rightarrow A$ is an abbreviation for $\overline{A \cdot B} + A$, which becomes, by (9), $\bar{A} + \bar{B} + A$. But $\bar{A} + \bar{B} + A = \bar{A} + A + \bar{B}$ [by (3) and (5)] $= I + \bar{B}$ [by (15)] $= I$ [by (11)], to complete the proof.

student need not dwell on gaining such facility, for in the next chapter a systematic and straightforward method will be given for doing such problems.

EXERCISES

Are the following tautologies?

	Answer
(*a*) $\bar{A} \cdot B \rightarrow A$	No
(*b*) $A \rightarrow A + B$	Yes
(*c*) $(A \rightarrow B) = (\bar{B} \rightarrow \bar{A})$	Yes
(*d*) $(A \cdot B \cdot C \rightarrow D) = (\bar{D} \cdot B \cdot C \rightarrow \bar{A})$	Yes

Another Method for Algebraic Proof. There is still another way for proving equations, i.e., to make both sides of the equals sign look identical. For example, let us prove that $(A \rightarrow B) = (\bar{B} \rightarrow \bar{A})$. From the definition of → we have $\bar{A} + B = B + \bar{A}$ or $\bar{A} + B = \bar{A} + B$, by (21) and (3). As another example, let us show that

$$A \rightarrow (B \rightarrow C) = A \cdot B \rightarrow C$$

By the definition of → the left-hand side becomes $\bar{A} + \bar{B} + C$. Similarly the right-hand side becomes $\overline{A \cdot B} + C$; by using (9) this becomes $\bar{A} + \bar{B} + C$, as desired.

10-7. Boolean Algebra as the Algebra of Sets

Sets. We have introduced Boolean algebra as the algebra of propositional logic. However, Boolean algebra can also be interpreted as the algebra of sets, or classes. A *class* is a collection of objects all of which have a certain property; such a well-defined collection of objects is often called a *set*. For example, the collection of all automobiles manufactured in the year 1956 and still in operating condition is a set. Or the collection of people who registered at the national meeting of the Institute of Radio Engineers in 1960 forms a set. However, most often for the purposes of mathematics a set is visualized as the area inside a closed curve drawn on a piece of paper: the set is roughly interpreted as the collection of points that lie inside the closed curve (see Fig. 10-1). The objects that belong to the set are called *elements* of the set. For example, the point p is an element of the set (the interior of the circle) of Fig. 10-1. Sets are often denoted by letters; for example, we call the set in Fig. 10-1 the *set A*.

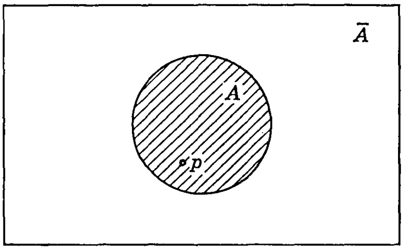


FIG. 10-1. A, $\bar{A}$.

Just as the prime purpose of propositional logic was to determine the truth value of a proposition, so the prime purpose of the algebra of sets is to determine *whether or not a point is in a set.* In fact the only difference between the propositional calculus of logic and the algebra of sets is that the algebra of sets deals with a very restricted kind of propositions. These are propositions of the form "*The point p is in the set A*" and combinations of such propositions with *and*, *or*, and *not.* Hence the crosshatched area of Fig. 10-1 represents all those points for which the proposition "The point p is in the set A" is true. Let us symbolize this proposition by A. Then $\bar{A}$ is true for all points p not lying in the crosshatched area.

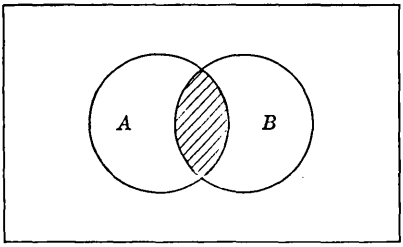


FIG. 10-2. $A \cdot B$.

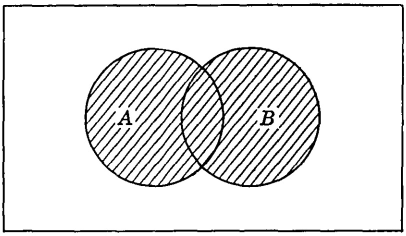


FIG. 10-3. $A + B$.

Intersection and Union. All the points for which $A \cdot B$ is true (i.e., for which "the point p is in the set A, *and* the point p is in the set B" is true) are represented by the crosshatched area of Fig. 10-2. All the points for which $A + B$ is true (i.e., for which "the point p is in the set A, *or* the point p is in the set B" is true) are represented by the crosshatched area in Fig. 10-3. The crosshatched area in Fig. 10-2 is called the *intersection* of the set A with the set B, that is, the set of points that lie in both A *and* B. The crosshatched area in Fig. 10-3 is called the *union* of the set A with the set B, that is, the set of points that lie in either A *or* B or both.

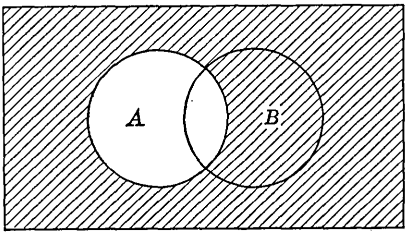


FIG. 10-4. $\bar{A}$.

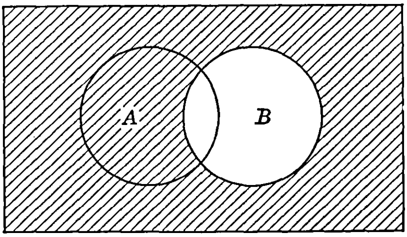


FIG. 10-5. $\bar{B}$.

Now, as we have seen, Boolean algebra holds for propositions, and hence it must also hold for our set interpretation of propositions. Hence we can visualize some of the relations of Boolean algebra by means of the set diagrams. For example, consider the relation $\overline{A \cdot B} = \bar{A} + \bar{B}$.

We shall show that $\overline{A \cdot B}$ and $\bar{A} + \bar{B}$ represent the same area. Figure 10-4 gives the area for $\bar{A}$, Fig. 10-5 for $\bar{B}$, and Fig. 10-6 for $\bar{A} + \bar{B}$. But, observing Fig. 10-2 for $A \cdot B$, we see that the area for $\bar{A} + \bar{B}$ is the same as that for $\overline{A \cdot B}$.

Just as in the propositional logic, given the truth value of the elementary elements, the truth value of a combined element can be determined, so, given the location of a point p with respect to the *elementary sets* A, B, . . . , we can determine if it is in the set described by a combined element. For example, if a point p is in set A, not in set B, not in set C, then is it in the set for which $(\bar{A} + \bar{B}) \cdot \bar{C}$ is true? The solution is given in Fig. 10-7. The crosshatched areas are all points for which $(\bar{A} + \bar{B}) \cdot \bar{C}$ is true, and the point p is in set A, but not in set B or set C. Hence p is in the set $(\bar{A} + \bar{B}) \cdot \bar{C}$.

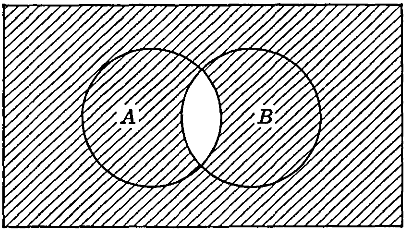


FIG. 10-6. $\bar{A} + \bar{B}$.

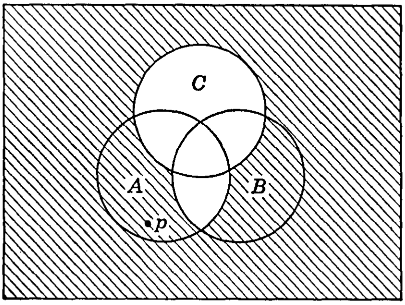


FIG. 10-7. $(\bar{A} + \bar{B}) \cdot \bar{C}$.

EXERCISES

(*a*) Show by set diagrams that $A + B \cdot C = (A + B) \cdot (A + C)$.

If a point is in set A, not in set B, and not in set C, then is it in the following sets, i.e., in the sets for which the following combined elements are true?

	Answer
(*b*) $(A + B) \cdot \bar{C}$	Yes
(*c*) $A + B \cdot \bar{C}$	Yes
(*d*) $A \cdot B \cdot \bar{C}$	No
(*e*) $A \cdot (B + \bar{C})$	Yes
(*f*) $\bar{B} + A \cdot C$	Yes

(*g*) Draw the set diagram for $A \rightarrow B$.
(*h*) Draw the set diagram for $A = B$.

10-8. Digital-computer Circuits

In this section and the following sections we shall begin to see why the study of Boolean algebra is important for digital-computer circuit design. However, first we must make clear just what a digital-computer circuit is (see Fig. 10-8).

The Condition-function table. Consider a digital-computer-circuit black box with, say, two input wires called wire A and wire B and one output wire called wire C. Suppose that only one of two voltages may appear on wire A, to be specific, say a 0-volt and a 1-volt signal; the same holds for wire B. Hence a table can be made describing all possible input conditions:

	Possible input conditions			
	0	1	2	3
Wire A	0	1	0	1
Wire B	0	0	1	1
Wire C	0	0	0	1

Let condition 3 be the case when there is a 1-volt signal on both wires A and B; condition 2 be the case when there is a 0-volt signal on wire A, 1 volt on wire B; condition 1 when there is 1 volt on wire A, 0 volts on wire B; condition 0 when there are 0-volt signals on both wires A and B. Then the function of the black box will be completely specified if we tell what voltage will appear on wire C for each of the four input conditions on wires A and B. For instance, suppose that 1 volt appears on wire C only when 1 volt appears on both wires *A and B* and that 0 volts appear on wire C otherwise. Then the table of input conditions can be completed for the output wire C as shown, and then the function of the black box is completely specified by the condition-function table. Such a black box is a type of digital-computer circuit.

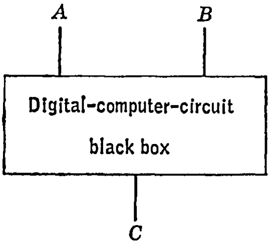


FIG. 10-8. Digital-computer black box.

Hence we define a (binary) digital-computer circuit as a circuit black box the input wires of which are limited to *two* voltage levels, the output wires of which are limited to *two* voltage levels, and such that the voltage levels of the output wires depend on the present voltage levels of the input wires.†

† Circuits that seem to depend on the past history of the input-voltage levels as well can always be interpreted in terms of present input-voltage levels provided that the number of "inputs" is increased; this will be discussed in a later chapter.

EXERCISE

(*a*) Make a condition-function table for a circuit with three input wires A, B, C and one output wire D such that D will have a unit voltage signal whenever A has a unit voltage signal and B has a unit voltage signal, or else when C has a zero voltage signal; otherwise D has a zero voltage signal.

Solution

	Possible input conditions							
	0	1	2	3	4	5	6	7
Wire A.........	0	1	0	1	0	1	0	1
Wire B....	0	0	1	1	0	0	1	1
Wire C.........	0	0	0	0	1	1	1	1
Wire D.........	1	1	1	1	0	0	0	1

10-9. Boolean Algebra as the Algebra of Digital-computer Circuits

We have introduced Boolean algebra as the algebra of propositional logic and have shown that it is in addition the algebra of sets. Boolean algebra can also be interpreted as the algebra of digital-computer circuits. Just as the prime purpose of propositional logic was to determine the truth value of a proposition and that of the algebra of sets was to determine whether or not a point is in a set, the prime purpose of the algebra of digital-computer circuits is to determine the signal-voltage level on a wire (e.g., either a 0-volt or a 1-volt signal). In fact, just as in the algebra of sets, the algebra of digital-computer circuits deals with a restricted kind of proposition. These are propositions of the form "The wire X has a 1-volt signal" and combinations of such propositions with *and*, *or*, and *not*.

Suppose that we symbolize by X the proposition "The wire X has a 1-volt signal." Then we can symbolize, for example, the statement in Exercise *a* of Sec. 10-8 that describes when wire D will have a 1-volt signal as follows: $A \cdot B + \bar{C}$. In fact, when that combined element is true, then D is to be true; that is, $D = A \cdot B + \bar{C}$. Hence we can symbolize the conditions that describe the output of a computer circuit in terms of its input by means of Boolean algebra.

The condition-function table now becomes analogous to the truth tables of propositional logic when 0 corresponds to false and 1 corresponds to true. In fact, given the Boolean algebraic conditions for D to have a 1-volt signal, a function table can be constructed directly in a manner similar to the construction of truth tables.

For example, suppose that the conditions for output wire D of a computer circuit were given in terms of the voltage signals on input wires A, B, and C by

$$D = \bar{A} + B \cdot C$$

Then the condition-function table would be:

	Possible input conditions							
	0	1	2	3	4	5	6	7
Wire A.........	0	1	0	1	0	1	0	1
Wire B.........	0	0	1	1	0	0	1	1
Wire C.........	0	0	0	0	1	1	1	1
Wire D.........	1	0	1	0	1	0	1	1

For example, if A is true (i.e., if "Wire A has a 1-volt signal" is true) and if B is true and if C is true, then $\bar{A}$ is false, but $B \cdot C$ is true, whence $\bar{A} + B \cdot C$ is true and therefore D is true (i.e., "Wire D has a 1-volt signal" is true). Hence for condition 7 we put a unit in the row for wire D, and so forth.

Immediately we see that the description of the same black-box computer circuit can be stated in more than one way in terms of Boolean algebra. For example, if an output wire D is given in terms of the input wires A, B, C as $D = \bar{A} + B \cdot C$, then, since

$$\bar{A} + B \cdot C = (\bar{A} + B) \cdot (\bar{A} + C)$$

we could equivalently write $D = (\bar{A} + B) \cdot (\bar{A} + C)$. On the other hand both $\bar{A} + B \cdot C$ and $(\bar{A} + B) \cdot (\bar{A} + C)$ have the same truth table and consequently the same condition-function table.

10-10. From Circuit Diagram to Boolean Function

The Gates. We shall presently show that any such digital-computer-circuit black box can be constructed out of three circuit elements, called, respectively, an *and gate*, an *or gate*, and an *inverter*.† However, the three kinds of circuit elements must first be defined. They are all digital-computer circuits and can therefore be defined by a condition-function table.

The *and* gate has two inputs A and B and one output X, which takes the voltage values given by the following table:

	Possible input conditions			
	0	1	2	3
Wire A..........	0	1	0	1
Wire B..........	0	0	1	1
Wire X..........	0	0	0	1

† Consideration of "flip-flops" will be deferred until a later chapter.

In words, wire X will have a 1-volt signal if and only if the wires A *and* B each have a 1-volt signal. One possible method for constructing such a circuit is illustrated in Fig. 10-9. The arrows indicate the direction of flow of *electron current;* T_1 and T_2 are diodes, and R is a resistor. When either wire A or wire B has a 0-volt signal, current will flow through the resistor until X is at 0 volts; when all of wires A and B are at +1 volt, then no current will flow through R and wire X will be at +1 volt.†

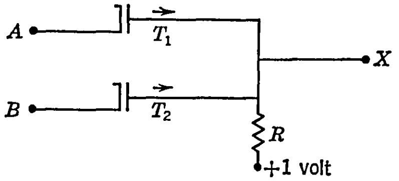


FIG. 10-9. Diode *and* circuit.

The *or* gate has two inputs, A, B, and one output, X, which takes the voltage values given by the following table:

	Possible input conditions			
	0	1	2	3
Wire A..........	0	1	0	1
Wire B..........	0	0	1	1
Wire X..........	0	1	1	1

In words, wire X will have a 1-volt signal if and only if at least one of wire A or B has a 1-volt signal. The schematic for such a circuit is illustrated in Fig. 10-10. Whenever any one of A or B or C becomes +1 volt, electron current will flow until X is at +1 volt.

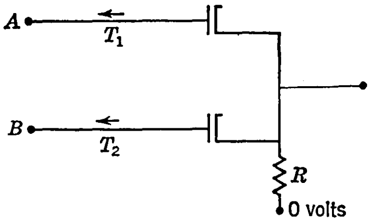


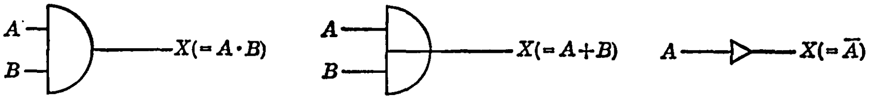
FIG. 10-10. Diode *or* circuit.

† Detailed description of *and* gates, *or* gates, and *inverters* is given in Part 5. We merely indicate how these might be made to enable the student to obtain a more concrete picture at this time.

An *inverter* has one input wire A and one output wire X. Wire X will have a 1-volt signal if wire A is at 0 volts, and wire X will have a 0-volt signal if wire A is at 1 volt. The table becomes:

	Possible input conditions	
	0	1
Wire A..........	0	1
Wire X..........	1	0

Symbolized Gates. Rather than draw the schematic for an *and* gate, *or* gate, or *inverter*, we use standardized symbols to represent them, as shown in Fig. 10-11.



And gate

Or gate

Inverter

FIG. 10-11. Symbolized gates.

We can generalize the *and* gate and *or* gate symbols for more than two inputs as shown in Figs. 10-12 and 10-13.

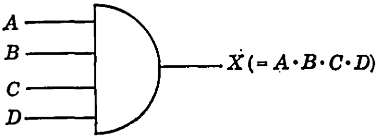


FIG. 10-12. Symbol for *and* gate with four inputs.

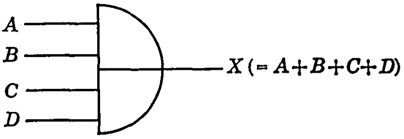


FIG. 10-13. Symbol for *or* gate with four inputs.

From the Circuit Diagram to the Boolean Function. Complicated digital-computer circuits can be built up by connecting the output of such gates to the input terminals of other gates. For example, consider Fig. 10-14. The outputs X and Y of the *and* gates are connected to the inputs of the *or* gate. Hence the output Z of the *or* gate then depends on the original inputs A, B, C, and D of the *and* gates. What is the precise dependence? Note that $X = A \cdot B$ and $Y = C \cdot D$ whereas $Z = X + Y$. Hence $Z = A \cdot B + C \cdot D$. In other words, the circuit of Fig. 10-14 can be thought of as a black box with input A, B, C, and D and output Z such that $Z = A \cdot B + C \cdot D$.

Hence, given a digital-computer-circuit diagram, the Boolean expression for its output can now readily be derived. For example, consider the circuit of Fig. 10-15. The problem is to write Z in terms of A and B, as given by the circuit diagram. This is done as follows: $Z = X \cdot Y$, $X = A + B$, $Y = \bar{C}$; therefore $Z = (A + B) \cdot \bar{C}$. For another example, consider the circuit of Fig. 10-16. Here $Z = X + Y$, $X = A \cdot B$, $Y = U \cdot V$, $U = \bar{A}$, and $V = \bar{B}$. Putting these together, we find that $Z = A \cdot B + \bar{A} \cdot \bar{B}$.

Summary. The procedure for writing the output Boolean functions in terms of the input elements (wires) for a given circuit diagram is as follows: First, assign letters to all input wires, all internal connecting wires, and all output wires. Second, proceed to write down the successive functions for each gate or inverter. Third, substitute in the equations until the outputs are given in terms of the inputs.

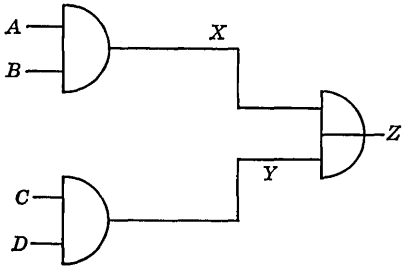


FIG. 10-14. $Z = X + Y = A \cdot B + C \cdot D$.

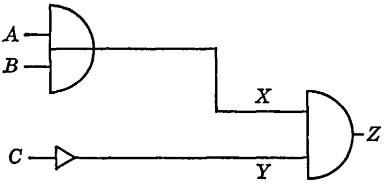


FIG. 10-15. $Z = X \cdot Y = (A + B) \cdot \bar{C}$.

The relations between the propositional calculus of symbolic logic and the circuit interpretation of Boolean algebra can now be reviewed. The symbol A corresponds to the proposition "Wire A has a 1-volt signal." Hence consider the propositional interpretation of the answer to the second example: $Z = A \cdot B + \bar{A} \cdot \bar{B}$. This says that "The output wire Z has a 1-volt signal" has the same truth values as "Either both input wires A and B have a 1-volt signal, or neither A nor B has a 1-volt signal." In other words, wire Z will have a 1-volt signal whenever either both A and B have 1 volt or neither A nor B has a 1-volt signal; otherwise wire Z will have a 0-volt signal.

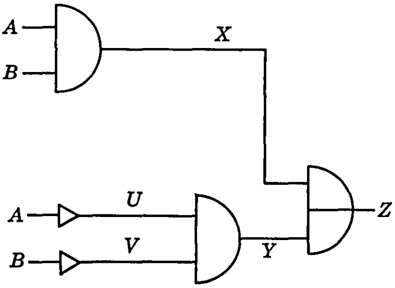


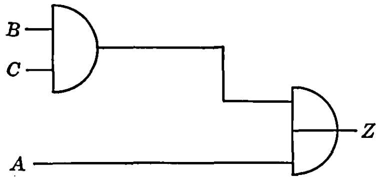
FIG. 10-16. $Z = X + Y = A \cdot B + U \cdot V = A \cdot B + \bar{A} \cdot \bar{B}$.

EXERCISES

Determine the Boolean function (i.e., the combined element) of the inputs A, B, C, and D that corresponds to the outputs Y and Z of the following diagrams:

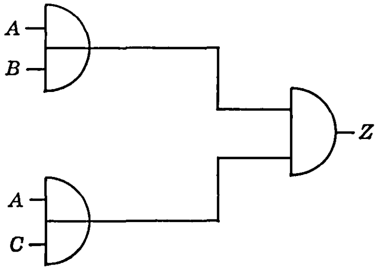
Solution

(a)



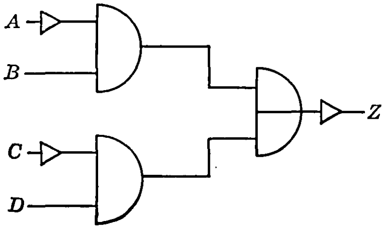
$Z = A + B \cdot C$

(b)



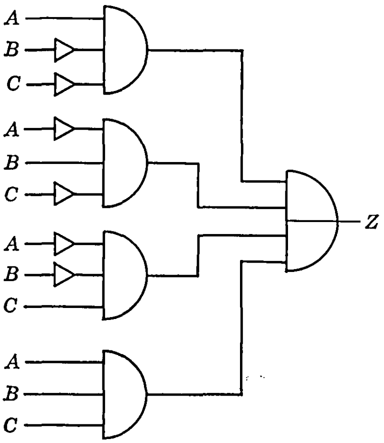
$Z = (A + B) \cdot (A + C)$

(c)

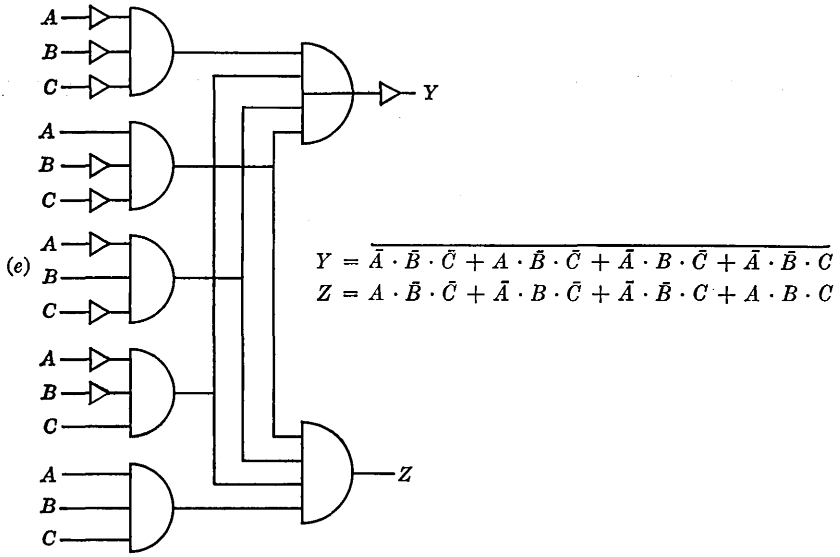


$Z = \overline{\bar{A} \cdot B + \bar{C} \cdot D}$

(d)



$$Z = A \cdot \bar{B} \cdot \bar{C} + \bar{A} \cdot B \cdot \bar{C} + \bar{A} \cdot \bar{B} \cdot C + A \cdot B \cdot C$$



(e)

$$Y = \overline{\bar{A} \cdot \bar{B} \cdot \bar{C} + A \cdot \bar{B} \cdot \bar{C} + \bar{A} \cdot B \cdot \bar{C} + \bar{A} \cdot \bar{B} \cdot C}$$
$$Z = A \cdot \bar{B} \cdot \bar{C} + \bar{A} \cdot B \cdot \bar{C} + \bar{A} \cdot \bar{B} \cdot C + A \cdot B \cdot C$$

10-11. From Boolean Function to Circuit Diagram

In this section is illustrated how to proceed in the direction opposite to that of Sec. 10-10: given a desired output Boolean function in terms of the inputs, draw a circuit design. Let Z be the ouput and A and B the inputs. Suppose that it is desired to design a circuit so that

$$Z = A \cdot B + \bar{A} \cdot \bar{B}$$

The right-hand side is the sum of two combined elements; so let

$$Z = X + Y$$

where $X = A \cdot B$ and $Y = \bar{A} \cdot \bar{B}$. Now Y is the product of two combined elements: let $Y = U \cdot V$, where $U = \bar{A}$, $V = \bar{B}$. Then we draw

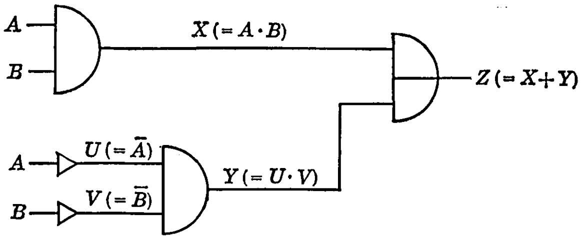


FIG. 10-17. $Z = X + Y = A \cdot B + U \cdot V = A \cdot B + \bar{A} \cdot \bar{B}$.

Z as the result of $X + Y$, X as $A \cdot B$, Y as $U \cdot V$, U as $\bar{A}$, and V as $\bar{B}$ (see Fig. 10-17). For another example, consider

$$Z = A \cdot \bar{B} \cdot \bar{C} + \bar{A} \cdot B \cdot \bar{C} + A \cdot B \cdot \bar{C}$$

Let $Z = U + V + W$, where $U = A \cdot \bar{B} \cdot \bar{C}$, $V = \bar{A} \cdot B \cdot \bar{C}$, and $W = A \cdot B \cdot \bar{C}$. Hence the diagram is as in Fig. 10-18. This example is interesting for the following reason: Note that

$$\begin{aligned} A \cdot \bar{B} \cdot \bar{C} + \bar{A} \cdot B \cdot \bar{C} + A \cdot B \cdot \bar{C} &= (A \cdot \bar{B} + \bar{A} \cdot B + A \cdot B) \cdot \bar{C} \\ &= (A \cdot (\bar{B} + B) + \bar{A} \cdot B) \cdot \bar{C} = (A + \bar{A} \cdot B) \cdot \bar{C} = (A + B) \cdot \bar{C} \end{aligned}$$

or $A \cdot \bar{B} \cdot \bar{C} + \bar{A} \cdot B \cdot \bar{C} + A \cdot B \cdot \bar{C} = (A + B) \cdot \bar{C}$, which is just the output expression for the second example of the previous section (Sec. 10-10). In other words, the above circuit diagram gives the same output as the circuit diagram of the second example of the previous section.

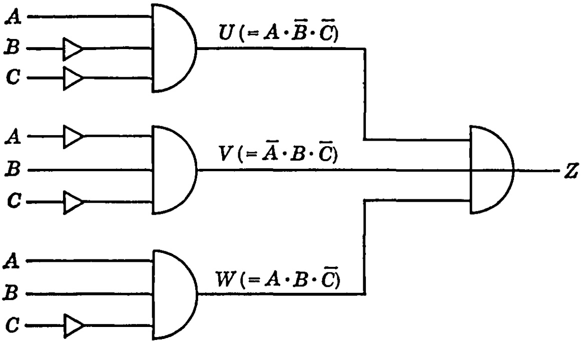


FIG. 10-18. $Z = U + V + W = A \cdot \bar{B} \cdot \bar{C} + \bar{A} \cdot B \cdot \bar{C} + A \cdot B \cdot \bar{C}$.

Hence we see that the design of a digital-computer-circuit black box is *not* unique and, in general, that there are many ways of arranging the *and* gates, *or* gates, and *inverters* to obtain equivalent results. How to choose the best design is discussed in the next chapter.

EXERCISES

If Z is the output and A, B, C, and D are inputs, draw the circuit diagrams for computer circuits such that:

(*a*) $Z = A \cdot \bar{B} + \bar{A} \cdot B$.

(*b*) $Z = (A \cdot \bar{B} + \bar{A} \cdot B) \cdot \bar{C} + (A \cdot B + \bar{A} \cdot \bar{B}) \cdot C$.

(*c*) $Z = (A + \bar{B}) \cdot (C + \bar{D})$.

(*d*) $Z = (A + \bar{B} \cdot C) \cdot (\bar{A} + B \cdot \bar{C}) + D$.

10-12. Additional Topics

a. Axiomatic Development of Boolean Algebra, and References. Boolean algebra, or the propositional calculus of symbolic logic, can be developed from several different sets of postulates. As an example, consider a set of postulates defined in terms of the following "fundamental concepts": a set S of elements ($a,b,c, \ldots$); $a + b$, the result of the binary operation $+$; and $\bar{a}$, the result of the operation $\bar{\ }$. The postulates are:

1. If a and b are elements of S, then $a + b$ is an element of S.
2. If a is an element of S, then $\bar{a}$ is an element of S.
3. $a + b = b + a$.

4. $(a + b) + c = a + (b + c)$.
5. $\overline{(\bar{a} + \bar{b})} + \overline{(\bar{a} + b)} = a$.

The equals sign $(=)$ is interpreted as an equivalence relation (see Sec. 7-8), and in addition the following *substitution rule* is assumed to hold: If $x = y$, then $f(x,a,b,c, \ldots) = f(y,a,b,c, \ldots)$. From these five postulates and the properties of $=$, all the equations of Boolean algebra can be proved (see Sec. 10-6). Evidently $+$ can be interpreted as the logical sum; the logical product is *defined* as $a \cdot b = \overline{\bar{a} + \bar{b}}$. The tautology I is defined by $I = a + \bar{a}$; 0 is defined by $0 = a \cdot \bar{a}$. For example, let us prove that $a \cdot b = b \cdot a$. By the definition $a \cdot b = \overline{(\bar{a} + \bar{b})}$; by (3), $\overline{(\bar{a} + \bar{b})} = \overline{(\bar{b} + \bar{a})}$; but $\overline{(\bar{b} + \bar{a})} = b \cdot a$, as desired. Can you prove that $a + (b \cdot c) = (a + b) \cdot (a + c)$?

A system of postulates is called *independent* when it is impossible to deduce from any portion of the system a conclusion deducible from any other portion. Can you prove that the five postulates given above are independent? Since the above set of postulates used the set S and the operations $+$ and $^-$, we denote this set as the $(S, +, ^-)$ postulate set. Another set of postulates, denoted by $(A, +, \cdot)$, is as follows:

1. There is an element 0 in A such that $0 + a = a$ for all a in A.
2. There is an element 1 in A such that $1 \cdot a = a$ for all a in A.
3. $0 \cdot a = 0$ for all a in A.
4. $1 + a = 1$ for all a in A.
5. A consists of *two* elements (that is, 0, 1).

Two sets of postulates **P** and **P′** are called *equivalent* if starting with the postulates **P**, after finding suitable definitions for the primitive concepts of **P′** in terms of the primitive symbols of **P**, the postulates of **P′** can be derived. Can you show that the postulate sets $(S, +, ^-)$ and $(A, +, \cdot)$ as given above are equivalent? This latter set is closely related to the techniques to be considered in the next chapter.

The following references deal with the foundations and the axiomatic approach to Boolean algebras:

Bernstein, B. A.: A Set of Four Independent Postulates for Boolean Algebra, *Trans. Am. Math. Soc.*, vol. 17, 1916; Sets of Postulates for the Logic of Propositions, *Trans. Am. Math. Soc.*, vol. 28, 1926.

Boole, G.: "George Boole's Collected Logical Works" (1859), vols. 1 and 2, P. E. B. Jourdain (ed.), The Open Court Publishing Company, La Salle, Ill., 1940; "An Investigation of the Laws of Thought," Dover Publications, New York, 1951.

Huntington, E. V.: Sets of Independent Postulates for the Algebra of Logic, *Trans. Am. Math. Soc.*, vol. 5, pp. 288–309, 1904; A New Set of Independent Postulates for the Algebra of Logic, *Proc. Natl. Acad. Sci. U.S.*, vol. 18, 1932; New Sets of Independent Postulates for the Algebra of Logic, *Trans. Am. Math. Soc.*, vol. 35, 1933.

Jevons, W. S.: "Pure Logic (1864) and Other Minor Works," R. Adamson and H. A. Jevons (eds.), The Macmillan Company, London and New York, 1890.

Schroeder, E.: "Algebra der Logik," vol. 1, Teubner Verlagsgesellschaft, Leipzig, 1890.

A discussion of the axiomatic method and techniques for proving the independence of and equivalence between axiom systems is found in R. L. Wilder, "Introduction to the Foundations of Mathematics," John Wiley & Sons, Inc., New York, 1952.

b. Formal Rules of Inference. We have been discussing above axioms from which the identities or tautologies of Boolean algebra may be derived. We have tacitly assumed that the "usual rules" of logic are to be used in such derivations. But it is just these rules themselves which we are proposing to study! This is the problem which is considered in logic texts and which will now be briefly treated. We are dealing with propositions whose content may be true or false and with tautologies that are stated in terms of propositions and that are themselves the rules of logic. However, we must start with primitive tautologies (axioms) and some primitive laws of deduction. One such primitive set is as follows: The primitive tautologies are:

1. $(x + x) \rightarrow x$.
2. $x \rightarrow (x + y)$.
3. $(x + y) \rightarrow (y + x)$.
4. $(x + y) \rightarrow ((z + x) \rightarrow (z + y))$.

The primitive rules are: (1) The *rule of substitution* (*dictum de omni*); we may substitute for a propositional variable any other variable or combination of variables, provided that the substitution is made wherever that sentential variable occurs. (2) The *rule of inference* (*modus ponens*); if x and $x \rightarrow y$ are tautologies, then y is a tautology. From these we can prove all the tautologies of Boolean algebra. For example, let us prove that, if $z \rightarrow x$ and $x \rightarrow y$ are tautologies, then $z \rightarrow y$ is a tautology. To show this, first note that by our rule of substitution we shall let $\bar{z}$ replace z in (4), obtaining as a tautology $(x \rightarrow y) \rightarrow ((\bar{z} + x) \rightarrow (\bar{z} + y))$ or $(x \rightarrow y) \rightarrow ((z \rightarrow x) \rightarrow (z \rightarrow y))$. Now, by the rule of inference, since $x \rightarrow y$ is a tautology, then $(z \rightarrow x) \rightarrow (z \rightarrow y)$ is a tautology; by our rule of inference again, since $z \rightarrow x$ is a tautology, then $z \rightarrow y$ is a tautology, as desired.

For detailed discussions of the formal theory of symbolic logic, the reader is referred to the following:

Church, A.: Introduction to Mathematical Logic, Part I, *Ann. Math. Studies* 13, Princeton University Press, Princeton, N.J., 1944.

Cohen, M., and E. Nagel: "An Introduction to Logic and the Scientific Method," Harcourt, Brace and Company, Inc., New York, 1934.

Hilbert, D., and W. Ackerman: "Principles of Mathematical Logic," R. E. Luce (ed.), Chelsea, New York, 1950.

Quine, Willard van Orman: "Mathematical Logic," rev. ed., Harvard University Press, Cambridge, Mass., 1951.

Reichenbach, H.: "Elements of Symbolic Logic," The Macmillan Company, New York, 1950.

Rosser, J. B.: "Logic for Mathematicians," McGraw-Hill Book Company, Inc., New York, 1953.

Whitehead, A. N., and B. Russell: "Principia Mathematica," vol. 1, 1910, vol. 2, 1912, vol. 3, 1913, Cambridge University Press, New York (2d ed., 1925–1927).

c. Other Logical Operations. As we have observed above, Boolean algebra can be accomplished with just two operations, $+$ and $^-$, since the $\cdot$ operation can be given in terms of $+$ and $^-$ as $a \cdot b = \overline{\bar{a} + \bar{b}}$. There are in fact single operations in terms of which Boolean algebra can be accomplished. Consider the operation $p_1(a,b)$ defined in terms of $+$ and $^-$ as follows: $p_1(a,b) = \overline{a + b}$. To show that Boolean algebra can be accomplished in terms of this single operation, all we need do is define $+$, $\cdot$, and $^-$ in terms of $p_1(a,b)$. Thus, as can easily be checked from our definition, $\bar{a} = p_1(a,a)$; $a + b = p_1[p_1(a,b),p_1(a,b)]$; and $a \cdot b = p_1[p_1(a,a),p_1(b,b)]$. Another such single operation is $p_2(a,b) = \overline{a \cdot b}$. Here we have $\bar{a} = p_2(a,a)$, $a + b = p_2[p_2(a,a),p_2(b,b)]$, and $a \cdot b = p_2[p_2(a,b),p_2(a,b)]$. Note that I and 0 can be defined in terms of p_1 or p_2. For p_1 we have $I = p_1\{p_1[a,p_1(a,a)],p_1[a,p_1(a,a)]\}$, and then $0 = p_1(I,I)$. Can you define I and 0 in terms of p_2? If we admit I and 0 as primitive concepts, then there are four further single operations that can be defined, namely, $p_3(a,b) = \bar{a} + b$, $p_4(a,b) = a + \bar{b}$, $p_5(a,b) = \bar{a} \cdot b$, $p_6(a,b) = a \cdot \bar{b}$. For example, for p_3 we have

$\bar{a} = p_3(a,0)$, $a + b = p_3[p_3(a,0),b]$, and $a \cdot b = p_3\{p_3[a,p_3(b,0)],0\}$. Can you show that p_4, p_5, and p_6 are each also valid single operations? What algebraic properties do $p_1, \ldots, p_6$ have? The single operation p_1 was first introduced by H. M. Sheffer [A Set of Five Independent Postulates for Boolean Algebras, *Trans. Am. Math. Soc.,* vol. 14, pp. 481–488, 1913, where $p_1(a,b)$ was denoted by $a|b$, commonly known as *Sheffer's stroke*]. A single primitive formula can be used in terms of p_1 (instead of the four given in Additional Topic b above), namely,

$$p_1[p_1(x,p_1\{y,z\}),p_1(p_1\{u,p_1[u,u]\},p_1\{p_1[v,y],p_1[p_1(x,v),p_1(x,v)]\})]$$

This was first developed by J. Nicod (A Reduction in the Number of the Primitive Propositions of Logic, *Proc. Cambridge Phil. Soc.,* vol. 19, pp. 32–42, 1916).

d. Logical Translation of English. The limitation on the practical application of the propositional calculus arises from the difficulty of translation of the practical worldly situations into these precise statements. This translation, the first step in formulating the problem for solution, is by no means obvious and requires the work of a trained logical analyst. Part of this difficulty is the prevalent loose and often obscure use of language, and in particular of those very conjunctions which form the logical operations.

TABLE 10-1. TRANSLATIONS OF ENGLISH CONNECTIVES

English connective	*Logical translation*
Not A	$\bar{A}$
A and B	$A \cdot B$
A or (inclusive) B; A or B or both	$A + B$
A or (exclusive) B; either A or B	$A \neq B$, $A \cdot \bar{B} + \bar{A} \cdot B$
A but B	$A \cdot B$
A although B	$A \cdot B$
A unless B	$\bar{B} \rightarrow A$, $B + A$
A on condition that B	$B \rightarrow A$, $\bar{B} + A$
A if B	$B \rightarrow A$, $\bar{B} + A$
If A, then B; A implies B	$A \rightarrow B$, $\bar{A} + B$
A only if B†	$A \rightarrow B$, $\bar{A} + B$
Not unless A then B	$\bar{A} \rightarrow \bar{B}$, $A + \bar{B}$
A provided that B	$B \rightarrow A$, $\bar{B} + A$
A as well as B	$A \cdot B$
A if and only if B	$A = B$, $A \cdot B + \bar{A} \cdot \bar{B}$
Not both A and B	$\overline{A \cdot B}$, $\bar{A} + \bar{B}$
Neither A nor B	$\bar{A} \cdot \bar{B}$, $\overline{A + B}$
When A, then B	$A \rightarrow B$, $\bar{A} + B$
A because B	$B \rightarrow A$, $\bar{B} + A$

† Often imprecisely synonymous with "A if and only if B."

For example, the word *unless* as used in the combined proposition "A unless B" is defined by logicians as $A + B$, since "A unless B" is considered equivalent to "A if not B," that is, "if not B, then A," which is written symbolically as $\bar{B} \rightarrow A$. However, since not all users of the word *unless* are familiar with the logicians' delineation of the term, one often finds it employed with an entirely different connotation. For instance, "A unless B" is often used to denote "If B, then not A," symbolically $B \rightarrow \bar{A}$, or $\bar{B} + \bar{A}$. A third possibility for "A unless B" might be "A, but if B then not A," which becomes $A \cdot (\bar{B} + \bar{A})$, or $A \cdot \bar{B}$. It is thus evident that in the formulation of the propositional elements from the original raw sentences one must have enough familiarity with the subject matter to be able to discern the *intent* of a sentence in its given context.

In Table 10-1 we have suggested some translations for various English connectives. Caution must be used for the translation of *or* to distinguish between the *inclusive or*, that is, *A or B or both*, and the *exclusive or*, that is, either *A or B* but not both. Also care must be observed in using "If A then B," or "A implies B." Strict logical interpretation is $A \rightarrow B$, or $\bar{A} + B$. Consider, for example, $A \rightarrow B \cdot C$. This is equivalent to $(A \rightarrow B) \cdot (A \rightarrow C)$, which is reasonable in terms of the English translation. Similarly $A \rightarrow (B + C)$ is equivalent to $(A \rightarrow B) + (A \rightarrow C)$, which again is reasonable. However, $A \cdot B \rightarrow C$ is equivalent to $(A \rightarrow C) + (B \rightarrow C)$, which upon English translation is not obvious without considerable reflection; i.e., "Both A and B imply C" is the same as "A implies C, or B implies C, or both A implies C and B implies C." Similarly $(A + B) \rightarrow C$ is equivalent to $(A \rightarrow C) \cdot (B \rightarrow C)$, which again is not particularly obvious upon English translation.

For further discussion of this topic see these works:

Allen, L. E.: Toward More Clarity in Business Communications by Modern Logical Methods, *Management Sci.*, vol. 5, no. 1, pp. 121–135, October, 1958.

Carnap, R.: "The Logical Syntax of Language," Routledge and Kegan Paul, Ltd., London, 1949.

Ledley, R. S.: Mathematical Foundations and Computational Methods for a Digital Logic Machine, *J. Operations Research Soc. Am.*, vol. 2, no. 3, pp. 249–274, August, 1954. (See also letters on Logical Translation of English by L. Davidson, *J. Operations Research Soc. Am.*, vol. 3, no. 2, p. 466, November, 1954, and J. W. Gilmore, *J. Operations Research Soc. Am.*, vol. 3, no. 1, p. 104, February, 1955.)

Reichenbach, H.: "Elements of Symbolic Logic," The Macmillan Company, New York, 1947.

CHAPTER 11

THE DESIGNATION NUMBERS AND THE DESIGN OF FUNCTION CIRCUITS

11-1. Introduction

Boolean Algebra. In the last chapter Boolean algebra was introduced by means of its logical interpretation. Then it was shown how Boolean algebra could also be applied to describe relationships between sets and between input and output wires of digital-computer circuitry. This latter application is, of course, of primary interest to us. It was shown how to draw a circuit diagram corresponding to a Boolean function and, conversely, how to write a Boolean function, corresponding to a circuit diagram, for the output wires in terms of the input wires.

However, certain problems appeared in this study that were not resolved. For instance, it was seen that there are in general many circuit designs that will do the same job. Also the algebraic manipulations of Boolean algebra seem rather difficult, requiring considerable skill and imagination together with a thorough knowledge of many of the relationships given in Sec. 10-6. This chapter will present systematic computational methods that enable the relatively easy and straightforward solutions to these and many more problems that may arise in the design of computer circuitry.

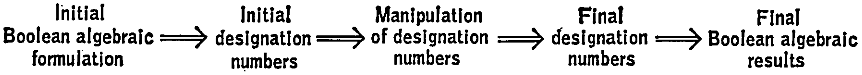

Initial Boolean algebraic formulation
Initial designation numbers
Manipulation of designation numbers
Final designation numbers
Final Boolean algebraic results

FIG. 11-1. Procedural chain for algebraic manipulation.

Computational Methods and Circuit Design. The computational methods to be given depend on the systematic manipulation of *designation numbers* that will be described in this chapter. The procedure is first to turn the initial Boolean algebraic functions into designation numbers, then to perform the indicated computations with these numbers, and finally to transform the resulting numbers back into the final desired Boolean algebraic expressions. In this way algebraic manipulations are replaced by the systematic numerical computations to be described. This procedure is diagramed in Fig. 11-1.

As we have seen, two disciplines of study are brought together for the design of computer circuitry. The first is that of Boolean algebra, which

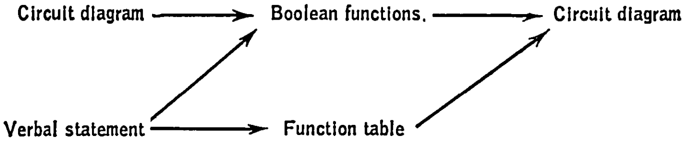


FIG. 11-2. From verbal statement to circuit diagram.

has broad applications to many fields. The second is that which is specific to digital-computer circuitry. Figure 11-2 diagrams procedures of the latter kind, some of which were described in the last chapter.

The first parts of the present chapter will be devoted to the procedures diagramed in Fig. 11-1; the rest of the chapter will be concerned with the procedures of Fig. 11-2 and their combination with those of Fig. 11-1. In fact the entire circuit-design problem can be summarized by the diagram in Fig. 11-3. The double-lined arrows refer to procedures that involve only Boolean algebra and the computational methods; the single-lined arrows refer to procedures unique to the digital-computer circuit problem.

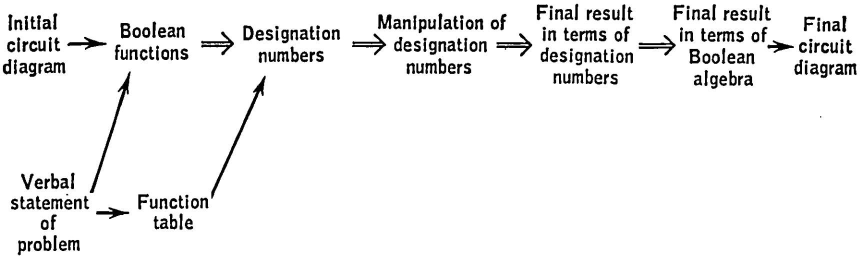

Initial circuit diagram
Boolean functions
Designation numbers
Manipulation of designation numbers
Final result in terms of designation numbers
Final result in terms of Boolean algebra
Final circuit diagram
Verbal statement of problem
Function table

FIG. 11-3. Combination of Figs. 11-1 and 11-2. The circuit-design problem outlined.

11-2. The Designation Numbers

The computational methods to be described involve binary numbers called *designation numbers.* To every element, elementary or combined, we associate a designation number. In doing a problem, the designation-number representation for each proposition is obtained; the computations are performed in terms of these numbers; and finally the answer is reinterpreted in terms of propositions or sentences. The procedure for going from propositions to designation numbers will be considered first, and that for returning from designation numbers to propositions will follow.

The Basis. The designation numbers for the elementary elements are assigned first. Such an assignment is called a basis. One such basis for a system of three elementary elements is

$$\begin{array}{rl} & {\scriptstyle 0123\quad 4567} \\ \#A = & 0101\quad 0101 \\ \#B = & 0011\quad 0011 \\ \#C = & 0000\quad 1111 \end{array}$$

where the upper small numbers merely number the positions of the columns of the basis.[†] For a system of n elementary elements,[‡] there are 2^n digits, or bits, in each designation number and hence 2^n columns in a basis. Any permutation of the columns of the basis shown will result in a different but nonetheless valid basis. The columns of a valid basis for n elementary elements need only represent all the 2^n possible combinations of 0 and 1 taken n at a time. Hence there can be $2^n!$ different bases.

One advantage of the type of basis illustrated above is that it may be written out directly for any number of elementary elements as follows: the number for the first elementary element alternates zeros and units; for the second alternates pairs of zeros and units; for the third alternates 4 zeros and 4 units; for the fourth would alternate 8 zeros and 8 units, and so forth, each such designation number being extended to comprise the full complement of 2^n bits. The column positions of the basis are numbered from the left, from 0 through $2^n - 1$. Such a basis is called a standard basis, and unless otherwise stated *it is always assumed that all designation numbers refer to this type of basis.* Following this convention we have, for example, in the case of four elementary elements, the standard basis:

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
$\#A =$	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1
$\#B =$	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1
$\#C =$	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1
$\#D =$	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1

Observe that each column of a standard basis, read from the bottom to the top, forms a binary number which is the same as the number of the column position. For example, column 3 is $\begin{matrix}1\\1\\0\\0\end{matrix}$ corresponding to $0011 = 3$ (*decimal*), and column 14 is $\begin{matrix}0\\1\\1\\1\end{matrix}$ corresponding to $1110 = 14$ (*decimal*).

Determining the Designation Number of a Boolean Function. In order to find $\#(A + B)$, we define an operation of "logical addition," to be performed on $\#A$ and $\#B$, wherein $0 + 0 = 0$, $0 + 1 = 1 + 0 = 1$, and $1 + 1 = 1$ *without carry.* For example, with respect to the standard basis for two elementary elements,

$$\begin{array}{r} \#A = 0101 \\ \#B = 0011 \\ \hline \#(A + B) = 0111 \end{array}$$

To find $\#(A \cdot B)$, we similarly define the position-by-position operation of "logical multiplication," wherein $0 \cdot 0 = 0$, $0 \cdot 1 = 1 \cdot 0 = 0$, and $1 \cdot 1 = 1$. For example,

$$\begin{array}{r} \#A = 0101 \\ \#B = 0011 \\ \hline \#(A \cdot B) = 0001 \end{array}$$

[†] The # symbol means "The designation number of . . . ."

[‡] Also called *basic symbols.*

To find $\#\bar{A}$, we *invert* each digit of $\#A$, that is, change 0 to 1 and 1 to 0. For example,

$$\#\bar{A} = 1010$$

With these operations on designation numbers in mind, we determine the designation number of a combined element merely by performing the indicated operations with respect to the chosen basis. For example, to find $\#(\bar{A} + B \cdot C)$, we have, with respect to a basis for three elementary elements,

$$\begin{array}{r@{\;=\;}l}
\#B & 0011 \quad 0011 \\
\#C & 0000 \quad 1111 \\ \hline
\#(B \cdot C) & 0000 \quad 0011 \\
\#\bar{A} & 1010 \quad 1010 \\ \hline
\#(\bar{A} + B \cdot C) & 1010 \quad 1011
\end{array}$$

There should be no confusion as to the dual role of the symbols $+, \cdot, \bar{\ }$. When applied to propositions $A, B, C, \ldots$, they are the logical operations *or*, *and*, and *not*, respectively; when applied to designation numbers $\#A, \#B, \#C, \ldots$, they are the numerical operations *logical add*, *logical multiply*, and *invert*, respectively.

Important Results. Some important results of this digitalization of propositions are: (1) $\#I = 1111 \cdots$; that is, $\#I$ has all units. (2) $\#X = \#Y$ if, and only if, $X = Y$. (3) $X \to Y$ if, and only if, $\#Y$ has units in at least those positions which correspond to the units of $\#X$.

For example, to demonstrate equivalences, we simply calculate the designation number of each side; if they are identical, the equation holds. To show that $A \cdot B + A \cdot C + \bar{B} \cdot C = A \cdot B + \bar{B} \cdot C$, we find

$$\#(A \cdot B) = 0001 \quad 0001 \qquad \#(A \cdot C) = 0000 \quad 0101$$

and $$\#(\bar{B} \cdot C) = 0000 \quad 1100$$

whence $$\#(A \cdot B + A \cdot C + \bar{B} \cdot C) = 0001 \quad 1101$$

but $\#(A \cdot B + \bar{B} \cdot C) = 0001\ 1101$ also, whence the equivalence must hold. To show that an implication is true, simply show that the implied proposition has units in at least the same position as the implier. For example, to demonstrate that $A \cdot B + \bar{B} \cdot C \to A + C$, we note that $\#(A \cdot B + \bar{B} \cdot C) = 000\overset{3}{1}\ \overset{4}{1}\overset{5}{1}0\overset{7}{1}$, which has units in positions 3, 4, 5, and 7; also $\#(A + C) = 010\overset{3}{1}\ \overset{4}{1}\overset{5}{1}1\overset{7}{1}$, which obviously has units in at least these positions.

Basis and Truth Tables. To see the relationship between the truth tables and the basis, observe that, if in the truth table of Sec. 10-5, T is replaced by 1 and F is replaced by 0 and the columns are written as rows, then a basis results, together with the designation number of $\bar{A} + B \cdot C$.

$$\begin{array}{l@{\;=\;}l}
0101 \quad 0101 & \#A \\
0011 \quad 0011 & \#B \\
0000 \quad 1111 & \#C \\ \hline
1010 \quad 1011 & \#(\bar{A} + B \cdot C)
\end{array}$$

The reason for this is as follows: The main feature of our basis is that the 2^n columns represent all the 2^n possible arrangements of n digits, each place of which can be 0 or 1. This corresponds precisely to all possible truth values of n elementary elements.

The limitations of the truth-table method as compared with our numerical basis hinge on the fact that the *order* of the columns of the basis is fixed by the basis, while in the truth table it makes no difference in what order the rows are considered. (The truth table illustrated above was chosen so as to compare with our basis.) Thus merely to specify a particular combined element it is necessary to present a complete "two-dimensional" truth table. The usefulness and applicability of truth tables have therefore been confined primarily to elementary didactic discussions and visual aid rather than as a notational representation for extensive computational techniques.

EXERCISES

With respect to standard bases determine the following:

	Solution
(*a*) $\#(A + \bar{B})$	1101
(*b*) $\#(A \cdot \bar{B})$	0100
(*c*) $\#(A \cdot \bar{B} + \bar{C})$	1111 0100
(*d*) $\#[A \cdot (\bar{B} + \bar{C})]$	0101 0100
(*e*) $\#[A \cdot B + \bar{A} \cdot \bar{B}]$	1001
(*f*) $\#[A \cdot \bar{B} + \bar{A} \cdot B]$	0110
(*g*) $\#[A \cdot \bar{B} + C \cdot \bar{D}]$	0100 1111 0100 0100
(*h*) $\#[A \cdot (B + \bar{C}) + \bar{A} \cdot (\bar{B} + C)]$	1101 1011
(*i*) $\#[A \cdot B \cdot C + \bar{A} \cdot \bar{B} \cdot \bar{C}]$	1000 0001
(*j*) $\#[(A + B + C) \cdot (\bar{A} + \bar{B} + \bar{C})]$	0111 1110
(*k*) $\#[A \cdot \bar{B} \cdot C + \bar{C} \cdot (D + B \cdot \bar{E}) + \bar{D} \cdot E]$	0011 0100 1111 0100 1111 1111 1111 0100

11-3. The First and Second Canonical Forms

From Designation Number to Boolean Function. Given a designation number, there is a systematic method for finding its symbolic Boolean algebraic representation in several different forms. For an example of five different equivalent representations of the same designation number, consider the number 0111 0100. (1) The first canonical form of this is $A \cdot \bar{B} \cdot \bar{C} + \bar{A} \cdot B \cdot \bar{C} + A \cdot B \cdot \bar{C} + A \cdot \bar{B} \cdot C$. (2) The second canonical form is $(A + B + C) \cdot (A + B + \bar{C}) \cdot (A + \bar{B} + \bar{C}) \cdot (\bar{A} + \bar{B} + \bar{C})$. (3) The simplest sum-of-products form is $A \cdot \bar{B} + B \cdot \bar{C}$. (4) The simplest product-of-sums form is $(A + B) \cdot (\bar{B} + \bar{C})$. (5) A mongrel form is $(A + B) \cdot \bar{C} + A \cdot \bar{B} \cdot C$. All these combined elements are equivalent. In this section systematic methods for obtaining the first two forms are given; the methods for finding the latter three forms are given in subsequent sections.

The First Canonical Form. The so-called *first canonical form,* also called the disjunctive normal form, is a sum of elementary products. An *elementary product* is a product that contains *all* the elementary elements or their negations. For example, all possible elementary products for three elementary elements are $\bar{A} \cdot \bar{B} \cdot \bar{C}$, $A \cdot \bar{B} \cdot \bar{C}$, $\bar{A} \cdot B \cdot \bar{C}$, $A \cdot B \cdot \bar{C}$, $\bar{A} \cdot \bar{B} \cdot C$, $A \cdot \bar{B} \cdot C$, $\bar{A} \cdot B \cdot C$, and $A \cdot B \cdot C$. Observe that the designation of an elementary product always has a single unit. For example,

$$\begin{aligned}
\#(\bar{A} \cdot \bar{B} \cdot \bar{C}) &= 1000 \quad 0000 \\
\#(A \cdot \bar{B} \cdot \bar{C}) &= 0100 \quad 0000 \\
\#(\bar{A} \cdot B \cdot \bar{C}) &= 0010 \quad 0000 \\
\#(A \cdot B \cdot \bar{C}) &= 0001 \quad 0000 \\
\#(\bar{A} \cdot \bar{B} \cdot C) &= 0000 \quad 1000 \\
\#(A \cdot \bar{B} \cdot C) &= 0000 \quad 0100 \\
\#(\bar{A} \cdot B \cdot C) &= 0000 \quad 0010 \\
\#(A \cdot B \cdot C) &= 0000 \quad 0001
\end{aligned}$$

For n elementary elements there are 2^n such products. The first canonical form of a designation number is the sum of only those elementary products each of which contributes a unit to the number. For example, 0100 0011 has units in positions 1, 6, and 7 and can therefore be constructed by summing the elementary products whose designation numbers have units in positions 1, 6, and 7. Hence, we have

$$\begin{array}{r}
\#(A \cdot \bar{B} \cdot \bar{C}) = 0100 \quad 0000 \\
\#(\bar{A} \cdot B \cdot C) = 0000 \quad 0010 \\
\#(A \cdot B \cdot C) = 0000 \quad 0001 \\
\hline
\#(A \cdot \bar{B} \cdot \bar{C} + \bar{A} \cdot B \cdot C + A \cdot B \cdot C) = 0100 \quad 0011
\end{array}$$

i.e., the first canonical form representation of 0100 0011 is

$$A \cdot \bar{B} \cdot \bar{C} + \bar{A} \cdot B \cdot C + A \cdot B \cdot C$$

The Second Canonical Form. To find the *second canonical form,* also called the conjunctive normal form, note that the designation number of the *sum* of all the elementary elements or their negations has exactly one zero:

$$\begin{aligned}
\#(\bar{A} + \bar{B} + \bar{C}) &= 1111 \quad 1110 \\
\#(A + \bar{B} + \bar{C}) &= 1111 \quad 1101 \\
\#(\bar{A} + B + \bar{C}) &= 1111 \quad 1011 \\
\#(A + B + \bar{C}) &= 1111 \quad 0111 \\
\#(\bar{A} + \bar{B} + C) &= 1110 \quad 1111 \\
\#(A + \bar{B} + C) &= 1101 \quad 1111 \\
\#(\bar{A} + B + C) &= 1011 \quad 1111 \\
\#(A + B + C) &= 0111 \quad 1111
\end{aligned}$$

These sums are called *elementary sums;* for n elements there are 2^n elementary sums. The conjunctive normal form corresponding to a given designation number is the product of those elementary sums which correspond to the zeros in the designation number. For example, for

$$\begin{array}{l}
\scriptstyle 0\ \ 23\ \ 45 \\
0100 \quad 0011
\end{array}$$

we have zeros in positions 0, 2, 3, 4, and 5, and hence

$$\#[(A + B + C) \cdot (A + \bar{B} + C) \cdot (\bar{A} + \bar{B} + C) \cdot (A + B + \bar{C}) \cdot (\bar{A} + B + \bar{C})] \\ = 0100 \quad 0011$$

This follows because each elementary sum in the product contributes one of the zeros of the designation number.

$$\begin{array}{rl}
\#(A + B + C) = & 0111 \quad 1111 \\
\#(A + \bar{B} + C) = & 1101 \quad 1111 \\
\#(\bar{A} + \bar{B} + C) = & 1110 \quad 1111 \\
\#(A + B + \bar{C}) = & 1111 \quad 0111 \\
\#(\bar{A} + B + \bar{C}) = & 1111 \quad 1011 \\
\hline
\#(\text{product}) = & 0100 \quad 0011
\end{array}$$

as desired.

EXERCISES

Find the first and second canonical form corresponding to the following designation numbers:

Answer

(*a*) 1001 0110 $\#[\bar{A} \cdot \bar{B} \cdot \bar{C} + A \cdot B \cdot \bar{C} + A \cdot \bar{B} \cdot C + \bar{A} \cdot B \cdot C]$
$\#[(\bar{A} + B + C) \cdot (A + \bar{B} + C) \cdot (A + B + \bar{C}) \cdot (\bar{A} + \bar{B} + \bar{C})]$

(*b*) 0101 0101 $\#[A \cdot B \cdot C + A \cdot B \cdot \bar{C} + A \cdot \bar{B} \cdot C + A \cdot \bar{B} \cdot \bar{C}]$
$\#[(A + B + C) \cdot (A + \bar{B} + C) \cdot (A + B + \bar{C}) \cdot (A + \bar{B} + \bar{C})]$

(*c*) 1000 0001 $\#[\bar{A} \cdot \bar{B} \cdot \bar{C} + A \cdot B \cdot C]$
$\#[(\bar{A} + B + C) \cdot (A + \bar{B} + C) \cdot (\bar{A} + \bar{B} + C) \cdot (A + B + \bar{C}) \cdot (\bar{A} + B + \bar{C}) \cdot (A + \bar{B} + \bar{C})]$

(*d*) 0111 1110 $\#[A \cdot \bar{B} \cdot \bar{C} + \bar{A} \cdot B \cdot \bar{C} + A \cdot B \cdot \bar{C} + \bar{A} \cdot \bar{B} \cdot C + A \cdot \bar{B} \cdot C + \bar{A} \cdot B \cdot C]$
$\#[(A + B + C) \cdot (\bar{A} + \bar{B} + \bar{C})]$

(*e*) 1011 $\#[\bar{A} \cdot \bar{B} + \bar{A} \cdot B + A \cdot B]$
$\#[\bar{A} + B]$

(*f*) 0010 $\#(\bar{A} \cdot B)$
$\#[(A + B) \cdot (\bar{A} + B) \cdot (\bar{A} + \bar{B})]$

(*g*) 1011 0001 0110 0101 $\#[\bar{A} \cdot \bar{B} \cdot \bar{C} \cdot \bar{D} + \bar{A} \cdot B \cdot \bar{C} \cdot \bar{D} + A \cdot B \cdot \bar{C} \cdot \bar{D} + A \cdot B \cdot C \cdot \bar{D} + A \cdot \bar{B} \cdot \bar{C} \cdot D + \bar{A} \cdot B \cdot \bar{C} \cdot D + A \cdot \bar{B} \cdot C \cdot D + A \cdot B \cdot C \cdot D]$
$\#[(\bar{A} + B + C + D) \cdot (A + B + \bar{C} + D) \cdot (\bar{A} + B + \bar{C} + D) \cdot (A + \bar{B} + \bar{C} + D) \cdot (A + B + C + \bar{D}) \cdot (\bar{A} + \bar{B} + C + \bar{D}) \cdot (A + B + \bar{C} + \bar{D}) \cdot (A + \bar{B} + \bar{C} + \bar{D})]$

11-4. Included and Nonincluded Elementary Elements

Elimination Pairs of Positions. Certain information about a Boolean function can be obtained from the designation number even before the Boolean function is explicitly displayed. For example, 0011 1111 can be written as a function in which A does not appear. To see this, consider the first canonical form of 0011 1111, namely,

$$\bar{A} \cdot B \cdot \bar{C} + A \cdot B \cdot \bar{C} + \bar{A} \cdot \bar{B} \cdot C + A \cdot \bar{B} \cdot C + \bar{A} \cdot B \cdot C + A \cdot B \cdot C$$

This can be factored as follows:

$$(\bar{A} + A) \cdot B \cdot \bar{C} + (\bar{A} + A) \cdot \bar{B} \cdot C + (\bar{A} + A) \cdot B \cdot C$$

but $\bar{A} + A = I$, whence the expression reduces to

$$B \cdot \bar{C} + \bar{B} \cdot C + B \cdot C$$

in which A does not appear explicitly. This example illustrates how it may be possible to combine pairs of elementary products so as to eliminate an elementary element from appearing explicitly in the form of a Boolean expression. On the other hand, if only one of a pair of elementary products is in the canonical form, then A could not be eliminated; e.g., consider 0001 1111. The canonical form is

$$A \cdot B \cdot \bar{C} + \bar{A} \cdot \bar{B} \cdot C + A \cdot \bar{B} \cdot C + \bar{A} \cdot B \cdot C + A \cdot B \cdot C$$

and on combining pairs

$$A \cdot B \cdot \bar{C} + (\bar{A} + A) \cdot \bar{B} \cdot C + (\bar{A} + A) \cdot B \cdot C$$

or

$$A \cdot B \cdot \bar{C} + \bar{B} \cdot C + B \cdot C$$

in which A appears explicitly. (It can be shown that no equivalent function exists in which A does not appear.) Hence we have shown that it is plausible to expect that the necessary occurrence or nonoccurrence (i.e., necessary inclusion or noninclusion) of an elementary element can be determined by observing the canonical form or, what is close to the same thing, by directly observing the designation number. Such is the case, and the purpose of this section is to present rules for such determinations.

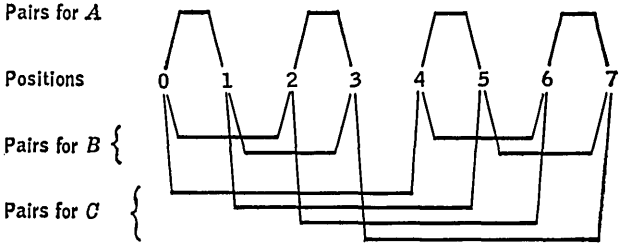


FIG. 11-4. Elimination-pair positions for three variables.

These rules are given as properties of so-called *elimination pairs of positions* of the designation number. Each elementary element has a set of pairs of positions. For three elementary elements, with respect to the standard basis the numbers of the positions of these pairs are

A:	(0,1)	(2,3)	(4,5)	(6,7)
B:	(0,2)	(1,3)	(4,6)	(5,7)
C:	(0,4)	(1,5)	(2,6)	(3,7)

The elimination pairs may be visualized as shown in Fig. 11-4. The extension of this system to four elementary elements follows directly:

A:	(0,1)	(2,3)	(4,5)	(6,7)	(8,9)	(10,11)	(12,13)	(14,15)
B:	(0,2)	(1,3)	(4,6)	(5,7)	(8,10)	(9,11)	(12,14)	(13,15)
C:	(0,4)	(1,5)	(2,6)	(3,7)	(8,12)	(9,13)	(10,14)	(11,15)
D:	(0,8)	(1,9)	(2,10)	(3,11)	(4,12)	(5,13)	(6,14)	(7,15)

Nonincluded Elementary Elements. Now that we have defined the elimination pairs with respect to the standard basis, the rules can be stated.

RULE 1. A Boolean function can be written that does not involve explicitly an elementary element if in the designation number all the elimination pairs of positions for that elementary element contain either both zeros or both units.

For example, consider $\overset{0123}{0011}$ $\overset{4567}{1111}$. The elimination pairs for A satisfy rule 1, for positions (6,7) contain 11, positions (4,5) contain 11, positions (2,3) contain 11, and positions (0,1) contain 00. Hence, as we have seen, a Boolean function for this designation number can be written in which A does not appear explicitly.

Consider as another example $\overset{0123}{1101}$ $\overset{4567}{1101}$. Here rule 1 is satisfied for the elimination pairs of C; that is, positions (3,7) contain 11, positions (2,6) contain 00, positions (1,5) contain 11, and positions (0,4) contain 11. In fact its Boolean function is just $A + \bar{B}$, in which C does not explicitly appear. In such a case we say that C is a nonincluded elementary element. In the previous example A was the nonincluded elementary element.

Included Elementary Elements. Next we shall give a rule for telling from the designation number whether an elementary element must appear explicitly in every form of the Boolean expression or whether the negation of the element must appear. However, we must first discuss the meaning of the *appearance* of an elementary element or its negation.

An elementary element or its negation is said to appear in a Boolean function if it appears when the function is reduced to one in which a *bar* is applied only to individual elementary elements. For instance, A appears in the expression $(A + B) \cdot C$, whereas $\bar{A}$ does not appear. However, if we consider the expression $\overline{A \cdot B + C}$, it is not clear whether it is A or $\bar{A}$ which appears; if this is reduced to $(\bar{A} + \bar{B}) \cdot \bar{C}$ [that is, $\overline{A \cdot B + C} = (\bar{A} + \bar{B}) \cdot \bar{C}$] we see that $\bar{A}$ appears and A does not. Similarly, for example, $\overline{\bar{A} \cdot B} + C = A + \bar{B} + C$, in which A appears, and $\overline{\overline{\bar{A} \cdot B} + C} = \bar{A} \cdot B \cdot \bar{C}$, in which $\bar{A}$ appears.

RULE 2. An elementary element (as distinguished from the negation of an elementary element) must appear explicitly in every Boolean functional representation of a designation number if at least one of the elimination pairs of positions of that elementary element contains 01.

RULE 3. The negation of an elementary element must appear explicitly in every Boolean functional representation of a designation number if at least one of the elimination pairs of positions of that elementary element contains 10. For example, consider

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	0	0	1	1	1	0	1	1	1	1	1	1	1	0	1

Here A must appear since the elimination pair (2,3) contains 01. Both B and $\bar{B}$ must appear since the elimination pair (1,3) contains 01 and the pair (4,6) contains 10. Also C and $\bar{C}$ must both appear since the elimination pair (1,5) contains 01 and (10,14) contains 10. Only D appears since (0,8), (1,9), and (2,10) each contain 01, but (3,11), (4,12), (5,13), and (7,15) each contain 11 and (6,14) contains 00. In fact the Boolean function is $A \cdot B + \bar{B} \cdot C + \bar{C} \cdot D$. In this example we call A, B, $\bar{B}$, C, $\bar{C}$, and D *included* elementary elements.

EXERCISES

Find the included and nonincluded elementary elements for the following designation numbers:

		Answer	
		Included	*Nonincluded*
(*a*)	0101 0101	A	B, C
(*b*)	1100 0011	B, $\bar{B}$, C, $\bar{C}$	A
(*c*)	0111 0111	A, B	C
(*d*)	1111 1100 1100 1100	$\bar{B}$, $\bar{C}$, $\bar{D}$	A
(*e*)	0000 1111 1010 1010	$\bar{A}$, C, D, $\bar{D}$	B

11-5. Mongrel Forms

Examples of Mongrel Forms. The *mongrel form* is not a unique form; i.e., two people determining a mongrel form for the same designation number may not arrive at the same Boolean function. In general, in the mongrel form, anything goes, just as long as the final Boolean function actually represents the designation number under consideration. For example, consider the designation number 0001 0111. We note that

$$\begin{array}{rl}
\#A & = 0101 \quad 0101 \\
\#B & = 0011 \quad 0011 \\
\#C & = \underline{0000 \quad 1111} \\
\#(A \cdot B) & = 0001 \quad 0001 \\
\#(A + B) & = 0111 \quad 0111
\end{array}$$

But since $\#\bar{C} = 1111\ 0000$, then $\#A \cdot B \cdot \bar{C} = 0001\ 0000$; also, since $\#C = 0000\ 1111$, then $\#(A + B) \cdot C = 0000\ 0111$, whence

$$0001 \quad 0111 = \#[A \cdot B \cdot \bar{C} + (A + B) \cdot C]$$

or $A \cdot B \cdot \bar{C} + (A + B) \cdot C$ is a mongrel form. The mongrel form is particularly useful in more complicated cases. For example, consider 0001 0111 0111 0001 1110 1110 1000 1000. Here we have the basis

$\#A$	= 0101	0101	0101	0101	0101	0101	0101	0101
$\#B$	= 0011	0011	0011	0011	0011	0011	0011	0011
$\#C$	= 0000	1111	0000	1111	0000	1111	0000	1111
$\#D$	= 0000	0000	1111	1111	0000	0000	1111	1111
$\#E$	= 0000	0000	0000	0000	1111	1111	1111	1111

Since

$\#(A \cdot B)$	= 0001	0001	0001	0001	0001	0001	0001	0001
$\#(A + B)$	= 0111	0111	0111	0111	0111	0111	0111	0111
$\#(\bar{A} + \bar{B})$	= 1110	1110	1110	1110	1110	1110	1110	1110
$\#\bar{A} \cdot \bar{B}$	= 1000	1000	1000	1000	1000	1000	1000	1000

and

$\#(\bar{C} \cdot \bar{D} \cdot \bar{E})$	= 1111	0000	0000	0000	0000	0000	0000	0000
$\#(C \cdot \bar{D} \cdot \bar{E})$	= 0000	1111	0000	0000	0000	0000	0000	0000
$\#(\bar{C} \cdot D \cdot \bar{E})$	= 0000	0000	1111	0000	0000	0000	0000	0000
$\#(C \cdot D \cdot \bar{E})$	= 0000	0000	0000	1111	0000	0000	0000	0000
$\#(\bar{D} \cdot E)$	= 0000	0000	0000	0000	1111	1111	0000	0000
$\#(D \cdot E)$	= 0000	0000	0000	0000	0000	0000	1111	1111

we can immediately write down the following mongrel form:

$$\begin{aligned}&(A \cdot B) \cdot (\bar{C} \cdot \bar{D} \cdot \bar{E}) + (A + B) \cdot (C \cdot \bar{D} \cdot \bar{E}) + (A + B) \cdot (\bar{C} \cdot D \cdot \bar{E}) \\ &\qquad + (A \cdot B) \cdot (C \cdot D \cdot \bar{E}) + (\bar{A} + \bar{B}) \cdot (\bar{D} \cdot E) + (\bar{A} \cdot \bar{B}) \cdot (D \cdot E)\end{aligned}$$

A useful fact to remember in writing mongrel forms is that

$$0110 = \#(A \cdot \bar{B} + \bar{A} \cdot B)$$

and

$$1001 = \#(A \cdot B + \bar{A} \cdot \bar{B})$$

Of course this can be generalized to observe, for example, that

$$0000 \quad 1111 \quad 1111 \quad 0000 \quad 0000 \quad 1111 \quad 1111 \quad 0000 = \#(C \cdot \bar{D} + \bar{C} \cdot D)$$

and

$$1111 \quad 0000 \quad 0000 \quad 1111 \quad 1111 \quad 0000 \quad 0000 \quad 1111 = \#(C \cdot D + \bar{C} \cdot \bar{D})$$

Applying this to 0001 0111 0111 0001 1110 1110 1000 1000, we find the mongrel form

$$\begin{aligned}&[(A \cdot B) \cdot (C \cdot D + \bar{C} \cdot \bar{D}) + (A + B) \cdot (C \cdot \bar{D} + \bar{C} \cdot D)] \cdot \bar{E} \\ &\qquad + [(\bar{A} + \bar{B}) \cdot \bar{D} + \bar{A} \cdot \bar{B} \cdot D] \cdot E\end{aligned}$$

The writing of "good" mongrel forms depends on ingenuity and practice. Of course the individual component products and sums need not be written out explicitly as was done above. For example, consider 0110 1001 1001 0110. A little thought will enable the following mongrel form to be written down immediately:

$$(A \cdot \bar{B} + \bar{A} \cdot B) \cdot (C \cdot D + \bar{C} \cdot \bar{D}) + (A \cdot B + \bar{A} \cdot \bar{B}) \cdot (C \cdot \bar{D} + \bar{C} \cdot D)$$

EXERCISES

Write down directly mongrel forms for:

					"Good" solution
(*a*)	0110	1001			$(A \cdot \bar{B} + \bar{A} \cdot B) \cdot \bar{C} + (A \cdot B + \bar{A} \cdot \bar{B}) \cdot C$
(*b*)	1001	0110	0110	1001	$(A \cdot B + \bar{A} \cdot \bar{B}) \cdot (C \cdot D + \bar{C} \cdot \bar{D})$
					$+ (A \cdot \bar{B} + \bar{A} \cdot B) \cdot (C \cdot \bar{D} + \bar{C} \cdot D)$
(*c*)	0010	0100	0010	0010	$\bar{A} \cdot B \cdot (\bar{C} + D) + (A \cdot \bar{B}) \cdot (C \cdot \bar{D})$
(*d*)	1010	0010	0010	0010	$\bar{A} \cdot \bar{C} \cdot \bar{D} + \bar{A} \cdot B$
(*e*)	0111	1011	1101	1110	$(A + \bar{B}) \cdot \bar{C} \cdot \bar{D} + (\bar{A} + B) \cdot C \cdot \bar{D}$
					$+ (A + \bar{B}) \cdot \bar{C} \cdot D + (\bar{A} + \bar{B}) \cdot C \cdot D$
(*f*)	1011	0111	1011	0111	$(\bar{A} + B) \cdot (\bar{C} \cdot \bar{E}) + (A + B) \cdot C \cdot \bar{E} + (\bar{C} + D) \cdot E$
	1111	0000	1111	1111	

11-6. Simplest Sum-of-products Representation†

The Role of the Prime Implicants. This symbolic representation of a designation number has the property of being a *sum of products with the least number of operations* · *and* + . There may be several such forms associated with a single designation number. A systematic process can be described that will result in such a form. To understand this process, let us first discuss the general nature of a product.

The number of letters that enter into a product is called the number of *terms* in the product. Thus $A \cdot \bar{B} \cdot C$ is a three-term product, $A \cdot \bar{B}$ is a two-term product, and A is called a one-term product. In general, for n basic symbols, the designation number of an n-term product has 2^0 $(=1)$ unit, of an $(n - 1)$-term product the designation number has 2^1 units, of an $(n - 2)$-term product it has 2^2 units, . . . , of an $(n - s)$-term product the designation number has 2^s units. For example, $\#A \cdot \bar{B} \cdot C = 0000\ 0100$ with 1 unit, $\#A \cdot \bar{B} = 0100\ 0100$ with 2 units, and $\#A = 0101\ 0101$ with 4 units. Another observation about products that follows directly is that, if P and P' are products such that $\#P \rightarrow \#P'$, then P' is formed from P by deleting some of the terms of P. For example, $\#A \cdot \bar{B} \cdot C \rightarrow \#A \cdot \bar{B}$, the latter being formed by deleting C from the former. The converse is evidently also true.

Now we can turn to the discussion of the simplest sum-of-products form. If $\#F$ is the designation number in question, consider a simplest sum-of-products representation of $\#F$. Clearly each product P_i of the sum must be such that $\#P_i \rightarrow \#F$ (that is, $\#P_i$ cannot have units in positions where $\#F$ has zeros), for otherwise the sum of the products would not have the same designation number as $\#F$. Also, if this sum is indeed the *simplest* sum of products, then certainly no product P_i of this sum can be replaced by a product P'_i such that $\#P_i \rightarrow P'_i$, for then the resulting

† The method of this section is an adaptation of McCluskey's variation of Quine's method of simplification. (See E. J. McCluskey, Jr., Minimization of Boolean Functions, *Bell System Tech. J.*, vol. 35, p. 1417, November, 1956. See also W. V. Quine, The Problem of Simplifying Truth Functions, *Am. Math. Monthly*, vol. 59, no. 8, pp. 521–531, October, 1952.)

sum of products would become even simpler (i.e., one of the products would have a term deleted). In other words, *in order that a product P be considered at all for inclusion in a simplest sum-of-products form for $\#F$, it must be such that:*

1. $\#P \rightarrow \#F$.
2. *There exists no P' such that $\#P \rightarrow \#P'$ and $\#P' \rightarrow \#F$.*

If P has the first property, it is called an *implicant* of F; if it also has the second, it is a *prime implicant* of F.

For example if $\#F = 0111\ 0100$, then $A \cdot \bar{B} \cdot \bar{C}$, $\bar{A} \cdot B \cdot \bar{C}$, $A \cdot B \cdot \bar{C}$, $A \cdot \bar{B} \cdot C$, $A \cdot \bar{C}$, $A \cdot \bar{B}$, and $B \cdot \bar{C}$ are all implicants of F, as can be readily checked, but only $A \cdot \bar{C}$, $A \cdot \bar{B}$, and $B \cdot \bar{C}$ are *prime implicants* of F. To show that $A \cdot \bar{B}$ is a prime implicant of F, note that the only products implied by $A \cdot \bar{B}$ are A and $\bar{B}$. But $\#A \not\rightarrow \#F$, and $\#\bar{B} \not\rightarrow \#F$, and hence $A \cdot \bar{B}$ is a prime implicant of F. (On the other hand, $A \cdot \bar{B} \cdot C$ is not a prime implicant because $\#A \cdot \bar{B} \cdot C \rightarrow \#A \cdot \bar{B}$ and $\#A \cdot \bar{B} \rightarrow \#F$.)

There are two stages in determining the simplest sum-of-products form of F. The *first* is the systematic generation of *all* the *prime* implicants of F; the *second* is the selection from this collection of a subcollection of products the sum of which will truly be the simplest sum of products, as defined above. We shall now describe in detail how to generate all the prime implicants of a given designation number. Then we shall show how with the aid of a so-called *prime-implicant chart* the final simplest sum of products is chosen.

The First Stage—Generating the Prime Implicants. The rationale for generating the prime implicants can best be understood by examining the first canonical form. For 0111 0100 the first canonical form is $A \cdot \bar{B} \cdot \bar{C} + \bar{A} \cdot B \cdot \bar{C} + A \cdot B \cdot \bar{C} + A \cdot \bar{B} \cdot C$. The following reductions can be made by combining pairs of products:

$$A \cdot \bar{B} \cdot \bar{C} + A \cdot \bar{B} \cdot C = A \cdot \bar{B} \cdot (\bar{C} + C) = A \cdot \bar{B} \cdot I = A \cdot \bar{B}$$

$$\bar{A} \cdot B \cdot \bar{C} + A \cdot B \cdot \bar{C} = (\bar{A} + A) \cdot B \cdot \bar{C} = I \cdot B \cdot \bar{C} = B \cdot \bar{C}$$

Thus the first canonical form reduces to $A \cdot \bar{B} + B \cdot \bar{C}$. We can determine whether or not two products can be combined by comparing the pair of associated basis columns: if the two columns differ in just one row, the terms can be combined. For instance, the pair $A \cdot \bar{B} \cdot \bar{C}$, $A \cdot \bar{B} \cdot C$ corresponds to the columns $\begin{smallmatrix}1\\0\\0\end{smallmatrix}$, $\begin{smallmatrix}1\\0\\1\end{smallmatrix}$, which differ only in the third row; similarly the pair $\bar{A} \cdot B \cdot \bar{C}$, $A \cdot B \cdot \bar{C}$ corresponds to the columns $\begin{smallmatrix}0\\1\\0\end{smallmatrix}$, $\begin{smallmatrix}1\\1\\0\end{smallmatrix}$, which differ only in the first row. Based on these concepts, the first stage involves three steps. We shall illustrate these steps by following through a specific example.

Consider the following designation number in four basic symbols:

$$\begin{array}{cccccccccccccccc}
0 & 1 & & 3 & 4 & & 6 & 7 & & 9 & 10 & 11 & & 13 & 14 & 15 \\
1 & 1 & 0 & 1 & 1 & 0 & 1 & 1 & 0 & 1 & 1 & 1 & 0 & 1 & 1 & 1
\end{array}$$

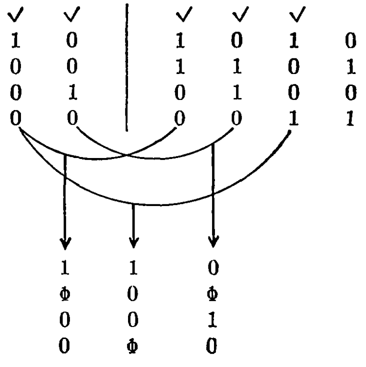
The first step is to display only the columns of the basis that correspond to the *units* of the given designation number,

0	1	3	4	6	7	9	10	11	13	14	15
0	1	1	0	0	1	1	0	1	1	0	1
0	0	1	0	1	1	0	1	1	0	1	1
0	0	0	1	1	1	0	0	0	1	1	1
0	0	0	0	0	0	1	1	1	1	1	1

The second step is to arrange these columns in order by the number of units contained in each column.

0-unit columns	1-unit columns	2-unit columns	3-unit columns	4-unit columns
0	10	1010	1110	1
0	00	1101	1101	1
0	01	0100	1011	1
0	00	0011	0111	1

The third step is an iterative process of combining pairs of columns that differ only in a single row. For example, $\begin{smallmatrix}0\\0\\0\\0\end{smallmatrix}$ and $\begin{smallmatrix}1\\0\\0\\0\end{smallmatrix}$ are combined to form $\begin{smallmatrix}\Phi\\0\\0\\0\end{smallmatrix}$, where the symbol Φ means 0 or 1. Similarly $\begin{smallmatrix}0\\0\\0\\0\end{smallmatrix}$ and $\begin{smallmatrix}0\\0\\1\\0\end{smallmatrix}$ are combined to form $\begin{smallmatrix}0\\0\\\Phi\\0\end{smallmatrix}$. Note that, since the pairs are to differ in a single row only, each u-unit column need be compared with only $(u + 1)$-unit columns. To *combine* with a u-unit column, a $(u + 1)$-unit column must have u of its $u + 1$ units in the same rows as the u-unit column [i.e., the u-unit column must imply the $(u + 1)$-unit column]. Thus the 1-unit columns are combined with the 2-unit columns as follows:



```
✓  ✓  |  ✓  ✓  ✓
1  0  |  1  0  1  0
0  0  |  1  1  0  1
0  1  |  0  1  0  0
0  0  |  0  0  1  1

   1  1  0
   Φ  0  Φ
   0  0  1
   0  Φ  0
```

When a pair of columns are combined, a check is placed above them to record the fact. As will be seen shortly, this checking process is of the utmost importance. In this way the adjacent collections of u-unit columns and $(u + 1)$-unit columns are successively compared and checked. In our example, this results in

√	√√	√√√√	√√√√	√
0	10	1010	1110	1
0	00	1101	1101	1
0	01	0100	1011	1
0	00	0011	0111	1

Φ0	110	11Φ011Φ0	111Φ
00	Φ0Φ	1111Φ011	11Φ1
0Φ	001	Φ0110Φ0Φ	1Φ11
00	0Φ0	0Φ0Φ1111	Φ111

The process is continued on this and each successive set of combined columns. But this time for two combined columns to be further combined they must have Φs as well as units in the same rows. The process ends when no further combinations can be formed. For our example the rest of the work is shown below:

	√√	√√√√√√√√	√√√√
Φ0	110	11Φ011Φ0	111Φ
00	Φ0Φ	1111Φ011	11Φ1
0Φ	001	Φ0110Φ0Φ	1Φ11
00	0Φ0	0Φ0Φ1111	Φ111

1	~~1~~	1	~~1~~	Φ	~~Φ~~	1	~~1~~	Φ	~~Φ~~
Φ	~~Φ~~	1	~~1~~	1	~~1~~	Φ	~~Φ~~	1	~~1~~
0	~~0~~	Φ	~~Φ~~	1	~~1~~	Φ	~~Φ~~	Φ	~~Φ~~
Φ	~~Φ~~	Φ	~~Φ~~	Φ	~~Φ~~	1	~~1~~	1	~~1~~

Duplicate columns have been crosshatched above. The prime implicants can all now be read from the *unchecked* columns. For example, $\begin{smallmatrix}\Phi\\0\\0\\0\end{smallmatrix}$ corresponds to $\bar{B} \cdot \bar{C} \cdot \bar{D}$, or $\begin{smallmatrix}1\\\Phi\\0\\\Phi\end{smallmatrix}$ corresponds to $A \cdot \bar{C}$, and so forth. For our example the prime implicants are thus $\bar{B} \cdot \bar{C} \cdot \bar{D}$, $\bar{A} \cdot \bar{B} \cdot \bar{D}$, $\bar{A} \cdot C \cdot \bar{D}$, $A \cdot \bar{C}$, $A \cdot B$, $B \cdot C$, $A \cdot D$, and $B \cdot D$.

The Second Stage—Using the Prime-implicant Chart. A chart showing the prime implicants is used in selecting for inclusion in the simplest sum of products those prime implicants which will produce the given designation number with the least number of operations · and + . The chart is simply a display of the prime-implicant designation-number unit positions. For our example this chart is shown in Table 11-1. (Note that, to determine, for instance, the unit positions of $\#\bar{B} \cdot \bar{C} \cdot \bar{D}$, we simply observe that $\bar{B} \cdot \bar{C} \cdot \bar{D}$ corresponded to $\begin{smallmatrix}\Phi\\0\\0\\0\end{smallmatrix}$. The Φ in the first row means that A was eliminated between the *two* columns $\begin{smallmatrix}0\\0\\0\\0\end{smallmatrix}$ and $\begin{smallmatrix}1\\0\\0\\0\end{smallmatrix}$; the zeros indicate that B, C, and D must be complemented in the two positions.

As another illustration, $A \cdot \bar{C}$ corresponds to $\begin{smallmatrix}1\\ \Phi\\ 0\\ \Phi\end{smallmatrix}$; the Φs indicate that B and D have been eliminated, and the zero indicates that C must be complemented.)

To choose the appropriate prime implicants for inclusion in the simplest sum of products, *first* note that $A \cdot D$ and $B \cdot D$ *must* be included because the units of $\#F$ in positions 10 and 13 are covered only by $B \cdot D$ and $A \cdot D$, respectively. If either $B \cdot D$ or $A \cdot D$ were left out of our sum, the designation number of the sum, lacking units in positions 10 or 13, could never be the same as $\#F$. Prime implicants that must for this reason be included are called *essential prime implicants.* Thus part of our sum must be $A \cdot D + B \cdot D$.

TABLE 11-1. PRIME-IMPLICANT CHART

	Unit positions of the given designation number $\#F$											
	0	1	3	4	6	7	9	10	11	13	14	15
$\bar{B} \cdot \bar{C} \cdot \bar{D}$	0	1										
$\bar{A} \cdot \bar{B} \cdot \bar{D}$	0			4								
$\bar{A} \cdot C \cdot \bar{D}$				4	6							
$A \cdot \bar{C}$		1	3				9		11			
$A \cdot B$			3			7			11			15
$B \cdot C$					6	7					14	15
$\longrightarrow A \cdot D$							9		11	⑬		15
$\longrightarrow B \cdot D$								⑩	11		14	15

Next note that $\#(A \cdot D + B \cdot D)$ covers positions 9, 10, 11, 13, 14, and 15, leaving only positions 0, 1, 3, 4, 6, and 7 to be covered by additional prime implicants. The *second* step in using the prime-implicant chart is, then, to choose in the best possible way from the *nonessential prime implicants* those combinations whose sum will cover unit positions of $\#F$ not already covered by the sum of the essential prime implicants. There is no known systematic procedure for doing this, except for trying all combinations. For our illustration it can be seen that $\bar{A} \cdot \bar{B} \cdot \bar{D} + A \cdot \bar{C} + B \cdot C$ will cover positions 0, 1, 3, 4, 6, and 7 in the best way. (Note that $\bar{B} \cdot \bar{C} \cdot \bar{D} + \bar{A} \cdot C \cdot \bar{D} + A \cdot B$ will also cover positions 0, 1, 3, 4, 6, and 7, but has one more operation $\cdot$ than the best choice.)

Thus we finally have for the simplest sum-of-products form of the designation number 1101 1011 0111 0111

$$A \cdot D + B \cdot D + \bar{A} \cdot \bar{B} \cdot \bar{D} + A \cdot \bar{C} + B \cdot C$$

Summary. Summarizing, the process of finding the simplest sum-of-products form for a given designation number $\#F$ has two stages. The first stage is the determination of all the prime implicants of #F by combining pairs of basis columns that correspond to the unit positions of #F. To aid the formation of the combinations, the columns are first ordered into a 0-unit collection, 1-unit collection, 2-unit collection, etc. The prime implicants are represented by those columns which do not enter into any further combinations, i.e., which are left unchecked when all combinations have been made. The second stage is to choose in the best way those prime implicants which are to be included in the simplest sum of products. As an aid to making this choice the prime-implicant chart is constructed, and from this essential prime implicants are noted. Then from the nonessential prime implicants a best sum is chosen to cover those units of #F not covered by the sum of the essential prime implicants. The simplest sum of products then becomes the sum of the essential prime implicants and this best sum of the nonessential prime implicants.

EXERCISES

Find the simplest sum-of-products forms of the following designation numbers:

		Solution
(*a*)	0111 0101	$A + B \cdot \bar{C}$
(*b*)	0110 1001	$A \cdot \bar{B} \cdot \bar{C} + \bar{A} \cdot B \cdot \bar{C} + \bar{A} \cdot \bar{B} \cdot C + A \cdot B \cdot C$
(*c*)	0100 0100 1111 0100	$A \cdot \bar{B} + \bar{C} \cdot D$
(*d*)	0010 0100 0010 0010	
(*e*)	0011 1010 1001 0010 1000 1101 0100 1011	

The *simplest product-of-sums* form is a product of sums that has the least number of operations + and $\cdot$. Find the simplest product-of-sums forms of each of the following designation numbers. (HINT: Work with the zeros instead of the units.)

		Solution
(*f*)	1001 1010	$(\bar{A} + B) \cdot (A + \bar{B} + C) \cdot (\bar{A} + \bar{B} + \bar{C})$
(*g*)	1111 0011 0001 1000	$(B + \bar{C} + D) \cdot (B + C + \bar{D}) \cdot (A + \bar{B} + \bar{D})$ $\cdot (\bar{A} + \bar{C} + \bar{D})$
(*h*)	0100 1001 1101 0110	
(*i*)	0110 0101 1001 1010	
(*j*)	0001 0011 0111 1111	

11-7. Obtaining Essential Prime Implicants Directly[†]

Preliminary Notational Review. Let us recall the meaning of our Φ notation. If $\substack{1\\0\\0\\0}$ is column 1 of a basis, then $\substack{1\\0\\0\\\Phi}$ represents both $\substack{1\\0\\0\\0}$ and $\substack{1\\0\\0\\1}$. Similarly $\substack{1\\\Phi\\\Phi\\\Phi}$ represents the 2^3 basis columns $\substack{1\\0\\0\\0}, \substack{1\\0\\0\\1}, \substack{1\\0\\1\\0}, \substack{1\\1\\0\\0}, \substack{1\\0\\1\\1}, \substack{1\\1\\0\\1}, \substack{1\\1\\1\\0}, \substack{1\\1\\1\\1}$. If a column has

[†] The method of this section is based on R. D. Elbourn's adaptation of a method developed by Harris. (See B. Harris, An Algorithm for Determining Minimal Representation of a Logic Function, *IRE Trans. on Electronic Computers*, vol. EC-6, pp. 103-108, June, 1957.)

Φ in p of its rows, it represents 2^p basis columns, in which the p rows are filled in all 2^p possible ways. Recall also that we interpret $\begin{smallmatrix}1\\0\\0\\\Phi\end{smallmatrix}$ to mean $A \cdot \bar{B} \cdot \bar{C}$, with D eliminated; similarly $\begin{smallmatrix}1\\\Phi\\\Phi\\\Phi\end{smallmatrix}$ means A, with B, C, and D eliminated, and so forth. We interpret a basis column as a binary number, with the least significant digit on top, and thus the rows in a basis column are associated with successive powers of 2; for example, for four variables

$$\begin{aligned} A &\leftrightarrow 2^0 = 1 \\ B &\leftrightarrow 2^1 = 2 \\ C &\leftrightarrow 2^2 = 4 \\ D &\leftrightarrow 2^3 = 8 \end{aligned}$$

Hence we shall refer to the *power position* of a bit in a column (e.g., column 1, namely, $\begin{smallmatrix}1\\0\\0\\0\end{smallmatrix}$, has a unit in power position 1, and $\begin{smallmatrix}1\\0\\0\\1\end{smallmatrix}$ has units in power positions 1 and 8). Finally, observe that the 2^p basis column numbers represented by a column with p of its components Φ can be found by taking the number formed with the Φs read as zeros and adding all combinations of powers of 2 corresponding to the power positions of the Φs. For example, $\begin{smallmatrix}1\\0\\0\\\Phi\end{smallmatrix}$ represents basis columns 1 and $1 + 8 = 9$, and $\begin{smallmatrix}1\\\Phi\\\Phi\\\Phi\end{smallmatrix}$ represents basis columns 1, $1 + 8 = 9$, $1 + 4 = 5$, $1 + 2 = 3$, $1 + 8 + 4 = 13$, $1 + 8 + 2 = 11$, $1 + 4 + 2 = 7$, and $1 + 8 + 4 + 2 = 15$.

Finding the Essential Prime Implicants Directly. In essence, in the previous simplification method we tried all combinations to obtain columns with the most Φs by a uniform "building-up" process. These columns then represented the prime implicants, and we had to make a prime-implicant table in order to choose the essential prime implicants. Now, however, we shall approach the problem from a different point of view. *We focus our attention on a single column at a time* (a column corresponding to a unit of the given designation number); *if we can show that this column is in a* single *prime implicant, then that prime implicant must be essential* (in such a case we say that the essential prime implicant is based on this column). If we make this examination for each (unit) column, then clearly we shall have determined all *essential* prime implicants, because an essential prime implicant is a prime implicant that has at least one (unit) column contained in no other prime implicant.

To find whether a column is in a single prime implicant, first find all columns that differ from it in only one power position, i.e., by only 1 bit. Suppose that there are p such power positions (that is, p columns). Then determine whether or not columns occur that have *all* possible combinations of 0 and 1 in these power positions, that is, 2^p columns altogether. If so, the original column is in an essential prime implicant.

Suppose, for example, we are given a column $\begin{smallmatrix}1\\0\\0\\0\end{smallmatrix}$. The first step is to find all columns which differ from this by only 1 bit. Suppose that columns $\begin{smallmatrix}1\\0\\0\\1\end{smallmatrix}$, $\begin{smallmatrix}1\\0\\1\\0\end{smallmatrix}$, and $\begin{smallmatrix}1\\1\\0\\0\end{smallmatrix}$ also occurred (but no others that differed from $\begin{smallmatrix}1\\0\\0\\0\end{smallmatrix}$ by 1 bit). Then—and this is the crux of our whole argument—the column $\begin{smallmatrix}1\\0\\0\\0\end{smallmatrix}$ will be in a single prime implicant (i.e., essential) only if columns $\begin{smallmatrix}1\\0\\1\\1\end{smallmatrix}$, $\begin{smallmatrix}1\\1\\0\\1\end{smallmatrix}$, $\begin{smallmatrix}1\\1\\1\\0\end{smallmatrix}$, and $\begin{smallmatrix}1\\1\\1\\1\end{smallmatrix}$ also occurred, for then we would have the total combination written as $\begin{smallmatrix}1\\\Phi\\\Phi\\\Phi\end{smallmatrix}$. The essential prime implicant would be in this case simply A. On the other hand, suppose, say, that $\begin{smallmatrix}1\\1\\1\\1\end{smallmatrix}$ were missing; then we have the following reduction by our above simplification method:

$$\begin{array}{c|ccc|ccc}1&1&1&1&1&1&1\\0&0&0&1&0&1&1\\0&0&1&0&1&0&1\\0&1&0&0&1&1&0\end{array}, \text{ then } \begin{array}{ccc|cccccc}1&1&1&1&1&1&1&1&1\\0&0&\Phi&0&\Phi&0&\Phi&1&1\\0&\Phi&0&\Phi&0&1&1&0&\Phi\\\Phi&0&0&1&1&\Phi&0&\Phi&0\end{array}, \text{ then } \begin{array}{ccc}1&1&1\\0&\Phi&\Phi\\\Phi&0&\Phi\\\Phi&\Phi&0\end{array}; \text{ that is, } \begin{smallmatrix}1\\0\\0\\0\end{smallmatrix} \text{ is in}$$

three prime implicants.

Systematic Method of Procedure. A systematic method of procedure is as follows: Write out the columns corresponding to units of the designation number. For any given column under consideration find another column that also occurs differing in 1 bit, say in power position p_1. Next find a second column that occurs differing from the original columns in 1 bit, say in power position p_2; then make sure the other column occurs that completes the 2^2 combinations of 0 and 1 in both these power positions, p_1 and p_2. Next find a third column that occurs differing in 1 bit from the original column, say in power position p_3; make sure all other columns also occur that complete all 2^3 combinations of 0 and 1 in these three power positions p_1, p_2, p_3; and so forth. This process will stop either when for a certain number of power positions all combinations do not occur (in which case our original column is in more than one prime implicant) or when no further columns occur that differ from the original column in only one power position (in which case the original column is in an essential prime implicant).

This process can be tabulated. Let us illustrate with the following designation number:

$$\begin{array}{cccccccccccccccc} &1& &3&4&5&6&7& &9& &11& &13&14&15\\0&1&0&1&1&1&1&1&0&1&0&1&0&1&1&1\end{array}$$

The first step is to make the prime-implicant table format with the successive power positions to the right,

											Power positions			
											D	*C*	*B*	*A*
1	4	3	5	6	9	7	11	13	14	15	8	4	2	1

.

We have grouped the column numbers according to the number of units in the column. Start, say, with the leftmost column as the original given column, and write its binary form (as a row for convenience) under the power positions; place an x under the 1 to record that this is the column under consideration:

											D	*C*	*B*	*A*
1	4	3	5	6	9	7	11	13	14	15	8	4	2	1
x											**0**	**0**	**0**	**1**

Now ask: Does the column occur that differs from the original column in power position 8, that is, does column 1 + 8 = 9 occur? It does; so we record this by placing an 8 under column number 9 and changing 0001 to Φ001 since both these columns do occur:

											D	*C*	*B*	*A*
1	4	3	5	6	9	7	11	13	14	15	8	4	2	1
x					**8**						**Φ**	0	0	1

Next ask: Does the column occur that differs from the original column in power position 4, that is, does column 1 + 4 = 5 occur? Yes. Do all 2^2 combinations occur? To answer this, note that 9 + 4 = 13, and column 13 does occur, so the answer is "yes." Thus we can change Φ001 to ΦΦ01, and record the work as follows:

											D	*C*	*B*	*A*
1	4	3	5	6	9	7	11	13	14	15	8	4	2	1
x			**4**		**8**			**4**			**Φ**	**Φ**	0	1

Next, try power position 2: column 1 + 2 = 3 occurs; so try 5 + 2 = 7, 9 + 2 = 11, and 13 + 2 = 15. All these columns occur, and hence all combinations of 0 and 1 occur in the three power positions 2, 4, and 8; hence we have

											D	*C*	*B*	*A*
1	4	3	5	6	9	7	11	13	14	15	8	4	2	1
x		**2**	**4**		**8**	**2**	**2**	**4**		**2**	**Φ**	**Φ**	**Φ**	1

Next try power position 1. Note that, since column 1 has a unit in power position 1, the column that differs from column 1 in this position only is column 1 − 1 = 0. But column 0 does not occur, and therefore column 1 is in a single, hence essential, prime implicant, namely, A.

At this point we are ready to see the *main advantage of the present method.* Since we have found an essential prime implicant that covers columns 1, 3, 5, 9, 7, 11, 13, and 15, *we need no longer worry about covering them.* So our problem is reduced to covering only columns 4, 6, and 14. Thus we *check off* all columns except these and choose column 4 as our next given column,

✓		✓	✓		✓	✓	✓	✓		✓	D	C	B	A	Essential
1	4	3	5	6	9	7	11	13	14	15	8	4	2	1	prime implicant
x		2	4		8	2	2	4		2	Φ	Φ	Φ	1	**A**
	x										**0**	**1**	**0**	**0**	

The process for column 4 is the same. *However, note that, in considering a power position corresponding to a zero in the original column, we* add *this power of* 2 *in order to determine what combinations of columns should occur, i.e., change the zero to a unit; if the power position corresponds to a unit in the original column, we would* subtract *this power of* 2 *in order to determine what columns should occur, i.e., change the* 1 *to a* 0. The work for column 4 becomes

✓	✓	✓	✓	✓	✓	✓	✓	✓		✓	D	C	B	A	Essential
1	4	3	5	6	9	7	11	13	14	15	8	4	2	1	prime implicant
x		2	4		8	2	2	4		2	Φ	Φ	Φ	1	A
	x		**1**	**2**		**1**					0	1	**Φ**	**Φ**	$\mathbf{\bar{D} \cdot C}$

Here column 4 + 8 = 12 did not occur, nor did column 4 − 4 = 0. But column 4 + 2 = 6 did occur, and so did columns 4 + 1 = 5 and 6 + 1 = 7. Thus we can check columns 4 and 6. Only column 14 remains unchecked; so we choose this column as the next given column. The work appears as follows:

✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	D	C	B	A	Essential
1	4	3	5	6	9	7	11	13	14	15	8	4	2	1	prime implicant
x		2	4		8	2	2	4		2	Φ	Φ	Φ	1	A
	x		1	2		1					0	1	Φ	Φ	$\bar{D} \cdot C$
				8		**1**			x	**1**	**Φ**	**1**	**1**	**Φ**	**C · B**

Column 14 − 8 = 6 occurs; column 14 − 4 = 10 does not occur; column 14 − 2 = 12 does not occur; columns 14 + 1 = 15, 6 + 1 = 7 do occur. Hence $C \cdot B$ is an essential prime implicant.

Examples and Discussion. Consider the following designation number, and let us find its simplest sum-of-products form:

```
  1   3     5 6 7   8
0 1 0 1   0 1 1 1   1 0 0 0   0 0 0 0
                   16 17   19   20 21 22 23       26 27       30
                    1  1  0 1    1  1  1  1   0 0  1  1   0 0  1 0
```

The work format appears as follows:

✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓		E	D	C	B	A	Essential
1	8	16	3	5	6	17	20	7	19	21	22	26	23	27	30	16	8	4	2	1	prime implicants
x			2	4		16		2	2	4			2			Φ	0	Φ	Φ	1	$\bar{D} \cdot A$
	x															0	1	0	0	0	$E \cdot D \cdot \bar{C} \cdot B \cdot \bar{A}$
		x				1	4			1						1	0	Φ	0	Φ	$E \cdot \bar{D} \cdot \bar{B}$
					x			1			16		1			Φ	0	1	1	Φ	$\bar{D} \cdot C \cdot B$
												x		①	4	1	1	Φ	1	0	
												1		x		1	1	0	1	Φ	$E \cdot D \cdot \bar{C} \cdot B$
											8	④			x	1	Φ	1	1	0	

Note that in adding (or subtracting) to a column number the result is always in an adjacent group of column numbers (to the right in adding, to the left in subtracting). For column 1 we find $1 + 16 = 17$, which occurs; $1 + 8 = 9$, which does not occur; $1 + 4 = 5$, $17 + 4 = 21$, both of which occur; $1 + 2 = 3$, $5 + 2 = 7$, $17 + 2 = 19$, and $21 + 2 = 23$, all of which occur; but $1 - 1 = 0$ does not occur. Thus an essential prime implicant has been found, namely, $\bar{D} \cdot A$, and column numbers 1, 3, 5, 17, 7, 19, 21, and 23 need not be used as a given column again. Note that nothing combined with column 8. When column number 26 is the given column, note that neither $26 - 16 = 10$ nor $26 - 8 = 18$ occurs; but $26 + 4 = 30$ occurs. Continuing, $26 - 2 = 24$ does not occur, but $26 + 1 = 27$ does occur; however, $30 + 1 = 31$ does not occur, and *no* essential prime implicant is found—we encircle the 1 to show that it did not combine with 30. Similarly, when 30 is the given column number, $30 - 8 = 22$ occurs, and $30 - 4 = 26$ occurs; but $22 - 4 = 18$ does not occur, and so we have encircled the 4. The only column not covered by an essential prime implicant is column 30; this can be covered in two ways, namely, by $E \cdot D \cdot B \cdot \bar{A}$ or by $E \cdot C \cdot B \cdot \bar{A}$. Hence the result is

$$A \cdot \bar{D} + \bar{A} \cdot \bar{B} \cdot \bar{C} \cdot D \cdot \bar{E} + \bar{B} \cdot \bar{D} \cdot E + B \cdot C \cdot \bar{D} + B \cdot \bar{C} \cdot D \cdot E + \begin{cases} \bar{A} \cdot B \cdot D \cdot E \\ \bar{A} \cdot B \cdot C \cdot E \end{cases}$$

Consider a six-variable problem with units in the following designation-number positions: 1, 3, 4, 5, 7, 13, 17, 19, 20, 23, 31, 33, 35, 36, 37, 39, 44, 45, 49, 51, 52, 53, 55, 60, and 61. The work is shown in Table 11-2, in which the columns are grouped appropriately. Here we have introduced a variation of the present method. For example, starting with column number 1 we find that $1 + 32 = 33$ occurs; $1 + 16 = 17$ and $33 + 16 = 49$ both occur; $1 + 8 = 9$ does not occur; but although $1 + 4 = 5$ occurs, $17 + 4 = 21$ does not occur, and so we encircle 4. Now instead of proceeding as above let us see if we can at this point make further use of the information so far obtained, namely, Φ Φ 0 0 0 1. To do this, suppose that we transfer to column number 17 as our given column; we indicate this transfer by putting a slash through the 16 below column number 17, as is shown. Then Φ Φ 0 0 0 1 still holds for column 17, and we need not try power 32 or power 16 again or, for that matter, power 4 (why?). But we must try power 8 again, for we have never tried it on this column (why not?). Thus $17 + 8 = 25$ does not occur; but $17 + 2 = 19$, $1 + 2 = 3$, $33 + 2 = 35$, $49 + 2 = 51$ all occur, giving us Φ Φ 0 0 Φ 1. Finally $17 - 1 = 16$ does not occur; so we have discovered an essential prime implicant, *based on column* 17 rather than on the column we started with, column 1. The next two rows of work in Table 11-2 also indicate a transfer to some other column number in finding the essential prime implicant.

Next observe the fourth row of work. Here we meet several further circumstances, which we shall explain by means of the illustration. The original column number is 7; $7 + 32 = 39$ occurs; $7 + 16 = 23$ and $39 + 16 = 55$ occur; $7 + 8 = 15$ does not occur; $7 - 4 = 3$, $23 - 4 = 19$, $39 - 4 = 35$, and $55 - 4 = 51$ all occur, giving so far Φ Φ 0 Φ 1 1. Next observe that $7 - 2 = 5$, $3 - 2 = 1$, $19 - 2 = 17$, $35 - 2 = 33$ all occur but that $23 - 2 = 21$ does not. Transferring to 23 as above, we need not try powers 32, 16, 4, or 2 again. Trying 8, we find that $23 + 8 = 31$ occurs but that $3 + 8 = 11$ does not occur. We could try to transfer once again, completing the crossing off of the 16 below column number 23 for the record. *But we must transfer to a column with the following properties:* (1) *it is not covered by a previously determined essential prime implicant and* (2) *it does not combine with powers that have already been tried and circled.* If the transfer were made to a column already covered, then this column obviously could not be the basis for another essential prime implicant; if the column to which we transferred combined with a circled power, then this power clearly would not again allow the completion of an essential prime implicant. For our illustration it is seen that no further column satisfying Φ Φ 0 Φ 1 1 as well as our conditions 1 and 2 exists. Thus we must go on to a new row of work, i.e., another uncovered column. Note that since we already have determined that no column satisfying Φ Φ 0 Φ 1 1 can be the basis of an essential prime implicant we should not go on to any of these columns either. For our illustration the next given column will be 44, and the rest of the work is completed as above.

EXERCISES

Find the simplest sum-of-products forms of the following designation numbers by the method of this section:

(*a*) 0111 0101.
(*b*) 0110 1001.
(*c*) 0100 0100 1111 0100.
(*d*) 0010 0100 0010 0010.
(*e*) 0011 1010 1001 0010 1000 1101 0100 1011.

Find the *simplest product-of-sums* forms of each of the following designation numbers. (HINT: Work with the zeros instead of the units.)

(*f*) 1001 1010.

Table 11-2. The Simplest Sum of Products Is $A \cdot \bar{C} \cdot \bar{D} + \bar{A} \cdot \bar{B} \cdot C \cdot \bar{D} + A \cdot \bar{B} \cdot C \cdot \bar{E} + \bar{B} \cdot C \cdot F + A \cdot B \cdot C \cdot E \cdot \bar{F} + \bar{D} \cdot B \cdot A$, Where Only the Last Term Is Not an Essential Prime Implicant†

✓	✓	✓	✓	✓	✓	✓	✓		✓	✓	✓	✓	✓	✓	✓	✓		✓	✓	✓	✓	✓		✓	*F*	*E*	*D*	*C*	*B*	*A*	Essential prime implicants
1	4	3	5	17	20	33	36	7	13	19	35	37	44	49	52	23	39	45	51	53	60	31	55	61	32	16	8	4	2	1	
x		2	④	~~16~~		32				2	2			16					2						Φ	Φ	0	0	Φ	1	$\bar{D} \cdot \bar{C} \cdot A$
	x		①		~~16~~		32								16										Φ	Φ	0	1	0	0	$\bar{D} \cdot C \cdot \bar{B} \cdot \bar{A}$
④			x						8			32						8							Φ	0	Φ	1	0	1	$\bar{E} \cdot C \cdot \bar{B} \cdot A$
②		4	②	②		②		x		4	4					~~16~~	32		4			⑧	16		Φ	Φ	0	Φ	1	1	
							8					1	x		8			1		1	16			1	1	Φ	Φ	1	0	Φ	$F \cdot C \cdot \bar{B}$
																8						x			0	1	Φ	1	1	1	$\bar{F} \cdot E \cdot C \cdot B \cdot A$

† Note that Φ Φ 0 Φ 1 1, corresponding to $\bar{D} \cdot B \cdot A$, in the fourth row covers the unchecked columns. By our method of procedure this is as good as any other product which covers them, and hence we include $\bar{D} \cdot B \cdot A$ in our simplest sum of products.

(*g*) 0100 1001 1101 0110.
(*h*) 1111 0011 0001 1000.
(*i*) 0110 0101 1001 1010.
(*j*) 0001 0011 0111 1111.

11-8. Simplification of Computer-circuit Design

The Simplification Procedure. The problem of the simplification of computer circuits can be stated as follows: Given a computer-circuit design, how can this circuit be redesigned using fewer *and* gates and *or* gates such that the output of the newly designed circuit is the same Boolean function (of the inputs) as the output of the original design?

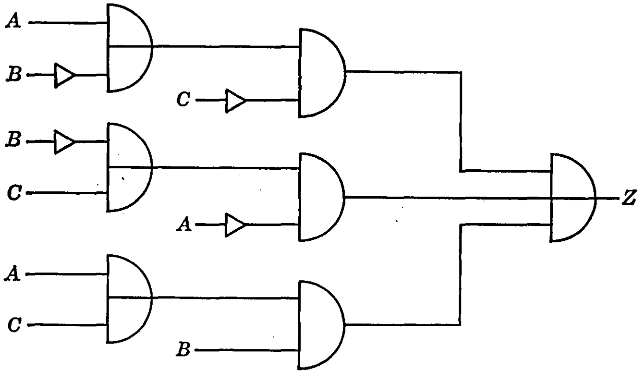


FIG. 11-5. Circuit for $Z = \bar{C} \cdot (A + \bar{B}) + \bar{A} \cdot (\bar{B} + C) + B \cdot (A + C)$.

We shall illustrate the procedure by means of an example. Consider the circuit of Fig. 11-5.

By the methods of Sec. 10-10

$$Z = \bar{C} \cdot (A + \bar{B}) + \bar{A} \cdot (\bar{B} + C) + B \cdot (A + C)$$

With respect to the standard basis we then find $\#Z = 1101\ 1011$. One simplest sum-of-products form is $\bar{A} \cdot \bar{B} + A \cdot \bar{C} + B \cdot C$, whence the simplified circuit can be made as in Fig. 11-6.

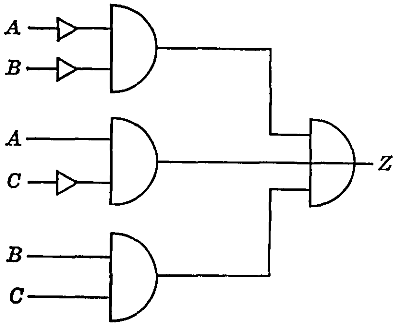


FIG. 11-6. Simplest sum-of-products form for Z of Fig. 11-5.

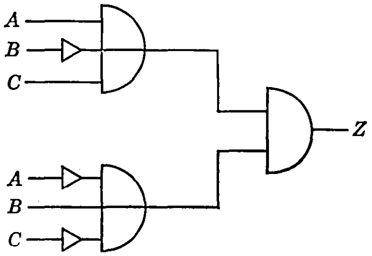


FIG. 11-7. Simplest product-of-sums form for Z of Fig. 11-5.

On the other hand the simplest product-of-sums form is

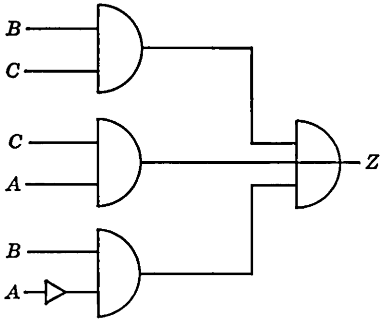
$$(A + \bar{B} + C) \cdot (\bar{A} + B + \bar{C})$$

and another simplified circuit is as in Fig. 11-7.

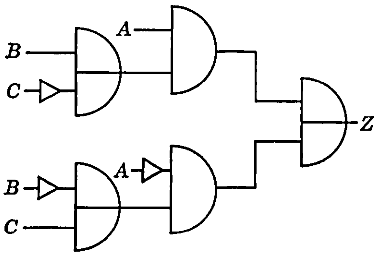
Summary. The method for simplifying a circuit is *first* to form the Boolean function for the output in terms of the inputs, *second* to form the designation number of this function, *third* to find the simplest sum-of-products or product-of-sums form for this designation number, and *fourth* to draw the circuit diagram corresponding to the simplest form.

EXERCISES

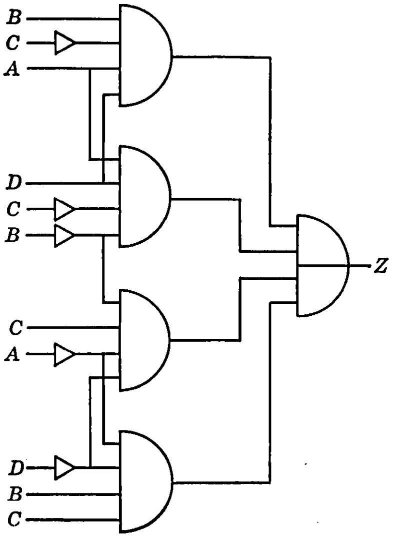
Simplify the following circuits:



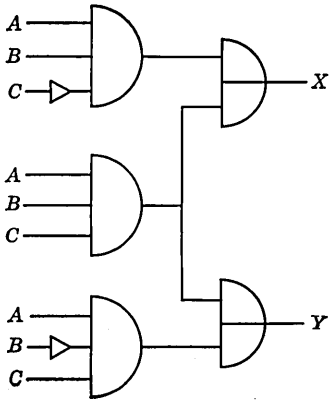
(a)



(b)



(c)



(d)

This last problem is of interest since it has two outputs, X and Y. The procedure is to treat each output separately at first, obtaining the designation number and the simplest form for each output. Then those wires which are common to both outputs (written in simplest form) are utilized in this dual capacity. Hence A goes to gates for both X and Y.

11-9. The Design of Circuits That Compute Functions

The Condition-function Table and the Basis. The following discussion about the relationship between the condition-function table (of Sec. 10-8) and the basis (of Sec. 11-2), albeit seeming trivial perhaps in the light of the contents of this chapter, is nevertheless of fundamental importance to the whole concept of the design of computer circuits. A clear understanding of this relationship should be grasped by the student.

The relationship between the function table and the basis is analogous to the relationship of the truth tables and the basis. The columns of both function table and basis are composed of all possible combinations of the n units and zeros, for a system of n elementary elements. However, the *order* of the columns of a *basis* is fixed. *Hence, if the function table is considered as a basis, with the input wires as elementary elements, the output-wire row giving the voltage signal* (*as* 0 *or* 1) *becomes simply a designation number*—in fact *the* designation number of the Boolean function of the output in terms of the input wires.

For example, consider the problem of Sec. 10-8. The condition-function table of the solution was

Wire A.........	0	1	0	1	0	1	0	1
Wire B.........	0	0	1	1	0	0	1	1
Wire C.........	0	0	0	0	1	1	1	1
Wire D.........	1	1	1	1	0	0	0	1

where A, B, C are input wires and D is the output. In order to reinterpret this function table as a basis and designation number, we let the designation numbers of the input wires be the same as the designation numbers of the elementary elements. Hence we have

$$\begin{array}{l}
\#A = 0101\ \ 0101 \\
\#B = 0011\ \ 0011 \\
\underline{\#C = 0000\ \ 1111} \\
\#D = 1111\ \ 0001
\end{array}$$

In terms of A, B, C we have $D = \bar{C} + A \cdot B$; that is, the designation number of D with respect to the standard basis for three elementary elements is $\#D = 1111\ 0001 = \#(\bar{C} + A \cdot B)$. *Henceforth we shall use the basis notation where its meaning as a condition-function table will be understood.*

As another example, consider the following condition-function table:

$$\begin{array}{l}
\#A = 0101 \quad 0101 \\
\#B = 0011 \quad 0011 \\
\#C = 0000 \quad 1111 \\
\hline
\#Z = 1010 \quad 1011
\end{array}$$

where A, B, and C are the inputs and Z is the output. To find Z in terms of A, B, and C, consider Z as a designation number, whence we have $Z = \bar{A} + B \cdot C$ as the simplest sum-of-products form.

Verbally Described Problems. In general the condition-function table is *not* given explicitly; rather some verbal description is given from which the function table can be constructed. Once the function table has been constructed, the Boolean function of the outputs is determined, usually in simplest form. Finally the circuit diagram is drawn. This circuit is then a circuit that will perform the duties described in the original verbal form of the problem.

For example, suppose that it is desired to design a circuit with three inputs and one output, such that, whenever an even number of inputs have a unit signal voltage, so will the output; otherwise the output is to have a zero signal voltage. The function table thus becomes

$$\begin{array}{l}
\#A = 0101 \quad 0101 \\
\#B = 0011 \quad 0011 \\
\#C = 0000 \quad 1111 \\
\hline
\#Z = 1001 \quad 0110
\end{array}$$

whence $Z = (A \cdot B + \bar{A} \cdot \bar{B}) \cdot \bar{C} + (A \cdot \bar{B} + \bar{A} \cdot B) \cdot C$. Hence the circuit of Fig. 11-8 will do the job.

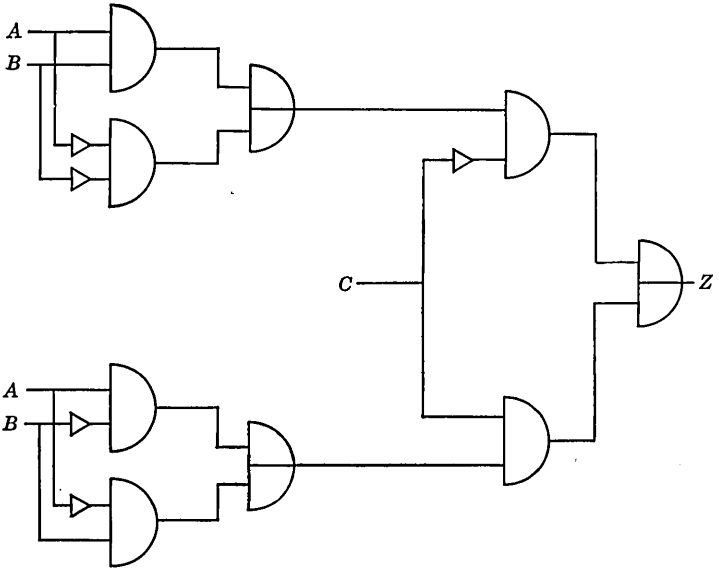


FIG. 11-8. Circuit for $Z = (A \cdot B + \bar{A} \cdot \bar{B}) \cdot \bar{C} + (A \cdot \bar{B} + \bar{A} \cdot B) \cdot C$. Z is 1 volt when an even number of inputs are at 1 volt.

In many problems the voltage on the input wires is taken to represent a bit in a binary number. For example, if there were three input wires, the voltage on wire A might represent the bit in the least significant position of a 3-bit binary number, B the next most significant bit, and C the most significant bit. For instance, to represent 5, or binary 101, wire A would have a 1-volt signal, wire B a 0-volt signal, and wire C a 1-volt signal. Similarly the output wires would represent the bits of an output binary number.

Suppose that we are to design a circuit that forms $3N$, where N represents a 3-bit number. Since N can go only up to 7, then the output can go only up to 21, and five output wires are necessary. Let N_1, N_2, and N_3 represent the input wires (instead of A, B, C as described above), and let M_1, M_2, M_3, M_4, and M_5 represent the output wires (see Fig. 11-9).

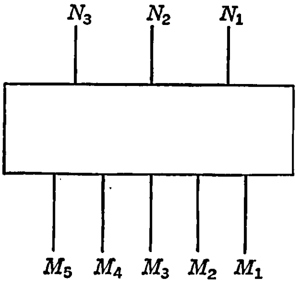


FIG. 11-9. Input-output wires of circuit to form $M = 3N$.

Now consider

$$\begin{array}{rl} & 0123\quad 4567 \\ \#N_1 = & 0101\quad 0101 \\ \#N_2 = & 0011\quad 0011 \\ \#N_3 = & 0000\quad 1111 \end{array}$$

The columns represent the possible states of the wires N_1, N_2, and N_3. In fact, if the input number were 5, then the state of the wires would be as shown in column 5, that is, the column would form the binary number 5 with the top row the least significant bit, the bottom row the most significant bit. Now, if $N = 0$, then $3N = 0$; if $N = 1$, $3N = 3$, etc. Hence we have

$$\begin{array}{rl} N = & 0, 1, 2, 3, 4, 5, 6, 7 \\ 3N = & 0, 3, 6, 9, 12, 15, 18, 21 \end{array}$$

or

$$\begin{array}{rl} \#N_1 = & 0101\quad 0101 \\ \#N_2 = & 0011\quad 0011 \\ \#N_3 = & 0000\quad 1111 \\ \hline \#M_1 = & 0101\quad 0101 \\ \#M_2 = & 0110\quad 0110 \\ \#M_3 = & 0010\quad 1101 \\ \#M_4 = & 0001\quad 1100 \\ \#M_5 = & 0000\quad 0011 \end{array}$$

This tabulation gives for any state of the N_1, N_2, and N_3 wires the desired state of the M_1, M_2, M_3, M_4, and M_5 wires. For instance, if $N = 5$, then $3N = 15$; that is, if

	Volts
Wire N_1 =	1
Wire N_2 =	0
Wire N_3 =	1

then

	Volts
Wire M_1	= 1
Wire M_2	= 1
Wire M_3	= 1
Wire M_4	= 1
Wire M_5	= 0

where of course 101 is 5, 1111 is fifteen, in binary. Now we can read off the answers:

$$M_1 = N_1$$
$$M_2 = N_1 \cdot \bar{N}_2 + \bar{N}_1 \cdot N_2$$
$$M_3 = \bar{N}_1 \cdot N_2 \cdot \bar{N}_3 + (N_1 + \bar{N}_2) \cdot N_3$$
$$M_4 = N_1 \cdot N_2 \cdot \bar{N}_3 + \bar{N}_2 \cdot N_3$$
$$M_5 = N_2 \cdot N_3$$

The circuit then becomes as in Fig. 11-10.

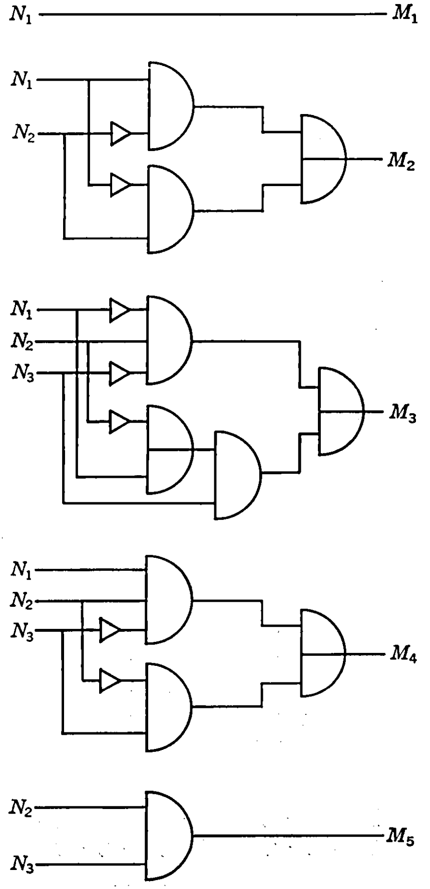


FIG. 11-10. Circuitry for the output wires of M.

Consider now a more complicated problem. Suppose that we are to design a circuit to have four inputs, N_1, N_2, M_1, and M_2. Let N_1 and N_2 represent a 2-bit binary number N with N_1 the least significant bit, and let M_1 and M_2 represent a binary number M with M_1 the least significant bit. The circuit is to produce the product of the two input numbers. Since each input number has 2 bits, the largest product is $3 \times 3 = 9$. Hence 4 bits for the resulting product outputs, i.e., four output wires, are sufficient. Hence a listing of the states becomes

N	0, 1, 2, 3	0, 1, 2, 3	0, 1, 2, 3	0, 1, 2, 3
M	0, 0, 0, 0	1, 1, 1, 1	2, 2, 2, 2	3, 3, 3, 3
$N \times M$	0, 0, 0, 0	0, 1, 2, 3	0, 2, 4, 6	0, 3, 6, 9

whence the function table becomes

$$\begin{array}{l}
N\left\{\begin{array}{l} \#N_1 = 0101 \quad 0101 \quad 0101 \quad 0101 \\ \#N_2 = 0011 \quad 0011 \quad 0011 \quad 0011 \end{array}\right. \\
M\left\{\begin{array}{l} \#M_1 = 0000 \quad 1111 \quad 0000 \quad 1111 \\ \#M_2 = 0000 \quad 0000 \quad 1111 \quad 1111 \end{array}\right. \\
\hline
N \times M\left\{\begin{array}{l} \#R_1 = 0000 \quad 0101 \quad 0000 \quad 0101 \\ \#R_2 = 0000 \quad 0011 \quad 0101 \quad 0110 \\ \#R_3 = 0000 \quad 0000 \quad 0011 \quad 0010 \\ \#R_4 = 0000 \quad 0000 \quad 0000 \quad 0001 \end{array}\right.
\end{array}$$

Hence we have

$$\begin{aligned}
R_1 &= N_1 \cdot M_1 \\
R_2 &= N_2 \cdot M_1 \cdot \bar{M}_2 + N_1 \cdot \bar{M}_1 \cdot M_2 + \bar{N}_1 \cdot N_2 \cdot M_1 + N_1 \cdot \bar{N}_2 \cdot M_2 \\
R_3 &= N_2 \cdot \bar{M}_1 \cdot M_2 + \bar{N}_1 \cdot N_2 \cdot M_2 \\
R_4 &= N_1 \cdot N_2 \cdot M_1 \cdot M_2
\end{aligned}$$

The multiplying circuit can then be drawn from this result.

Consider an example of a decoding circuit. Suppose that the input code is 3 bits (i.e., three input wires) and that the output code is 3 bits. The relation between the input code and output code is

Input code	0123	4567
Output code	0647	0377

i.e., if 1 goes in, then 6 must come out; if 5 goes in, 3 must come out, etc. The function table for this decoding scheme is

$$\begin{array}{l}
\#A = 0101 \quad 0101 \\
\#B = 0011 \quad 0011 \\
\#C = 0000 \quad 1111 \\
\hline
\#X = 0001 \quad 0111 \\
\#Y = 0101 \quad 0111 \\
\#Z \doteq 0111 \quad 0011
\end{array}$$

whence

$$\begin{aligned}
X &= A \cdot B + A \cdot C + B \cdot C \\
Y &= A \cdot C + B \cdot C + A \cdot \bar{C} \\
Z &= B \cdot C + A \cdot \bar{C} + \bar{C} \cdot B
\end{aligned}$$

and the circuit diagram appears in Fig. 11-11.

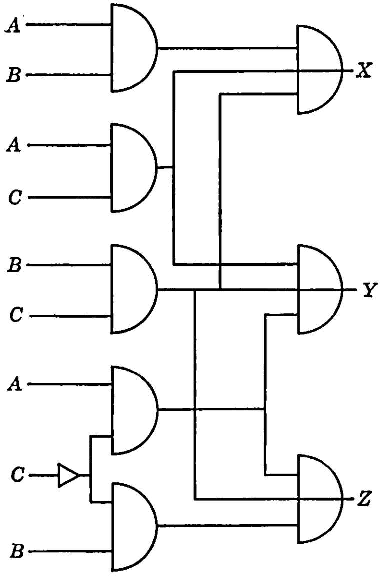


FIG. 11-11. A decoding circuit. Input code of 0123 4567 corresponds to output code of 0647 0377.

EXERCISES

Design circuits that perform the following functions:

(*a*) A parallel digital squaring circuit where the input is a 4-bit binary number N and the output is N^2.

(*b*) A circuit that forms the positive values of $2N - 1$ for N a 4-bit binary number (and forms zero for $N = 0$).

(*c*) A circuit the input of which is the 4-bit binary number N, the output of which is $N + 3$ in binary.

(*d*) A circuit that will produce a unit output only when an odd number of its four input wires contain unit signal voltages.

11-10. Synchronous Recursive-function Circuits

Input as a Function of the Output. Until this section the output of a circuit depended on the inputs, but the inputs of the circuit have not depended in any way on the output of the circuit. In this section circuits will be considered in which the inputs depend on the outputs. Foı example, consider a digital-computer-circuit black box with three inputs and three outputs such that the outputs are connected to the inputs (see Fig. 11-12). Hence the future output depends on the past output. Such an output function is called a *recursive function.* Two kinds of inputs can be distinguished, *external inputs,* which are not related to the output, and *internal inputs,* which are related to the output. In Fig. 11-12 P is an external input, and A, B, and C are internal inputs.

Consider the following example of a recursive-function circuit: Let A and B represent a 2-bit binary-number input in the manner of the last section, and let A' and B' represent a 2-bit output number. Let P represent another input wire which is attached to a push button; i.e., when the push button is pushed, a unit signal voltage will appear on P. The circuit is to work as follows: Suppose that initially the output "number" is 0; when the button is pushed, the output should change to 1; when the button is pushed a second time, the output is to change to 2; when pushed a third time, the output should change to 3; and when pushed again, the output should return to 0 again. This cycle can be repeated. Such a circuit is called a *counter* since it counts how many times the button is pushed (but only up to three!).

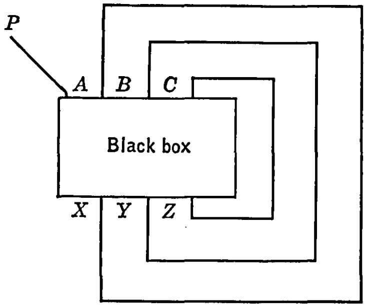


FIG. 11-12. A recursive circuit with four inputs and three outputs.

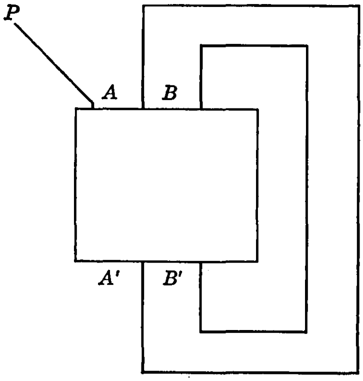


FIG. 11-13. A two-digit counter.

The output number is to become the future input number, i.e., the circuit is to be wired as in Fig. 11-13. Hence the output immediately becomes the internal input. The function table becomes

	0123	4567
$\#A =$	0101	0101
$\#B =$	0011	0011
$\#P =$	0000	1111
$\#A' =$	0101	1010
$\#B' =$	0011	0110

To see how this function table was derived, consider, for example, column 2. Here the input number is $\genfrac{}{}{0pt}{}{0}{1}$, or 2, and the push button is not being pushed. Let us assume that, if the push button is not pushed, then the output number is the same as the input number. Hence the output number becomes also $\genfrac{}{}{0pt}{}{0}{1}$, or 2. A similar discussion also holds for columns 0, 1, and 3.

Now let us consider column 6, for example. Here the input number is $\frac{0}{1}$, or 2, and this time the push button is being pushed. Hence the output number must be 3, or $\frac{1}{1}$ as indicated in column 6. Similarly in column 4, the input number of 0 becomes the output number of 1, in column 5 the input number of 1 becomes the output number of 2, and in column 7 the input number of 3 becomes the output number of 0.

We can now observe that

$$\#A' = 0101 \quad 1010 = \#(A \cdot \bar{P} + \bar{A} \cdot P)$$
$$\#B' = 0011 \quad 0110 = \#[B \cdot \bar{P} + (A \cdot \bar{B} + \bar{A} \cdot B) \cdot P]$$

Timing. But wait! There is something wrong here! Suppose that the output number was, say, 2. Then immediately the input number will become 2 since A is connected directly to A' and B is connected directly to B'. If P is not being pushed, then everything stays in this state; i.e., the output that corresponds to an input of 2, with no button being pushed, is just 2. However, suppose that now the button is pushed. The output immediately changes to 3; this 3 immediately becomes the input. Suppose that the button is still being pushed; then the output immediately changes to 0, etc., before the button can be released. But we only wanted the output to change from 2 to 3 during a single push of the button.

The most obvious way out of this dilemma is to delay the output from becoming the new input for a long enough time so that the push button can be released. In such a case this is what will happen: Suppose initially, for example, that both output and input were 2 and the button is pushed. Then the output immediately becomes 3, but not the input; it stays at 2 because we now have a delay in propagation of the signals along the wires $A'A$ and $B'B$. Then the button is released, after which the 3 propagates to the input. But an input of 3 with the button not pushed produces an output of 3 as desired. Hence by the introduction of the delay the circuit has been made to do what we desired. The wiring of the circuit will now look like Fig. 11-14, where the delay is represented by a rectangle with a number in it that represents the time of the delay in microseconds.

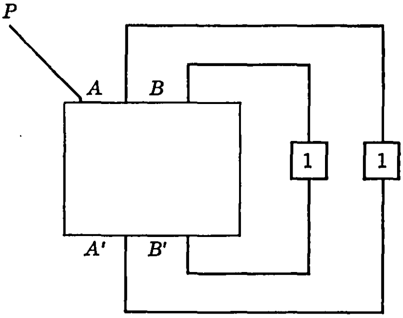


FIG. 11-14. Circuit of Fig. 11-13 with unit delays.

Hence the only other necessary condition is that the duration of the push-button signal is shorter than the delay. Of course the push-button signal is not really manually operated: it is an external electronic signal

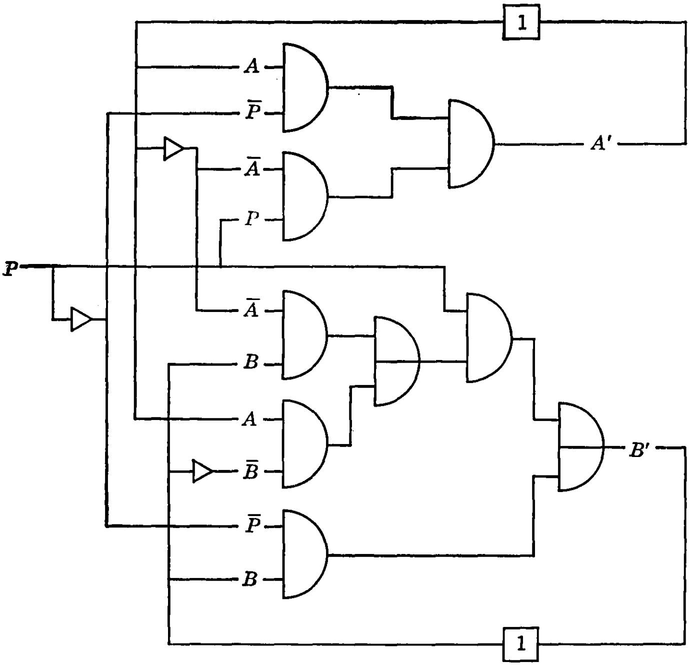


FIG. 11-15. The detailed circuitry of the two-digit counter.

generated in some other circuit. Using our Boolean-function derivation above, we can now draw the complete circuit diagram as in Fig. 11-15. Circuitry that depends on the synchronization of the pulses in this manner is called *synchronous circuitry.*

The Timing Diagram. The time sequencing of signals in the wires A, B, P, A', and B' can be visualized by the voltage-time diagram. Here the voltage as a function of time is plotted for each wire, as shown in

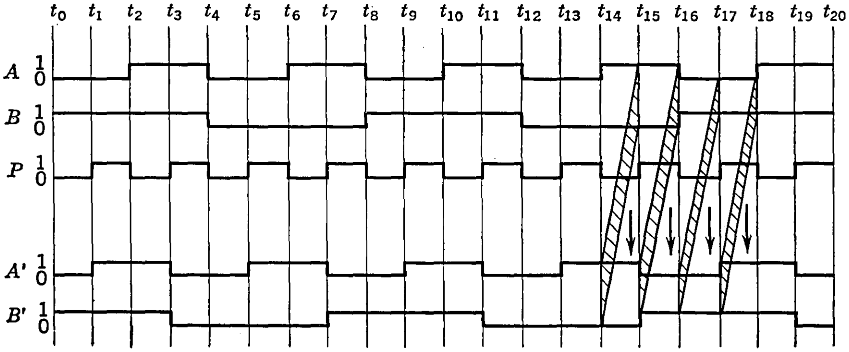


FIG. 11-16. Voltage levels vs. time for the two-digit counter.

Fig. 11-16. Figure 11-17 illustrates the voltage levels in wires A and B when the 2-bit binary number represented by A and B (with A the least significant bit) is 0, 1, 2, and 3, respectively. Let us return now to Fig. 11-16. At time t_0, suppose that both A, B and A', B' represent the "number" 2 and that the button was not being pushed. At t_1 (say 1 μsec later) the button is pushed; A', B' then form the "number" 3 almost immediately, but A, B stay the same. At t_2 the 3 has propagated through to A, B. Note that A', B' stay the same at 3 since A, B form 3 and the button has been released. When the button is again pushed at t_3, A', B' form 0 but A, B stay at 3 since the 0 has not propagated through to them yet. By t_4 this propagation has occurred, and A, B form 0, etc.

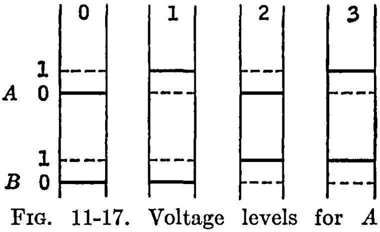


FIG. 11-17. Voltage levels for A and B.

Summary. Given the voltage levels of A, B, and P at t_i, the voltage levels for A' and B' at t_i are determined by means of the function table; the voltage levels of A and B at t_{i+1} are the same as those of A' and B' were at t_i (see crosshatching and arrows in Fig. 11-16).

We have shown how to design a recursive-function digital-computer circuit, i.e., a circuit whose present input depends on its past output. The computational procedure is almost the same as for the nonrecursive case discussed in the previous section. However, delay lines have to be included in the circuit to delay the output from changing the internal input too fast, and any external input signal must have a duration a little less than the delay time.

EXERCISES

(*a*) Consider a circuit with one internal input A, one output A', and one external input P, again a push button. If the input has a 1-volt signal and the button is pushed (i.e., wire P has a 1-volt signal), then the output changes to zero. If the output is zero volts and the button is pushed, the output signal changes to 1 volt. Design this circuit.

Solution

$$\begin{aligned} \#A &= 0101 \\ \#P &= 0011 \\ \hline \#A' &= 0110 = A \cdot \bar{P} + \bar{A} \cdot P \end{aligned}$$

The circuit diagram is as shown in Fig. 11-18.

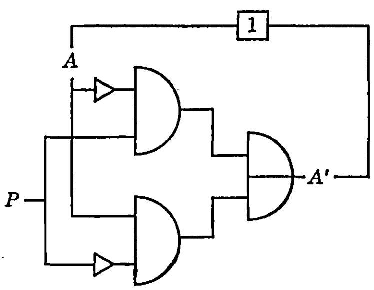


FIG. 11-18. Solution to Exercise *a*.

(*b*) Design a circuit that counts as follows: 0, 1, 3, 2, 0, 1, 3, 2, . . . (i.e., the output wires, say A', B', form a 2-bit binary number; the count is stepped by a push button satisfying the above requirements).

Solution. The function table is

$$\begin{array}{rl}
\#A &= 0101 \quad 0101\\
\#B &= 0011 \quad 0011\\
\#P &= 0000 \quad 1111\\
\hline
\#A' &= 0101 \quad 1100 = A \cdot \bar{P} + \bar{B} \cdot P\\
\#B' &= 0011 \quad 0101 = B \cdot \bar{P} + A \cdot P
\end{array}$$

The circuit diagram is as shown in Fig. 11-19.

(*c*) Draw the voltage-time diagram for Exercise *b*.

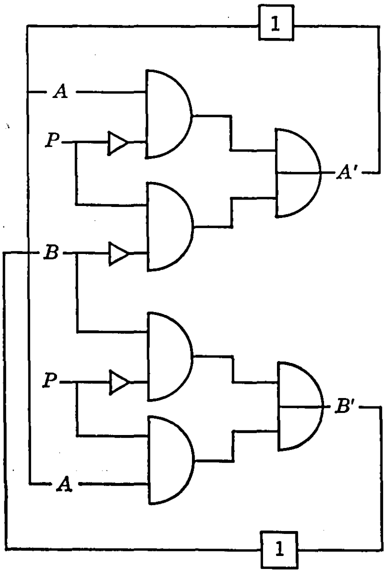


FIG. 11-19. Solution to Exercise *b*.

11-11. The States of Circuits

Boolean Matrices. Boolean matrices are involved in the discussion of the states of circuits. A *Boolean matrix* is a rectangular array of zeros and units, for example,

$$\begin{pmatrix} 01010 \\ 10010 \\ 01011 \end{pmatrix}$$

The columns are labeled by successively increasing integers starting with 0, from left to right. The rows are similarly labeled from top to bottom:

$$\begin{array}{cc} & j \; 01234 \\ i \; \begin{array}{c} 0 \\ 1 \\ 2 \end{array} & \begin{pmatrix} 01010 \\ 10010 \\ 01011 \end{pmatrix} \end{array}$$

The zeros and units are called the *elements* of the matrix. An element of the matrix can be represented by a_{ij}, where i is the row in which the element appears and j is the column in which it appears. (The mnemonic *R*oman *C*atholic can be used to recall that the first subscript refers to the row, the second subscript to the column.) Thus in our example $a_{13} = 1$, and $a_{20} = 0$. The whole matrix is represented symbolically by placing parentheses around the symbol for an element, thus: (a_{ij}). In summary a Boolean matrix is the same as the usual concept of a matrix except that the elements can only be *zero* or *unit* and the numbering of the rows and columns starts with 0.

Boolean matrix multiplication, indicated by the symbol $\otimes$, follows the same rules as ordinary matrix multiplication except that *logical multiplication and logical addition of the elements are employed.* That is to say, if (c_{ij}) is the matrix resulting from the Boolean multiplication of the matrix (a_{ik}) by the matrix (b_{kj}), that is, $(c_{ij}) = (a_{ik}) \otimes (b_{kj})$, then the elements of (c_{ij}) are determined by

$$c_{ij} = \sum_k a_{ik} \cdot b_{kj}$$

where $\cdot$ represents logical multiplication and $+$ now represents logical addition. For example, if

$$(a_{ik}) = \begin{array}{c} \\ 0 \\ 1 \\ 2 \end{array}\!\!\begin{array}{c} \begin{array}{ccccc} 0 & 1 & 2 & 3 & 4 \end{array} \\ \begin{pmatrix} 0 & 1 & 0 & 1 & 0 \\ 1 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 & 1 \end{pmatrix} \end{array} \qquad \text{and} \qquad (b_{jk}) = \begin{array}{c} \\ 0 \\ 1 \\ 2 \\ 3. \\ 4 \end{array}\!\!\begin{array}{c} \begin{array}{ccc} 0 & 1 & 2 \end{array} \\ \begin{Bmatrix} 1 & 0 & 1 \\ 0 & 0 & 0 \\ 1 & 1 & 0 \\ 0 & 0 & 0 \\ 0 & 1 & 1 \end{Bmatrix} \end{array}$$

then

$$\begin{pmatrix} 0 & 1 & 0 & 1 & 0 \\ 1 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 & 1 \end{pmatrix} \otimes \begin{Bmatrix} 1 & 0 & 1 \\ 0 & 0 & 0 \\ 1 & 1 & 0 \\ 0 & 0 & 0 \\ 0 & 1 & 1 \end{Bmatrix} = \begin{array}{c} \\ 0 \\ 1 \\ 2 \end{array}\!\!\begin{array}{c} \begin{array}{ccc} 0 & 1 & 2 \end{array} \\ \begin{pmatrix} 0 & 0 & 0 \\ 1 & 0 & 1 \\ 0 & 1 & 1 \end{pmatrix} \end{array} = (c_{ij})$$

To form c_{00}, note that

$$c_{00} = \sum_k a_{0k} \cdot b_{k0} = a_{00} \cdot b_{00} + a_{01} \cdot b_{10} + a_{02} \cdot b_{20} + a_{03} \cdot b_{30} + a_{04} \cdot b_{40}$$

EXERCISES

Show that

(*a*) $$\begin{pmatrix} 1 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \end{pmatrix} \otimes \begin{pmatrix} 1 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & 1 & 0 \end{pmatrix} = \begin{pmatrix} 1 & 1 & 0 \\ 0 & 0 & 1 \\ 0 & 0 & 0 \end{pmatrix}$$

(*b*) $$\begin{pmatrix} 1 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \end{pmatrix} \otimes \begin{pmatrix} 1 \\ 0 \\ 1 \\ 0 \end{pmatrix} = \begin{pmatrix} 1 \\ 1 \\ 0 \end{pmatrix}$$

(*c*) $$(0 \quad 1 \quad 1) \otimes \begin{pmatrix} 1 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \end{pmatrix} = (0 \quad 1 \quad 1 \quad 0)$$

(*d*) $$\begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \otimes \begin{pmatrix} 1 & 1 & 0 \\ 0 & 0 & 1 \\ 0 & 1 & 0 \end{pmatrix} = \begin{pmatrix} 1 & 1 & 0 \\ 0 & 0 & 1 \\ 0 & 1 & 0 \end{pmatrix}$$

States of a Circuit. Now we are ready to consider the definition of "states of a circuit." First let us restrict ourselves to recursive circuits where the outputs are connected to the inputs through delay lines; in addition suppose that the circuits have *no* external input and have the same number of output wires as input wires, as in Fig. 11-20. As we saw in the last section, at a given time the signals in the wires A', B', C'

are not necessarily the same as those in A, B, C. Hence we associate two "states" with a circuit: the state of the input wires and the state of the output wires. The *state of the input wires* at any given time is defined as the set of signals in the input wires at that time; the *state of the output wires* at any time is the set of signals in the output wires at that time. Recalling that the columns of the function-table basis give all possible states of the input wires, we can use the column number of the standard basis for a shorthand way of representing a state.

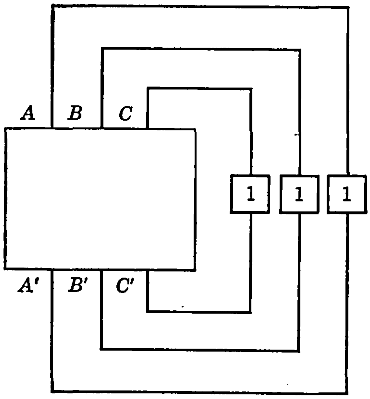


FIG. 11-20. Recursive circuit with no external inputs.

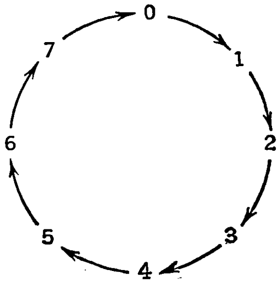


FIG. 11-21. Input-state diagram of circuitry: $A' = \bar{A}$ $B' = \bar{A} \cdot B + A \cdot \bar{B}$ $C' = A \cdot B \cdot \bar{C} + (\bar{A} + \bar{B}) \cdot C$

For example, suppose that the function table for a circuit were

$$\begin{array}{rl}
 & 0123\ \ 4567 \\
\#A = & 0101\ \ 0101 \\
\#B = & 0011\ \ 0011 \\
\#C = & 0000\ \ 1111 \\
\hline
 & 1234\ \ 5670 \\
\#A' = & 1010\ \ 1010 \\
\#B' = & 0110\ \ 0110 \\
\#C' = & 0001\ \ 1110
\end{array}$$

Then we let 0 represent the state $\overset{0}{\underset{0}{0}}$, let 1 represent the state $\overset{1}{\underset{0}{0}}$, 2 the state $\overset{0}{\underset{0}{1}}$, 3 the state $\overset{1}{\underset{0}{1}}$, and so forth. Hence according to the function table the input state 0 corresponds to the output state 1, the input state 1 corresponds to the output state 2, input 2 corresponds to output 3, input 3 to output 4, etc.

Suppose that initially we set this circuit so that the input state is 0 (that is, $\overset{0}{\underset{0}{0}}$) and then let it go. We can represent the successive input states as in Fig. 11-21. This is called an *input-state diagram* for a circuit.

There is another way to trace the successive input states of a circuit, using Boolean matrices. Here we suppose that the rows of the matrix are labeled by the input states, the columns by the output states. Then an element of the matrix is a unit, provided that its row and column are labeled by corresponding input-output states.

For example, consider the following function table:

$$\begin{array}{rl} & 0123 \\ \#A &= 0101 \\ \#B &= 0011 \\ \hline & 1230 \\ \#A' &= 1010 \\ \#B' &= 0110 \end{array}$$

Hence we have

$$\begin{array}{rcl} & \begin{array}{cccc} 0 & 1 & 2 & 3 \end{array} & \text{(output states 1 } \mu\text{sec later)} \\ \text{(Input states)} \begin{array}{c} 0 \\ 1 \\ 2 \\ 3 \end{array} & \begin{pmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \end{pmatrix} & \end{array}$$

Such a matrix is called the *state matrix* for the circuit. In the example $a_{01} = 1$ because input state 0 corresponds to output state 1, $a_{12} = 1$ because input state 1 corresponds to output state 2, and so forth.

Assuming, to be concrete, that the delay is 1 μsec, if this circuit is initially set at input state 0, it will have input state 1 after 1 μsec. This fact can be read directly from the input-state matrix. Now, after 2 μsec, the input state will be 2. If it was initially at input state 1, after 2 μsec it will be at input state 3, and so forth. This information can be obtained by multiplying the input-state matrix by itself and reading the resulting matrix.

$$\begin{pmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \end{pmatrix} \otimes \begin{pmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \end{pmatrix} = \begin{array}{rcl} & \begin{array}{cccc} 0 & 1 & 2 & 3 \end{array} & \text{(output states 2 } \mu\text{sec later)} \\ \begin{array}{c} 0 \\ 1 \\ 2 \\ 3 \end{array} & \begin{pmatrix} 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{pmatrix} & \end{array}$$

Hence, given an initial input state corresponding to one of the rows, the input state after 2 μsec can be taken from the matrix by observing the column number of the unit of that row. To find the states 3 μsec later, simply multiply this matrix again by the original one *on the right:*

$$\begin{pmatrix} 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{pmatrix} \otimes \begin{pmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \end{pmatrix} = \begin{array}{rcl} & \begin{array}{cccc} 0 & 1 & 2 & 3 \end{array} & \text{(output states 3 } \mu\text{sec later)} \\ \begin{array}{c} 0 \\ 1 \\ 2 \\ 3 \end{array} & \begin{pmatrix} 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{pmatrix} & \end{array}$$

The result is read the same way, etc.

Hence, to find the input-state matrix that gives the input state of a circuit after *n delay-time periods, multiply successively n times on the right by the initial input matrix.*

For example, let us find the input-state matrix after two, three, four, and five delay-time periods for the circuit with the following function table:

$$\begin{array}{r@{\;=\;}l} \#A & 0101 \\ \#B & 0011 \\ \hline \#A' & 1011 \\ \#B' & 0010 \end{array}$$

We then have

$$\begin{array}{c|cccc} & 0 & 1 & 2 & 3 \\ \hline 0 & 0 & 1 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 \\ 2 & 0 & 0 & 0 & 1 \\ 3 & 0 & 1 & 0 & 0 \end{array} \qquad \text{as original matrix}$$

$$\begin{pmatrix} 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 \end{pmatrix} \otimes \begin{pmatrix} 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{pmatrix} \qquad \text{after 1 delay time}$$

$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{pmatrix} \otimes \begin{pmatrix} 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 \end{pmatrix} = \begin{pmatrix} 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{pmatrix} \qquad \text{after 2 delay times}$$

$$\begin{pmatrix} 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{pmatrix} \otimes \begin{pmatrix} 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{pmatrix} \qquad \text{after 3 delay times}$$

$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{pmatrix} \otimes \begin{pmatrix} 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 \end{pmatrix} = \begin{pmatrix} 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{pmatrix} \qquad \text{after 4 delay times}$$

$$\begin{pmatrix} 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{pmatrix} \otimes \begin{pmatrix} 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{pmatrix} \qquad \text{after 5 delay times}$$

The student should draw the input-state diagram for this function table and compare the results with the matrices.

EXERCISES

Draw the input-state diagrams, and find the input-state matrix after two, three, four, five, six, seven, and eight delay-time periods of the circuits with the following function tables:

(*a*)
$$\begin{array}{r@{\;=\;}l} \#A & 0101 \\ \#B & 0011 \\ \hline \#A' & 1010 \\ \#B' & 0110 \end{array}$$

(*b*)
$$\begin{array}{r@{\;=\;}l} \#A & 0101 \\ \#B & 0011 \\ \hline \#A' & 0110 \\ \#B' & 1000 \end{array}$$

(*c*)

$$\begin{array}{rl}
\#A = & 0101 \quad 0101 \\
\#B = & 0011 \quad 0011 \\
\#C = & 0000 \quad 1111 \\
\hline
\#A' = & 1010 \quad 0111 \\
\#B' = & 1100 \quad 0011 \\
\#C' = & 0011 \quad 0011
\end{array}$$

(*d*)

$$\begin{array}{rl}
\#A = & 0101 \quad 0101 \\
\#B = & 0011 \quad 0011 \\
\#C = & 0000 \quad 1111 \\
\hline
\#A' = & 0011 \quad 1111 \\
\#B' = & 0011 \quad 0111 \\
\#C' = & 1001 \quad 1101
\end{array}$$

11-12. Additional Topics

a. Representations of Boolean Algebra. As we have pointed out in the previous chapter, the functions of Boolean algebra can be represented by areas of intersecting sets. Such diagrams are called Venn diagrams (attributed to Venn by Schroeder). Figure 11-22 is such a diagram for four variables.

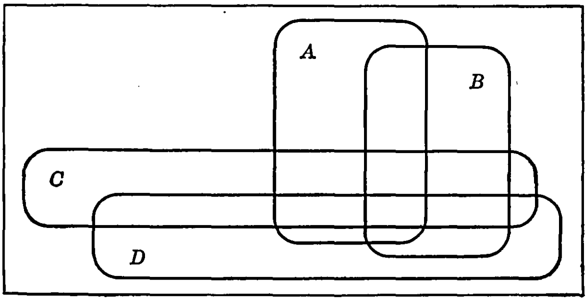


FIG. 11-22. Four-variable Venn diagram.

A geometric representation of a Boolean algebra may be obtained by considering the *columns of the basis as coordinates of points* that form a *hypercube.* For two variables we obtain a square (see Fig. 11-23*a*). The vertices are the elementary products and the edges are the functions formed by the sum of the adjacent vertices. For three variables we obtain a cube, where now the faces are the sum of the adjacent vertices

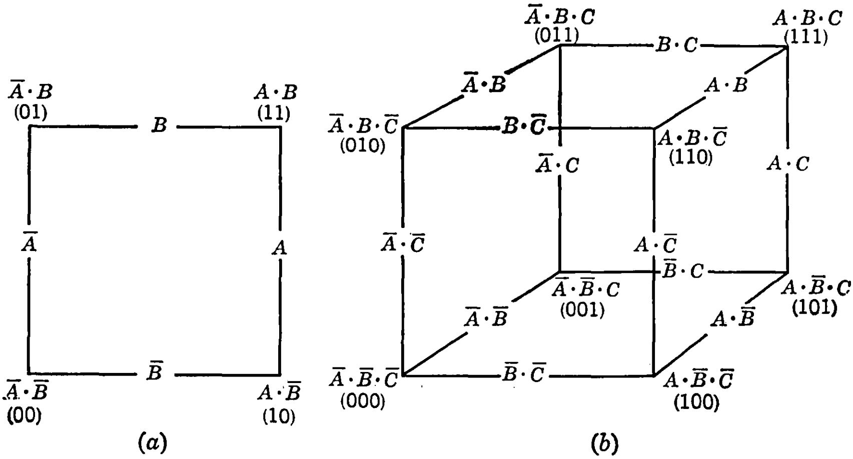


FIG. 11-23. Two- and three-dimensional hypercubes. The numbers represent basis columns.

(or edges), and so forth (see Fig. 11-23*b*). In Fig. 11-23*b* we have labeled the edges; can you label the faces?

Another representation of a Boolean lattice diagram is illustrated in Fig. 11-24, for a two-elementary-element system. A point Y on the lattice is said to *include* another point X on the lattice if a continuous upward path can be found from X to Y along the lines drawn on the lattice diagram. For example, on our lattice, $\bar{A} + \bar{B}$ includes $A \cdot \bar{B}$; $A + B$ includes A, and so forth. One point of the lattice is said to be *below* (*above*) another point if it is on a row below (above) the row of the other point. For example, on our lattice, $\bar{A} \cdot \bar{B}$ is below $\bar{B}$, $\bar{A} + \bar{B}$ is above $A \cdot \bar{B} + \bar{A} \cdot B$, etc.

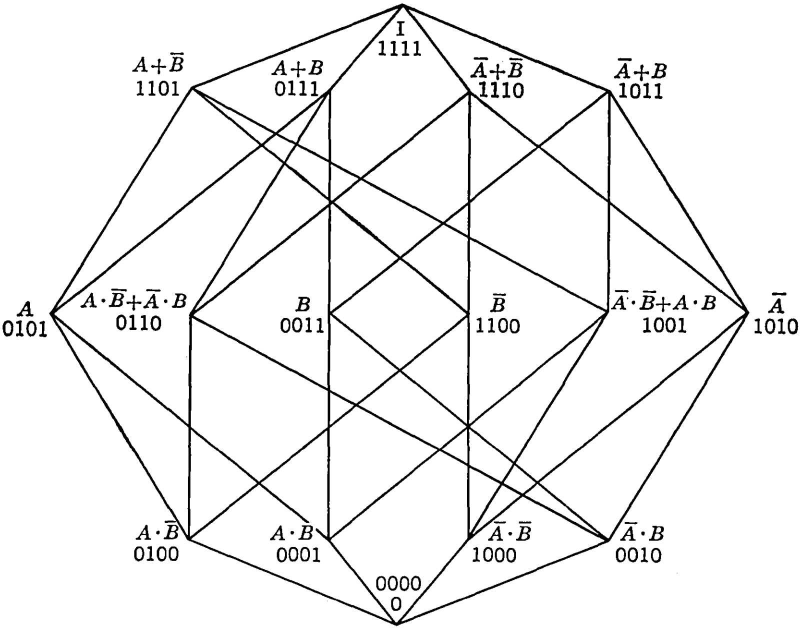


FIG. 11-24. The lattice diagram (two variables).

The lattice is so constructed that the logical sum of two elements is the lowest point which *includes* both elements. For example, $\bar{A} + \bar{B}$ is the logical sum of $\bar{A}$ and $\bar{B}$, $A + B$ is the logical sum of $A \cdot \bar{B}$ and B, etc. The logical product of two elements is the highest point which *is included by* both elements. For example, $A \cdot B$ is the logical product of A and B; $A \cdot \bar{B}$ is the logical product of $A + B$ and $\bar{B}$. The numbers now represent designation numbers (not basis columns as shown in Fig. 11-23).

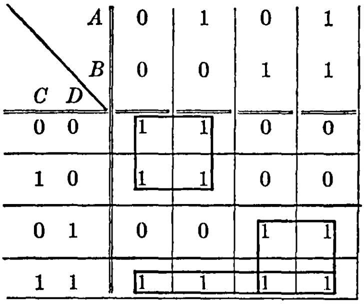
b. Chart Methods for Simplification of Boolean Functions. Besides the methods given in this chapter for finding the simplest sum-of-products form, there are some semisystematic, so-called chart methods of procedure. Consider the following four-variable basis and designation numbers:

A	0101	0101	0101	0101
B	0011	0011	0011	0011
C	0000	1111	0000	1111
D	0000	0000	1111	1111
	1100	1100	0011	1111

The work is arranged in a chart as follows:

A	0	1	0	1
B	0	0	1	1
C D				
0 0	1	1	0	0
1 0	1	1	0	0
0 1	0	0	1	1
1 1	1	1	1	1

The rows in the chart proper correspond to the successive groups of four positions of the designation number. In this form it is easier to see that the units in the bottom row correspond to all possible combinations of A and B and hence will be covered by $C \cdot D$. The four units in the upper left quarter correspond to all combinations of conditions on A and C and hence will be covered by $\bar{B} \cdot \bar{D}$; similarly the four units in the lower right quarter can be covered by $B \cdot D$:



A	0	1	0	1
B	0	0	1	1
C D				
0 0	1	1	0	0
1 0	1	1	0	0
0 1	0	0	1	1
1 1	1	1	1	1

Thus the simplest sum of products becomes $C \cdot D + \bar{B} \cdot \bar{D} + B \cdot C$. This method is due to Veitch (see reference below). A variation due to Karnaugh (see reference below) rearranges the columns and rows so that the units of A and C become adjacent. For example, consider the designation number 0101 0111 0101 1111 and the following chart:

A	0	1	1	0
B	0	0	1	1
C D				
0 0	0	1	1	0
1 0	0	1	1	1
1 1	1	1	1	1
0 1	0	1	1	0

From this it is seen that the simplest sum of products is $A + B \cdot C + C \cdot D$. Try this with a chart not having the rearranged rows and columns, and the advantages of the rearrangement will become clear. (Note that the rearranged chart is really just a Venn diagram.) For further information on these methods see the references below.

c. Decimal Method for Determining Included and Nonincluded Elementary Elements. First note that the elimination pairs for an A_i are simply those pairs of (positions corresponding to) basis columns that are identical except in the ith row; for example,

$$\begin{array}{ccc} & 9 & 13 \\ A_1 & 1 & 1 \\ A_2 & 0 & 0 \\ A_3 & 0 & 1 \\ A_4 & 1 & 1 \end{array}$$

are an elimination pair for A_3.

Next observe that, if p and q are decimal representations of an elimination pair of positions for A_i, then we have $p + 2^{i-1} = q$; for example, $9 + 2^{3-1} = 9 + 4 = 13$ as desired. Finally note that it is an easy matter to determine whether a basis column has a unit or a zero in the ith row directly from the decimal position number, for all we do is to divide it by 2^{i-1}. If the *integral part* of the result is *even*, the column has a zero in the ith row; if the integral part is *odd*, the column has a unit in the ith row (why?). For example, for $i = 3$ and the column in position 9, we have $9/2^{3-1} = \frac{9}{4} = 2\frac{1}{4}$; since 2 is even, there is a 0 in the third row of this column. For $i = 3$ and the column in position 13 we have $13/2^{3-1} = \frac{13}{4} = 3\frac{1}{4}$; since 3 is odd, there is a 1 in the third row of this column.

Now we are ready to present the decimal procedure in three steps. Consider, for example,

$$\begin{array}{cccccccccccccccc} & 1 & & & 4 & 5 & 6 & 7 & & 9 & & & 12 & 13 & 14 & 15 \\ 0 & 1 & 0 & 0 & 1 & 1 & 1 & 1 & 0 & 1 & 0 & 0 & 1 & 1 & 1 & 1 \end{array}$$

and A_3. The *first* step is to separate those positions having units in the designation number into *two* groups, corresponding to whether the position has a unit or a zero in the ith row:

Positions having zero in 3d row	*Positions having unit in 3d row*
1, 9	4, 5, 6, 7, 12, 13, 14, 15

Second, add 2^{i-1} to the positions having zero in the ith row: $1 + 4 = 5$, $9 + 4 = 13$. *Third,* compare the adjusted numbers with the numbers of the positions having units in the ith row: if the two groups of numbers are identical, neither A_i nor $\bar{A}_i$ is included; if a number appears in the unit group that does not appear in the adjusted zero group, then A_i is included; if a number appears in the adjusted zero group that does not appear in the unit group, then $\bar{A}_i$ is included. For our example we have the two groups:

Adjusted zero group	*Unit group*
5, 13	4, 5, 6, 7, 12, 13, 14, 15

Since numbers appear in the unit group that do not appear in the adjusted zero group (but not the reverse), we have that A_3 (but not $\bar{A}_3$) is included. For A_1 these groups are (5,7,13,15) and (1,5,7,9,13,15), whence A_1 is included; for A_2 these groups are

(3,6,7,11,14,15) and (6,7,14,15), whence $\bar{A}_2$ is included; for A_4 these groups are (9,12,13,14,15) and (9,12,13,14,15), whence A_4 is *not* included.

d. Decimal Method for Simplification.† The work of finding the simplest sum-of-products form by the method of Sec. 11-6 can be shortened by considering the decimal numbers denoting basis columns instead of the columns themselves. Consider again the example of that section, the designation number

$$\begin{array}{cccccccccccccccccc} & & 0 & 1 & & 3 & 4 & & 6 & 7 & & 9 & 10 & 11 & & 13 & 14 & 15 \\ \#F & = & 1 & 1 & 0 & 1 & 1 & 0 & 1 & 1 & 0 & 1 & 1 & 1 & 0 & 1 & 1 & 1 \end{array}$$

Instead of writing the columns in groups of ascending numbers of included units,

$$\begin{array}{c|cc|cc} 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 \end{array} \cdots$$

we merely write the groups of decimal numbers of those columns: 0; 1, 4; 3, 6, 9, 10; 7, 11, 13, 14; 15. Note that if a column is to combine with another in the next adjacent group then the number in the latter group must be *greater* than the number in the former by a power of 2. Hence we need list (in groups) for the combined result *only the least number, and the difference*, which corresponds to the power position in which the columns differ: 0-1 for columns 0 and 1, 0-4 for columns 0 and 4, 1-2 for 1 and 3, 1-8 for 1 and 9, 4-2 for 4 and 6 (but *not* 4-1 for 4 and 3), 3-4 for 3 and 7, etc. Just as combined columns recombine only if they have Φ in the same positions, so these pairs combine only if they have common differences: thus 1-2 and 9-2 combine, and we write 1-2,8 (also 1-8 and 3-8 combine to form 1-2,8). The work for this example would be tabulated as in Table 11-3. No combinations are possible in the final row of the table, because no pairs have the same two differences.

The prime implicants are represented by the unchecked numbers of Table 11-3: namely, 0-1; 0-4; 4-2; 1-2,8; 3-4,8; 6-1,8; 9-2,4; 10-1,4. The prime-implicant chart (see Table 11-1) is formed by adding to each least number all combinations of its associated differences. To find the expression for one of these combination numbers, we omit the variables whose power positions correspond to the differences and bar those variables whose powers do not add to make the first number, to the left of the dash. Thus 9-2,4 corresponds to $A \cdot D$ (since B and C are deleted and $1 + 8 = 9$) and 4-2 corresponds to $\bar{A} \cdot C \cdot \bar{D}$ (since B is deleted and $0 + 4 + 0 = 4$).

TABLE 11-3. WORK FOR THE DECIMAL METHOD

✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓
0 | 1; 4 | 3; 6; 9; 10 | 7; 11; 13; 14 | 15

✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓
0-1; 0-4 | 1-2; 1-8; 4-2 | 3-4; 3-8; 6-1; 6-8; 9-2; 9-4; 10-1; 10-4 | 7-8; 11-4; 13-2; 14-1

1-2,8; 1-8,2 | 3-4,8; 6-1,8; 6-8,1; 9-2,4; 9-4,2; 10-1,4; 10-4,1

or, on eliminating duplications,

1-2,8 | 3-4,8; 6-1,8; 9-2,4; 10-1,4

† This method was developed by James Bruce Wilson.

e. References on Simplification of Boolean Functions

Abhyankar, S.: Minimal "Sum of Products of Sums" Expressions of Boolean Functions, *IRE Trans. on Electronic Computers,* vol. EC-7, no. 4, pp. 268–276, December, 1958.

Harris, B.: An Algorithm for Determining Minimal Representation of a Logic Function, *IRE Trans. on Electronic Computers,* vol. EC-6, no. 2, pp. 103–108, June, 1957.

Karnaugh, G.: The Map Method for Synthesis of Combinational Logic Circuits, *Trans. AIEE,* pt. I, *Communs. and Electronics,* vol. 72, pp. 593–599, November, 1953.

Ledley, R. S.: Mathematical Foundations and Computational Methods for a Digital Logic Machine, *J. Operations Research Soc. Am.,* vol. 2, no. 3, pp. 249–279, August, 1954.

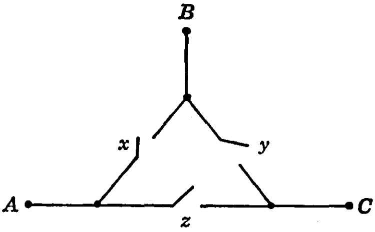
McCluskey, E. J., Jr.: Minimization of Boolean Functions, *Bell System Tech. J.,* vol. 35, pp. 1–28, November, 1956.

Quine, W. V.: A Way to Simplify Truth Functions, *Am. Math. Monthly,* vol. 62, pp. 627–631, November, 1955.

Roth, J. P.: Algebraic Topological Methods for the Synthesis of Switching System I, *Trans. Am. Math. Soc.,* vol. 88, no. 2, pp. 301–326, July, 1958.

Veitch, E. W.: A Chart Method for Simplifying Truth Functions, *Proc. Assoc. Computing Machinery,* May, 1952, pp. 127–133.

f. Switching Circuits and References. The *and, or,* and *not* gates that we have been considering are unidirectional—i.e., the signal is propagated in the forward direction only. In fact this is the only reason for having a special *or* gate, for otherwise the same result could be obtained with a simple soldered joint. However, there are circuits that can be described in terms of Boolean algebra, in which the signal can propagate in both forward and backward directions; these are called *switching circuits.* The usual implementation of switching circuits is with relays, although in special cases transistors can produce the same effect. It will suffice for our purposes to indicate switching circuits in terms of simple open or closed switches, as in the circuit of Fig. 11-25. Each switch is denoted by a Boolean variable, where a bar over the variable indicates an open switch, no bar a closed switch. The *transmission function* between two terminals indicates the switch conditions under which signal transmission can take place between these terminals, and is a Boolean function. Thus in the figure there will be transmission between A and B only when either switch x is closed,



$$T_{AB} = x+y\cdot z$$
$$T_{BC} = y+x\cdot z$$
$$T_{AC} = z+x\cdot y$$

FIG. 11-25. Switching circuit and transmission functions.

or both switches y *and* z are closed; thus the transmission function, denoted by T_{AB}, is given by $T_{AB} = x + y \cdot z$.

Although many of the problems of gating circuits and switching circuits are analogous, their characteristics can differ greatly. The first use of Boolean algebra in electrical engineering was made by C. E. Shannon in his well-known paper A Symbolic Analysis of Relay and Switching Circuits (*Trans. AIEE,* vol. 57, pp. 713–723, 1938). More recently F. E. Hohn and L. R. Schissler applied Boolean matrices to these problems in their paper Boolean Matrices and the Design of Combinational Relay Switching Circuits (*Bell System Tech. J.*, vol. 34, pp. 177–202, 1955). Their use of matrix methods is different, however, from those to be considered in Chap. 13 of this text, where we develop and apply the theory of Boolean matrices to gating circuits. Further reading on the design of switching circuits can be found in the works below:

Caldwell, S. H.: "Switching Circuits and Logical Design," John Wiley & Sons, Inc., New York, 1958.

Higonnet, R. A., and R. A. Grea: "Logical Design of Electrical Circuits," McGraw-Hill Book Company, Inc., New York, 1958.

Huffman, D. A.: The Design and Use of Hazard-free Switching Networks, *J. Assoc. Computing Machinery,* vol. 4, pp. 47–62, January, 1957; The Synthesis of Sequential Switching Networks, *J. Franklin Inst.*, vol. 257, nos. 2, 3, pp. 161–190, 275–303, March, April, 1954.

Humphrey, W. S., Jr.: "Switching Circuits," McGraw-Hill Book Company, Inc., New York, 1958.

Keister, W., A. E. Ritchie, and S. H. Washburn: "The Design of Switching Circuits," D. Van Nostrand Company, Inc., Princeton, N.J., 1951.

Mealy, G. H.: A Method for Synthesizing Sequential Circuits, *Bell System Tech. J.*, vol. 34, pp. 1045–1080, September, 1955.

CHAPTER 12

ELEMENTARY COMPUTATIONAL METHODS IN CIRCUIT DESIGN

12-1. Introduction

The material of this chapter and Chaps. 13 and 14 is based on the original research of the author. It is here presented for the first time in a complete and continuous development.

The previous two chapters have discussed (1) the method of writing the Boolean function for a circuit diagram and conversely drawing the circuit diagram that corresponds to a Boolean function, and (2) the method of writing the designation number for a Boolean function and conversely determining a Boolean function corresponding to a designation number. The present chapter deals with computational methods that involve designation numbers and illustrates applications of these methods to circuit design.

Three main topics are considered in this chapter. First the concept of constraints and the related concepts of logical dependence and independence are introduced and applied to circuit design. Next methods are given for solving Boolean equations, and applications of these methods to circuit design are discussed. Finally the concept of absolute simplest form is given, and applications are discussed. In each case the computational method is presented separately so that when applications to circuit design are given the student can concentrate solely on the engineering aspects of the problem.

12-2. Constraints

The Problem. Up to now we have been considering systems of *independent elementary elements.* That is, for elementary elements with the basis given above, the truth value of each elementary element is independent of the truth value of any other elementary element. It often happens, however, that this is not the case; the truth value of an elementary element *may* depend on the truth values of the other elementary elements *for a particular problem.* Such a dependence is given in terms of a combined element which is to remain true *for this problem.* For example, in a problem we might always have that $A \rightarrow B$. The effect of such a constraint is to limit the possible combinations of truth values that A and B can take simultaneously. If $A \rightarrow B$ is to hold, then B cannot be false when A is true; i.e., the combination of truth values A true, B false cannot occur.

Summarizing, *a constraint is a logical relation between the elementary elements (i.e., a combined element) which is to be considered as tautologically true (i.e., always true) because of the stated or intrinsic circumstances of the problem.* Put another way, constraints are factually true combined elements which are to remain true throughout a problem and therefore, with respect to this problem, can be considered as tautologies. In this section will be given a systematic computational method that automatically guarantees the maintenance of any desired constraints throughout the solution of a problem, so that all advantages produced by the constraints can be obtained in a straightforward manner.

In terms of digital-computer circuitry a constraint is a relation between the input-wire signals. The effect of constraints is to limit the number of input states; i.e., some simultaneous combinations of voltages on the wires are forbidden by the constraints. For example, suppose that a circuit had three input wires A, B, and C such that no two of the wires could both have 1-volt signals simultaneously, and it never occurs that all three of the wires have 0-volt signals simultaneously. These constraints can be written as follows: $A \cdot B = 0$, $A \cdot C = 0$, $B \cdot C = 0$, and $\bar{A} \cdot \bar{B} \cdot \bar{C} = 0$ (where we recall that A means that "wire A has a 1-volt signal" is true, etc.; see Chap. 10).

We have remarked that, when the usual basis is considered as a circuit function table, for n input wires there are 2^n columns in the basis (or table), one column for each of the 2^n possible input states. With constraints, however, some of these 2^n states are forbidden. Hence it seems plausible that in order to take account of the constraints during a problem we should simply cross off those columns in the basis which represent forbidden states of the particular problem—forming a new, reduced, constrained basis. Then we should proceed as described in the previous chapters, except that *all computational methods should be performed with respect to this constrained basis.* This is the method for taking account of constraints.

The first problem that arises is: Given a set of constraints as Boolean functions, which columns of the basis are to be crossed off to form the constrained basis? Conversely, if we are given the columns to be crossed off or the constrained basis, what is the Boolean form of the constraints? The next problem is: How are the computational methods of the previous chapter performed with respect to a constrained basis?

Determining the Constrained Basis. Let us consider the first of these problems, to find the constrained basis, given the constraints. Suppose that $f_1(A,B,C, \ldots)$, $f_2(A,B,C, \ldots)$, $\ldots$, $f_n(A,B,C, \ldots)$ are constraints which are all to be tautologically true simultaneously. We form the logical product of the designation numbers of the constraints,

$$\prod_{j=1}^{n} \#f_j(A,B,C, \ldots) = f_1(A,B,C, \ldots) \cdot f_2 \cdot f_3 \cdot \cdots \cdot f_n$$

It can be seen that the positions of the zeros in the resulting number correspond to input states that can never occur if the constraints are all to remain true. Hence *the constrained basis is formed by crossing off those columns of the usual basis that correspond to the positions of the zeros of the product number.*

For example, consider the designation numbers of the constraints of the above example,

$$\begin{array}{rl}
\#(A \cdot B = 0) = & 1110\quad 1110\\
\#(A \cdot C = 0) = & 1111\quad 1010\\
\#(B \cdot C = 0) = & 1111\quad 1100\\
\#(\bar{A} \cdot \bar{B} \cdot \bar{C} = 0) = & \underline{0111\quad 1111}\\
\text{Product} & 0110\quad 1000\\
 & 0\;\;\;\;3\quad\;\;\;567
\end{array}$$

Crossing off columns 0, 3, 5, 6, and 7 of the usual basis, we find for the constrained basis

$$\begin{array}{rl}
\#A = & 100\\
\#B = & 010\\
\#C = & 001
\end{array}$$

Conversely, given a constrained basis, the explicit form of the constraints can be determined as a Boolean function. We shall give this procedure by means of an example. Suppose that the constrained basis is

$$\begin{array}{rl}
\#A = & 0101\quad 01\\
\#B = & 0011\quad 00\\
\#C = & 0000\quad 11
\end{array}$$

The columns of this constrained basis correspond to the columns in positions 0, 1, 2, 3, 4, and 5 of the standard basis. Hence we know that the designation number of the constraint must have units in these positions, and zeros elsewhere: $\overset{0\,1}{11}\ \overset{2\,3}{11}\ \overset{4\,5}{11}\ 00$. With respect to the (unconstrained) standard basis this is the designation number of $\bar{B} + \bar{C}$, which is the desired constraint. Note that $\bar{B} + \bar{C}$ can equivalently be written as $\bar{B} + \bar{C} = I$ or $B \cdot C = 0$. In finding a constraint from a constrained basis only a single constraint, corresponding to the final product of constraints, will appear, as was the case in our example.

Designation Numbers with Respect to a Constrained Basis. As stated above, when constraints exist in a problem *the computational methods for that problem are to be carried out with respect to the constrained basis.* Hence we must be able (1) to find the designation number of a Boolean function with respect to the constrained basis and conversely (2) to find the Boolean function corresponding to a given designation number (in any of the various forms considered in Chap. 11) with respect to the constrained basis. These computations are identical with those discussed in Chap. 11 except that all procedures are *with respect to the constrained basis.* However, for clarity we shall illustrate these procedures.

For example, consider the following constrained basis:

$$\begin{array}{ll}
\#A & 0101\quad 01\\
\#B & 0011\quad 00\\
\#C & 0000\quad 11
\end{array}$$

To find $\#(\bar{A} + B \cdot C)$ with respect to this basis, we have

$$\begin{array}{rl}
\#B = & 0011\ \ 00 \\
\underline{\#C} = & \underline{0000\ \ 11} \\
\#(B \cdot C) = & 0000\ \ 00 \\
\underline{\#\bar{A}} = & \underline{1010\ \ 10} \\
\#(\bar{A} + B \cdot C) = & 1010\ \ 10
\end{array}$$

Note that in this case it is simply $\#\bar{A}$.

For the converse problem of finding the Boolean function of a given designation number with respect to a constrained basis, the designation number can be given in two different ways: either (1) with the same number of positions as columns of the constrained basis, in which case it is assumed that the positions of the designation number correspond to the columns of the constrained basis, or (2) with 2^n positions, for n elementary elements. In this second case the positions of the given designation numbers that correspond to the missing columns of the constrained basis are crossed off, and the second case is thereby reduced to the first case. For instance, suppose that it were desired to find the Boolean function corresponding to 1010 1011 with respect to the constrained basis given in the above example. Columns 6 and 7 of the standard basis are missing in this constrained basis; hence we cross off positions 6 and 7 of our number: 1010 10 ~~11~~.

The methods given in Chap. 11 for finding the simplest sum-of-products form can easily be adapted to take constraints into consideration. Every constrained basis column (i.e., eliminated column) can be used freely if it combines with nonconstrained columns corresponding to the units of the designation number. However, positions corresponding to constrained columns do not necessarily have to be covered by a prime implicant (as do the unit positions of the constrained designation number). In other words, the constrained positions can be thought of as units or zeros, whichever helps best to simplify the function.

For example, consider 1111 01 with respect to the constrained basis of the above example. By our first simplification method we have

				Constrained columns displayed for possible use	
✓	✓✓	✓✓			
0	10	11		0	1
0	01	10		1	1
0	00	01		1	1

✓✓	✓✓✓		
Φ0	1 1 Φ 0	11	Formed by combination with constrained columns
0Φ	Φ 0 1 1	1Φ	
00	0 Φ 0 Φ	Φ1	

Φ	1Φ
Φ	Φ1
0	ΦΦ

The table of prime implicants (omitting columns 6 and 7) becomes

	0	1	2	3	5
$\bar{C}$	0	1	2	3	
A		1		3	5
B			2	3	

whence $\bar{C}$ and A are essential prime implicants whose sum covers the unit positions of the constrained designation number; thus

$$1111\ \ 01 = \#(A + \bar{C})$$

The second method given in Chap. 11 for finding the simplest sum of products can likewise be adapted to handle constrained designation numbers. Consider again the number 1111 01 as an example:

							C	B	A	
0	1	2	3	5	6	7	4	2	1	
x	1	2	1				0	Φ	Φ	$\bar{C}$
	4		2	x		2	Φ	Φ	1	A

Constrained positions

Note that we had to make use of constrained column 7 in order to form all combinations with B (power 2) in the second line of the illustration.

The methods for going from the constrained designation number to other Boolean function forms, e.g., the first and second canonical forms, follow easily.

Constrained Boolean Algebra. An important point arises from the last example. Note that, with respect to the constrained basis given above, many strange relations hold; for example,

$$A = A \cdot (\bar{B} + \bar{C}) \qquad A + B = A \cdot \bar{B} + B \cdot \bar{C} + A \cdot C$$

Such relations obviously do not hold with respect to the standard unconstrained basis. In other words, with respect to a constrained basis, we obtain a *constrained Boolean algebra* which is different from the unconstrained Boolean algebra described in Chap. 10. That is, additional relations hold in a constrained Boolean algebra that do not hold in the unconstrained algebra—but all relations that hold in the unconstrained algebra also hold in the constrained algebra. In general these additional relations in the constrained algebra often allow Boolean functions to be further simplified; that is, $A \cdot (\bar{B} + \bar{C})$ becomes just A, and $A \cdot \bar{B} + B \cdot \bar{C} + A \cdot C$ becomes $A + B$ with respect to the constrained basis of the above example. This of course means that in designing a circuit any knowledge of constraints should be used to greatest advantage for further simplifying the design. However, it is not always possible to simplify a function further; for instance, $\bar{A} \cdot \bar{C} + A \cdot C$ cannot be put into simpler form with respect to the constrained basis of the above example.

EXERCISES

Find the constrained basis for the following sets of constraints:

	Answer
(*a*) $A \cdot B = 0$	$\#A = 010$ $\#B = 001$
(*b*) $B \cdot C = 0$	$\#A = 0101 \quad 01$ $\#B = 0011 \quad 00$ $\#C = 0000 \quad 11$
(*c*) $\bar{A} + B \cdot C = 0$ $A \cdot C = 0$	$\#A = 11$ $\#B = 01$ $\#C = 00$
(*d*) $A \rightarrow C$ $C \rightarrow B$	$\#A = 0001$ $\#B = 0111$ $\#C = 0011$
(*e*) $A \cdot (B \cdot \bar{C} + \bar{B} \cdot C) = 0$ $A \cdot B \rightarrow 0$ $B + \bar{D} = I$	$\#A = 0100 \quad 000$ $\#B = 0010 \quad 111$ $\#C = 0001 \quad 101$ $\#D = 0000 \quad 011$

Find the constraints corresponding to the following constrained bases:

	Answer
(*f*) $\#A = 010$ $\#B = 001$	$\bar{A} + \bar{B} = I$
(*g*) $\#A = 111$ $\#B = 010$ $\#C = 001$	$A \cdot \bar{B} + A \cdot \bar{C} = I$
(*h*) $\#A = 0001$ $\#B = 0111$ $\#C = 0011$	$\bar{A} \cdot \bar{C} + B \cdot C = I$

With respect to the constrained basis given in the example above, find the designation numbers of:

	Answer
(*i*) $A \cdot \bar{B} + \bar{C}$	1111 01
(*j*) $A \cdot (\bar{B} + \bar{C})$	0101 01
(*k*) $A \cdot B \cdot C + \bar{A} \cdot \bar{B} \cdot \bar{C}$	1000 00
(*l*) $A \cdot (B + \bar{C}) + \bar{A} \cdot (\bar{B} + C)$	1101 10

With respect to the constrained basis given in the example above, find the simplest sum-of-products form of:

	Answer
(*m*) 0101 01	A
(*n*) 0111 01	$A + B$
(*o*) 1010 01	$\bar{A} \cdot \bar{C} + A \cdot C$
(*p*) 0101 0100	A
(*q*) 0111 0101	$A + B$
(*r*) 1010 0101	$\bar{A} \cdot \bar{C} + A \cdot C$

12-3. Logical Dependence and Independence

Logical Independence. Boolean functions are logically independent (with respect to each other) if they can take on all possible combinations of truth values. There are 2^n possible combinations of truth values that n Boolean functions can have (since each can take either of two truth values). Hence, in order to test the independence of n Boolean functions, we need examine only the combinations of truth values that can occur: if all 2^n are possible, then they are independent; otherwise they are not. This can be done by means of the designation numbers for the Boolean functions. The procedure is illustrated by means of an example.

Suppose that we desire to know whether or not the functions of $A \cdot B + \bar{A} \cdot \bar{B}$ and $\bar{B}$ are independent. Write their designation numbers on successive rows,

$$\begin{aligned} \#(A \cdot B + \bar{A} \cdot \bar{B}) &= 1001 \\ \#\bar{B} &= 1100 \end{aligned}$$

The columns represent all possible combinations of truth values. Since all four column combinations are there, we can say that $A \cdot B + \bar{A} \cdot \bar{B}$ and $\bar{B}$ are independent.

As another example, consider $A \cdot B + \bar{B} \cdot \bar{C}$ and $\bar{A} \cdot \bar{B} + B \cdot C$. We write

$$\begin{aligned} \#(A \cdot B + \bar{B} \cdot \bar{C}) &= 1101 \quad 0001 \\ \#(\bar{A} \cdot \bar{B} + B \cdot C) &= 1000 \quad 1011 \end{aligned}$$

Here $\begin{smallmatrix}1\\1\end{smallmatrix}$, $\begin{smallmatrix}1\\0\end{smallmatrix}$, $\begin{smallmatrix}0\\1\end{smallmatrix}$, and $\begin{smallmatrix}0\\0\end{smallmatrix}$ all appear; so the functions are, indeed, independent.

Consider next the three functions $A \cdot B + \bar{A} \cdot \bar{B}$, $\bar{B}$, and $A \cdot \bar{B} + \bar{A} \cdot B$. Write

$$\begin{aligned} \#(A \cdot B + \bar{A} \cdot \bar{B}) &= 1001 \\ \#\bar{B} &= 1100 \\ \#(A \cdot \bar{B} + \bar{A} \cdot B) &= 0110 \end{aligned}$$

Here we need $2^3 = 8$ possible combinations in order that the *three* functions be independent. But only four columns appear; hence the functions are *not* independent.

Logical Dependence. Boolean functions that are not logically independent are said to be *logically dependent.* This means that there must be some logical relations between them that can be expressed as a combined element. For instance, we know that $A \cdot B$ and $A + B$ are logically dependent because clearly $A \cdot B \rightarrow (A + B)$. Given a set of n Boolean functions that are logically dependent, the explicit form of this dependence can be found by means of the designation numbers, in two steps. This first step is to write their designation numbers in successive rows. In this array of designation numbers cross off columns so that no two identical columns remain. Since the functions are dependent, there will be fewer than 2^n columns remaining. Put these columns in their standard order, and consider this resulting array as a *constrained basis.*

The second step is to find the constraint in the same manner as in the previous section, remembering that the rows are labeled by the given functions (and not elementary elements). For example, consider the functions $\bar{B} \cdot \bar{C} + \bar{A} \cdot C$, $\bar{A} \cdot \bar{C} + \bar{B} \cdot C$, and $\bar{B}$. Let us denote these by f_1, f_2, and f_3, respectively. Then we have

$\# f_1 = 1100$ ~~1~~ 01 ~~0~~
$\# f_2 = 1010$ ~~1~~ 10 ~~0~~
$\# f_3 = 1100$ ~~1~~ 10 ~~0~~

and on rearrangement in standard order we obtain the constrained basis:

	012	567
$\# f_1 =$	010	101
$\# f_2 =$	001	011
$\# f_3 =$	000	111

Hence the designation number of the constraint is 1110 0111. This designation number is to be evaluated with respect to the following standard (unconstrained) basis (see Sec. 12-2):

$\# f_1 = 0101 \quad 0101$
$\# f_2 = 0011 \quad 0011$
$\# f_3 = 0000 \quad 1111$

whence we have $1110 \ 0111 = \#(\bar{f}_1 + \bar{f}_2) \cdot \bar{f}_3 + (f_1 + f_2) \cdot f_3$. Hence the logical dependence between f_1, f_2, and f_3 is given by

$$\#(\bar{f}_1 + \bar{f}_2) \cdot \bar{f}_3 + (f_1 + f_2) \cdot f_3 = I$$

as can be easily verified by substituting for f_1, f_2, and f_3 their corresponding functions in terms of A, B, and C.

EXERCISES

Are the following sets of functions independent?

	Answer
(*a*) $A \cdot B + \bar{A} \cdot \bar{B}$ $A \cdot \bar{B} + \bar{A} \cdot B$	No
(*b*) $\bar{A}$ $\bar{B}$	Yes
(*c*) $\bar{B}$ $\bar{A}$ $(A \cdot B + \bar{A} \cdot \bar{B}) \cdot \bar{C} + (A \cdot \bar{B} + \bar{A} \cdot B) \cdot C$	Yes
(*d*) $\bar{B} \cdot \bar{C} + \bar{A} \cdot C$ $\bar{A} \cdot \bar{C} + \bar{B} \cdot C$ $(A + \bar{B}) \cdot \bar{C} + A \cdot \bar{B} \cdot C$	Yes
(*e*) $\bar{B} \cdot \bar{C} + \bar{A} \cdot C$ $\bar{A} \cdot \bar{C} + \bar{B} \cdot C$ $\bar{B}$	No

Find the logical dependence between the following sets of functions:

	Answer
(*f*) $f_1 = A \cdot B + \bar{A} \cdot \bar{B}$ $f_2 = A \cdot \bar{B} + \bar{A} \cdot B$	$f_1 = \bar{f}_2$ (that is, $f_1 \cdot \bar{f}_2 + \bar{f}_1 \cdot f_2 = I$)
(*g*) $f_1 = \bar{A} \cdot B$ $f_2 = A + \bar{B}$	$f_1 = \bar{f}_2$
(*h*) $f_1 = \bar{A} \cdot B$ $f_2 = \bar{A} + B$	$f_1 \rightarrow f_2$ (that is, $\bar{f}_1 + f_2 = I$)

12-4. Constraints in Circuit Design

Simplification by Constraints. If some input states of a circuit are forbidden (i.e., it is known that these states will never occur), or if a logical relation (or dependence) is known to exist between the input wires of a circuit, then the circuit can be designed in terms of a constrained Boolean algebra. In other words, *the computation may be performed with respect to the constrained basis.*

There are important advantages in recognizing constraints in the inputs to a circuit. For as we mentioned in Sec. 12-2, Boolean functions, and hence the corresponding circuits, can often be further simplified in the constrained system. In this section illustrations will be given of the use of the computational methods associated with constraints in the design of circuits.

For example, suppose that it is desired to design a circuit with two input wires such that a 1-volt signal output will exist whenever there are either 0-volt signals on both wires A and B or else 1-volt signals on both A and B. Hence, if Z represents the output, we have for the function table

$$\#A = 0101$$
$$\underline{\#B = 0011}$$
$$\#Z = 1001 = \#(A \cdot B + \bar{A} \cdot \bar{B})$$

Hence the circuit appears as in Fig. 12-1.

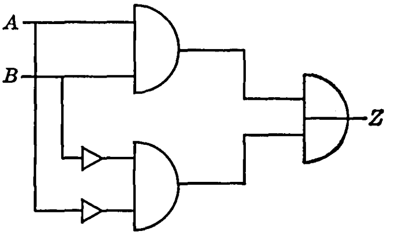


FIG. 12-1. Circuit for $Z = A \cdot B + \bar{A} \cdot \bar{B}$.

Now suppose, in addition, that it is known that wire B will have a unit signal whenever A has one; i.e., if A has a unit signal, then B has one, or $A \rightarrow B$. Since $\#(A \rightarrow B) = 1011$, the constrained basis becomes

$$\#A = 001$$
$$\underline{\#B = 011}$$
$$\#Z = 101 = \#(A + \bar{B})$$

whence the circuit design appears as in Fig. 12-2.

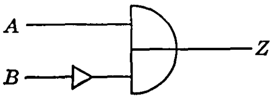


FIG. 12-2. Circuit for $Z = A + \bar{B}$.

Hence the introduction of the constraint enabled a significant simplification in the circuit design.

Examples. As another example, suppose that it is desired to design a counter, with three input and three output wires, that counts: 1, 2, 3, 4, 5, 1, 2, 3, ... The function table for such a counter becomes

$$\begin{array}{rl} \#A &= 0101 \;\; 0101 \\ \#B &= 0011 \;\; 0011 \\ \#C &= 0000 \;\; 1111 \\ \hline \#A' &= 010 \;\; 11 \\ \#B' &= 110 \;\; 00 \\ \#C' &= 001 \;\; 10 \end{array}$$

The statement of the problem does not specify the output states corresponding to input states 0, 6, and 7. In fact, if the counter is initially set at input state 1, and if it is designed correctly, input states 0, 6, and 7 will never occur. Hence a constrained basis can be used,

$$\begin{array}{rl} \#A &= 101 \;\; 01 \\ \#B &= 011 \;\; 00 \\ \#C &= 000 \;\; 11 \\ \hline \#A' &= 010 \;\; 11 = \#(\bar{A} + C) \\ \#B' &= 110 \;\; 00 = \#(\bar{A} + \bar{B}) \cdot \bar{C} \\ \#C' &= 001 \;\; 10 = \#(A \cdot B + \bar{A} \cdot \bar{B}) \end{array}$$

and the circuit is as in Fig. 12-3.

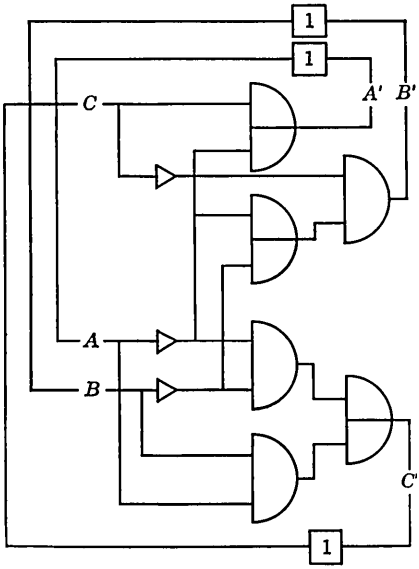


FIG. 12-3. Counter that counts 1, 2, 3, 4, 5, 1, 2, 3, ...

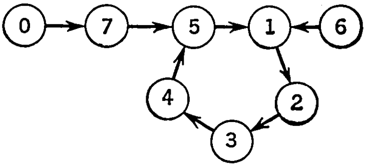


FIG. 12-4. State diagram of counter of Fig. 12-3 with initial erroneous input 0, 6, or 7.

It is interesting to find out what would happen in this circuit if the initial input state were erroneously 0, 6, or 7. To determine this, we simply list the designation numbers (with respect to the standard unconstrained base),

$$\begin{array}{rl} \#(\bar{A} + C) &= 1010 \;\; 1111 \\ \#(\bar{A} + \bar{B}) \cdot \bar{C} &= 1110 \;\; 0000 \\ \#(A \cdot B + \bar{A} \cdot \bar{B}) &= 1001 \;\; 1001 \end{array}$$

whence the state diagram becomes as in Fig. 12-4. Thus there is no difficulty, as all three states return to the loop.

Consider next a four-input circuit, two wires of which represent one 2-bit binary number N and the other two wires of which represent another binary number M. The problem is to design this circuit so that the result will be $N - M$, where it is known beforehand that $N \geq M$; that is, a negative number will never result. The function table for this circuit becomes

N_1	0101	0101	0101	0101
N_2	0011	0011	0011	0011
M_1	0000	1111	0000	1111
M_2	0000	0000	1111	1111
Z_1	0101	010	01	0
Z_2	0011	001	00	0

where input states 4, 8, 9, 12, 13, and 14 can never occur according to the statement of the problem. Hence a constrained basis is used,

N_1	0101	101	01	1	
N_2	0011	011	11	1	
M_1	0000	111	00	1	
M_2	0000	000	11	1	
Z_1	0101	010	01	0	$= \#(N_1 \cdot \bar{M}_1 + \bar{N}_1 \cdot M_1)$
Z_2	0011	001	00	0	$= \#(N_1 + \bar{M}_1) \cdot N_2 \cdot \bar{M}_2$

and the circuit becomes as in Fig. 12-5.

As another example, suppose that it is desired to design a circuit that will have a 1-volt output signal whenever at least one, and no more than two, of the three input wires have 1-volt signals. Calling the inputs U, V, and W and the output Z, we have for the function table

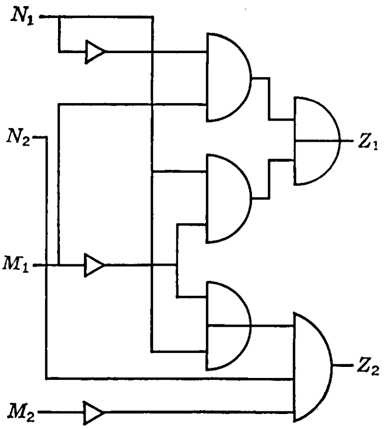


FIG. 12-5. Circuit forming $N - M$ $(N \geq M)$.

U	0101	0101	
V	0011	0011	
W	0000	1111	
Z	0111	1110	$= U \cdot \bar{V} + \bar{U} \cdot V + U \cdot \bar{W} + \bar{U} \cdot W$

Now suppose that U, V, and W are themselves outputs of a certain circuit: $U = \bar{B} \cdot \bar{C} + \bar{A} \cdot C$, $V = \bar{A} \cdot \bar{C} + \bar{B} \cdot C$, and $W = \bar{B}$. The question is: Should Z be formed as the above function, or can Z be simplified owing to our additional knowledge of U, V, and W? To answer this question, we must determine whether U, V, and W are independent or whether they are dependent. It turns out that they are dependent (see Exercise *e* of Sec. 12-3). Thus the next step is to find explicitly this dependence,

$$\begin{aligned}
\#U &= \#(\bar{B} \cdot \bar{C} + \bar{A} \cdot C) = 1100 \quad 1010 \\
\#V &= \#(\bar{A} \cdot \bar{C} + \bar{B} \cdot C) = 1010 \quad 1100 \\
\#W &= \qquad \#\bar{B} \qquad\quad = 1100 \quad 1100
\end{aligned}$$

The method of Sec. 12-3 gives the designation number of the constraint between U, V, and W as 1110 0111. This means that the input states 3 and 4 cannot occur. We evaluate Z with respect to the constrained basis,

$$\begin{array}{l|ll}
U & 010 & 101 \\
V & 001 & 011 \\
W & 000 & 111 \\
\hline
Z & 011 & 110 = U \cdot \bar{V} + \bar{U} \cdot V
\end{array}$$

which is simpler than the case above, where we knew nothing about the dependence between U, V, and W. Hence the circuit is simply as in Fig. 12-6.

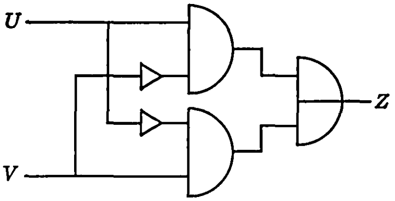


FIG. 12-6. Simplified circuit diagram.

EXERCISES

(*a*) Design a circuit that computes $2N - 7$, where it is known that negative results will never occur and N is a 3-bit binary number.

(*b*) Design a circuit that computes $2N - 7$, where it is known that positive results will never occur and N is a 3-bit binary number.

(*c*) Design a counter that counts successively in even numbers from 0 through 14, then returns to 0 and repeats.

(*d*) Design a counter that counts successively in odd numbers from 1 through 15 and then returns to 1 and repeats.

(*e*) Design a circuit with four inputs U, V, W, and X that produces a unit output only when the input state is a multiple of 3. Assume that U, V, W, and X are the outputs of the following circuits:

$$\begin{aligned}
U &= A \cdot \bar{B} + \bar{A} \cdot C \\
V &= A \cdot B + B \cdot C + \bar{A} \cdot \bar{C} + \bar{B} \cdot \bar{C} \\
W &= (A + \bar{B}) \cdot (B + \bar{C}) \\
X &= A \cdot (B + \bar{C}) + \bar{A} \cdot (\bar{B} + C)
\end{aligned}$$

12-5. Linear Boolean Equations

Boolean Equations. Just as in physics the solutions to many problems are obtained by considering ordinary algebraic equations in one or more unknowns, so in circuit design solutions are obtained by considering Boolean algebraic equations. An example of a Boolean algebraic equation is $X \cdot (A + B) = A \cdot B \cdot C$, *where X is an as yet unknown Boolean function of A, B, and C.* Boolean algebraic equations may be solved by means of the designation numbers.

For example, in order to find an X that satisfies

$$X \cdot (A + B) = A \cdot B \cdot C$$

we observe that $\#(A \cdot B \cdot C) = 0000\ 0001$ and that

$$\#(A + B) = 0111 \quad 0111$$

Since $\#(A \cdot B \cdot C)$ must be identical to $\#[X \cdot (A + B)]$ if the equation is to hold true, $\#X$ must be of the form _000 _001, where the two blanks may be filled with either zeros or units. There are thus exactly four nonequivalent combined elements that satisfy this equation, corresponding to the four designation numbers 0000 0001, 1000 0001, 0000 1001, and 1000 1001. It can be verified that they are $A \cdot B \cdot C$, $A \cdot B \cdot C + \bar{A} \cdot \bar{B} \cdot \bar{C}$, $C \cdot (A \cdot B + \bar{A} \cdot \bar{B})$, and $\bar{A} \cdot \bar{B} + A \cdot B \cdot C$.

Similarly we can solve an implication, for example,

$$X \cdot (A + B + \bar{C}) \rightarrow A \cdot B + \bar{B} \cdot C$$

Now $\#(A + B + \bar{C}) = 1111\ 0111$, while $\#(A \cdot B + \bar{B} \cdot C) = 0001\ 1101$, whence X must have the form 000_ __0_. There are therefore $2^4 = 16$ solutions, or nonequivalent X's, that satisfy this equation. These can be seen to be 0, $A \cdot B \cdot \bar{C}$, $\bar{A} \cdot \bar{B} \cdot C$, $A \cdot \bar{B} \cdot C$, $A \cdot B \cdot C$, $A \cdot B \cdot \bar{C} + \bar{A} \cdot \bar{B} \cdot C$, $A(B \cdot \bar{C} + \bar{B} \cdot C)$, $A \cdot B$, $\bar{B} \cdot C$, $(\bar{A} \cdot \bar{B} + A \cdot B) \cdot C$, $A \cdot C$, $A \cdot B \cdot \bar{C} + \bar{B} \cdot C$, $A \cdot B + \bar{A} \cdot \bar{B} \cdot C$, $A \cdot (B + C)$, $(\bar{B} + A) \cdot C$, and $A \cdot B + \bar{B} \cdot C$.

Since in general there will be many solutions to such equations as those considered above, it is probable that solutions satisfying certain given conditions will usually be desired. For example, it might be desired to find only those solutions X of the implication

$$X \cdot (A + B + \bar{C}) \rightarrow A \cdot B + \bar{B} \cdot C$$

which also imply $A \cdot B + \bar{A} \cdot \bar{B}$ and satisfy

$$X + B \cdot \bar{C} + A \cdot B \cdot C = C \cdot (A + \bar{B}) + B \cdot \bar{C}$$

Since $\#(A \cdot B + \bar{A} \cdot \bar{B}) = 1001\ 1001$, positions 1, 2, 5, and 6 must be zero; and since

$$\#[C \cdot (A + \bar{B}) + B \cdot \bar{C}] = 0011\ 1101$$

and

$$\#(B \cdot \bar{C} + A \cdot \bar{B} \cdot C) = 0011\ 0100$$

X must have units in positions 4 and 7 and zeros in 0, 1, and 6. Thus X must be of the form 000_ 1001, where only position 3 is arbitrary. (Notice that the conditions on X must be consistent with respect to each other.) Hence, of the original 16 possibilities for X, only 2 are left, $0000\ 1001 = \#[C \cdot (A \cdot B + \bar{A} \cdot \bar{B})]$ and $0001\ 1001 = \#(A \cdot B + \bar{A} \cdot \bar{B} \cdot C)$.

Of course there are equations that have no solution, for example,

$$A + C + X \rightarrow A \cdot B + \bar{B} \cdot C$$

Some Further Notation. At this point it is important to discuss some further notation. We have noted above that occasionally subscripts were used to distinguish among elementary elements, as in $A_1, A_2, \ldots, A_n$. Superscripts, on the other hand, will be used to denote position in a designation number. *Hence* $\#B^i$ *represents the bit,* 0 *or* 1, *in the ith position* of $\#B$. For example, if $\#B = 0011\ 0011$, then $\#B^3 = 1$, $\#B^4 = 0$, etc.

In this notation we can conveniently rewrite our well-known rules for logical addition, multiplication, and negation of designation numbers as follows:

$$\begin{array}{ll} \#(A + B)^i = \#A^i + \#B^i & \text{where } 1 + 1 = 1 + 0 = 0 + 1 = 1 \\ & \text{and } 0 + 0 = 0 \\ \#(A \cdot B)^i = \#A^i \cdot \#B^i & \text{where } 1 \cdot 0 = 0 \cdot 1 = 0 \cdot 0 = 0 \text{ and } 1 \cdot 1 = 1 \\ \text{and} \quad \#\overline{A^i} = 0, 1 & \text{when } \#A^i = 1, 0, \text{ respectively} \end{array}$$

It is occasionally important to tell explicitly what the elementary elements of a basis are. Thus, *if we wish to talk about a basis for A, B, and C, we would write* $b[A,B,C]$, while for U, V, and W we would write $b[U,V,W]$.

Our superscripts can be used again to denote a position—this time of a column of a basis. Hence $b[A,B,C]^5$ would represent the fifth column of the (standard) basis or

$$\begin{array}{c} 1 \\ 0 \\ 1 \end{array}$$

Solution to Linear Equations. An equation of the type

$$R + P \cdot X + Q \cdot \bar{X} = S$$

where R, P, Q, and S are known elements (elementary or combined) and X is an unknown element (elementary or combined), is called a *unilateral linear equation in a single unknown* X. The solutions to this equation are combined elements which, when substituted for X in the equation, make the equation true. The solutions are found in terms of designation numbers as follows (where X^i represents the bit 0 or 1 in the ith position of $\#X$):

For the equation $R + P \cdot X + Q \cdot \bar{X} = S$ we can show that, in the designation number of X,

$$\begin{array}{ll} X^i = 1 & \text{if } \overline{S^i} \cdot \overline{(R^i + P^i)} + S^i \cdot (R^i + P^i) = 1 \\ X^i = 0 & \text{if } \overline{S^i} \cdot \overline{(R^i + Q^i)} + S^i \cdot (R^i + Q^i) = 1 \end{array}$$

for i taking the values of all positions. If both conditions are satisfied for a position, two solutions are possible for that position; if neither is satisfied for a position, no solution exists. These formulas can be easily verified as follows: Suppose that $S^i = 1$ for some i; then if $X^i = 1$, $\overline{X^i} = 0$, and the unit of S^i must be contributed by either R^i or $P^i \cdot X^i$, whence

$S^i \cdot (R^i + P^i) = 1$; but if $S^i = 0$ and $X^i = 1$, $\overline{X^i} = 0$, then both R^i and P^i must be zero, or $\overline{S^i} \cdot \overline{R^i} \cdot \overline{P^i} = \overline{S^i} \cdot (\overline{R^i + P^i}) = 1$. Similarly for $X^i = 0$, $\overline{X^i} = 1$.

As an example, consider the equation

$$\bar{A} \cdot \bar{B} \cdot \bar{C} + B \cdot X + (A + \bar{C}) \cdot \bar{X} = A \cdot B + \bar{A} \cdot \bar{C}$$

Here with respect to the usual basis

$$\begin{array}{llll} & & & 45 \\ \#S = & \#(A \cdot B + \bar{A} \cdot \bar{C}) & = 1011 & 0001 \\ \#P = & \#B & = 0011 & 0011 \\ \#R = & \#(\bar{A} \cdot \bar{B} \cdot \bar{C}) & = 1000 & 0000 \\ \#Q = & \#(A \cdot \bar{C}) & = 1111 & 0101 \end{array}$$

Let us illustrate the use of the formulas for the specific cases of $i = 5$. We have

$$\begin{aligned} \overline{S^5} \cdot (\overline{R^5 + P^5}) + S^5 \cdot (R^5 + P^5) &= \bar{0} \cdot (\overline{0 + 0}) + 0 \cdot (0 + 0) \\ &= 1 \cdot (1) + 0 \cdot (0) = 1 \end{aligned}$$

and hence X^5 can be 1. Also,

$$\begin{aligned} \overline{S^5} \cdot (\overline{R^5 + Q^5}) + S^5 \cdot (R^5 + Q^5) &= \bar{0} \cdot (\overline{0 + 1}) + 0 \cdot (0 + 1) \\ &= 1 \cdot (0) + 0 \cdot (1) = 0 \end{aligned}$$

and hence X^5 *cannot* be 0. Thus 1 is written for the fifth position of the designation number of X.

Consider $i = 4$. We have

$$\begin{aligned} \overline{S^4} \cdot (\overline{R^4 + P^4}) + S^4 \cdot (R^4 + P^4) &= \bar{0} \cdot (\overline{0 + 0}) + 0 \cdot (0 + 0) \\ &= 1 \cdot (1) + 0 \cdot (0) = 1 \\ \overline{S^4} \cdot (\overline{R^4 + Q^4}) + S^4 \cdot (R^4 + Q^4) &= \bar{0} \cdot (\overline{0 + 0}) + 0 \cdot (0 + 0) \\ &= 1 \cdot (1) + 0 \cdot (0) = 1 \end{aligned}$$

Hence X^4 can be 0 or 1 (producing two different solutions). We shall denote this by writing Φ in the fourth position of the designation number for X.

Following this procedure for $i = 0, 1, 2, 3, 4, 5, 6$, and 7, we find $\#X = \Phi 1 \Phi \Phi \;\; \Phi 1 0 \Phi$. Hence there are $2^5 = 32$ solutions.

An equation of the type $R + P \cdot X = Q \cdot \bar{X} + S$ is called a bilateral equation in a single unknown X. The solutions for

$$R + P \cdot X = Q \cdot \bar{X} + S$$

are

$$\begin{array}{ll} X^i = 1 & \text{if } \overline{S^i} \cdot (\overline{R^i + P^i}) + S^i \cdot (R^i + P^i) = 1 \\ X^i = 0 & \text{if } \overline{R^i} \cdot (\overline{S^i + Q^i}) + R^i \cdot (S^i + Q^i) = 1 \end{array}$$

for i taking the values of all positions. If neither condition is satisfied for a position, no solution exists.

For example, consider the equation

$$(B \cdot \bar{C} + A \cdot \bar{B} \cdot C) + (C + \bar{A} \cdot B) \cdot X$$
$$= (\bar{A} \cdot \bar{C} + B \cdot \bar{C} + A \cdot \bar{B} \cdot C) \cdot \bar{X} + \bar{B} \cdot C$$

Here

$$\begin{aligned} \#S &= \#(\bar{B} \cdot C) &&= 0000 \quad 1100 \\ \#P &= \#(C + \bar{A} \cdot B) &&= 0010 \quad 1111 \\ \#R &= \#(B \cdot \bar{C} + A \cdot \bar{B} \cdot C) &&= 0011 \quad 0100 \\ \#Q &= \#(\bar{A} \cdot \bar{C} + B \cdot \bar{C} + A \cdot \bar{B} \cdot C) &&= 1011 \quad 0100 \end{aligned}$$

and therefore $\#X = 1\Phi00 \quad 1\Phi00$, with $2^2 = 4$ solutions.

Simultaneous Linear Equations. Consider now m simultaneous equations with a single unknown of either the unilateral or the bilateral type or both. The following procedure will produce all solutions each of which satisfies all the m equations. We first take the solutions of each equation and place them in a column beneath each other. To find those solutions which satisfy all the m equations simultaneously, we consider each digital position separately and "multiply" the respective digits of various solutions according to the following table (where Φ is used as above):

"Multiply"	0	1	Φ
0	0	None	0
1	None	1	1
Φ	0	1	Φ

For example, let us find the solution which satisfies both the equations solved above:

$$\begin{aligned} \text{Solution to 1st equation} &= \Phi 1 \Phi\Phi \quad \Phi 10 \Phi \\ \text{Solution to 2d equation} &= 1\Phi 00 \quad 1\Phi 00 \\ \text{Desired result} = \text{“product” of solutions} &= 1100 \quad 1100 = \#\bar{B} \end{aligned}$$

Hence $X = \bar{B}$ is the only solution that satisfies both the above equations.

EXERCISES

(*a*) Solve the following equation for X as a function of A, B, and C:

$$\bar{A} \cdot \bar{B} \cdot \bar{C} + X \cdot \bar{C} \cdot (A + B) + \bar{X} \cdot C \cdot (\bar{A} + \bar{B}) = A \cdot \bar{C} + \bar{A} \cdot C - \bar{B} \cdot \bar{C}$$

Solution. $A \cdot \bar{C}$, $(A + B) \cdot \bar{C}$, $A \cdot (B + \bar{C})$, $A \cdot B + \bar{B} \cdot \bar{C}$.

(*b*) How many solutions are there to the following equation?

$$X \cdot (A \cdot B \cdot C + \bar{A} \cdot \bar{B}) = A \cdot B \cdot C + \bar{A} \cdot \bar{B} \cdot \bar{C} \qquad \textit{Ans. } 2^5 = 32$$

(*c*) How many solutions are there to the following implication?

$$A \cdot B + \bar{A} \cdot \bar{B} \cdot \bar{C} \rightarrow X \cdot (A \cdot B + B \cdot C + A \cdot C + \bar{B} \cdot \bar{C}) \qquad \textit{Ans. } 2^5 = 32$$

(*d*) Find X if

$$X \cdot (A \cdot B \cdot C + \bar{A} \cdot \bar{B}) = A \cdot B \cdot C + \bar{A} \cdot \bar{B} \cdot \bar{C}$$

(*e*) Find X if

$$X \cdot \bar{C} \cdot (A + B) + \bar{A} \cdot C = \bar{X} \cdot \bar{C} \cdot (\bar{A} + \bar{B}).$$

(*f*) Solve the equations in (*d*) and (*e*) simultaneously.
Solution. $\#X = 1000\ 0101 = \#(\bar{A} \cdot \bar{B} \cdot \bar{C} + A \cdot C)$.

12-6. The General Method for Solution to Any Number of Simultaneous Equations in Any Number of Unknowns

Equivalence. The general method for obtaining all solutions to an *equivalence* in many unknowns is best illustrated by an example. Consider the equation

$$A + B \cdot X \cdot \bar{Y} + C \cdot Y = \bar{A} \cdot \bar{X} \cdot \bar{Y} + \bar{B} \cdot \bar{X} + (C + A \cdot B) \qquad (12\text{-}1)$$

Recall that in the solution of equivalence equations the fundamental principle is that *the designation numbers of each side of the equation must be identical.* The system to be given here is designed to facilitate the calculation of those designation numbers for the unknown elements X and Y which fulfill this condition. The designation numbers of the unknowns are calculated one position at a time so that this position becomes the same for the total designation number of each side of the equation.

The solutions for each position i of the unknowns must be among the set: $(X^i,Y^i) = (0,0); (1,0); (0,1); (1,1)$. With the aid of the work sheet shown in Table 12-1 each of these pairs is substituted into the equation; if a pair makes both sides equal, it is admitted as a possible solution.

TABLE 12-1. WORK SHEET

Coefficients and designation numbers				Corresponding unknown	Possible pairs of solutions			
					X^i: 0	1	$\boxed{0}$	1
					Y^i: 0	0	1	1
	0123	$\boxed{4}$	567					
$\#A$ =	0101	0	101	I	1	1	1	1
$\#B$ =	0011	0	011	$X \cdot \bar{Y}$	0	1	0	0
$\#C$ =	0000	1	111	Y	0	0	1	1
$\#\bar{A}$ =	1010	1	010	$\bar{X} \cdot \bar{Y}$	1	0	0	0
$\#\bar{B}$ =	1100	1	100	$\bar{X}$	1	0	1	0
$\#(C + A \cdot B)$ =	0001	1	111	I	1	1	1	1

RESULT ARRAY

i	0	1	2	3	4	5	6	7
$\#X$	1, 1	0, 0	0, 1	0, 1, 0, 1	0, 1	0, 1, 0, 1	1, 0, 1	0, 1, 0, 1
$\#Y$	0, 1	0, 1	1, 1	0, 0, 1, 1	1, 1	0, 0, 1, 1	0, 1, 1	0, 0, 1, 1

On the work sheet are put the designation numbers of the coefficient of each unknown and the value corresponding to that unknown computed with each of the possible pairs. Thus in our example the coefficient of $X \cdot \bar{Y}$ is B, 0011 0011. When the pair (0,0) is substituted into $X \cdot \bar{Y}$, the result is zero; for (1,0), a unit; for (0,1), a zero; for (1,1), a zero. Each line is treated similarly; but observe that for terms not involving either unknown the tautological element I is placed in the unknown column. To check the ith position of both sides of the equation for a pair of solutions, the column corresponding to that solution is logically multiplied by the column of the coefficients corresponding to the ith position. The results above and those below the line are added logically, and the two sums are compared. If the sums are the same, that pair of solutions is admissible for that position. Consider in our example the pair (0,1) and the fourth position (see Table 12-2). Then

TABLE 12-2

$$\begin{array}{c}
0 \cdot 1 = 0 \\
0 \cdot 0 = 0 \\
1 \cdot 1 = 1 \\
\hline
1 \cdot 0 = 0 \\
1 \cdot 1 = 1 \\
1 \cdot 1 = 1
\end{array}$$

$$0 \cdot 1 + 0 \cdot 0 + 1 \cdot 1 = 1$$

as does

$$1 \cdot 0 + 1 \cdot 1 + 1 \cdot 1 = 1$$

Therefore the pair (0,1) is admissible as a solution for the fourth position and is recorded in the result array. The pair (1,1) is also a solution for this position, but (1,0) and (0,0) are not. When this procedure is repeated for all combinations, we obtain the results as shown in Table 12-1. There are thus

$$2 \times 2 \times 2 \times 4 \times 2 \times 4 \times 3 \times 4 = 3{,}072$$

different nonequivalent pairs of solutions to the equivalence equation of the example. Note that *all* nonequivalent pairs of solutions have been obtained.

Simultaneous Equivalences. The general method for obtaining all solutions to any number of *simultaneous equivalences* in many unknowns is again best illustrated by an example.

Consider the two simultaneous equations

$$A + B \cdot X \cdot \bar{Y} + C \cdot Y = \bar{A} \cdot \bar{X} \cdot \bar{Y} + \bar{B} \cdot \bar{X} + (C + A \cdot B) \quad (12\text{-}1)$$
$$B \cdot X \cdot Y + C \cdot (A + B) \cdot Y = \bar{B} \cdot \bar{X} \cdot \bar{Y} \quad (12\text{-}2)$$

We want to find combined elements that when substituted for X and Y in the equations will make both true simultaneously. The method for obtaining these solutions is evidently to record in the table of results only

TABLE 12-3. WORK SHEET

Coefficients and designation numbers			Corresponding unknown	Possible pairs of solutions			
				X^i: 0	1	0	1
				Y^i: 0	0	1	1
	0123	4567					
$\#A$	= 0101	0101	I	1	1	1	1
$\#B$	= 0011	0011	$X \cdot \bar{Y}$	0	1	0	0
$\#C$	= 0000	1111	Y	0	0	1	1
$\#\bar{A}$	= 1010	1010	$\bar{X} \cdot \bar{Y}$	1	0	0	0
$\#\bar{B}$	= 1100	1100	$\bar{X}$	1	0	1	0
$\#(C + A \cdot B)$	= 0001	1111	I	1	1	1	1
$\#B$	= 0011	0011	$X \cdot Y$	0	0	0	1
$\#[C \cdot (A + B)]$	= 0000	0111	Y	0	0	1	1
$\#\bar{B}$	= 1100	1100	$\bar{X} \cdot \bar{Y}$	1	0	0	0

RESULT ARRAY

i	0	1	2	3	4	5	6	7
$\#X$	1, 1	0	0	0, 1, 0	0, 1	1	1	0, 1
$\#Y$	0, 1	1	1	0, 0, 1	1, 1	0	0	0, 0

those position solutions which satisfy *both* the simultaneous equations. The procedure is analogous to that of the single equation (see Table 12-3). Now there remain $2 \times 1 \times 1 \times 3 \times 2 \times 1 \times 1 \times 2 = 24$ nonequivalent pairs of solutions to both these equations.

Simultaneous Equivalences and Implications. The solution to an *implication* in several unknowns is obtained in the way described above for an equivalence, except that, for the implication $L \rightarrow R$, it is not necessary that R and L have identical numbers; but R must have a unit at least wherever L has a unit. Thus for implications one follows the whole procedure described above, except that for any digital position the result *is not* recorded only when there is a calculated unit above the line corresponding to a calculated zero below (here we assume L above the line and R below).

The extension of these results to simultaneous implications as well as to mixed systems of simultaneous equivalences and implications is immediate. For example, consider the following mixed system of simultaneous statements:

$$A + B \cdot X \cdot \bar{Y} + C \cdot Y = \bar{A} \cdot \bar{X} \cdot \bar{Y} + \bar{B} \cdot \bar{X} + (C + A \cdot B) \qquad (12\text{-}1)$$

$$B \cdot X \cdot Y + C \cdot (A + B) \cdot Y = \bar{B} \cdot \bar{X} \cdot \bar{Y} \qquad (12\text{-}2)$$

$$A \cdot C \cdot (X + Y) + A \cdot B \cdot (X + \bar{Y}) \rightarrow X \cdot Y + A \cdot C \qquad (12\text{-}3)$$

TABLE 12-4. WORK SHEET

Coefficients and designation numbers			Corresponding unknown	Possible pairs of solutions				
				X^i:	0	1	0	1
	0123	4567		Y^i:	0	0	1	1
$\#A =$	0101	0101	I		1	1	1	1
$\#B =$	0011	0011	$X \cdot \bar{Y}$		0	1	0	0
$\#C =$	0000	1111	Y		0	0	1	1
$\#\bar{A} =$	1010	1010	$\bar{X} \cdot \bar{Y}$		1	0	0	0
$\#\bar{B} =$	1100	1100	$\bar{X}$		1	0	1	0
$\#(C + A \cdot B) =$	0001	1111	I		1	1	1	1
$\#B =$	0011	0011	$X \cdot Y$		0	0	0	1
$\#[C \cdot (A + B)] =$	0000	0111	Y		0	0	1	1
$\#\bar{B} =$	1100	1100	$\bar{X} \cdot \bar{Y}$		1	0	0	0
$\#(A \cdot C) =$	0000	0101	$X + Y$		0	1	1	1
$\#(A \cdot B) =$	0001	0001	$X + \bar{Y}$		1	1	0	1
$\#I =$	1111	1111	$X \cdot Y$		0	0	0	1
$\#(A \cdot C) =$	0000	0101	I		1	1	1	1

RESULT ARRAY

i	0	1	2	3	4	5	6	7
$\#X$	1, 1	0	0	0	0, 1	1	1	0, 1
$\#Y$	0, 1	1	1	1	1, 1	0	0	0, 0

The work leading to the results is shown in Table 12-4. Note that a dashed line is used to denote implication. There remain only

$$2 \times 1 \times 1 \times 1 \times 2 \times 1 \times 1 \times 2 = 8$$

pairs of nonequivalent solutions for this system of two equivalences and one implication. Written out explicitly, the eight pairs of solutions are

$$\begin{array}{ll}
X = \bar{A} \cdot (B \cdot C + \bar{B} \cdot \bar{C}) + A \cdot \bar{B} \cdot C & X = C \cdot (A + B) + \bar{A} \cdot \bar{B} \cdot \bar{C} \\
Y = \bar{C} \cdot (A + B) + \bar{A} \cdot \bar{B} \cdot C & Y = \bar{C} \cdot (A + B) + \bar{A} \cdot \bar{B} \cdot C \\
X = \bar{A} \cdot (B \cdot C + \bar{B} \cdot \bar{C}) + A \cdot \bar{B} \cdot C & X = C \cdot (A + B) + \bar{A} \cdot \bar{B} \cdot \bar{C} \\
Y = \bar{C} + \bar{A} \cdot \bar{B} & Y = \bar{C} + \bar{A} \cdot \bar{B} \\
X = C \cdot (\bar{A} + \bar{B}) + \bar{A} \cdot \bar{B} & X = C + \bar{A} \cdot \bar{B} \\
Y = \bar{C} \cdot (A + B) + \bar{A} \cdot \bar{B} \cdot C & Y = \bar{C} \cdot (A + B) + \bar{A} \cdot \bar{B} \cdot C \\
X = C \cdot (\bar{A} + \bar{B}) + \bar{A} \cdot \bar{B} & X = C + \bar{A} \cdot \bar{B} \\
Y = \bar{C} + \bar{A} \cdot \bar{B} & Y = \bar{C} + \bar{A} \cdot \bar{B}
\end{array}$$

One can verify any of these pairs by direct substitution back into the original three statements.

EXERCISES

Solve the following sets of simultaneous Boolean equations for the unknowns X and Y:

(*a*) $(A + B) \cdot X + A \cdot \bar{X} \cdot \bar{Y} = A,\ \bar{A} \cdot \bar{Y} + X = A + B,\ X \cdot Y \rightarrow B.$
Solution. $X = A,\ Y = A \cdot B + \bar{A} \cdot \bar{B};\ X = A,\ Y = A + \bar{B}.$

WORK SHEET FOR EXERCISE *a*

Coefficients and designation numbers	Corresponding unknown	Possible pairs of solutions				
0123		X^i:	0	1	0	1
		Y^i:	0	0	1	1
$\#(A + B) = 0111$	X		0	1	0	1
$\#A = 0101$	$\bar{X} \cdot \bar{Y}$		1	0	0	0
$\#A = 0101$	I		1	1	1	1
$\#\bar{A} = 1010$	$\bar{Y}$		1	1	0	0
$\#I = 1111$	X		0	1	0	1
$\#(A + B) = 0111$	I		1	1	1	1
$\#I = 1111$	$X \cdot Y$		0	1	0	0
$\#\bar{B} = 1100$	I		1	1	1	1

RESULT ARRAY

i	0	1	2	3
$\#X$	0	1, 1	0	1
$\#Y$	1	0, 1	0	1

$0101 = A$
$1001 = A \cdot B + \bar{A} \cdot \bar{B}$

$0101 = A$
$1101 = A + \bar{B}$

(*b*) $X \cdot Y = A \cdot \bar{B},\ A + B = \bar{A} \cdot Y + X,\ (A + B) \cdot X + A \cdot Y = A.$
Solution. $X = A,\ Y = A \cdot \bar{B} + \bar{A} \cdot B.$

(*c*) $X \cdot Y = A \cdot \bar{B},\ \bar{A} \cdot B = \bar{A} \cdot Y + X,\ (A + B) \cdot X + A \cdot Y = A.$
No solution. Why?

(*d*) $X \cdot (A + B) + Y \cdot (A + C) = (\bar{X} \cdot \bar{Y} + X \cdot Y) \cdot (\bar{A} + \bar{B} + \bar{C}),$
$X \cdot (A \cdot B + \bar{A} \cdot \bar{B}) \rightarrow Y + A \cdot B + C.$

There are four sets of solutions:

$X = A \cdot \bar{C} + (\bar{A} + \bar{B}) \cdot C$
$Y = \bar{C} + A \cdot \bar{B} + \bar{A} \cdot B$

$X = A \cdot \bar{C} + (\bar{A} + \bar{B}) \cdot C$
$Y = \bar{C} + (\bar{A} + \bar{B}) \cdot C$

$X = (A + B) \cdot \bar{C} + (\bar{A} + \bar{B}) \cdot C$
$Y = \bar{C} + A \cdot \bar{B} + \bar{A} \cdot B$

$X = (A + B) \cdot \bar{C} + (\bar{A} + \bar{B}) \cdot C$
$Y = \bar{C} + (\bar{A} + \bar{B}) \cdot C$

12-7. Solution to Equations in Circuit Design

Examples. In circuit design, problems that involve solutions of equations are often of an exploratory nature—for example, they may arise in the investigation of various alternative circuits. The method of solution to equations enables the design engineer to determine what additional circuitry will be needed if combinations of existing circuits are to be used to produce a given desired result. As in ordinary algebraic problems the formulation of the equations requires ingenuity in interpreting the physical situations in terms of mathematical concepts.

For example, suppose that the circuit shown in Fig. 12-7 had already been constructed. It is desired to have the output be

$$Z = A \cdot B + A \cdot \bar{C} + B \cdot \bar{C} + \bar{B} \cdot C$$

The problem is to design two circuits, with inputs A, B, and C, such that when their outputs are attached to input wires X and Y, the desired Z will be obtained. As the circuit now stands,

$$Z = (A + B) \cdot X \cdot \bar{Y} + (\bar{A} + \bar{B}) \cdot Y$$

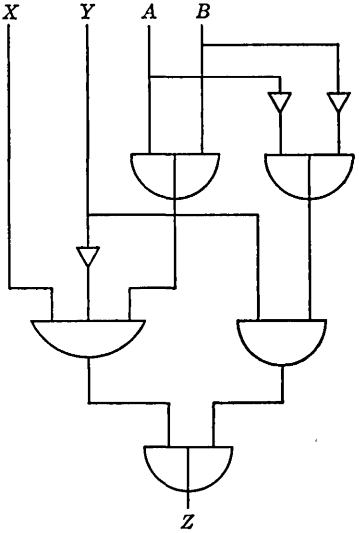


FIG. 12-7. Circuit output of $Z = (A + B) \cdot X \cdot \bar{Y} + (\bar{A} + \bar{B}) \cdot Y$.

and hence we want to find $X = X(A,B,C)$, $Y = Y(A,B,C)$ such that

$$(A + B) \cdot X \cdot \bar{Y} + (\bar{A} + \bar{B}) \cdot Y = A \cdot B + A \cdot \bar{C} + B \cdot \bar{C} + \bar{B} \cdot C$$

Solving this equation by the method of the previous section, we obtain the result array:

i	0	1	2	3	4	5	6	7
$\#X$	0, 1	1, 0, 1	1, 0, 1	1	0, 1	1, 0, 1	0	1
$\#Y$	0, 0	0, 1, 1	0, 1, 1	0	1, 1	0, 1, 1	0	0

and hence there are $2 \cdot 3 \cdot 3 \cdot 1 \cdot 2 \cdot 3 \cdot 1 \cdot 1 = 108$ sets of solutions. The problem now reduces to choosing from these a *simple* solution. Note that we might choose

$$\#X = 0101 \quad 0101$$
$$\#Y = 0\phi10 \quad 1\phi00$$

where $X = A$. However, even though we would have $2 \times 2 = 4$ choices for Y, none of them is very simple.

From the result array it is easily determined by means of the elimination pairs that Y must include $\bar{B}$ and C. Choosing the only Y that does not involve A, we find $\# Y = 0000 \quad 1100 = \bar{B} \cdot C$. With this Y we must have $\# X = \Phi 111 \quad \Phi\Phi 01$, from which we can choose

$$\# X = 1111 \quad 0101 = A + \bar{C}$$

Thus as a simple solution to the problem we have the circuits of Fig. 12-8. This example has illustrated that solving the required equation is not always enough. From the various alternative solutions must be chosen the one that is most desirable and suited to the situation at hand. Often the criterion of simplicity is used.

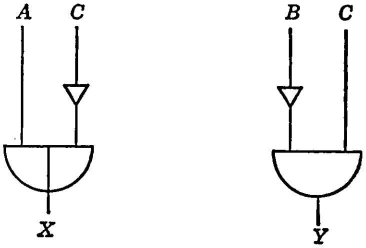


FIG. 12-8. Circuits for $X = A + \bar{C}$, $Y = \bar{B} \cdot C$.

As another example, suppose that the circuit of Fig. 12-9 has already been constructed and that our problem is to design circuits with outputs X and Y such that $Z = \bar{A} \cdot B \cdot \bar{C} + (\bar{A} + B) \cdot C$ and such that neither X nor Y will involve A. First note that

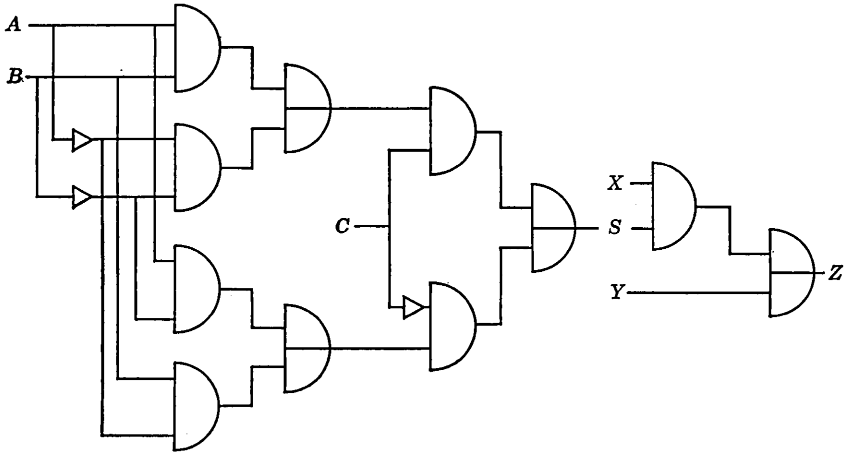


FIG. 12-9. Circuit for $Z = X \cdot [(A \cdot B + \bar{A} \cdot \bar{B}) \cdot C + (A \cdot \bar{B} + \bar{A} \cdot B) \cdot \bar{C}] + Y$.

$$S = (A \cdot B + \bar{A} \cdot \bar{B}) \cdot C + (A \cdot \bar{B} + \bar{A} \cdot B) \cdot \bar{C}$$

whence we wish to solve the following equation for X and Y in terms of A, B, and C:

$$X \cdot [(A \cdot B + \bar{A} \cdot \bar{B}) \cdot C + (A \cdot \bar{B} + \bar{A} \cdot B) \cdot \bar{C}] + Y \\ = \bar{A} \cdot B \cdot \bar{C} + (\bar{A} + B) \cdot C$$

Observing the result array with respect to the elimination pairs of A, we find the following set of solutions not involving A:

$$X = B + C \qquad Y = B \cdot C$$

As an example in simultaneous equations, suppose that the two circuits Z_1 and Z_2 shown in Fig. 12-10 are already constructed. It is desired to design two circuits such that, when their outputs are connected to X and Y of Z_1, there will result $Z_1 = A \cdot \bar{B} + \bar{A} \cdot B + C$ and, when connected to Z_2, there will result $Z_2 = A \cdot \bar{B} \cdot \bar{C} + (A \cdot B + \bar{A} \cdot \bar{B}) \cdot C$. Here we are required to solve the two simultaneous equations

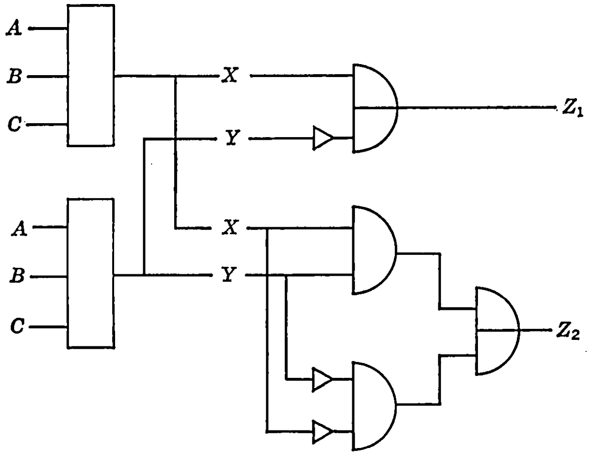


FIG. 12-10. Circuits for $Z_1 = X + \bar{Y}$, $Z_2 = X \cdot Y + \bar{X} \cdot \bar{Y}$.

$$X + \bar{Y} = A \cdot \bar{B} + \bar{A} \cdot B + C$$
$$X \cdot Y + \bar{X} \cdot \bar{Y} = A \cdot \bar{B} \cdot \bar{C} + (A \cdot B + \bar{A} \cdot \bar{B}) \cdot C$$

One simple result is $X = \bar{A} \cdot B + C$ and $Y = A \cdot B + \bar{A} \cdot \bar{B}$.

EXERCISES

(*a*) Suppose that the circuit of Fig. 12-11 has already been constructed and that it is desired to have $Z = A \cdot B \cdot \bar{C} + (\bar{A} \cdot B + A \cdot \bar{B}) \cdot C$. The problem is to design

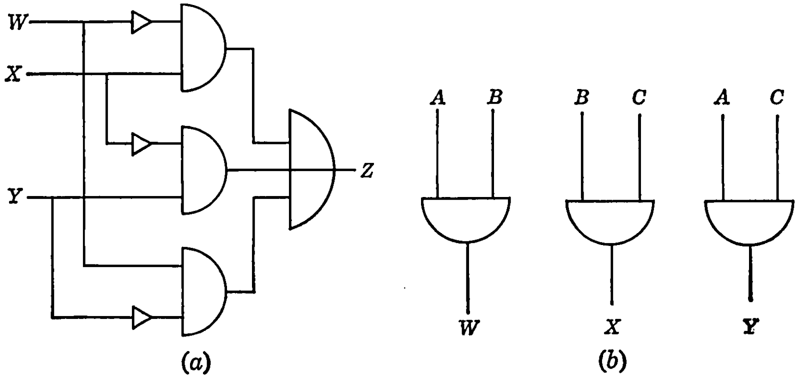


FIG. 12-11. (*a*) Circuit for Exercise *a*; (*b*) a simple solution.

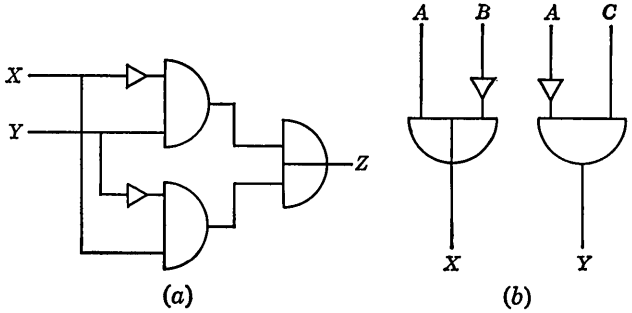


FIG. 12-12. (*a*) Circuit for Exercise *b*; (*b*) a simple solution.

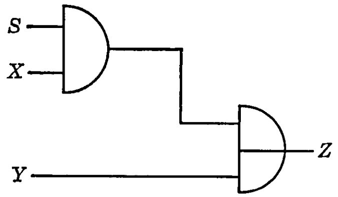


FIG. 12-13. Circuit for Exercise *c*.

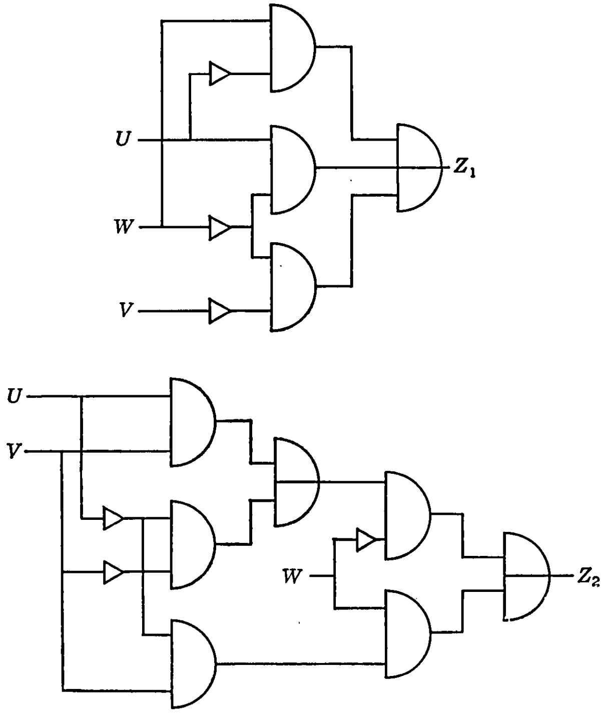


FIG. 12-14. Circuits for Exercise *d*.

simple circuits which would each have inputs A, B, and C such that, when their outputs are connected to W, X, and Y, the output of the given circuit is the desired Z. One solution is shown in the figure.

(b) Suppose that the circuit of Fig. 12-12 had already been constructed. Design simple circuits with outputs to be connected to X and Y to give $Z = (A + \bar{B}) \cdot \bar{C} + (A + B) \cdot C$. One simple solution is shown in the figure.

(c) Suppose that a circuit had been constructed with output $S = (A \cdot B + \bar{A} \cdot \bar{B}) \cdot \bar{C} + (A \cdot \bar{B} + \bar{A} \cdot B) \cdot C$. It is desired to transform this circuit as shown in Fig. 12-13 so that the output will be $Z = (A + \bar{B}) \cdot \bar{C} + (\bar{A} + \bar{B}) \cdot C$, as shown. The problem is to find simple circuits for X and Y. One solution is $X = B$ and $Y = \bar{B}$.

(d) Suppose that the two circuits of Fig. 12-14 have already been constructed. Design three circuits such that, if their outputs are connected, respectively, to U, V, and W, then $Z_1 = A \cdot \bar{B} + \bar{A} \cdot B + C$ and $Z_2 = A \cdot \bar{B} \cdot \bar{C} + (A \cdot B + \bar{A} \cdot \bar{B}) \cdot C$. One set of solutions is $U = A \cdot (\bar{B} + \bar{C})$, $V = \bar{B} \cdot \bar{C} + A \cdot B$, and $W = B$.

12-8. The Absolute Simplest Form and Change of Variables

The Absolute Simplest Form. It can be shown that a designation number for n elementary elements with u units *cannot be represented as a combined element with fewer than $n - k - 1$ operations* of the type $\cdot$ and $+$, where k is the largest integer such that 2^k is a factor of u. An element with this minimum number of operations is called an *absolute simplest form* and can easily be systematically constructed. Some of them are given in Table 12-5.

A method for constructing an absolute simplest form, given n and u, involves four steps as follows: (1) First write u as a binary number. (2) Fill in zeros to the left of this binary number to make n bits altogether. (3) Starting at the left, number these bits by $n, n - 1, n - 2, \ldots, 1$. (4) Then, denoting the elementary elements by $A_n, A_{n-1}, \ldots, A_1$, an absolute simplest form is obtained by working successively from left to right and corresponding to the ith bit placing "$A_i \cdot ($" if the ith bit is a zero and placing "$A_i + ($" if the ith bit is a unit. This process ends on the last (rightmost) *unit* of the binary number u.

For example, let us find an absolute simplest form for a designation number with ten units and five elementary elements, that is, $n = 5$ and $u = 10$ (*decimal*). Since $u = 1010$ (*binary*), we affix one zero to the left and number the positions, obtaining

$$\begin{array}{ccccc} 0 & 1 & 0 & 1 & 0 \\ 5 & 4 & 3 & 2 & 1 \end{array}$$

An absolute simplest form is then obtained as indicated,

$$\begin{array}{ccccc} 0 & 1 & 0 & 1 & 0 \\ 5 & 4 & 3 & 2 & 1 \end{array}$$
$$A_5 \cdot (A_4 + (A_3 \cdot (A_2))) = A_5 \cdot (A_4 + A_3 \cdot A_2)$$

where the final expression is in terms of the more common use of parentheses.

As another example, let us construct an absolute simplest form for a designation number that contains 31,792 units in 17 variables. The

TABLE 12-5. ABSOLUTE SIMPLEST FORMS FOR n ELEMENTARY ELEMENTS WHOSE DESIGNATION NUMBERS CONTAIN u UNITS

	$n = 2$	$n = 3$	$n = 4$	$n = 5$
$u = 1$	$A_2 \cdot A_1$	$A_3 \cdot A_2 \cdot A_1$	$A_4 \cdot A_3 \cdot A_2 \cdot A_1$	$A_5 \cdot A_4 \cdot A_3 \cdot A_2 \cdot A_1$
$u = 2$	A_2	$A_3 \cdot A_2$	$A_4 \cdot A_3 \cdot A_2$	$A_5 \cdot A_4 \cdot A_3 \cdot A_2$
$u = 3$	$A_2 + A_1$	$A_3 \cdot (A_2 + A_1)$	$A_4 \cdot A_3 \cdot (A_2 + A_1)$	$A_5 \cdot A_4 \cdot A_3 \cdot (A_2 + A_1)$
$u = 4$	I	A_3	$A_4 \cdot A_3$	$A_5 \cdot A_4 \cdot A_3$
$u = 5$		$A_3 + A_2 \cdot A_1$	$A_4 \cdot (A_3 + A_2 \cdot A_1)$	$A_5 \cdot A_4 \cdot (A_3 + A_2 \cdot A_1)$
$u = 6$		$A_3 + A_2$	$A_4 \cdot (A_3 + A_2)$	$A_5 \cdot A_4 \cdot (A_3 + A_2)$
$u = 7$		$A_3 + A_2 + A_1$	$A_4 \cdot (A_3 + A_2 + A_1)$	$A_5 \cdot A_4 \cdot (A_3 + A_2 + A_1)$
$u = 8$		I	A_4	$A_5 \cdot A_4$
$u = 9$			$A_4 + A_3 \cdot A_2 \cdot A_1$	$A_5 \cdot (A_4 + A_3 \cdot A_2 \cdot A_1)$
$u = 10$			$A_4 + A_3 \cdot A_2$	$A_5 \cdot (A_4 + A_3 \cdot A_2)$
$u = 11$			$A_4 + A_3 \cdot (A_2 + A_1)$	$A_5 \cdot (A_4 + A_3 \cdot (A_2 + A_1))$
$u = 12$			$A_4 + A_3$	$A_5 \cdot (A_4 + A_3)$
$u = 13$			$A_4 + A_3 + A_2 \cdot A_1$	$A_5 \cdot (A_4 + A_3 + A_2 \cdot A_1)$
$u = 14$			$A_4 + A_3 + A_2$	$A_5 \cdot (A_4 + A_3 + A_2)$
$u = 15$			$A_4 + A_3 + A_2 + A_1$	$A_5 \cdot (A_4 + A_3 + A_2 + A_1)$
$u = 16$			I	A_5
$u = 17$				$A_5 + A_4 \cdot A_3 \cdot A_2 \cdot A_1$
$u = 18$				$A_5 + A_4 \cdot A_3 \cdot A_2$
$u = 19$				$A_5 + A_4 \cdot A_3 \cdot (A_2 + A_1)$
$u = 20$				$A_5 + A_4 \cdot A_3$
$u = 21$				$A_5 + A_4 \cdot (A_3 + A_2 \cdot A_1)$
$u = 22$				$A_5 + A_4 \cdot (A_3 + A_2)$
$u = 23$				$A_5 + A_4 \cdot (A_3 + A_2 + A_1)$
$u = 24$				$A_5 + A_4$
$u = 25$				$A_5 + A_4 + A_3 \cdot A_2 \cdot A_1$
$u = 26$				$A_5 + A_4 + A_3 \cdot A_2$
$u = 27$				$A_5 + A_4 + A_3 \cdot (A_2 + A_1)$
$u = 28$				$A_5 + A_4 + A_3$
$u = 29$				$A_5 + A_4 + A_3 + A_2 \cdot A_1$
$u = 30$				$A_5 + A_4 + A_3 + A_2$
$u = 31$				$A_5 + A_4 + A_3 + A_2 + A_1$
$u = 32$				I

binary representation of 31,792, with two zeros affixed to the left, is

$$\begin{array}{ccccccccccccccccc} 0 & 0 & 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 \\ 17 & 16 & 15 & 14 & 13 & 12 & 11 & 10 & 9 & 8 & 7 & 6 & 5 & 4 & 3 & 2 & 1 \end{array}$$

Thus an absolute simplest form (removing unnecessary parentheses) will be

$$A_{17} \cdot A_{16} \cdot (A_{15} + A_{14} + A_{13} + A_{12} + A_{11} \\ + A_{10} \cdot A_9 \cdot A_8 \cdot A_7 \cdot (A_6 + A_5))$$

Corresponding to a given n and u, there are in general many possible absolute simplest forms. For example, for $n = 3$, $u = 5$, we could have as well as $A_3 + A_2 \cdot A_1$ the 24 nonequivalent functions given in Table 12-6. In fact it might be wise to say that the absolute simplest *form* in this case is $X_3 + X_2 \cdot X_1$, where X_1 can be A_1 or $\bar{A}_1$, A_2 or $\bar{A}_2$, or A_3 or $\bar{A}_3$ provided that the same symbol is not used for more than one X_i. Note, however, that, since the operations $\cdot$ and $+$ are each commutative, not all possible combinations give nonequivalent functions.

TABLE 12-6. ALL 24 POSSIBLE ABSOLUTE SIMPLEST FORMS FOR $n = 3, u = 5$

$A_3 + A_2 \cdot A_1$	$A_2 + A_3 \cdot A_1$	$A_1 + A_3 \cdot A_2$
$\bar{A}_3 + A_2 \cdot A_1$	$\bar{A}_2 + A_3 \cdot A_1$	$\bar{A}_1 + A_3 \cdot A_2$
$A_3 + \bar{A}_2 \cdot A_1$	$A_2 + \bar{A}_3 \cdot A_1$	$A_1 + \bar{A}_3 \cdot A_2$
$A_3 + A_2 \cdot \bar{A}_1$	$A_2 + A_3 \cdot \bar{A}_1$	$A_1 + A_3 \cdot \bar{A}_2$
$\bar{A}_3 + \bar{A}_2 \cdot A_1$	$\bar{A}_2 + \bar{A}_3 \cdot A_1$	$\bar{A}_1 + \bar{A}_3 \cdot A_2$
$\bar{A}_3 + A_2 \cdot \bar{A}_1$	$\bar{A}_2 + A_3 \cdot \bar{A}_1$	$\bar{A}_1 + A_3 \cdot \bar{A}_2$
$A_3 + \bar{A}_2 \cdot \bar{A}_1$	$A_2 + \bar{A}_3 \cdot \bar{A}_1$	$A_1 + \bar{A}_3 \cdot \bar{A}_2$
$\bar{A}_3 + \bar{A}_2 \cdot \bar{A}_1$	$\bar{A}_2 + \bar{A}_3 \cdot \bar{A}_1$	$\bar{A}_1 + \bar{A}_3 \cdot \bar{A}_2$

Change of Variables. The question naturally arises as to the value of absolute simplest forms. The answer to this concerns the concept of *change of variables,* which we shall now discuss. The concept of change of variables in Boolean algebra is similar to the concept of change of variables in ordinary algebra. If A, B, and C are elementary elements, we perform a change of variables by letting

$$A = A(A',B',C') \qquad B = B(A',B',C') \qquad C = C(A',B',C')$$

where A', B', C' are *new* variables.

If A, B, and C are independent, we must be sure that the new variables are independent, i.e., that no dependence between A, B, and C is introduced by the particular Boolean functions chosen for $A(A',B',C')$, $B(A',B',C')$, and $C(A',B',C')$. For example, suppose that we wish to change from variables A and B to new variables A' and B' by the *transformation* $A = A' \cdot B' + \bar{A}' \cdot \bar{B}'$, $B = \bar{A}'$. It can easily be seen with respect to the standard basis $b[A',B']$ that $A' \cdot B' + \bar{A}' \cdot \bar{B}'$ and $\bar{A}'$ are independent; so the transformation is *valid,* i.e., keeps A and B independent. *Hence all Boolean functions of A and B can now be written in terms of A' and B'.* For example, in the new variables

$$A \cdot \bar{B} + \bar{A} \cdot B = (A' \cdot B' + \bar{A}' \cdot \bar{B}') \cdot A' + (\overline{A' \cdot B' + \bar{A}' \cdot \bar{B}'}) \cdot \bar{A}' = B'$$

In addition the new (primed) variables can be found in terms of the old (unprimed) variables. In our case $A' = \bar{B}$, $B' = A \cdot \bar{B} + \bar{A} \cdot B$.

Now we can again consider the problem of the value of the absolute simplest form. For any combined element we can, by counting the units in its designation number, find its corresponding absolute simplest form. This form can always be obtained by making a change of variables; i.e., *a particular change of variables can be made so that the given Boolean function will be transformed into its absolute simplest form in terms of the new variables.* The need for such a technique in circuit design can be understood from the following consideration: First the circuit that generates an absolute simplest form cannot be further simplified. In some cases there will be a saving in circuitry if a given function is formed by transforming to new variables and passing these through a circuit which generates the absolute simplest form. In fact it is often possible to find a single transformation that will put several output functions into their absolute simplest forms!

The procedure for determining that change of variables which transforms a Boolean function into its absolute simplest form is described in the next section.

EXERCISES

(*a*) What is the least number of + and · operations required in the Boolean representation of a designation number for 30 elementary elements which has 24 units? (*Ans.* $30 - 3 - 1 = 26$.) For 20 elementary elements and 16 units? (*Ans.* $20 - 4 - 1 = 15$.)

(*b*) Find the absolute simplest forms for $n = 5$ and $u = 0, 1, 2, \ldots, 32$.

(*c*) Find a single transformation that will take $A \cdot \bar{B} + \bar{A} \cdot C$ and $A \cdot B + \bar{B} \cdot (\bar{A} + \bar{C})$ into absolute simplest form.

HINT: Solve the simultaneous equations

$$A \cdot \bar{B} + \bar{A} \cdot C = C'$$
$$A \cdot B + \bar{B} \cdot (\bar{A} + \bar{C}) = A' \cdot B' + C'$$

for the "unknowns" A, B, and C in terms of A', B', and C'. One solution is $A = C'$, $B = A' + B' + \bar{C}'$, $C = C'$. Show that a "change of variables" to the absolute simplest form does not exist in this case. That is, from the result array show that independent solutions do not exist.

12-9. Transformation to the Absolute Simplest Form

The Theory. Suppose that we had a combined element $X(A,B,C, \ldots)$ and we wished to know *what change of variables* $A = A(A',B',C', \ldots)$, $B = B(A',B',C', \ldots)$, $C = C(A',B',C', \ldots)$, ... would transform $X(A,B,C, \ldots)$ into $Y(A',B',C', \ldots)$, that is,

$$X(A,B,C, \ldots) = Y(A',B',C', \ldots)$$

If a transformation **R** could be found such that

$$\mathbf{R}[X(A,B,C, \ldots)] = Y(A,B,C, \ldots)$$

with the additional properties that

$$\mathbf{R}[X + Y] = \mathbf{R}[X] + \mathbf{R}[Y] \qquad \mathbf{R}[X \cdot Y] = \mathbf{R}[X] \cdot \mathbf{R}[Y]$$

and

$$\mathbf{R}[\bar{X}] = \overline{\mathbf{R}[X]}$$

then our problem would be solved. For then

$$X(A,B,C, \ldots) = \mathbf{R}^{-1}[Y(A,B,C, \ldots)]$$
$$= Y(\mathbf{R}^{-1}[A],\mathbf{R}^{-1}[B],\mathbf{R}^{-1}[C], \ldots)$$

or

$$A' = \mathbf{R}^{-1}[A] \qquad B' = \mathbf{R}^{-1}[B] \qquad C' = \mathbf{R}^{-1}[C] \qquad \text{etc.}$$

whence $A = \mathbf{R}[A']$, $B = \mathbf{R}[B']$, $C = \mathbf{R}[C']$, etc.

Use of Permutation Matrices. The work is of course done with the designation numbers; R turns out to be a *Boolean matrix* such that $\# Y$ is $\# X$ multiplied by R. For example, suppose that it is desired to find the change of variables that will make $\bar{A} = A' \cdot \bar{B}' + \bar{A}' \cdot B'$, that is, $X = \bar{A}$ and $Y = A' \cdot \bar{B}' + \bar{A}' \cdot B'$. Since $\# X = 1010$ and $\# Y = 0110$, we may take

$$\mathbf{R} = \begin{pmatrix} 0100 \\ 0001 \\ 0010 \\ 1000 \end{pmatrix}, \text{ since } \begin{array}{c} \scriptstyle 0123 \\ (1010) \end{array} \otimes \begin{pmatrix} 0100 \\ 0001 \\ 0010 \\ 1000 \end{pmatrix} = \begin{array}{c} \scriptstyle 0123 \\ (0110) \end{array}$$

(See Sec. 11-11 for a description of Boolean matrix multiplication. Note that since 1010 is a row, we must multiply by a matrix on the *right*, as indicated, in order to obtain another row.) Such a matrix is called a *permutation matrix*, since it has the effect simply of permuting the positions of the designation number it multiplies. Here the unit in position 0 is carried into position 1, the zero in position 1 is carried into position 3, the unit in position 2 is carried into position 2, and the zero in position 3 is carried into position 0.

This is in fact the manner in which the matrix R is derived for a problem. Decide for each position of $\# X$ into what position its bit should be carried so that the result will look like $\# Y$. Then, if position k is to be carried into position h, the matrix element $R_{kh} = 1$, otherwise zero. Note that in the above example we could as well have had position 0 carried into position 2, position 1 into position 0, position 2 into position 1, and position 3 into position 3. In this case $R_{02} = 1$, $R_{10} = 1$, $R_{21} = 1$, and $R_{33} = 1$, whence

$$\mathbf{R} = \begin{pmatrix} 0010 \\ 1000 \\ 0100 \\ 0001 \end{pmatrix}$$

Checking,

$$(1010) \otimes \begin{pmatrix} 0010 \\ 1000 \\ 0100 \\ 0001 \end{pmatrix} = (0110)$$

Now that we have found R, we can determine the change of variables as follows:

$$\# A = \mathbf{R}[\# A'] = (0101) \otimes \begin{pmatrix} 0010 \\ 1000 \\ 0100 \\ 0001 \end{pmatrix} = (1001)$$

whence, with respect to $b[A',B']$, $1001 = \#(A' \cdot B' + \bar{A}' \cdot \bar{B}')$. Also,

$$\# B = \mathbf{R}[\# B'] = (0011) \otimes \begin{pmatrix} 0010 \\ 1000 \\ 0100 \\ 0001 \end{pmatrix} = (0101) = \# A'$$

Hence $A = A' \cdot B' + \bar{A}' \cdot \bar{B}'$ and $B = A'$ is the desired change of variables. To check this, note that

$$\bar{A} = \overline{(A' \cdot B' + \bar{A}' \cdot \bar{B}')} = A' \cdot \bar{B}' + \bar{A}' \cdot B'$$

as desired.

The inverse change of variables, that is, $A' = A'(A,B,C, \ldots)$, $B' = B'(A,B,C, \ldots)$, $C' = C'(A,B,C, \ldots)$, . . . , can be found from $A' = \mathbf{R}^{-1}[A]$, $B' = \mathbf{R}^{-1}[B]$, $C' = \mathbf{R}^{-1}[C]$, ... Given the Boolean *permutation* matrix, the matrix $\mathbf{R}^{-1}$ *is found by interchanging the rows and columns of* $\mathbf{R}$; for example, if

$$\mathbf{R} = \begin{pmatrix} 0010 \\ 1000 \\ 0100 \\ 0001 \end{pmatrix}$$

then $$\mathbf{R}^{-1} = \begin{pmatrix} 0100 \\ 0010 \\ 1000 \\ 0001 \end{pmatrix} \quad \text{and} \quad \mathbf{R} \otimes \mathbf{R}^{-1} = \begin{pmatrix} 1000 \\ 0100 \\ 0010 \\ 0001 \end{pmatrix}$$

In our example, then,

$$\#A' = (0101) \otimes \begin{pmatrix} 0100 \\ 0010 \\ 1000 \\ 0001 \end{pmatrix} = (0011) = \#B$$

$$\#B' = (0011) \otimes \begin{pmatrix} 0100 \\ 0010 \\ 1000 \\ 0001 \end{pmatrix} = (1001) = \#(A \cdot B + \bar{A} \cdot \bar{B})$$

To check that this *is truly* the inverse change of variables, note that

$$\begin{aligned} A' \cdot \bar{B}' + \bar{A}' \cdot B' &= B \cdot \overline{(A \cdot B + \bar{A} \cdot \bar{B})} + \bar{B} \cdot (A \cdot B + \bar{A} \cdot \bar{B}) \\ &= B \cdot (A \cdot \bar{B} + \bar{A} \cdot B) + \bar{A} \cdot \bar{B} = \bar{A} \cdot B + \bar{A} \cdot \bar{B} = \bar{A}(B + \bar{B}) = \bar{A} \end{aligned}$$

as desired.

Simultaneous Change of Variables. If we want a change of variables such that, in addition to $X(A,B, \ldots) = Y(A',B', \ldots)$, we also have $U(A,B, \ldots) = V(A',B', \ldots)$ and $W(A,B, \ldots) = T(A',B', \ldots)$, . . . , then we write the designation numbers under each other,

$$\begin{array}{ll} \#X(A,B, \ldots) & \#Y(A',B', \ldots) \\ \#U(A,B, \ldots) & \#V(A',B', \ldots) \\ \#W(A,B, \ldots) & \#T(A',B', \ldots) \\ \cdots\cdots\cdots & \cdots\cdots\cdots \end{array}$$

The transformation now must be designed to permute the *columns* of the right-hand array so that it will look like the left-hand array. For example, suppose that we desired to have a change of variables so that

$\bar{A} + \bar{B} = A' + B'$ and $\bar{A} = A' \cdot \bar{B}' + \bar{A}' \cdot B'$. Form

$$\begin{array}{rcl@{\qquad}lcl} & & \scriptstyle 0\,1\,2\,3 & \scriptstyle 0\,1\,2\,3 & & \\ \#(\bar{A} + \bar{B}) & = & 1110 & 0111 & = & \#(A' + B') \\ \#\bar{A} & = & 1010 & 0110 & = & \#(A' \cdot \bar{B}' + \bar{A}' \cdot B') \end{array}$$

The method for forming the permutation transformation matrix R is similar to the above. Decide for each column in the left-hand array into what column it should be carried so that the final array will look like the right-hand array. Then, if column k is to be carried into column h, the element $R_{kh} = 1$; otherwise it is zero. For example, we can have column 0 carried into column 2, column 1 into column 3, column 2 into column 1, and column 3 into column 0. Then we would have $R_{02} = 1$, $R_{13} = 1$, $R_{21} = 1$, and $R_{30} = 1$:

$$\mathbf{R} = \begin{pmatrix} 0010 \\ 0001 \\ 0100 \\ 1000 \end{pmatrix}$$

Checking,

$$\begin{pmatrix} 1110 \\ 1010 \end{pmatrix} \otimes \begin{pmatrix} 0010 \\ 0001 \\ 0100 \\ 1000 \end{pmatrix} = \begin{pmatrix} 0111 \\ 0110 \end{pmatrix}$$

Hence the change of variables becomes

$$\#A = \mathbf{R}[\#A'] = (0101) \otimes \begin{pmatrix} 0010 \\ 0001 \\ 0100 \\ 1000 \end{pmatrix} = (1001) = \#(A' \cdot B' + \bar{A}' \cdot \bar{B}')$$

$$\#B = \mathbf{R}[\#B'] = (0011) \otimes \begin{pmatrix} 0010 \\ 0001 \\ 0100 \\ 1000 \end{pmatrix} = (1100) = \#(\bar{B}')$$

To check this, note that

$$\begin{aligned} \bar{A} + \bar{B} &= (\overline{A' \cdot B' + \bar{A}' \cdot \bar{B}'}) + (\bar{\bar{B}}') = A' \cdot \bar{B}' + \bar{A}' \cdot B' + B' \\ &= A' \cdot \bar{B}' + B' = A' + B' \end{aligned}$$

and $$\bar{A} = (\overline{A' \cdot BI + \bar{A}' \cdot \bar{B}'}) = A' \cdot \bar{B}' + \bar{A}' \cdot B'$$

The inverse change of variables is found by considering $\mathbf{R}^{-1} = \begin{pmatrix} 0001 \\ 0010 \\ 1000 \\ 0100 \end{pmatrix}$:

$$\#A' = \mathbf{R}^{-1}[\#A] = (0101) \otimes \begin{pmatrix} 0001 \\ 0010 \\ 1000 \\ 0100 \end{pmatrix} = (0110) = \#(A \cdot \bar{B} + \bar{A} \cdot B)$$

$$\#B' = \mathbf{R}^{-1}[\#B] = (0011) \otimes \begin{pmatrix} 0001 \\ 0010 \\ 1000 \\ 0100 \end{pmatrix} = (1100) = \#(\bar{B})$$

Change to Absolute Simplest Form. With the above computational procedures we are now able to generate transformations to the absolute simplest form. For example, consider $A \cdot (\bar{C} + B) + \bar{C} \cdot B + \bar{A} \cdot \bar{B} \cdot C$ with the designation number 0111 1001, where $n = 3$, $u = 5$. The absolute simplest form is $A' \cdot B' + C'$, with designation number 0001 1111. Hence **R** could be

$$\mathbf{R} = \begin{pmatrix} 0100 & 0000 \\ 0001 & 0000 \\ 0000 & 0001 \\ 0000 & 1000 \\ & \\ 0000 & 0010 \\ 0010 & 0000 \\ 1000 & 0000 \\ 0000 & 0100 \end{pmatrix}$$

whence the change of variables

$$\begin{aligned} A &= B' \cdot \bar{C}' + \bar{B}' \cdot C' \\ B &= \bar{A}' \cdot \bar{B}' + A' \cdot C' \\ C &= \bar{A}' \cdot (B' + \bar{C}') + A' \cdot \bar{B}' \cdot C' \end{aligned}$$

will reduce the given element to its absolute simplest form. However, suppose that we also desire to have $A = A'$ and $B = B'$, as well as $A \cdot (\bar{C} + B) + \bar{C} \cdot \bar{B} + \bar{A} \cdot \bar{B} \cdot C = A' \cdot B' + C'$. Under these conditions the transformation **R** becomes

$$\mathbf{R} = \begin{pmatrix} 1000 & 0000 \\ 0000 & 0100 \\ 0000 & 0010 \\ 0001 & 0000 \\ & \\ 0000 & 1000 \\ 0100 & 0000 \\ 0010 & 0000 \\ 0000 & 0001 \end{pmatrix}$$

which gives as the required change of variables

$$\begin{aligned} A &= A' \\ B &= B' \\ C &= A' \cdot (\bar{B}' \cdot \bar{C}' + B' \cdot C') + \bar{A}' \cdot (\bar{B}' \cdot C' + B' \cdot \bar{C}') \end{aligned}$$

EXERCISES

Find the inverse change of variables of:

(*a*) $A = B' \cdot \bar{C}' + \bar{B}' \cdot C'$, $B = \bar{A}' \cdot \bar{B}' + A' \cdot C'$, $C = \bar{A}' \cdot (B' + \bar{C}') + A' \cdot \bar{B}' \cdot C'$.

(*b*) $A = A'$, $B = B'$, $C = A' \cdot (\bar{B}' \cdot \bar{C}' + B' \cdot C') + \bar{A}' \cdot (\bar{B}' \cdot C' + B' \cdot \bar{C}')$.

12-10. The Absolute Simplest Form in Circuit Design

As an example, suppose that it is desired to construct a circuit with the following *four* outputs:

$$\begin{aligned}
E_1 &= \bar{A}_1 \cdot \bar{A}_2 \cdot \bar{A}_3 + (A_1 \cdot \bar{A}_2 + \bar{A}_1 \cdot A_2) \cdot A_3 \\
E_2 &= (A_1 \cdot A_2 + \bar{A}_1 \cdot \bar{A}_2) \cdot \bar{A}_3 + (A_1 \cdot \bar{A}_2 + \bar{A}_1 \cdot A_2) \cdot A_3 \\
E_3 &= \bar{A}_1 + A_2 \cdot \bar{A}_3 + \bar{A}_2 \cdot A_3 \\
E_4 &= \bar{A}_1 \cdot \bar{A}_2 + A_3 \cdot (\bar{A}_1 + \bar{A}_2) + A_1 \cdot A_2 \cdot \bar{A}_3
\end{aligned}$$

Here $n = 3$, and for E_1, $u = 3$; for E_2, $u = 4$; for E_3, $u = 6$; for E_4, $u = 5$. We have for the respective absolute simplest forms

$$E_1 = A_3' \cdot (A_2' + A_1') \qquad E_3 = A_3' + A_2'$$
$$E_2 = A_3' \qquad E_4 = A_3' + A_2' \cdot A_1'$$

The problem is to design the change of variable circuits whose outputs are A_1', A_2', and A_3' (that is, the inverse change of variables is explicitly needed: $A_1' = \mathrm{R}^{-1}[A_1]$, $A_2' = \mathrm{R}^{-1}[A_2]$, and $A_3' = \mathrm{R}^{-1}[A_3]$). One valid

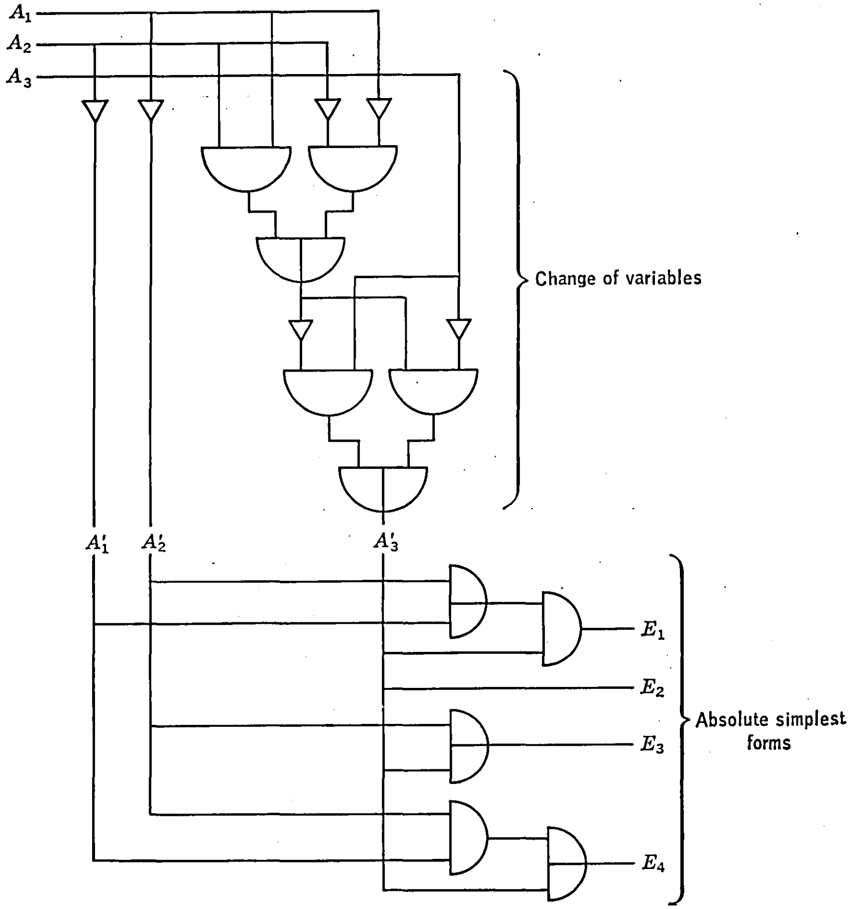


FIG. 12-15. Change of variables and resulting absolute-simplest-form circuitry for a multiple-output circuit.

solution is

$$\begin{aligned}
\#A_1' &= 1100 \quad 1100 = \#\bar{A}_2\\
\#A_2' &= 1010 \quad 1010 = \#\bar{A}_1\\
\#A_3' &= 1001 \quad 0110 = \#(A_1 \cdot A_2 + \bar{A}_1 \cdot \bar{A}_2) \cdot \bar{A}_3\\
&\qquad\qquad + (A_1 \cdot \bar{A}_2 + \bar{A}_1 \cdot A_2) \cdot A_3
\end{aligned}$$

Here 9 operations are required to make A_1', A_2', and A_3' and 5 more to make E_1, E_2, E_3, and E_4, or a total of 14 altogether, as compared with the 25 operations required to make the E_i directly from the A_i (see Fig. 12-15).

EXERCISES

(*a*) Supply the details for the above example.

(*b*) Suppose that it is desired to make a circuit with the following four output functions:

$$\begin{aligned}
E_1 &= \bar{A}_1 \cdot \bar{A}_3 + (A_1 + \bar{A}_2) \cdot A_3\\
E_2 &= (\bar{A}_1 + \bar{A}_2) \cdot \bar{A}_3 + (\bar{A}_1 + A_2) \cdot A_3\\
E_3 &= \bar{A}_1 \cdot A_2 \cdot \bar{A}_3 + (A_1 \cdot A_2 + \bar{A}_1 \cdot \bar{A}_2) \cdot A_3\\
E_4 &= (A_1 \cdot \bar{A}_2 + \bar{A}_1 \cdot A_2) \cdot \bar{A}_3 + (A_1 \cdot A_2 + \bar{A}_1 \cdot \bar{A}_2) \cdot A_3
\end{aligned}$$

The problem is to design change-of-variable circuits that will simplify the generation of E_1, E_2, E_3, and E_4.

One solution is to choose $A_1' \cdot A_3' + A_2'$, $A_1' + A_3'$, $(A_2' + A_3') \cdot A_1'$, and A_1', respectively, as the absolute simplest forms of E_1, E_2, E_3, and E_4. Then we find

$$\begin{aligned}
A_1' &= (\bar{A}_1 \cdot \bar{A}_2 + A_1 \cdot A_2) \cdot \bar{A}_3 + (\bar{A}_1 \cdot A_2 + A_1 \cdot \bar{A}_2) \cdot A_3\\
A_2' &= \bar{A}_1 \cdot \bar{A}_3 + \bar{A}_2 \cdot A_3\\
A_3' &= \bar{A}_1 \cdot \bar{A}_2 + A_2 \cdot A_3
\end{aligned}$$

Can you find a simpler change of variables?

12-11. Additional Topics

a. Solution of Equations by Matrices. In Table 12-2 we tested the pair $(x^4,y^4) = (0,1)$ by multiplying column 4 of the coefficient table by the third column of the unknown table, summing both above and below the line, and *comparing only the sums.* Note that this is equivalent to matrix multiplication, if one column were transposed to be a row. Similarly it is easily seen that, if the table of coefficients above (or below) the line, considered as a matrix, is multiplied *on the left* by the transpose of the matrix of the table of its unknowns, then the product gives, in row i, column j, the result of multiplying unknown column i by coefficient column j. Comparison of the products formed for the tables above and below the line (i.e., for the left- and right-hand sides of the equation) shows the allowable solutions for each column of the basis for A, B, C (reading the x, y pairs downward). A solution is allowable in an equivalence if in its corresponding position the two matrices have the same character, either both units or both zeros. A solution is allowable in an implication unless there is a unit in the corresponding position of the implier matrix (the tail of the arrow) and a zero in that position of the implicand matrix (the head of the arrow). For a set of equivalences and implications we can form a result matrix for each, with a unit in each position corresponding to an allowable solution, a zero otherwise; then those solutions which are allowable for all the equivalences and implications will have a unit in the corresponding position in every result matrix.

For example, consider the three simultaneous statements of Sec. 12-6,

$$A + B \cdot X \cdot \bar{Y} + C \cdot Y = \bar{A} \cdot \bar{X} \cdot \bar{Y} + \bar{B} \cdot \bar{X} + (C + A \cdot B) \qquad (12\text{-}1)$$

$$B \cdot X \cdot Y + C \cdot (A + B) \cdot Y = \bar{B} \cdot \bar{X} \cdot \bar{Y} \qquad (12\text{-}2)$$

$$A \cdot C \cdot (X + Y) + A \cdot B \cdot (X + \bar{Y}) \rightarrow X \cdot Y + A \cdot C \qquad (12\text{-}3)$$

Comparing with Table 12-4, we see that the matrices for Eq. (12-1) would be

$$\begin{pmatrix} 100 \\ 110 \\ 101 \\ 101 \end{pmatrix} \otimes \begin{pmatrix} 0101 & 0101 \\ 0011 & 0011 \\ 0000 & 1111 \end{pmatrix} = \begin{pmatrix} 0101 & 0101 \\ 0111 & 0111 \\ 0101 & 1111 \\ 0101 & 1111 \end{pmatrix} \quad \text{above the line}$$

and

$$\begin{pmatrix} 111 \\ 001 \\ 011 \\ 001 \end{pmatrix} \otimes \begin{pmatrix} 1010 & 1010 \\ 1100 & 1100 \\ 0001 & 1111 \end{pmatrix} = \begin{pmatrix} 1111 & 1111 \\ 0001 & 1111 \\ 1101 & 1111 \\ 0001 & 1111 \end{pmatrix} \quad \text{below the line}$$

Comparing these matrices for equality, we obtain the result matrix of allowable solutions:

$$\begin{pmatrix} 0101 & 0101 \\ 1001 & 0111 \\ 0111 & 1111 \\ 1011 & 1111 \end{pmatrix}$$

[Recalling that the first row (row 0) corresponds to the column for $(x^i,y^i) = (0,0)$, row 1 to $(x^i,y^i) = (1,0)$, row 2 to $(x^i,y^i) = (0,1)$, and row 3 to $(x^i,y^i) = (1,1)$, we see that this solution matrix corresponds exactly to the result array of Table 12-1.] Similarly Eq. (12-2) gives the matrices

$$\begin{pmatrix} 00 \\ 00 \\ 01 \\ 11 \end{pmatrix} \otimes \begin{pmatrix} 0011 & 0011 \\ 0000 & 0111 \end{pmatrix} = \begin{pmatrix} 0000 & 0000 \\ 0000 & 0000 \\ 0000 & 0111 \\ 0011 & 0111 \end{pmatrix}$$

and

$$\begin{pmatrix} 1 \\ 0 \\ 0 \\ 0 \end{pmatrix} \otimes \begin{pmatrix} 1100 & 1100 \end{pmatrix} = \begin{pmatrix} 1100 & 1100 \\ 0000 & 0000 \\ 0000 & 0000 \\ 0000 & 0000 \end{pmatrix}$$

The result matrix is

$$\begin{pmatrix} 0011 & 0011 \\ 1111 & 1111 \\ 1111 & 1000 \\ 1100 & 1000 \end{pmatrix}$$

Implication (12-3) gives the matrices

$$\begin{pmatrix} 01 \\ 11 \\ 10 \\ 11 \end{pmatrix} \otimes \begin{pmatrix} 0000 & 0101 \\ 0001 & 0001 \end{pmatrix} = \begin{pmatrix} 0001 & 0001 \\ 0001 & 0101 \\ 0000 & 0101 \\ 0001 & 0101 \end{pmatrix}$$

and

$$\begin{pmatrix} 01 \\ 01 \\ 01 \\ 11 \end{pmatrix} \otimes \begin{pmatrix} 1111 & 1111 \\ 0000 & 0101 \end{pmatrix} = \begin{pmatrix} 0000 & 0101 \\ 0000 & 0101 \\ 0000 & 0101 \\ 1111 & 1111 \end{pmatrix}$$

The result matrix has zeros only where the upper product has a unit and the lower a zero:

$$\begin{pmatrix} 1110 & 1111 \\ 1110 & 1111 \\ 1111 & 1111 \\ 1111 & 1111 \end{pmatrix}$$

Examining the three matrices for common units, we have the following compound-solution matrix, which corresponds exactly to the result array of Table 12-4:

$$\begin{pmatrix} 0000 & 0001 \\ 1000 & 0111 \\ 0111 & 1000 \\ 1100 & 1000 \end{pmatrix}$$

[More systematically the result matrices can be formed as in the equations $\#(\mathbf{A} = \mathbf{B}) = \#(\mathbf{A} \cdot \mathbf{B} + \bar{\mathbf{A}} \cdot \bar{\mathbf{B}})$ and $\#(\mathbf{A} \rightarrow \mathbf{B}) = \#(\bar{\mathbf{A}} + \mathbf{B})$, substituting the matrices for the letters on the right and performing the multiplication bit by bit, not row by column as in regular matrix multiplication. The compound solution is then the bit-by-bit product of the result matrices for each statement. The method presented here can of course be extended to any number of statements in any number of knowns and unknowns. If any column of the compound-solution matrix has all zeros and no units, then there is no solution to the problem, unless there exist some constraints on the knowns A, B, C, . . . which remove that column from their basis.]

b. Flip-Flop Problems. The use of flip-flops for storage in recursive circuits frequently generates problems that involve the solution of equations. For example, consider a flip-flop with two states Q and $\bar{Q}$ such that if it is in state Q then an input-pulse signal T will change its state to $\bar{Q}$ and if it is in state $\bar{Q}$ then an input-pulse signal T will change its state to Q; otherwise the flip-flop will stay in its present state. If Q' is the final state, then (see Fig. 12-16*a*)

$$Q' = T \cdot \bar{Q} + \bar{T} \cdot Q$$

(Can you develop this equation for Q'?) Now suppose that it is desired to make a recursive circuit with inputs A, B, and C, and with output C' connected to the input C, such that $C' = A \cdot B + (A + B) \cdot C$. If we are to use the above flip-flop, the problem is to design the input pulse T so that the output Q' is the desired C', that is, such that $Q' = C' = A \cdot B + (A + B) \cdot C$. Thus we desire to solve the equation

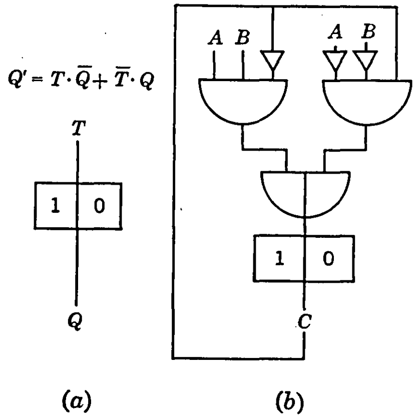


FIG. 12-16. Flip-flop representation and example of a circuit.

$$A \cdot B + (A + B) \cdot C = T \cdot \bar{C} + \bar{T} \cdot C$$

for the unknown T, where we have substituted C for Q (why?). The solution turns out to be $T = A \cdot B \cdot \bar{C} + \bar{A} \cdot \bar{B} \cdot C$ (see Fig. 12-16*b*).

Consider next a flip-flop with two inputs R and S such that an input pulse on S will change the state of the flip-flop to Q, while an input pulse on R will return the state to $\bar{Q}$. Both R and S cannot be pulsed at the same time (see Fig. 12-17*a*). The equations for this flip-flop are

$$Q' = S + \bar{R} \cdot Q \qquad \text{and} \qquad R \cdot S = 0$$

Suppose that we desired to make a circuit using two flip-flops, to have inputs A, B, and C and two outputs (one from each flip-flop):

$$C' = A \cdot B + (A + B) \cdot C \qquad \text{and} \qquad D' = (A \cdot \bar{B} + \bar{A} \cdot B) \cdot \bar{C} + (A \cdot B + \bar{A} \cdot \bar{B}) \cdot C$$

This set of equations must be solved:

$$\begin{aligned} A \cdot B + (A + B) \cdot C &= S_C + \bar{R}_C \cdot C \qquad & R_C \cdot S_C &= 0 \\ (\bar{A} \cdot B + A \cdot \bar{B}) \cdot \bar{C} + (A \cdot B + \bar{A} \cdot \bar{B}) \cdot C &= S_D + \bar{R}_D \cdot D \qquad & R_D \cdot S_D &= 0 \end{aligned}$$

The solutions (see Fig. 12-17*b*) are

$$\begin{aligned} R_C &= \bar{A} \cdot \bar{B} \qquad & R_D &= (\bar{A} \cdot B + A \cdot \bar{B}) \cdot C + (A \cdot B + \bar{A} \cdot \bar{B}) \cdot \bar{C} \\ S_C &= A \cdot B \qquad & S_D &= (\bar{A} \cdot B + A \cdot \bar{B}) \cdot \bar{C} + (A \cdot B + \bar{A} \cdot \bar{B}) \cdot C \end{aligned}$$

Many other kinds of flip-flops can be conceived. For example, suppose that in the R-S flip-flop we let both R and S be pulsed at the same time and that this causes the flip-flop to *change* state. Then the equation of the flip-flop would be

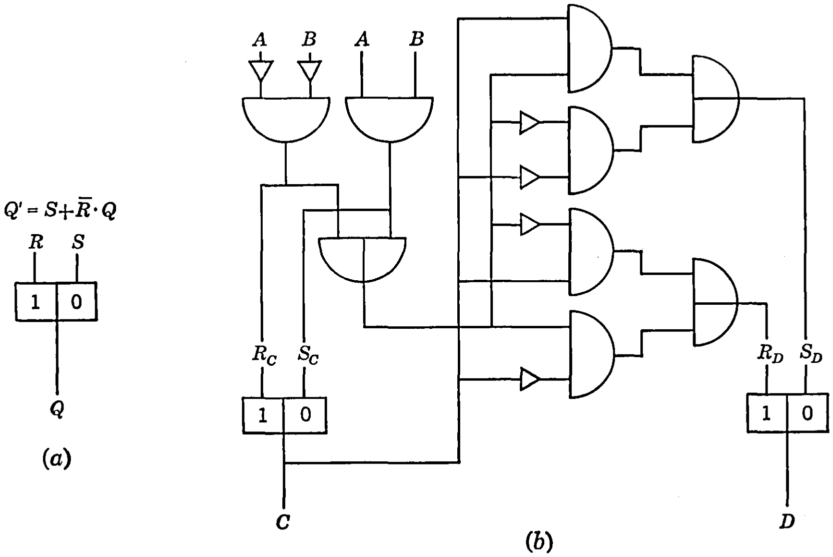


FIG. 12-17. Flip-flop representation and example of a circuit.

$$Q' = S \cdot \bar{Q} + \bar{R} \cdot Q$$

c. Absolute Simplest Form and Symmetries of Boolean Functions. As far as circuit design is concerned, the labels assigned to the input variables are immaterial, and whether the input element is called A_i or $\bar{A}_i$ is also immaterial. Thus only the Boolean *functional form* of the output of a circuit is important; all functions of the same form that differ by permutations or inversions of the variable labels represent the *same circuit*, so long as the substitutions are made consistently. For n variables, m of which must explicitly appear in the functional form, the number of circuits equivalent in the above sense becomes

$$2^{\binom{n}{m}}$$

which can be a very large number. The converse problem immediately suggests itself: How many nonequivalent (in the above sense) functional forms of n variables are there? This problem was considered by D. Slepian in his On the Numbers of Symmetry Types of Boolean Functions of n Variables, *Can. J. Math.*, vol. 5, pp. 185–193, 1953, where it was shown that the results of J. A. Todd, The Groups of Symmetries of the Regular Polytopes, *Proc. Cambridge Phil. Soc.*, vol. 27, pp. 212–231, 1931, could be applied.

We have observed such symmetries associated with the absolute simplest form above for n variables and u units in the designation number. This suggests the following extension of the above problem, also considered by Slepian: For n variables and u units how many different (in the above sense) functional forms are there? The absolute simplest form is evidently only one type of functional form.

d. Three-valued Logic. Until now we have considered circuits where only two signal levels can be distinguished, namely, 0 and 1. Suppose that three values of signal levels can be distinguished, say 0, 1, and 2. This leads to the consideration of three-valued logics. Since each input to a circuit can have three values, for a three-input circuit there are $3 \times 3 \times 3$ possible input conditions, namely,

$$\begin{array}{lllllllllll}
\#A_1 = & 012 & 012 & 012 & \quad & 012 & 012 & 012 & \quad & 012 & 012 & 012 \\
\#A_2 = & 000 & 111 & 222 & & 000 & 111 & 222 & & 000 & 111 & 222 \\
\#A_3 = & 000 & 000 & 000 & & 111 & 111 & 111 & & 222 & 222 & 222
\end{array}$$

This array forms a basis in a three-valued logic. For n variables there are 3^n columns in a basis. Our computational techniques in terms of designation numbers can then be directly extended as soon as we define the logical operations.

We wish to define a set of operations that will give us a *complete Boolean algebra.* A complete Boolean algebra is one in which all the $3^{(3^n)}$ possible designation numbers (i.e., Boolean functions) can be generated from the basis by means of the defined operations. There are many such sets of operations (just as there were many possible types of operations for the two-valued logic). We wish to choose a "simple" set that has "nice" algebraic properties. Analogous to our simple set +, ·, and ¯ of two-valued logic we define a set of three-valued operations as follows:

$$\begin{array}{cccccc}
2+0=2 & 2+1=2 & 2+2=2 & 1+0=1 & 1+1=1 & 0+0=0 \\
2\cdot 0=0 & 2\cdot 1=1 & 2\cdot 2=2 & 1\cdot 0=0 & 1\cdot 1=1 & 0\cdot 0=0
\end{array}$$
$$\bar{0}=1 \qquad \bar{1}=2 \qquad \text{and} \qquad \bar{2}=0$$

To show that this set results in a complete Boolean algebra, we need merely show that designation numbers can be generated that are all 0 except for a single 1 in any position or with a single 2 in any position. To see this, we observe that for one variable $\#A = 012$, $\#\bar{A} = 120$, and $\bar{\bar{A}} = 201$, whence

$$\begin{array}{llll}
\#\bar{A}\cdot\bar{\bar{A}} = 100 & \qquad & \#(\overline{A+\bar{\bar{A}}}) = 200 \\
\#A\cdot\bar{A} = 010 & & \#(\overline{A+\bar{\bar{A}}}) = 020 \\
\#A\cdot\bar{\bar{A}} = 001 & & \#(\overline{\bar{A}+\bar{\bar{A}}}) = 002
\end{array}$$

With these relations in mind can you generate the designation numbers required above for three variables? [HINT: To generate, for instance, 000 000 200 000 000 000 000 000 000, we simply form $\#(\overline{A_1+\bar{A}_1})\cdot\#(\overline{\bar{A}_2+\bar{\bar{A}}_2})\cdot\#(\overline{A_3+\bar{A}_3})$.]

What algebraic properties does this Boolean algebra have? Can you define other sets of three-valued-logic operations that will result in a complete Boolean algebra? What algebraic properties do these algebras have?

Circuit design in three-valued logic is analogous to that in two-valued logic. For example, let us design a three-valued serial adder (see Chap. 15) where the numbers are ternary, i.e., to the radix 3. We have the following function table:

$\#A$ =	012	012	012	012	012	012	012	012	012
$\#B$ =	000	111	222	000	111	222	000	111	222
$\#C$ =	000	000	000	111	111	111	222	222	222
Result $\#R$ =	012	120	201	120	201	012	201	012	120
Carry $\#C'$ =	000	001	011	001	011	111	011	111	112

Using a mongrel-form technique, we then find

$$\begin{aligned} R = {} & [A \cdot (\overline{B + \bar{B}}) + \bar{A} \cdot (\overline{B + \bar{\bar{B}}}) + \bar{\bar{A}} \cdot (\overline{\bar{B} + \bar{\bar{B}}})] \cdot (\overline{C + \bar{C}}) \\ & + [\bar{A} \cdot (\overline{B + \bar{B}}) + \bar{\bar{A}} \cdot (\overline{B + \bar{\bar{B}}}) + A \cdot (\overline{\bar{B} + \bar{\bar{B}}})] \cdot (\overline{C + \bar{\bar{C}}}) \\ & + [\bar{\bar{A}} \cdot (\overline{B + \bar{B}}) + A \cdot (\overline{B + \bar{\bar{B}}}) + \bar{A} \cdot (\overline{\bar{B} + \bar{\bar{B}}})] \cdot (\overline{\bar{C} + \bar{\bar{C}}}) \\ C = {} & (A \cdot \bar{\bar{A}} \cdot B \cdot \bar{B} + A \cdot B \cdot \bar{\bar{B}}) \\ & + (A \cdot \bar{\bar{A}} + A \cdot B \cdot \bar{B} + B \cdot \bar{\bar{B}}) \cdot C \\ & + (A \cdot \bar{B} \cdot \bar{\bar{B}} + B \cdot \bar{B} + A \cdot B) \cdot (\overline{\bar{C} + \bar{\bar{C}}}) \end{aligned}$$

Evidently our adder function table depended upon the interpretation of the values of signal levels 0, 1, and 2 as the ternary digits 0, 1, and 2, respectively. Other interpretations will result in other functions for R and C'. Can you design a three-valued serial adder for the other algebras that you generated above? (Note that $C' = 2$ only when $C = 2$. This would never happen in the real case, so that the basis may be constrained by the removal of its last third. How would this affect the forms of R and C' above?)

Can you extend the computational techniques given in this chapter to analogous methods for our three-valued logic?

e. Generalized Concept of the Searching Problem in Terms of Boolean Algebra. Referring back to Sec. 7-2, the searching problem can be easily couched in terms of Boolean algebra. This is accomplished by observing that an item is uniquely defined (at least as far as the search is concerned) by the pattern of characteristics that are and are not associated with it. With three characteristics A_1, A_2, and A_3 a total of eight items X_0, X_1, X_2, X_3, X_4, X_5, X_6, and X_7 can be distinguished,

$$\begin{array}{llll} X_0 = \bar{A}_1 \cdot \bar{A}_2 \cdot \bar{A}_3 & X_1 = A_1 \cdot \bar{A}_2 \cdot \bar{A}_3 & X_2 = \bar{A}_1 \cdot A_2 \cdot \bar{A}_3 & X_3 = A_1 \cdot A_2 \cdot \bar{A}_3 \\ X_4 = \bar{A}_1 \cdot \bar{A}_2 \cdot A_3 & X_5 = A_1 \cdot \bar{A}_2 \cdot A_3 & X_6 = \bar{A}_1 \cdot A_2 \cdot A_3 & X_7 = A_1 \cdot A_2 \cdot A_3 \end{array}$$

Here, for example, $X_5 = A_1 \cdot \bar{A}_2 \cdot A_3$ means that item X_5 is associated with characteristic A_1 and A_3 but not with A_2. If the basic symbols are chosen to be the characteristics, then the columns of the basis can represent items distinguishable by these characteristics, where a unit in a row of a column indicates that the item of that column is associated with the characteristic of that row and a zero means that it is not. Thus

	X_0	X_1	X_2	X_3	X_4	X_5	X_6	X_7
$\#A_1$ =	0	1	0	1	0	1	0	1
$\#A_2$ =	0	0	1	1	0	0	1	1
$\#A_3$ =	0	0	0	0	1	1	1	1

The simple searching problem is then solved as follows: To find all items associated with $A_1 \cdot A_2$, simply form

				X_3				X_7
$\#A_1 \cdot A_2$ =	0	0	0	1	0	0	0	1

whence X_3 and X_7 are the desired items. Of course there is not always an item associated with every combination of characteristics; in such a case the columns for which there is no associated item are omitted from the basis, i.e., a *constrained* basis is used. Thus Table 7-1 can be considered as a constrained basis.

With this interpretation the searching problem can be generalized as follows: Suppose that it is desired to find all items associated with characteristic A_1 and at least one of the characteristics A_2 or A_3, that is, associated with $A_1 \cdot (A_2 + A_3)$. We simply form $\#A_1 \cdot (A_2 + A_3) = 0001\ 0101$, whence the answer is X_3, X_5, and X_7. As another illustration, consider Table 7-1 (page 217), and find all articles associated with nuclear theory or nuclear plans or both. Observing that

$$\begin{array}{rcl}
\text{Nuclear} = \mathit{4.3} & = & 0000\quad 1001\quad 0011 \\
\text{Theory} = \mathit{4.4} & = & 0010\quad 0001\quad 0011 \\
\text{Plans} = \mathit{6.2} & = & 0100\quad 1001\quad 1110 \\
\hline
\text{we have}\qquad (\mathit{4.3}) \cdot [(\mathit{4.4}) + (\mathit{6.2})] & = & 0000\quad 1001\quad 0011
\end{array}$$

Hence articles **2.1**, **2.4**, **3.3**, and **3.4** by Nicholls, Seidle, Stockendal, and Tove are the final desired results. Summarizing, *when it is desired to find all items specified by a Boolean function of the characteristics, the designation number of the function is formed; the positions of the units indicate the desired items.*

One of the primary reasons for the advantages of the Tabledex method can be understood in terms of our Boolean algebraic interpretation of searching. When the search proceeds within a table, the problem has already been significantly reduced since in general only a small fraction of all the items are associated with a particular characteristic. In addition the constrained basis to which a table corresponds includes only rows corresponding to characteristics of a *greater* order than that of the table, thereby further reducing the basis and hence the searching problem. For example, from Table 7-1 it is seen that the column-constrained and row-reduced basis with respect to Table *5.1* of Fig. 7-3*c* on page 220 is

	2.2	**2.3**	**2.4**	**3.2**	**3.4**
5.2	1	1	0	1	0
5.3	0	1	1	0	0
6.1	1	0	0	0	0
6.2	0	0	1	1	0
7.1	1	1	0	1	0

f. Sentential Examples. In the Additional Topics of Chap. 10 we discussed the logical interpretation of English. We shall now present examples of applications of our computational methods to word or sentential problems. It is of historical interest to note that many such problems were presented in E. Schroder's "Algebra der Logik," Teubner Verlagsgesellschaft, Leipzig, 1890, but he did not have the advantages of our computational methods. (Our first two examples are taken from R. S. Ledley, Mathematical Foundations and Computational Methods for a Digital Logic Machine, *J. Operations Research Soc. Am.*, vol. 2, no. 3, pp. 249–274, August, 1954.)

First Example: Consistency, Redundancy. Suppose that an intelligence agency receives the following messages from fairly, though not completely, reliable sources, concerning fighter and bomber airplanes of the Lower Slobbovian air force:

1. A plane with jet engines and a short range is a bomber.
2. Conventional engine bombers have heavy armor.
3. Conventional engine fighters have a short range.
4. Long-range conventional engine planes have light armor.

5. Jet planes have heavy armor.
6. Heavy armored planes with a short range are fighters.
7. Either long-range planes or fighter planes have light armor.
8. Either conventional engine planes or short-range planes have heavy armor.

The intelligence agency now makes an analysis of these statements and in this connection desires the answers to the following questions:

a. Are these statements all consistent?

b. If not, suppose that there is a high probability that only one of them was incorrect; can a single statement be omitted so that the remaining ones are consistent, and if so, which statements are they?

c. Are any of the statements related?

d. Are any of the statements redundant, i.e., not necessary?

e. With various assumptions as to which is an incorrect statement, what conclusions can be drawn?

Solution. First note that only fighter and bomber airplanes are discussed by hypothesis and that an engine must be either jet or conventional; the opposite of long-range is short-range, and the opposite of light armor is heavy armor; and in addition, only airplanes, engines, range, and armor are discussed in the statements. Thus the eight statements can all be expressed in terms of the following four elementary elements (where "It" refers to "the plane"):

A = It is a fighter plane B = It has jet engines
$\bar{A}$ = It is a bomber plane $\bar{B}$ = It has conventional engines
C = It has a long range D = It has light armor
$\bar{C}$ = It has a short range $\bar{D}$ = It has heavy armor

Thus the first statement is: *It has jet engines and it has a short range* implies that *it is a bomber.* In symbols this is $B \cdot \bar{C} \to \bar{A}$. Now in terms of $+$ and $\cdot$ this means: $\overline{B \cdot \bar{C}} + \bar{A}$. In terms of the regular basis extended to four elementary elements find $\#(\overline{B \cdot \bar{C}} + \bar{A}) = 1110\ 1111\ 1110\ 1111$. Similarly we have for the whole set of statements

1. $B \cdot \bar{C} \to \bar{A}$:	$\overline{B \cdot \bar{C}} + \bar{A}$:	1110	1111	1110	1111
2. $\bar{A} \cdot \bar{B} \to \bar{D}$:	$\overline{\bar{A} \cdot \bar{B}} + \bar{D}$:	1111	1111	0111	0111
3. $A \cdot \bar{B} \to \bar{C}$:	$\overline{A \cdot \bar{B}} + \bar{C}$:	1111	1011	1111	1011
4. $\bar{B} \cdot C \to D$:	$\overline{\bar{B} \cdot C} + D$:	1111	0011	1111	1111
5. $B \to \bar{D}$:	$\bar{B} + \bar{D}$:	1111	1111	1100	1100
6. $\bar{D} \cdot \bar{C} \to A$:	$\overline{\bar{D} \cdot \bar{C}} + A$:	0101	1111	1111	1111
7. $A + C \to D$:	$\overline{A + C} + D$:	1010	0000	1111	1111
8. $\bar{B} + \bar{C} \to \bar{D}$:	$\overline{\bar{B} + \bar{C}} + \bar{D}$:	1111	1111	0000	0011

Now we are ready to answer the questions. First, note that the product of all the designation numbers is #0—in other words, always false. This means that the statements are not consistent and there must be a contradiction. So the answer to (*a*) is "no." For the answer to (*b*) observe that, if any or all of statements 1, 2, 3, and 4 were omitted, the remaining statements would still be contradictory. However, if any of statements 5, 6, 7, and 8 were omitted, no contradiction would result; i.e., the product of the remaining statements would have some units. The answer to (*b*) is thus "yes: 5, 6, 7, or 8." As far as (*c*) is concerned, if statements 7 and 8 are true, then 1, 2, 3, and 4 are all implied true. It is easily seen that statement 8 alone implies statement 2, while statement 7 alone implies statement 4. Thus, of statements 1, 2, 3, 4, 7, and 8, only statements 7 and 8 need be stated, since statements 1, 2, 3, and 4 are redundant; this answers (*d*). It would be most reasonable for one of the two nonrelated statements 5 and 6 to be taken as incorrect. Suppose that from other considerations it was decided that statement 6 is incorrect. Thus we are left with statements 1, 2, 3, 4, 5, 7, and 8, of which statements 1, 2, 3, and 4 are redundant; so we are left with statements 5, 7, and 8. Since these are all true at the same time, the product of their designation numbers, namely, 1010 0000 0000 0000, contains the most information. This is $\bar{A} \cdot \bar{C} \cdot \bar{D}$. Thus there is a type of airplane with the following characteristics: It is a bomber and has short range and heavy armor, which is not a particularly obvious observation to deduce from statements 5, 7, and 8. If it is assumed that statement 5 is incorrect, we are left with statements 6, 7, and 8—the product of which turns out to be $B \cdot C \cdot D$: a jet with long range and light armor. These last two conclusions are the most general statements that can be made under the circumstances presented in the problem, in the sense that they imply the most possible other statements. For example, if $\bar{A} \cdot \bar{C} \cdot \bar{D}$ is true, A is implied false, showing that statements 5, 7, and 8 imply no fighter airplanes.

Second Example: Change of Variables. Suppose that the army sent a military observer to observe the military tactics of the Blefuscudians during their war with the Lilliputians. The observer was to notice especially the relation between the physical circumstances of the battlefield and the infantry tactics used. After watching many battles the observer sent back his final report with four general observations:

1. In hilly terrain on clear days, localized infantry attacks were used in conjunction with long-range artillery, but no tanks.

2. On flat plains, at night or in bad weather, the Blefuscudians used short-range artillery but did not use a generalized infantry assault on a broad front that was supported by armored tanks.

3. In hilly terrain at night, or in bad weather in the daytime, armored tanks were used with localized attacks, or long-range artillery with an assault on a broad front was used.

4. In bad weather on flat terrain or at night, or in good weather in hilly country, either a localized attack was used or long-range artillery and tanks together with an assault on a broad front were resorted to by the Blefuscudians.

The problem is: From this report made by the military observer, what can the army learn about the effects on the Blefuscudians' infantry tactics of (*a*) flat terrain, (*b*) night fighting, and (*c*) bad weather? Conversely, under what physical circumstances will (*d*) an assault on a broad front be made, (*e*) long-range artillery be used, and (*f*) armored tanks be used? Suppose that the battle is to take place on the plains, on a clear day, what (*g*) will be the Blefuscudians' tactics?

To solve this problem, we first observe that terrain is either flat or hilly but not both, day is the opposite of night, good weather the opposite of bad weather. Also an infantry attack either is localized or is an assault on a broad front, only battles with infantry attacks being observed. In addition long-range and short-range artillery are opposites; the use of armored tanks and no armored tanks are opposites. Thus we symbolize the elementary elements by

A = (The terrain was) flat B = (It was) night C = (The weather was) bad
$\bar{A}$ = (The terrain was) hilly $\bar{B}$ = (It was) day $\bar{C}$ = (The weather was) good

In the primed set of elementary elements we leave the completion of the sentences understood,

A' = an assault on a broad front B' = long-range artillery C' = armored tanks
$\overline{A'}$ = localized attack $\overline{B'}$ = short-range artillery $\overline{C'}$ = no tanks

The four observations can now be symbolized thus:

1. $\bar{A} \cdot \bar{B} \cdot \bar{C} = \overline{A'} \cdot B' \cdot \overline{C'}$
2. $A \cdot (B + C) = \overline{B'} \cdot (\overline{A' \cdot C'})$
3. $\bar{A} \cdot B + \bar{B} \cdot C = \overline{A'} \cdot C' + A' \cdot B'$
4. $C \cdot (A + B) + \bar{A} \cdot \bar{C} = \overline{A'} + A' \cdot B' \cdot C'$

Evidently the primed elementary elements are independent, and so we have a change-of-variables problem, where to determine the answers to questions *a*, *b*, and *c* we solve for A, B, and C in terms of A', B', and C', and for A', B', and C' in terms of A, B, and C. The results are

a. $A = \overline{B'}$

b. $B = B' \cdot C' + \overline{B'} \cdot \overline{C'}$

c. $C = A' \cdot B' + \overline{A'} \cdot \overline{B'}$

d. $A' = A \cdot \bar{C} + \bar{A} \cdot C$

e. $B' = \bar{A}$

f. $C' = A \cdot \bar{B} + \bar{A} \cdot B$

However, there is another set of results, where (*a*), (*b*), (*e*), and (*f*) are the same but (*c*) and (*d*) become

c. $C = \overline{A'} \cdot (\overline{B'} + C') + A' \cdot B' \cdot \overline{C'}$ *d.* $A' = \bar{C} \cdot (A + B) + \bar{A} \cdot \bar{B} \cdot C$

The meaning of the first set of results is as follows: On flat terrain short-range artillery is used with the attack. At night the infantry attack is made with either long-range artillery and armored tanks or short-range artillery and no tanks. In bad weather either an assault on a broad front supported by long-range artillery or localized attacks supported by short-range artillery are made. Infantry assaults on a broad front are made on flat terrain in good weather or on hilly terrain in bad weather. Long-range artillery is used in hilly country. Armored tanks are used on flat plains in the daytime or on hilly country at night. The second set of results should also be interpreted.

To answer question *g*, consider first the first set of results: $A \cdot \bar{B} \cdot \bar{C} = A' \cdot \overline{B'} \cdot C'$, as can be easily demonstrated by the use of the designation numbers. Thus the Blefuscudians will use an infantry assault on a broad front supported by short-range artillery and armored tanks, according to the logical analysis of the military observer's report. Similarly the second set of results can be used to give an alternate procedure that might be used by the Blefuscudians, which is also a logical conclusion of the report. This complete analysis of the military observer's report would certainly be difficult to obtain by other means.

Third Example: Solution for Simultaneous Equivalence Equations. Suppose that there were two aircraft-identification towers on an island. For a few days the same enemy airplanes kept flying overhead. Recognition of the types of enemy airplanes observed was very difficult and led to some controversy between the two observer posts. It was thought, although it was far from certain, that there were four types of enemy airplanes. Call these types A, B, X, and Y: it was fairly certain that types A and B existed. On three successive days the reports from each post read as follows:

	Post 1	*Post* 2
1st day:	The planes were of both types X and Y	The planes were of type A and not type B
2d day:	The planes were of type A or B, or both	The planes were of type Y and not type A, or else they were of type X
3d day:	The planes were of type X and either type A or B, or both; or else they were type A and type Y	The planes were of type A

The question is: Is it at all possible, purely on the basis of these reports, that airplanes of type X and type Y were really airplanes of types A and B?

Solution. If we can solve successfully for X and Y in terms of A and B, then the answer would be "yes." The equations are

$$X \cdot Y = A \cdot \bar{B} \qquad A + B = \bar{A} \cdot Y + X \qquad (A + B) \cdot X + A \cdot Y = A$$

The solution:

$X = A$ that is, X could have been type A

$Y = A\bar{B} + \bar{A}B$ that is, Y could have been either type A or type B, but not both types

Now suppose that the data had been the same as above, except that on the second day post 1 said that the airplanes were of type B and not type A. Would it then be possible, in the light of the observers' data, for the enemy to have only two types of airplanes?

Solution. The equations

$$X \cdot Y = A \cdot \bar{B} \qquad \bar{A} \cdot B = \bar{A} \cdot Y + X \qquad (A + B) \cdot X + A \cdot Y = A$$

do not have a solution, and therefore no assumption as to the identity of types X and Y with types B and A can be made consistent with the observers' reports.

Fourth Example: Solutions to Simultaneous Implications. The board of directors of a certain manufacturing firm had a meeting to decide on their future business policies in the light of their goals. They discussed the relationships between money spent on advertising, the price of the commodity they manufactured, the per cent of the salesmen's commissions, and a possible change-over to an improved design of their product, and also the effects various courses of action would have on the final total profit of the firm and on the quantity of the product sold. The goals of the firm were clearly decided upon: to make a large profit, which was their responsibility as the directors of the firm, and to sell large quantities of the article they manufactured, so as to keep the factory operating at its maximum possible capacity. They all agreed on the following five points:

1. If the policy is either to change to a better-designed pattern or to cut the advertising expenditures, or if they give high salesmen's commissions or raise the retail price of the commodity, and if only a small number of articles are sold, then the profits will be small.
2. If large sums are spent in advertising and there is no alteration of design, or if there is a change-over to a new design but the advertising allotment is cut, then there cannot be both a large sales volume and large profit.
3. If the firm makes a large profit or if its sales volume is high, then the retail price of the articles is low and the advertising expenditure must be high; or else the retail price must be low with a small advertising budget, and there must also be a change to a new improved design coupled with low salesmen's commissions.
4. If the retail price is low, if there is no cut in advertising, and if there are high salesmen's commissions, or if the price is low and the salesmen's commissions are low but there is a change to an improved product design, then a great quantity of items must be sold.
5. If there are large advertising expenditures with a change-over to a new and improved design, but the retail price is kept low and many articles are sold, then a high profit will result.

The problem is: With these general rules as a guide what must be the policy decision of the board so as to have a large profit, to sell many articles, and both to have a large profit and to sell many articles?

To solve this problem first note that the five statements are combinations of the following sentences or propositions:

A, spend large sums of money on advertising.
B, change over to a better product design.
C, have a high retail price.
D, have high salesmen's commissions.
X, make a large profit.
Y, have a large sales volume.

The five points can be symbolized in terms of these sentences:

1. $(B + \bar{A} + D + C) \cdot \bar{Y} \rightarrow \bar{X}$
2. $A \cdot \bar{B} + \bar{A} \cdot B \rightarrow \overline{X \cdot Y}$
3. $X + Y \rightarrow A \cdot \bar{C} + \bar{A} \cdot B \cdot \bar{C} \cdot \bar{D}$
4. $A \cdot \bar{C} \cdot D + B \cdot \bar{C} \cdot \bar{D} \rightarrow Y$
5. $A \cdot B \cdot \bar{C} \cdot Y \rightarrow X$

There are three pairs of solutions to this set of five implications:

$$\begin{array}{lll} X = A \cdot \bar{C} \cdot (B + \bar{D}) & X = A \cdot B \cdot \bar{C} & X = A \cdot B \cdot \bar{C} \\ Y = \bar{C} \cdot (A \cdot D + B \cdot \bar{D}) & Y = \bar{C} \cdot (A + B \cdot \bar{D}) & Y = \bar{C} \cdot (A \cdot D + B \cdot \bar{D}) \end{array}$$

Two policies consistent with these five rules can be obtained from the first set of results:

A policy incorporating a large advertising campaign with low retail prices, and either a change-over to a better design or low salesmen's commissions, is necessary and sufficient to ensure large profits.

Low retail prices, and either a large amount of advertising together with high salesmen's commissions or a new design and low salesmen's commissions, are necessary and sufficient for a large sales volume.

(The second and third sets of results should also be interpreted.)

In order to achieve both goals of large profits and a large sales volume, it is necessary and sufficient to have a large advertising budget, low retail prices, and a change-over to a new design, according to the first set of results.

The second and third set of results indicate that a large profit implies a large sales volume, and a large profit is obtained through large advertising costs, low retail prices, and a change-over to a new design, which are the identical requirements indicated from the first solution above.

g. Additional aspects of the material presented in this chapter and Chaps. 13 and 14 can be found in the following references by R. S. Ledley: Mathematical Foundations and Computational Methods for a Digital Logic Machine, *J. Operations Research Soc. Am.*, vol. 2, no. 3, pp. 249–274, August, 1954; Digital Computational Methods in Symbolic Logic, with Examples in Biochemistry, *Proc. Natl. Acad. Sci. U.S.*, vol. 41, no. 7, pp. 498–511, July, 1955; Boolean Matrix Equations in Digital Circuit Design, *IRE Trans. on Electronic Computers*, vol. EC-8, no. 2, pp. 131–139, June, 1959.

CHAPTER 13

BOOLEAN MATRIX EQUATIONS AND THE FUNDAMENTAL FORMULAS

13-1. Introduction and Statement of the Problem

Computational Methods. The first chapter of Part 3 introduced Boolean algebra and logic and described how a Boolean function can be interpreted as a circuit design and how conversely a digital-circuit design can be interpreted as a Boolean function. However, the problem immediately arose that there are in general many circuit designs, some simpler, some more complicated, that produce equivalent results, since a Boolean function can be written in many different equivalent forms. Chapter 11 tackled this problem by introducing a numerical-computation method that systematically enabled the reduction of a Boolean function to certain standard equivalent forms. Chapter 12 continued to produce further applications of the digital computational methods: constraints, solution to Boolean equations, transformation to the absolute simplest form. These last two applications were relatively specific methods for handling certain important types of problems.

In this chapter and the next the computational methods are extended in a more general way. Many problems that arise in Boolean algebra of the kind we consider in circuit design can be formulated in terms of these methods. In fact the solution to equations and the change of variables become special cases of the methods described in these chapters. The applications presented in this textbook are of necessity very limited, but in general the realm of applicability of these computational methods in electronic circuit design is broad. We reemphasize that individual originality is required to make full use of these methods. Practical application of the methods in engineering research ultimately depends on the ingenuity and skill of the user.

Problems of These Chapters. The kind of digital-circuit-design problems solved by the methods presented in this and the next chapter can most easily be described by means of a generalized example. Figure 13-1 is a block diagram of a circuit that has been separated into the circuits f_1, f_2, and F. The inputs to circuit f_1 are the wires labeled A_1, A_2, and A_3. Its single output is labeled f_1, where of course the signals produced in f_1 represent some Boolean function of the signals in the inputs; that is, $f_1 = f_1(A_1,A_2,A_3)$. Function f_2 is considered similarly. The circuit F has as inputs the wires labeled f_1, f_2, X_1, and X_2 and as output the wire labeled F. The signal produced by F represents some Boolean function of the signals of its inputs, that is, $F = F(f_1,f_2,X_1,X_2)$. However, since $f_1 = f_1(A_1,A_2,A_3)$ and $f_2 = f_2(A_1,A_2,A_3)$, we can evidently also describe the signals in F as a Boolean function of A_1, A_2, A_3, X_1, and X_2. For clarity we write this as $E(A_1,A_2,A_3,X_1,X_2)$, where in a certain sense $F = E$.

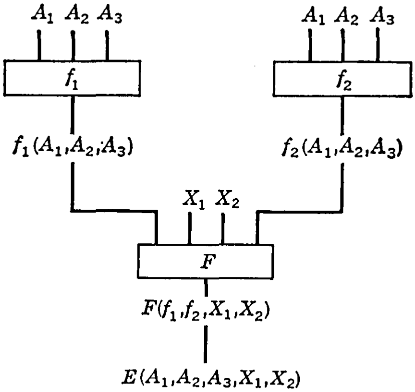


FIG. 13-1. Block diagram of circuits f_1, f_2, and F.

Now that we have described our example, let us see what elementary problems of circuit design can arise here. First, suppose the circuits f_1, f_2, and F have already been constructed. The problem might be to determine the result of wiring them together as in Fig. 13-1; that is, the problem would be to determine $E(A_1,A_2,A_3,X_1,X_2)$ (see Fig. 13-2a). Second, suppose that only the circuits f_1 and f_2 have been constructed and that it is desired to make another circuit F using as inputs f_1, f_2, X_1, and X_2 such that the end result will be the given desired function $E(A_1,A_2,A_3,X_1,X_2)$. The problem here is to determine the circuit design for F (see Fig. 13-2b). Third suppose that only the circuit F has been constructed and that it is desired to make circuits f_1 and f_2 so that when wired to F as in Fig. 13-1 the final result will be the given desired function $E(A_1,A_2,A_3,X_1,X_2)$. The problem in this case is to design f_1 and f_2 (see Fig. 13-2c). The general mathematical formulations of these problems are as follows:

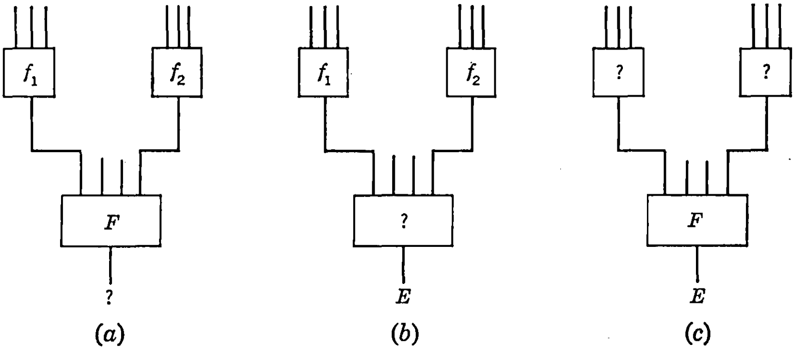


FIG. 13-2. Elementary problems of circuit design.

1. The function $F(f_1, \ldots ,f_J,X_1, \ldots ,X_K)$ is given explicitly, and the functions $f_1(A_1, \ldots ,A_I), \ldots , f_J(A_1, \ldots ,A_I)$ are each given explicitly; the problem is to find the unknown function

$$E(A_1, \ldots ,A_I,X_1, \ldots ,X_K)$$

2. The function $E(A_1, \ldots ,A_I,X_1, \ldots ,X_K)$ is given explicitly, and the functions $f_1(A_1, \ldots ,A_I), \ldots , f_J(A_1, \ldots ,A_I)$ are each given explicitly; the problem is to find the unknown function

$$F(f_1, \ldots ,f_J,X_1, \ldots ,X_K)$$

such that $E = F$.

3. Both the Boolean functions $E(A_1, \ldots ,A_I,X_1, \ldots ,X_K)$ and $F(A_1, \ldots ,A_I,X_1, \ldots ,X_K)$ are given explicitly; the problem is to find unknown functions $f_1(A_1, \ldots ,A_I), \ldots , f_J(A_1, \ldots ,A_I)$ such that $E = F$.

The first problem presents no difficulty, for in order to find E we merely substitute in F the explicit expression for each f_j of the A_i. The E so determined is always such that $F = E$.

The second and third problems are more difficult to solve, and systematic methods for their solution involving the designation numbers will be given in the next sections. A solution can always be found for the second problem; this is not so for the third problem. It may happen in certain instances that no solutions to the third problem exist. In such an event solutions exist only if certain constraints hold between $A_1, A_2, \ldots , A_I$. These constraints may then be obtained, as well as the solutions which are possible provided that these constraints occur.

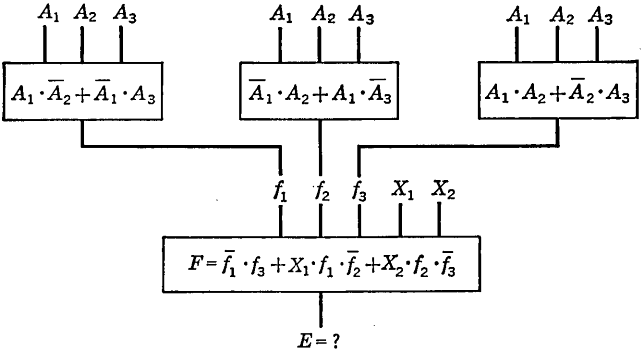
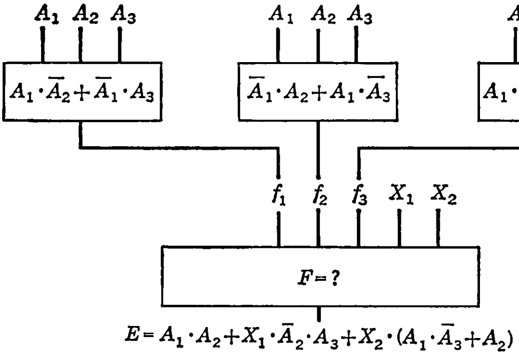

A_1 A_2 A_3
$A_1 \cdot \bar{A}_2 + \bar{A}_1 \cdot A_3$
A_1 A_2 A_3
$\bar{A}_1 \cdot A_2 + A_1 \cdot \bar{A}_3$
A_1 A_2 A_3
$A_1 \cdot A_2 + \bar{A}_2 \cdot A_3$
f_1 f_2 f_3 X_1 X_2
$F = \bar{f}_1 \cdot f_3 + X_1 \cdot f_1 \cdot \bar{f}_2 + X_2 \cdot f_2 \cdot \bar{f}_3$
$E = ?$

FIG. 13-3. Problem of type 1.

Examples. An example of a problem of type 1 is as follows: Suppose that three circuits, each with inputs A_1, A_2, and A_3, have been constructed with outputs $A_1 \cdot \bar{A}_2 + \bar{A}_1 \cdot A_3$, $\bar{A}_1 \cdot A_2 + A_1 \cdot \bar{A}_3$, and $A_1 \cdot A_2 + \bar{A}_2 \cdot A_3$, respectively. Also suppose that another circuit has been constructed with inputs f_1, f_2, f_3, X_1, and X_2 and with output $\bar{f}_1 \cdot f_3 + X_1 \cdot f_1 \cdot \bar{f}_2 + X_2 \cdot f_2 \cdot \bar{f}_3$. The problem is this: If the output of each of the former three circuits were connected, respectively, to inputs f_1, f_2, and f_3, what would be the output of the latter circuit in terms of A_1, A_2, A_3, and X_1 and X_2? The answer is $A_1 \cdot A_2 + X_1 \cdot \bar{A}_2 \cdot A_3 + X_2 \cdot (A_1 \cdot \bar{A}_3 + A_2)$ (see Fig. 13-3).

As an example of the problem of type 2, suppose that three circuits, each with inputs A_1, A_2, and A_3, have been constructed with outputs f_1, f_2, and f_3, such that $f_1 = A_1 \cdot \bar{A}_2 + \bar{A}_1 \cdot A_3$, $f_2 = \bar{A}_1 \cdot A_2 + A_1 \cdot \bar{A}_3$, and $f_3 = A_1 \cdot A_2 + \bar{A}_2 \cdot A_3$. Suppose that it is desired to use these circuits in the construction of another circuit F, such that the output of F is $E = A_1 \cdot A_2 + X_1 \cdot \bar{A}_2 \cdot A_3 + X_2 \cdot (A_1 \cdot \bar{A}_3 + A_2)$. The problem is this: How shall the outputs of the already constructed three circuits be gated so that E will result; i.e., what is the design of the circuit F that has inputs f_1, f_2, f_3, X_1, and X_2 and output E? It turns out that F should be $\bar{f}_1 \cdot f_3 + X_1 \cdot f_1 \cdot \bar{f}_2 + X_2 \cdot f_2 \cdot \bar{f}_3$ (see Fig. 13-4).



A_1 A_2 A_3
$A_1 \cdot \bar{A}_2 + \bar{A}_1 \cdot A_3$
A_1 A_2 A_3
$\bar{A}_1 \cdot A_2 + A_1 \cdot \bar{A}_3$
A_1 A_2 A_3
$A_1 \cdot A_2 + \bar{A}_2 \cdot A_3$
f_1 f_2 f_3 X_1 X_2
$F = ?$
$E = A_1 \cdot A_2 + X_1 \cdot \bar{A}_2 \cdot A_3 + X_2 \cdot (A_1 \cdot \bar{A}_3 + A_2)$

FIG. 13-4. Problem of type 2.

As an example of a type 3 problem, suppose that it is desired to make a circuit E with output

$$A_1 \cdot A_2 + X_1 \cdot \bar{A}_2 \cdot A_3 + X_2 \cdot (A_1 \cdot \bar{A}_3 + A_2)$$

In addition suppose that a certain circuit F has already been built which has inputs f_1, f_2, f_3, X_1, and X_2 and output

$$F = \bar{f}_1 \cdot f_3 + X_1 \cdot f_1 \cdot \bar{f}_2 + X_2 \cdot f_2 \cdot \bar{f}_3$$

The problem is this: What three circuits shall be constructed that have inputs A_1, A_2, and A_3 and outputs connected to inputs f_1, f_2, and f_3 of F, respectively, such that the output of the already constructed F will be the desired output E?

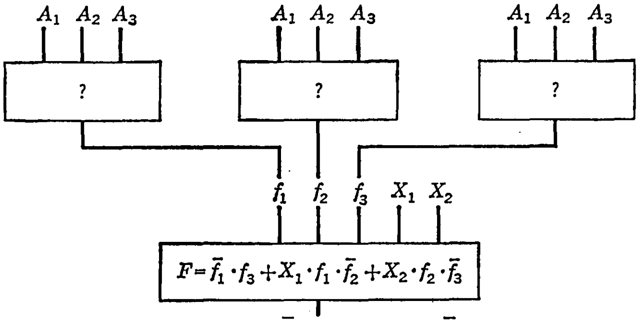
It turns out that the three circuits should be

$A_1 \cdot \bar{A}_2 + \bar{A}_1 \cdot A_3$ connected to f_1
$\bar{A}_1 \cdot A_2 + A_1 \cdot \bar{A}_3$ connected to f_2
and $A_1 \cdot A_2 + \bar{A}_2 \cdot A_3$ connected to f_3

(see Fig. 13-5).

As will be shown, there are many other kinds of digital-circuit-design problems that can be solved by the methods to be presented. However, these simple pictures serve to give the reader a general idea of the kind of problems that can be approached by means of the numerical computational methods given in these chapters.

Methods to Be Developed. In the methods that are the subject of this chapter and the next the computations are not always carried out in terms of the designation numbers. Rather, the designation numbers are turned into Boolean matrices; the computations involve these matrices. The resulting matrix is changed back into the designation numbers and the explicit Boolean function comprising the solution is derived from its number. This process is summarized in Fig. 13-6. It is interesting to note that the computations involving the matrices make use of only three pairs of fundamental formulas.


A_1 A_2 A_3
?
A_1 A_2 A_3
?
A_1 A_2 A_3
?
f_1 f_2 f_3 X_1 X_2
$F=\bar{f}_1 \cdot f_3 + X_1 \cdot f_1 \cdot \bar{f}_2 + X_2 \cdot f_2 \cdot \bar{f}_3$

$$E = A_1 \cdot A_2 + X_1 \cdot \bar{A}_2 \cdot A_3 + X_2 \cdot (A_1 \cdot \bar{A}_3 + A_2)$$

FIG. 13-5. Problem of type 3.

In this chapter we shall be primarily concerned with the statement of the computational methods and the proofs of the formulas; in the next chapter we shall present applications of the methods. Although the proofs of the formulas can substantially help in understanding the general approach to the applications, they are not a necessary part of the computational methods. Therefore, after Secs. 13-2 and 13-3, consideration of Secs. 13-4 through 13-6 *may be postponed without loss of continuity* until after Chap. 14 has been covered.

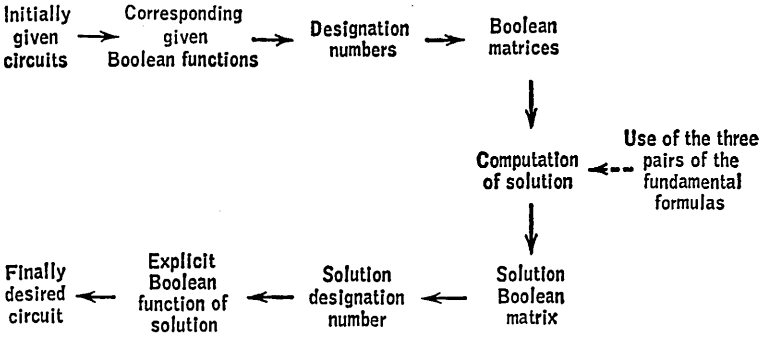

Initially given circuits
Corresponding given Boolean functions
Designation numbers
Boolean matrices
Computation of solution
Use of the three pairs of the fundamental formulas
Solution Boolean matrix
Solution designation number
Explicit Boolean function of solution
Finally desired circuit

FIG. 13-6. Summary of solution to circuit design.

Prerequisite to the proofs of the fundamental formulas is a discussion of the general Boolean matrix equation and its solution, given in Sec. 13-4. Then in Sec. 13-5 we apply the general solution to a special type of Boolean matrix equation, which is of considerable importance for our purposes. Finally, in Sec. 13-6 we develop the fundamental matrix equation that relates the Boolean matrix equations to the designation numbers of the circuit-design problem. Then proofs of the fundamental formulas follow easily. Figure 13-7 summarizes the true logical developmental relations among the sections of this chapter.

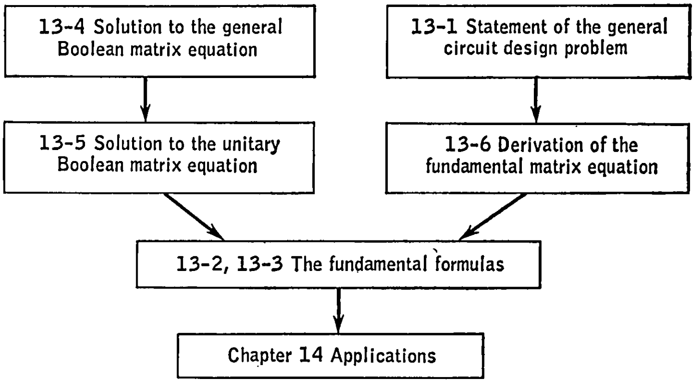

13-4 Solution to the general Boolean matrix equation
13-1 Statement of the general circuit design problem
13-5 Solution to the unitary Boolean matrix equation
13-6 Derivation of the fundamental matrix equation
13-2, 13-3 The fundamental formulas
Chapter 14 Applications

FIG. 13-7. Logical developmental relations among sections of Chap. 13.

13-2. Designation Numbers and Boolean Matrices

Functions to Matrices and Reverse. We shall see that the computational methods involve three Boolean matrices. One of these matrices is associated with $E(A_1, \ldots ,A_I,X_1, \ldots ,X_K)$ and is denoted by (E_{ki}); another is associated with $F(f_1, \ldots ,f_J,X_1, \ldots ,X_K)$ and is denoted by (F_{kj}); and the third is associated with the set of functions

$$f_1(A_1, \ldots ,A_I), \ldots , f_J(A_1, \ldots ,A_I)$$

and is denoted by (R_{ji}) (see Fig. 13-1).

The given functions of the problem are transformed into these matrices, the matrix for the desired function (or functions in the case of the f's) is derived from the given matrices, and the desired function is then explicitly obtained from its matrix. Going from the function to the matrix or from the matrix to the function involves the designation numbers. This section is devoted to the method for transforming function into matrix and back.

The Function E. The matrix (E_{ki}) can be derived from the Boolean function $E(A_1, \ldots ,A_I,X_1, \ldots ,X_K)$ in two steps: First form the designation number $\#E$ with respect to the basis

$$b[A_1,A_2, \ldots ,A_I,X_1,X_2, \ldots ,X_K]$$

For example, if $E = A_1 \cdot A_2 + X_1 \cdot \bar{A}_2 \cdot A_3 + X_2 \cdot (A_1 \cdot \bar{A}_3 + A_2)$, then with respect to $b[A_1,A_2,A_3,X_1,X_2]$ we have

$\#A_1 =$	0101	0101	0101	0101	0101	0101	0101	0101
$\#A_2 =$	0011	0011	0011	0011	0011	0011	0011	0011
$\#A_3 =$	0000	1111	0000	1111	0000	1111	0000	1111
$\#X_1 =$	0000	0000	1111	1111	0000	0000	1111	1111
$\#X_2 =$	0000	0000	0000	0000	1111	1111	1111	1111
$\#E =$	0001	0001	0001	1101	0111	0011	0111	1111

And, second, separate the positions of the designation number $\#E$ into 2^K successive groups of positions, with 2^I positions in each group. Index the groups from left to right by k ($k = 0, 1, 2, \ldots, 2^K - 1$) and the positions within each group by i ($i = 0, 1, 2, \ldots, 2^I - 1$). Then the *rows* of the matrix (E_{ki}) are simply these respective groups of positions. For our example, $K = 2$, $2^K = 4$, $I = 3$, $2^I = 8$, whence

$\#E =$	0001	0001	0001	1101	0111	0011	0111	1111
i	0123	4567	0123	4567	0123	4567	0123	4567
k	0		1		2		3	

and therefore

$$
(E_{ki}) = \begin{array}{cc} & \begin{array}{cc} i\ 0123 & 4567 \end{array} \\ \begin{array}{c} k\ 0 \\ 1 \\ 2 \\ 3 \end{array} & \begin{pmatrix} 0001 & 0001 \\ 0001 & 1101 \\ 0111 & 0011 \\ 0111 & 1111 \end{pmatrix} \end{array}
$$

What we are really doing is forming the successive rows of (E_{ki}) corresponding to the successive subbases for A_1, A_2, A_3 by appropriately "folding" the $\#E$, as indicated by the boxes.

The converse process, from (E_{ki}) to the explicit function E, is simply the reverse of the above procedure. Given (E_{ki}), we can immediately form $\#E$. Then $E(A_1, \ldots, A_I, X_1, \ldots, X_K)$ as an explicit Boolean function can be found in any form desired by means of the usual procedures with respect to $b[A_1,A_2, \ldots, A_I,X_1,X_2, \ldots, X_K]$.

The Function F. The matrix (F_{kj}) is derived similarly by folding $\#F$, where of course the basis $b[f_1,f_2, \ldots, f_J,X_1,X_2, \ldots, X_K]$ is used in computing $\#F$. Thus groups of $\#F$ are indexed by k ($k = 0, 1, 2, \ldots, 2^K - 1$) and the positions in each group by j ($j = 0, 1, 2, \ldots, 2^J - 1$). The *rows* of the matrix (F_{kj}) are simply these respective groups of positions. For example, if

$$F = \bar{f}_1 \cdot f_3 + X_1 \cdot f_1 \cdot \bar{f}_2 + X_2 \cdot f_2 \cdot \bar{f}_3$$

then with respect to $b[f_1,f_2,f_3,X_1,X_2]$ we have

$$\begin{array}{rcllllllll}
\#f_1 &=& 0101 & 0101 & 0101 & 0101 & 0101 & 0101 & 0101 & 0101\\
\#f_2 &=& 0011 & 0011 & 0011 & 0011 & 0011 & 0011 & 0011 & 0011\\
\#f_3 &=& 0000 & 1111 & 0000 & 1111 & 0000 & 1111 & 0000 & 1111\\
\#X_1 &=& 0000 & 0000 & 1111 & 1111 & 0000 & 0000 & 1111 & 1111\\
\#X_2 &=& 0000 & 0000 & 0000 & 0000 & 1111 & 1111 & 1111 & 1111\\
\hline
\#F &=& 0000 & 1010 & 0100 & 1110 & 0011 & 1010 & 0111 & 1110\\
\\
j & & 0123 & 4567 & 0123 & 4567 & 0123 & 4567 & 0123 & 4567\\
k & & \multicolumn{2}{c}{0} & \multicolumn{2}{c}{1} & \multicolumn{2}{c}{2} & \multicolumn{2}{c}{3}
\end{array}$$

and

$$(F_{kj}) = \begin{pmatrix} 0000 & 1010\\ 0100 & 1110\\ 0011 & 1010\\ 0111 & 1110 \end{pmatrix}$$

The converse process, finding F as an explicit Boolean function given (F_{kj}), is the reverse of the above procedure. First $\#F$ is formed, and from this the explicit Boolean function $F(f_1, \ldots ,f_J,X_1, \ldots ,X_K)$ is found with respect to basis $b[f_1,f_2, \ldots ,f_J,X_1,X_2, \ldots ,X_K]$.

The Set of Functions f_s. The matrix (R_{ji}) is formed in quite a different manner from the other matrices. To describe the relation between the function set $f_1(A_1, \ldots ,A_I), f_2(A_1, \ldots ,A_I), \ldots , f_J(A_1, \ldots ,A_I)$ and the matrix (R_{ji}), we first form the set of designation numbers for $f_1, f_2, \ldots , f_J$, each with respect to $b[A_1,A_2, \ldots ,A_I]$, and place these numbers in successive rows starting with $\#f_1$. For example, if

$$\begin{aligned}
f_1 &= A_1 \cdot \bar{A}_2 + \bar{A}_1 \cdot A_3\\
f_2 &= \bar{A}_1 \cdot A_2 + A_1 \cdot \bar{A}_3\\
f_3 &= A_1 \cdot A_2 + \bar{A}_2 \cdot A_3
\end{aligned}$$

then with respect to $b[A_1,A_2,A_3]$ we have

$$\begin{array}{rcll}
 & i & 0123 & 4567\\
\\
\#A_1 &=& 0101 & 0101\\
\#A_2 &=& 0011 & 0011\\
\#A_3 &=& 0000 & 1111\\
\hline
\#f_1(A_1,A_2,A_3) &=& 0100 & 1110\\
\#f_2(A_1,A_2,A_3) &=& 0111 & 0010\\
\#f_3(A_1,A_2,A_3) &=& 0001 & 1101\\
 & j & 0326 & 5534
\end{array}$$

Next index each column of $b[A_1, \ldots ,A_I]$ by i ($i = 0, 1, 2, \ldots , 2^I - 1$), and note below each column of the f_s array of designation numbers the binary number formed by that column (considering the top bit the least significant, the bottom bit the most significant). Notice that, if we form the basis $b[f_1,f_2, \ldots ,f_J]$ and index the columns by j ($j = 0, 1, 2, \ldots , 2^J - 1$), then the index of each column of the f_s array is the same as the index of the similar column of $b[f_1,f_2, \ldots ,f_J]$.

Now the elements of the Boolean matrix (R_{ji}) can be determined. An element R_{ji} of the matrix (R_{ji}) is a unit if corresponding to the ith column of $b[A_1, \ldots ,A_I]$ there is a column of the f_s array indexed by j; otherwise R_{ji} is zero. In our example we have the following correspondences: (j,i):(0,0), (3,1), (2,2), (6,3), (5,4), (5,5), (3,6), and (4,7). The only unit elements of the matrix (R_{ji}) are R_{00}, R_{31}, R_{22}, R_{63}, R_{54}, R_{55}, R_{36}, and R_{47}. We have for our example

$$(R_{ji}) = \begin{array}{r|ll|} & i\ 0123 & 4567 \\ \hline j\ 0 & 1000 & 0000 \\ 1 & 0000 & 0000 \\ 2 & 0010 & 0000 \\ 3 & 0100 & 0010 \\ & & \\ 4 & 0000 & 0001 \\ 5 & 0000 & 1100 \\ 6 & 0001 & 0000 \\ 7 & 0000 & 0000 \end{array}$$

Note that each column has just one unit but that this is not necessarily true of the rows.

The reverse process of generating $f_1(A_1, \ldots ,A_I)$, $f_2(A_1, \ldots ,A_I)$, $\ldots$, $f_J(A_1, \ldots ,A_I)$ from any given (R_{ji}) is not unique, in contrast to the other processes described in this section. Several sets of f_s are often possible, while, on the other hand, it may happen that there are no sets f_s corresponding to a given (R_{ji}). Two bases are consulted in this process, namely, $b[f_1, \ldots ,f_J]$ and $b[A_1, \ldots ,A_I]$, both written in the usual pattern. The desired solution is computed by means of the result array, as yet empty, with 2^I columns and J rows: the columns are indexed by i in order from left to right, the rows by $f_1, \ldots , f_J$, from top to bottom. For those pairs of indices j, i for which $R_{ji} = 1$, we place the jth column of $b[f_1, \ldots ,f_J]$ in the ith column of the result array. The rows of the result array thus filled are the designation numbers of the corresponding $f_1, \ldots , f_J$ with respect to the basis $b[A_1, \ldots ,A_I]$ and hence give the desired explicit functions

$$f_1(A_1, \ldots ,A_I), \ldots , f_J(A_1, \ldots ,A_I)$$

For example, consider

$$(R_{ji}) = \begin{array}{r|ll|} & i\ 0123 & 4567 \\ \hline j\ 0 & 1000 & 0000 \\ 1 & 0000 & 0000 \\ 2 & 0010 & 0000 \\ 3 & 0100 & 0010 \\ & & \\ 4 & 0000 & 0001 \\ 5 & 0000 & 1100 \\ 6 & 0001 & 0000 \\ 7 & 0001 & 0010 \end{array}$$

$b[f_1,f_2,f_3]$

j	0123	4567
$\#f_1 =$	0101	0101
$\#f_2 =$	0011	0011
$\#f_3 =$	0000	1111

RESULT ARRAY

i	0	1	2	3	4	5	6	7
$\#f_1(A_1,A_2,A_3)$	0	1	0	0, 1	1	1	1, 1	0
$\#f_2(A_1,A_2,A_3)$	0	1	1	1, 1	0	0	1, 1	0
$\#f_3(A_1,A_2,A_3)$	0	0	0	1, 1	1	1	0, 1	1

whence there are four possible sets of solutions which with respect to the usual basis $b[A_1,A_2,A_3]$ are seen to be

$$\begin{aligned} f_1 &= A_1 \cdot \bar{A}_2 + \bar{A}_1 \cdot A_3 \\ f_2 &= \bar{A}_1 \cdot A_2 + A_1 \cdot \bar{A}_3 \\ f_3 &= A_1 \cdot A_2 + \bar{A}_2 \cdot A_3 \end{aligned} \qquad \begin{aligned} f_1 &= A_1 \cdot \bar{A}_2 + \bar{A}_1 \cdot A_3 + A_1 \cdot \bar{A}_3 \\ f_2 &= \bar{A}_1 \cdot A_2 + A_1 \cdot \bar{A}_3 \\ f_3 &= A_1 \cdot A_2 + A_3 \end{aligned}$$

$$\begin{aligned} f_1 &= A_1 \cdot \bar{A}_2 + \bar{A}_1 \cdot A_3 \\ f_2 &= \bar{A}_1 \cdot A_2 + \bar{A}_1 \cdot \bar{A}_3 \\ f_3 &= A_1 \cdot A_2 + A_3 \end{aligned} \qquad \begin{aligned} f_1 &= A_1 \cdot \bar{A}_2 + \bar{A}_1 \cdot A_3 + A_1 \cdot \bar{A}_3 \\ f_2 &= \bar{A}_1 \cdot A_2 + A_1 \cdot \bar{A}_3 \\ f_3 &= A_1 \cdot A_2 + \bar{A}_2 \cdot A_3 \end{aligned}$$

If a column of (R_{ji}) were all zeros, then the corresponding column of the result array would be left blank and no solutions would exist.

EXERCISES

(*a*) Find the matrix (E_{ki}) for $E = A_1 \cdot \bar{A}_2 \cdot X_1 + \bar{A}_1 \cdot A_2 \cdot \bar{X}_1$.

Solution. $(E_{ki}) = \begin{pmatrix} 0010 \\ 0100 \end{pmatrix}$.

(*b*) If $(E_{ki}) = (0110)$, find E.

Solution. $E = A_1 \cdot \bar{A}_2 + \bar{A}_1 \cdot A_2$.

(*c*) Find the matrix (F_{kj}) for $F = [(f_1 = X_1) \cdot (f_2 = X_2)]$. [HINT: Recall that $(f_1 = X_1)$ can be written as $f_1 \cdot X_1 + \bar{f}_1 \cdot \bar{X}_1$.]

Solution. $(\mathrm{F}_{kj}) = \begin{pmatrix} 1000 \\ 0100 \\ 0010 \\ 0001 \end{pmatrix}$.

(*d*) If $(F_{kj}) = \begin{pmatrix} 01 \\ 10 \\ 10 \\ 01 \end{pmatrix}$, find F.

Solution. $F = (f_1 \cdot \bar{X}_1 + \bar{f}_1 \cdot X_1) \cdot \bar{X}_2 + (f_1 \cdot X_1 + \bar{f}_1 \cdot \bar{X}_1) \cdot X_2$.

(*e*) Find the matrix (R_{ji}) for

$$\begin{aligned} f_1 &= A_1 \cdot \bar{A}_2 + \bar{A}_1 \cdot A_3 \\ f_2 &= A_1 \cdot A_2 + \bar{A}_2 \cdot A_3 \end{aligned}$$

Solution. $(R_{ji}) = \begin{pmatrix} 1010 & 0000 \\ 0100 & 0010 \\ 0001 & 0001 \\ 0000 & 1100 \end{pmatrix}$.

(*f*) If $(R_{ji}) = \begin{pmatrix} 1100 \\ 1010 \end{pmatrix}$, find the set f_s.

Solution. None.

(*g*) If $(R_{ji}) = \begin{pmatrix} 1010 & 0000 \\ 0101 & 0010 \\ 0001 & 0101 \\ 0000 & 1100 \end{pmatrix}$, find the set or sets f_s.

13-3. Antecedences and Consequences, and the Fundamental Formulas

Antecedence and Consequence Solutions. In Sec. 13-1 (see Fig. 13-1) we described three types of problems; these can be summarized in terms of the set of functions $f_1(A_1, \ldots ,A_I), \ldots , f_J(A_1, \ldots ,A_I)$, the functions $F(f_1, \ldots ,f_J,X_1, \ldots ,X_K)$, $E(A_1, \ldots ,A_J,X_1, \ldots ,X_K)$, and the equivalence equation†

$$F(f_1(A_1, \ldots ,A_I), \ldots ,f_J(A_1, \ldots ,A_I),X_1, \ldots ,X_K) = E(A_1, \ldots ,A_I,X_1, \ldots ,X_K) \quad (13\text{-}1)\blacklozenge$$

For type 1 problems the function F and the set of functions $f_1, \ldots , f_J$ are given, and the problem is to find a solution E satisfying Eq. (13-1). In type 2 problems the function E and the set of functions $f_1, \ldots , f_J$ are given, and the problem is to find a solution F that satisfies Eq. (13-1). For type 3 problems the functions F and E are given, and the problem is to find the set of solutions $f_1, \ldots , f_J$ that satisfies Eq. (13-1).

It turns out, however, that it is more convenient to solve problems of types 2 and 3 in two parts, one involving the equation

$$F_a(f_1(A_1, \ldots ,A_I), \ldots ,f_J(A_1, \ldots ,A_I),X_1, \ldots ,X_K) \rightarrow E(A_1, \ldots ,A_I,X_1, \ldots ,X_K) \quad (13\text{-}2)\blacklozenge$$

and the other involving the equation

$$E(A_1, \ldots ,A_I,X_1, \ldots ,X_K) \rightarrow F_c(f_1(A_1, \ldots ,A_I), \ldots ,f_J(A_1, \ldots ,A_I),X_1, \ldots ,X_K) \quad (13\text{-}3)\blacklozenge$$

For if a solution satisfies both Eqs. (13-2) *and* (13-3), *then it satisfies Eq·* (13-1); that is, if $F_a = F_c = F$, then $F = E$. Solutions of Eq. (13-2) are called *antecedence solutions*, and solutions of Eq. (13-3) are called *consequence solutions.*

The meaning of the antecedence and consequence solutions can be understood as follows: Recall that the function E (that is, the final desired output of the circuit) gives those combinations of input conditions that are to result in zero and unit output signals. An antecedence solution represents a circuit design that will produce a zero output signal for *at least all* input conditions for which E is to be zero; it may have a zero output signal for other input conditions as well. A consequence solution represents a circuit design that will produce a *unit* output signal for *at least all* input conditions for which E is to be a unit; it may have a unit output signal for other input conditions as well. Thus antecedence and consequence solutions do not necessarily represent circuits that have

† Black diamonds indicate the more important formulas.

an output identical to E but do represent circuits whose outputs are "close" to that of E.

For example, if the designation number of E is 0110, then the designation numbers of all possible antecedence solutions are 0100, 0010, and 0000, while the designation numbers of all possible consequence solutions are 1110, 0111, and 1111.

As is illustrated by this example, it is not very difficult to produce antecedence and consequence solutions to a given equation. The importance of antecedence and consequence solutions arises from the fact that circuit-design problems always require antecedence and consequence solutions of a specified form, or involving only certain specified input wires, or both.

For example, suppose that $E = (A_1 \cdot \bar{A}_2 + \bar{A}_1 \cdot A_2) \cdot X_1$ and that it is desired to find antecedence solutions of the form $F_a = f_1 \cdot f_2 \cdot X_1$. Since E and F are given and we want to find f_1, f_2, this is a type 3 problem. First observe that with respect to $b[A_1,A_2,X_1]$ we find $\#E = 0000\ 0110$. An antecedence solution will have fewer units than $\#E$ in its designation number, as, for example, $0000\ 0100 = \#A_1 \cdot \bar{A}_2 \cdot X_1$. So, if we let $f_1 = A_1$ and $f_2 = \bar{A}_2$, we have found an antecedence solution, i.e.,

$$F_a(f_1(A_1,A_2),f_2(A_1,A_2,),X_1) = f_1(A_1,A_2) \cdot f_2(A_1,A_2) \cdot X_1 = A_1 \cdot \bar{A}_2 \cdot X_1$$

and $A_1 \cdot \bar{A}_2 \cdot X_2 \rightarrow (A_1 \cdot \bar{A}_2 + \bar{A}_1 \cdot A_2) \cdot X_1$ as desired.

As another example, suppose that $E = (A_1 \cdot \bar{A}_2 + \bar{A}_1 \cdot A_2) \cdot X_1$ as above, that $f_1 = A_1$, $f_2 = \bar{A}_2$, and that we desire to find a consequence solution $F_c(f_1,f_2,X_1)$. Since E, f_1, and f_2 are given and we want to find F_c, this is a type 2 problem. Again $\#E = 0000\ 0110$. A consequence solution will have more units than $\#E$ in its designation number—for example, $0000\ 0111 = \#(A_1 + A_2) \cdot X_1$. So, if we let

$$F_c = (f_1 + \bar{f}_2) \cdot X_1$$

we have found a consequence solution, for

$$\begin{aligned} F_c(f_1(A_1,A_2),f_2(A_1,A_2),X_1) &= (f_1(A_1,A_2) + \bar{f}_2(A_1,A_2)) \cdot X_1 \\ &= (A_1 + A_2) \cdot X_1 \end{aligned}$$

and $(A_1 \cdot \bar{A}_2 + \bar{A}_1 \cdot A_2) \cdot X_1 \rightarrow (A_1 + A_2) \cdot X_1$ as desired.

Note that the trial-and-error methods of these examples is far from desirable. We shall now present the fundamental formulas that enable the systematic computation of antecedence and consequence solutions.

The Fundamental Formulas. The formulas are given in terms of the Boolean matrices developed in the previous section. There are six fundamental formulas, an antecedence formula and a consequence formula for each of the three types of problems. Observe that in terms of the matrices the problems of types 1, 2, and 3 can be distinguished as follows:

For type 1 the matrices (F_{kj}) and (R_{ji}) are given, and the problem is to determine (E_{ki}).

For type 2 the matrices (E_{ki}) and (R_{ji}) are given, and the problem is to determine (F_{kj}).

For type 3 the matrices (E_{ki}) and (F_{kj}) are given, and the problem is to determine (R_{ji}).

Let us first recall (see Additional Topics, Sec. 13-7) that, if (M_{pq}) is a Boolean matrix, then $(\bar{M}_{pq})$ is the matrix formed by changing all zero elements of (M_{pq}) to units and all unit elements of (M_{pq}) to zeros. Also (M_{qp}), the matrix formed by interchanging the rows and columns of (M_{pq}), is called the transpose of (M_{pq}). For example, if $(M_{pq}) = \begin{pmatrix} 01 \\ 01 \end{pmatrix}$, then $(\bar{M}_{pq}) = \begin{pmatrix} 10 \\ 10 \end{pmatrix}$ and $(M_{qp}) = \begin{pmatrix} 00 \\ 11 \end{pmatrix}$. Also remember from Sec. 11-11 that Boolean matrix multiplication, indicated by $\otimes$, is identical to ordinary matrix multiplication, except that the logical sum and logical product replace the ordinary sum and product.

We now present the formulas that enable antecedence and consequence solutions to be determined for each of the above types of problems, where the subscripts a and c are used to distinguish between the determination of matrices corresponding to antecedence or consequence solutions:

	Antecedence	Consequence	
Type 1	$(F_{kj}) \otimes (R_{ji}) = (E_{ki})$	$(\bar{F}_{kj}) \otimes (R_{ji}) = (\bar{E}_{ki})$	
Type 2	$(R_{ji}) \otimes (\bar{E}_{ik}) = (\bar{F}_{jk})_a$	$(R_{ji}) \otimes (E_{ik}) = (F_{jk})_c$	(13-4) ◆
Type 3	$(F_{jk}) \otimes (\bar{E}_{ki}) = (\bar{R}_{ji})_a$	$(\bar{F}_{jk}) \otimes (E_{ki}) = (\bar{R}_{ji})_c$	

Properties of the Solutions. As will be shown in the succeeding sections, the solutions have the following properties:

TYPE 1 PROBLEMS. In problems of type 1 the antecedence and consequence solutions E are the same (and hence there was no necessity for using subscripts a and c); the formulas give two ways of determining E. In this case we *always* have $F = E$. The principal application of the formulas for type 1 is as a quick method for performing substitutions: we are given $F(f_1, \ldots, f_J, X_1, X_2, \ldots, X_K)$, and then the $E(A_1, \ldots, A_I, X_1, \ldots, X_K)$ determined by the formula is the result of substituting the given functions $f_1(A_1, \ldots, A_I), \ldots, f_J(A_1, \ldots, A_I)$ into F.

TYPE 2 PROBLEMS. A solution to problems of type 2 is in the form of a single function. Most often there are multiple solutions; i.e., there are many functions which are each good solutions. There is *always at least one* antecedence or consequence solution. Suppose that F_1, F_2, $\ldots$, F_N are the multiple solutions for an antecedence problem of type 2; then $F_n \rightarrow E$ $(n = 1, 2, \ldots, N)$. A surprising property of such a list of multiple antecedence solutions is that one of these solutions is implied by all the rest; i.e., there exists an F_m among the solutions such that $F_n \rightarrow F_m$ for all n. We call this the solution "closest" to E, because $F_m \rightarrow E$, but for all other solutions $F_n \rightarrow F_m \rightarrow E$.

Similarly suppose that $F_1, F_2, \ldots, F_N$ are the multiple solutions for a consequence problem of type 2; then $E \rightarrow F_n$ ($n = 1, 2, \ldots, N$). Again, there always exists a closest solution to E; that is, there exists some solution F_m such that F_m implies all other solutions: $E \rightarrow F_m \rightarrow F_n$ for all n. These closest solutions can easily be picked out and are usually the most useful.

An important application of the formulas for problems of type 2 is as an alternative method for determining any constraints between given functions. Use for (E_{ik}) the single-column matrix (column vector) of 2^I *rows every element of which is a unit,* and determine (R_{ji}) as usual from the given functions $f_1(A_1, \ldots, A_I), \ldots, f_J(A_1, \ldots, A_I)$ between which the constraint is being determined. A solution $F(f_1, \ldots, f_J)$ that is *both* an antecedence and consequence solution is the desired constraint. If no solution exists that is both an antecedence and consequence solution, then the functions $f_1, \ldots, f_J$ are independent (see Sec. 14-2).

TYPE 3 PROBLEMS. The solutions to problems of type 3 are in the form of a set of functions. It is possible to have multiple solutions, i.e., many sets of functions which are perfectly good solutions, analogous to multiple solutions to algebraic equations. Occasionally one of these sets of solutions contains a function $f = I$ or $f = 0$, that is, $\#f =$ all units, or $\#f =$ all zeros; these are *trivial* solutions. It is also possible that no solution exists. As we have indicated above, when no solution exists conditions can be determined under which solutions will exist. These conditions are given in terms of constraints between the $A_1, A_2, \ldots, A_I$. The general method for determining these constraints is to compute the Boolean row vector or single-row matrix (C_i) from the formula

$$(V_j) \otimes (R_{ji}) = (C_i)$$

where (V_j) is a row vector consisting of all units. Then (C_i) is the designation number of the desired constraints or conditions for the existence of solutions, referred to $b[A_1, \ldots, A_I]$. In order to impose this constraint, new bases constrained by the condition must now *replace* $b[A_1, \ldots, A_I, X_1, \ldots, X_K]$ and $b[A_1, \ldots, A_I]$, and the solution is found as described above except that the range of i is from 0 to $u - 1$, where u is the number of units in (C_i).

As mentioned in Sec. 13-1, the reader may go directly to Chap. 14 for immediate examples of applications of these formulas.

EXERCISES

(*a*) If $E = \bar{A}_2 \cdot X_1 + \bar{A}_1 \cdot \bar{X}_1$, $f_1 = A_1 \cdot \bar{A}_2$, and $f_2 = \bar{A}_1 \cdot A_2$, find both antecedence and consequence solutions $F_a(f_1,f_2,X_1)$ and $F_c(f_1,f_2,X_1)$. Show that $F_a \rightarrow E$ and $E \rightarrow F_c$. (HINT: Substitute for f_1 and f_2 in terms of A_1 and A_2 in the functions F_a and F_c, find their designation numbers, and compare with the designation number of E.)

(*b*) If $E = \bar{A}_2 \cdot X_1 + \bar{A}_1 \cdot \bar{X}_1$ and $F = (f_1 + f_2) \cdot X_1 + f_1 \cdot f_2 \cdot \bar{X}_1$, find the sets of antecedence and consequence solutions f_s.

(*c*) From Exercise *b* choose the set of antecedence solutions $\#f_1 = 1000$, $\#f_2 = 1100$, and show that for antecedence $F_a \rightarrow E$, using the formula of type 1. Choose the set of consequence solutions $\#f_1 = 1010$, $\#f_2 = 1110$, and show that $E \rightarrow F_c$, using the formula of type 1.

13-4. Solution to the General Boolean Matrix Equation†

The General Boolean Matrix Equation. Consider the matrix equation

$$(a_{ij}) \otimes (x_{jk}) = (b_{ik}) \qquad (13\text{-}5)\blacklozenge$$

where (a_{ij}), (x_{jk}), and (b_{ik}) are Boolean matrices. Suppose that (a_{ij}) and (b_{ik}) are known and that the problem is to determine the Boolean matrix (x_{jk}) that satisfies this equation. This is similar to the problem of the solution of ordinary matrix equations, but the conditions for the existence of a solution and the character of the solutions when they exist differ greatly. For example, consider the following Boolean matrix equation:

$$\begin{pmatrix} 1000 & 0100 \\ 0010 & 0001 \\ 1010 & 0111 \\ 1001 & 1000 \end{pmatrix} \otimes (x_{jk}) = \begin{pmatrix} 1110 \\ 1011 \\ 1111 \\ 0110 \end{pmatrix}$$

As can be readily checked, each of the following is a solution:

$$\begin{pmatrix} 0110 \\ 1111 \\ 1011 \\ 0110 \\ \\ 0110 \\ 1110 \\ 1111 \\ 1011 \end{pmatrix} \quad \begin{pmatrix} 0110 \\ 0000 \\ 1011 \\ 0000 \\ \\ 0000 \\ 1000 \\ 0000 \\ 0000 \end{pmatrix} \quad \begin{pmatrix} 0010 \\ 0000 \\ 0000 \\ 0100 \\ \\ 0000 \\ 1100 \\ 0000 \\ 1011 \end{pmatrix} \quad \begin{pmatrix} 0000 \\ 0000 \\ 0010 \\ 0000 \\ \\ 0110 \\ 1110 \\ 0000 \\ 1001 \end{pmatrix}$$

In fact there are *more than* 2^{17} possible solutions to this equation. On the other hand, the equation

$$\begin{pmatrix} 1010 & 1100 \\ 0110 & 0011 \\ 1010 & 0111 \\ 1001 & 0110 \end{pmatrix} \otimes (x_{jk}) = \begin{pmatrix} 0010 \\ 1101 \\ 0000 \\ 1001 \end{pmatrix}$$

has only one solution, namely,

$$\begin{pmatrix} 0000 \\ 1101 \\ 0000 \\ 1001 \\ \\ 0010 \\ 0000 \\ 0000 \\ 0000 \end{pmatrix}$$

† The sections that follow presuppose the reader's familiarity with the operations and properties of Boolean matrices given in the Additional Topics of this chapter (Sec. 13-7).

and the equation

$$\begin{pmatrix} 1010 & 0100 \\ 0110 & 0011 \\ 1010 & 0111 \\ 1001 & 0110 \end{pmatrix} \otimes (x_{jk}) = \begin{pmatrix} 0010 \\ 1101 \\ 1111 \\ 1001 \end{pmatrix}$$

has no solution.

Solution to the Equation. We shall show below that a necessary and sufficient condition that a solution to Eq. (13-5) exists is that†

$$(a_{ij}) \otimes \overline{((a_{ij})^T \otimes (\overline{b_{ik}}))} = (b_{ik}) \qquad (13\text{-}6)\blacklozenge$$

Similarly a necessary and sufficient condition for a solution to exist for

$$(x_{kj}) \otimes (a_{ji}) = (b_{ki}) \qquad (13\text{-}7)$$

is that

$$\overline{((\overline{b_{ki}}) \otimes (a_{ji})^T)} \otimes (a_{ji}) = (b_{ki}) \qquad (13\text{-}8)$$

Evidently, if condition (13-6) holds, then

$$(x'_{jk}) = \overline{(a_{ij})^T \otimes (\overline{b_{ik}})} \qquad (13\text{-}9)\blacklozenge$$

is a solution to Eq. (13-5) (this, by the way, proves sufficiency). In fact such an (x'_{jk}) is a very special solution, for as we shall see, if another solution (x_{jk}) to Eq. (13-5) exists, then $(x_{jk}) \rightarrow (x'_{jk})$. That is, (x'_{jk}) is implied by all other solutions; we shall therefore say that it is the *least upper bound* of solutions to Eq. (13-5). Analogous results are true for Eq. (13-7), for which any solution $(x_{kj}) \rightarrow (x'_{kj})$, where

$$(x'_{kj}) = \overline{(\overline{b_{ki}}) \otimes (a_{ji})^T} \qquad (13\text{-}10)$$

Proofs. Proof of the necessity of condition (13-6) [or (13-8)], i.e., that if a solution exists then Eq. (13-9) [or (13-10)] gives one, is based on the following considerations: Recall that from the definition of Boolean matrix multiplication, given in Sec. 11-11, Eq. (13-5) is equivalent to the set of (I times K) equations

$$\sum_j a_{ij} \cdot x_{jk} = b_{ik} \qquad (13\text{-}11)$$

where $\sum\limits_j$ indicates that the logical summation is to be taken of all the logical products over the range of j. From these equations it is clear that each column of (x_{jk}) depends only on the matrix (a_{ij}) and the corresponding column of (b_{ik}) and is independent of the other columns of (b_{ik}). Hence, if the existence condition can be shown to hold for each column of Eq. (13-5), it holds for the entire matrix. Similarly Eq.

† The superscript T denotes transposition, i.e., for example, $(a_{ij})^T = (a_{ji})$—see Sec 13-7.

(13-7) is equivalent to the set of equations

$$\sum_j x_{kj} \cdot a_{ji} = b_{ki} \tag{13-12}$$

from which it is seen that the rows are independent of each other.

We consider as an example the first column of Eq. (13-5) (that is, $k = 0$). This column must satisfy the following subset of Eq. (13-11):

$$\sum_j a_{ij} \cdot x_{j0} = b_{i0} \tag{13-13}$$

Let us assume that a solution exists; then consider those i for which $b_{i0} = 0$. In such an ith row, for each j such that $a_{ij} = 1$, the corresponding x_{j0} *must* be zero in order that $\sum_j a_{ij} \cdot x_{j0} = 0$. For each j such that $a_{ij} = 0$ the corresponding x_{j0} can be either 1 or 0. Next consider those i for which $b_{i0} = 1$: here there must be at least one value for j for which both $a_{ij} = 1$ and $x_{j0} = 1$. By the assumption that a solution exists *it will be obtained* if we make $x_{j0} = 1$ for all j except those j where $b_{i0} = 0$ *and* where $a_{ij} = 1$ for at least one value of i require that $x_{j0} = 0$. Let us denote this set of x_{j0} by (x'_{i0}).

The condition that $x'_{j0} = 0$ can be expressed as $(b_{i0} + \bar{a}_{ij}) = 0$ for at least one i. In other words, $\prod_i (b_{i0} + \bar{a}_{ij}) = 0$, or equivalently

$$\sum_i \bar{b}_{i0} \cdot a_{ij} = 1$$

Since otherwise $x'_{j0} = 1$, we can set $\bar{x}'_{j0} = \sum_i \bar{b}_{i0} \cdot a_{ij}$. This is equivalent to the matrix equation

$$(x'_{j0}) = \overline{(a_{ij})^T \otimes (\overline{b_{i0}})} \tag{13-14}$$

Equation (13-9) follows by extension. Equation (13-10) follows by similar reasoning.

We have shown that if there is any solution to Eq. (13-13) then x'_{j0} is also a solution; we shall now show that if x_{j0} satisfies Eq. (13-13) then $x_{j0} \rightarrow x'_{j0}$. For suppose there exists a solution to Eq. (13-13), say (z_{j0}). Clearly $z_{j0} = 1$ cannot occur for any j for which $x'_{j0} = 0$, by the definition of (z_{j0}), and therefore $z_{j0} \rightarrow x'_{j0}$. [Note that we say that $(M_{jk}) \rightarrow (N_{jk})$ when $M_{jk} \rightarrow N_{jk}$ for each k and j.]

The matrices (x'_{jk}) and (x'_{kj}) have a property of great importance in the cases where Eqs. (13-5) and (13-7) have no solution. Observe that in any case we have formed (x'_{j0}) such that $(a_{ij}) \otimes (x'_{j0})$ has zeros at least wherever (b_{i0}) has zeros. Therefore $(a_{ij}) \otimes (x'_{j0}) \rightarrow (b_{i0})$. By extension it follows that

$$(a_{ij}) \otimes (x'_{jk}) \rightarrow (b_{ik}) \tag{13-15}\blacklozenge$$

even when no solution (x_{jk}) exists for Eq. (13-5). Similarly, for the problems represented by Eq. (13-7),

$$(x'_{kj}) \otimes (a_{ji}) \rightarrow (b_{ki}) \tag{13-16}$$

For example, consider $\begin{pmatrix}11\\00\end{pmatrix} \otimes (x_{jk}) = \begin{pmatrix}10\\11\end{pmatrix}$; since $(x'_{jk}) = \begin{pmatrix}10\\10\end{pmatrix}$, we find $\begin{pmatrix}11\\00\end{pmatrix} \otimes \begin{pmatrix}10\\10\end{pmatrix} = \begin{pmatrix}10\\00\end{pmatrix} \rightarrow \begin{pmatrix}10\\11\end{pmatrix}$.

Character of Solutions to the Equation. We have shown above that (x'_{kj}) is the least upper bound to solutions to Eq. (13-5), if any solutions exist. However, there is in general *no* unique *greatest lower bound.* For example, consider the equation

$$\begin{pmatrix}101\\011\end{pmatrix} \otimes (x_{jk}) = \begin{pmatrix}10\\11\end{pmatrix}$$

There are 5 solutions to this equation, which are shown in Fig. 13-8. In this case there are 2 *lowest* solutions (dark parentheses), i.e., solutions that are *implied by no other solutions.* In the example shown above with the over 2^{17} solutions there are 72 lowest solutions. The lowest solutions have an interesting property: if (y_{jk}) is any lowest solution of Eq. (13-5), then all matrices (w_{jk}) such that

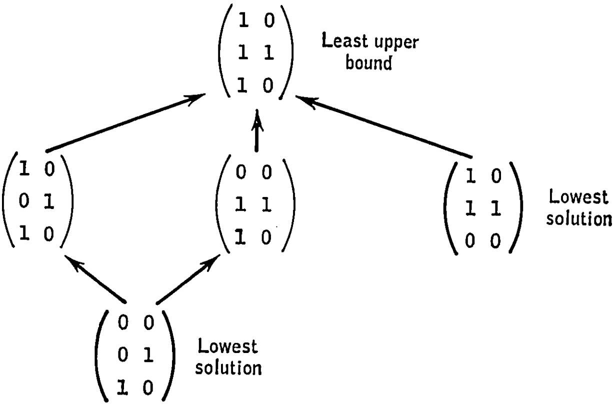


FIG. 13-8. Five solutions to the equation $\begin{pmatrix}101\\011\end{pmatrix} \otimes (x_{jk}) = \begin{pmatrix}10\\11\end{pmatrix}$.

$$(y_{jk}) \rightarrow (w_{jk}) \rightarrow (x'_{jk}) \tag{13-17}\blacklozenge$$

are also solutions. In fact the totality of solutions for a Boolean matrix equation is found as follows: Compute (x'_{jk}); then find all the lowest solutions; finally for each lowest solution generate all solutions (w_{jk}). The solution (x'_{jk}) is found by means of the above formulas; and we shall discuss below how to find the lowest solutions (y_{jk}). To find the solutions (w_{jk}), simply start from the corresponding lowest solution (y_{jk}), and replace those zeros of (y_{jk}) that do *not* correspond to zeros of (x'_{jk})

with units, one at a time, two at a time, and so forth. Of course analogous remarks hold for (x'_{kj}) corresponding to Eq. (13-7), that is,

$$(y_{kj}) \longrightarrow (w_{kj}) \longrightarrow (x'_{kj}) \tag{13-18}$$

For the example equation above

$$(x'_{jk}) = \overline{\begin{pmatrix}101\\011\end{pmatrix}^T \otimes \overline{\begin{pmatrix}10\\11\end{pmatrix}}} = \overline{\begin{pmatrix}10\\01\\11\end{pmatrix} \otimes \begin{pmatrix}01\\00\end{pmatrix}} = \overline{\begin{pmatrix}01\\00\\01\end{pmatrix}} = \begin{pmatrix}10\\11\\10\end{pmatrix}$$

Since $(a_{ij}) \otimes (x'_{jk}) = \begin{pmatrix}010\\011\end{pmatrix} \otimes \begin{pmatrix}10\\11\\10\end{pmatrix} = \begin{pmatrix}10\\11\end{pmatrix} = (b_{ik})$, a solution exists. It will be shown below that there are two lowest solutions, namely, $\begin{pmatrix}00\\01\\10\end{pmatrix}$ and $\begin{pmatrix}10\\11\\00\end{pmatrix}$. Hence we have for (w_{jk}) the matrices

$$\begin{pmatrix}00\\01\\10\end{pmatrix} \quad \begin{pmatrix}10\\01\\10\end{pmatrix} \quad \begin{pmatrix}00\\11\\10\end{pmatrix} \quad \begin{pmatrix}10\\11\\00\end{pmatrix} \quad \begin{pmatrix}10\\11\\10\end{pmatrix}$$

There are only these five different solutions, as shown in Fig. 13-8.

Computing the Lowest Solutions.† The problem of computing all solutions, if any solution exists, now reduces to the determination of the lowest solutions (y_{jk}). Here the method is only semisystematic. Let us present the method by means of an example. Consider the equation

$$\begin{pmatrix}1000 & 0100\\0010 & 0001\\1010 & 0111\\1001 & 1000\end{pmatrix} \otimes (x_{jk}) = \begin{pmatrix}1110\\1011\\1111\\0110\end{pmatrix} \qquad \text{whence } (x'_{jk}) = \begin{pmatrix}0110\\1111\\1011\\0110\\ \\0110\\1110\\1111\\1011\end{pmatrix}$$

The possible columns of the matrix (y_{jk}) are determined one at a time in three steps. Let us start with (y_{j0}). (1) Form the array composed of only those rows of (a_{ij}) which correspond to units of the column (b_{i0}). In our example these are rows 0, 1, and 2. (2) Copy under this array as a row the (x'_{j0}) column of (x'_{jk}). Then cross off those columns of the array corresponding to the zeros of (x'_{j0}), thus:

$$\begin{array}{ccccc}
0 & 12 & 3 & 4 & 567\\
\cancel{1} & 00 & \cancel{0} & \cancel{0} & 100\\
\cancel{0} & 01 & \cancel{0} & \cancel{0} & 001\\
\cancel{1} & 01 & \cancel{0} & \cancel{0} & 111\\
\hline
\cancel{0} & 11 & \cancel{0} & \cancel{0} & 111
\end{array}$$

† A systematic method for finding all solutions will be found in Sec. 13-7, Additional Topics.

(3) The final step is semisystematic and is concerned only with the remaining columns of the array, that is, 1, 2, 5, 6, and 7. The goal is to choose columns whose logical sum is all units, i.e., whose logical sum yields the three units of column (b_{i0}). For example, choose columns 2 and 5. Then one possibility for (y_{j0}) is a column with units in rows 2 and 5. However, the process of deriving columns for (y_{j0}) in this manner must proceed in the following order: First try to choose a single column of the array that is all units. Next try for pairs of columns of the array whose logical sum is all units; then for triplets of columns; etc. *Each such choice* determines a possible column for (y_{j0}) that has units only in rows corresponding to the columns of the array included in the choice. However, to ensure the lowest solutions the successive choices must adhere to the following rule: The present choice *must include* at least one column of the array *not* included in each of the previous choices and must *not include* a column of the array that *is* included in each of the previous choices. When this can no longer be done, all possible columns for (y_{i0}) have been found. Thus the possible columns are chosen so that each produces column (b_{i0}) in the multiplication but so that no (y_{i0}) implies any other. The process is repeated for (y_{i1}), (y_{i2}), etc.

For our illustration, to determine possible columns for (y_{j0}) first note that no single column of the array has all units. So try for pairs. First choose array columns 2 and 5. Take array columns 5 and 7 as the second choice, which differs appropriately from the first choice, array columns 2 and 5. No more pair choices exist, nor are there any triplet choices. (We need not look for quadruplet choices, since the array has only three rows.) Hence (y_{i0}) can be either

$$\begin{array}{cc} 0 & 0\\ 1 & 0\\ 2\rightarrow & 1\\ 3 & 0\\ 4 & 0\\ 5\rightarrow & 1\\ 6 & 0\\ 7 & 0 \end{array} \qquad \text{or} \qquad \begin{array}{cc} 0 & 0\\ 1 & 0\\ 2 & 0\\ 3 & 0\\ 4 & 0\\ 5\rightarrow & 1\\ 6 & 0\\ 7\rightarrow & 1 \end{array}$$

Similarly for our example we find

$$(y_{j1})\colon \begin{array}{ccc} 1 & 0 & 0\\ 0 & 0 & 0\\ 0 & 0 & 0\\ 0 & 1 & 0\\ \\ 0 & 0 & 1\\ 0 & 1 & 1\\ 0 & 0 & 0\\ 0 & 0 & 0 \end{array} \qquad (y_{j2})\colon \begin{array}{cccccc} 1 & 1 & 0 & 0 & 0 & 0\\ 0 & 0 & 0 & 0 & 0 & 0\\ 1 & 0 & 1 & 1 & 0 & 0\\ 0 & 0 & 0 & 1 & 0 & 1\\ \\ 0 & 0 & 1 & 0 & 1 & 0\\ 0 & 0 & 1 & 1 & 1 & 1\\ 0 & 0 & 0 & 0 & 0 & 0\\ 0 & 1 & 0 & 0 & 1 & 1 \end{array} \qquad (y_{j3})\colon \begin{array}{cc} 0 & 0\\ 0 & 0\\ 1 & 0\\ 0 & 0\\ \\ 0 & 0\\ 0 & 0\\ 0 & 0\\ 0 & 1 \end{array}$$

Hence, to determine all the (y_{jk}), we take all combinations of the two possibilities for column 0, the three for column 1, the six for column 2, and the two for column 3, to make altogether $2 \times 3 \times 6 \times 2 = 72$ lowest matrices (y_{jk}), that is,

$$\begin{pmatrix} 0110 \\ 0000 \\ 1011 \\ 0000 \\ \\ 0000 \\ 1000 \\ 0000 \\ 0000 \end{pmatrix} \quad \begin{pmatrix} 0110 \\ 0000 \\ 0011 \\ 0000 \\ \\ 0000 \\ 1000 \\ 0000 \\ 1000 \end{pmatrix} \quad \begin{pmatrix} 0010 \\ 0000 \\ 1011 \\ 0100 \\ \\ 0000 \\ 1100 \\ 0000 \\ 0000 \end{pmatrix} \quad \begin{pmatrix} 0010 \\ 0000 \\ 0011 \\ 0100 \\ \\ 0000 \\ 1100 \\ 0000 \\ 1000 \end{pmatrix}$$

and so forth.

EXERCISES

(*a*) Find all solutions of $\begin{pmatrix} 101 \\ 011 \end{pmatrix} \otimes (x_{jk}) = \begin{pmatrix} 10 \\ 11 \end{pmatrix}$.

(*b*) Find all solutions of $\begin{pmatrix} 1001 \\ 0101 \end{pmatrix} \otimes (x_{jk}) = \begin{pmatrix} 10 \\ 11 \end{pmatrix}$, and graph their implications as in Fig. 13-8.

(*c*) Find the least-upper-bound solution and all lowest solutions of

$$\begin{pmatrix} 1101 & 0000 \\ 0100 & 1011 \\ 1001 & 1000 \\ 0010 & 0111 \end{pmatrix} \otimes (x_{jk}) = \begin{pmatrix} 1011 \\ 1111 \\ 1101 \\ 1001 \end{pmatrix}$$

(*d*) Find the least-upper-bound solution and all lowest solutions of

$$(x_{kj}) \otimes \begin{pmatrix} 10 \\ 01 \\ 00 \\ 11 \end{pmatrix} = \begin{pmatrix} 11 \\ 01 \end{pmatrix}$$

13-5. Solution to the Unitary Matrix Equation

The Unitary Matrix Equation. The matrix equation

$$(a_{kj}) \otimes (U_{ji}) = (b_{ki}) \qquad (13\text{-}19)\blacklozenge$$

where (U_{ji}) is a matrix that has precisely 1 unit in each column,[†] is called a *unitary* (or pseudopermutation) matrix equation and is of particular importance for our purposes. The effect of multiplying a matrix on the right by (U_{ji}) is to permute the columns of that matrix; if element $U_{ji} = 1$, then column j will go into column i. For example, let

$$(U_{ji}) = \begin{pmatrix} 0001 \\ 0000 \\ 1000 \\ 0110 \end{pmatrix} \quad \text{then} \begin{pmatrix} 1100 \\ 1010 \\ 0011 \\ 1100 \end{pmatrix} \otimes \begin{pmatrix} 0001 \\ 0000 \\ 1000 \\ 0110 \end{pmatrix} = \begin{pmatrix} 0001 \\ 1001 \\ 1110 \\ 0001 \end{pmatrix}$$

[†] To avoid confusion, we shall not include in our discussion matrices having just 1 unit in each *row*. Such a matrix permutes the rows of another matrix which it multiplies *on the left*. Our work can be generalized to these matrices by applying the relation on tranposes of products (see Exercise *e*).

where column 0 went to column 3, column 2 to column 0, and column 3 to column 1 and also to column 2. It is clear that if a matrix is complemented and its columns permuted the result is the same as if its columns were permuted first and then complemented; thus

$$\overline{(a_{kj}) \otimes (U_{ji})} = (\overline{a_{kj}}) \otimes (U_{ji}) \qquad (13\text{-}20)\blacklozenge$$

In general there may arise three types of equations associated with Eq. (13-19) involving unitary matrices, corresponding (1) to (b_{ki}) unknown, (2) to (a_{kj}) unknown, and (3) to (U_{ji}) unknown except for its unitary nature.

Solutions to Type 1 *Equations.* For an equation of the first type a matrix (a_{kj}) and a unitary matrix (U_{ji}) are known, and a matrix (b_{ki}) is sought which satisfies Eq. (13-19). Here the solution is obvious: (b_{ki}) is obtained directly by multiplication. Obviously (b_{ki}) so obtained is unique. We note also that by applying Eq. (13-20) we obtain

$$(\overline{a_{kj}}) \otimes (U_{ji}) = (\overline{b_{ki}}) \qquad (13\text{-}21)$$

Solutions to Type 2 *Equations.* For an equation of the second type a matrix (b_{ki}) and a unitary matrix (U_{ji}) are known, and a matrix (a_{kj}) is sought which satisfies Eq. (13-19). This is an equation of the type illustrated by Eq. (13-7), and we obtain from Eq. (13-10) the least upper bound of solutions,

$$(a'_{kj}) = \overline{(\overline{b_{ki}}) \otimes (U_{ji})^T} \qquad (13\text{-}22)$$

Complementing and applying the relation that the transpose of a product of matrices is the product of the transposes of the factors taken in reverse order, we obtain

$$(\overline{a'_{jk}}) = (U_{ji}) \otimes (\overline{b_{ik}}) \qquad (13\text{-}23)\blacklozenge$$

where $(b_{ik}) = (b_{ki})^T$, etc. (In what follows we usually indicate a transpose by this method of transposing the indices.)

A consequence of Eq. (13-20) is that the set of solutions for type 2 equations, if any exist, *will have a greatest lower bound as well as a least upper bound.* We see that Eq. (13-21) must hold for every solution (a_{kj}) to Eq. (13-19). If we now consider Eq. (13-21) as an equation of the second type in $(\overline{a_{kj}})$ as the unknown, we can solve *for a least upper bound* for $(\overline{a_{kj}})$, namely,

$$(\overline{a^*_{kj}}) = \overline{(b_{ki}) \otimes (U_{ji})^T}$$

Complementing and transposing, we have

$$(a^*_{jk}) = (U_{ji}) \otimes (b_{ik}) \qquad (13\text{-}24)\blacklozenge$$

Since $(\overline{a^*_{jk}})$ is the least upper bound of $(\overline{a_{jk}})$, then (a^*_{jk}) is the greatest lower bound of (a_{jk}) and hence is *the unique lowest solution to the equation of type* 2. The totality of matrices (a_{jk}) such that

$$(a^*_{jk}) \rightarrow (a_{jk}) \rightarrow (a'_{jk}) \qquad (13\text{-}25)\blacklozenge$$

are all the solutions to Eq. (13-19), considered as a type 2 equation (see Exercise d below).

Solutions to Type 3 *Equations.* For an equation of the third type matrices (a_{kj}) and (b_{ki}) are known, and a *unitary* matrix (U_{ji}) is sought that will satisfy Eq. (13-19). This is an equation of the type illustrated by Eq. (13-5), and we obtain from Eq. (13-9) the least upper bound of solutions [such that $(U_{ji}) \to (x'_{ji})$],

$$(x'_{ji}) = \overline{(a_{jk}) \otimes (\overline{b_{ki}})} \qquad \text{(13-26)}\blacklozenge$$

From Eq. (13-15) we see that $(a_{kj}) \otimes (x'_{ji}) \to (b_{ki})$. If $(U_{ji}) \to (x'_{ji})$, it follows (see Sec. 11-11) that

$$(a_{kj}) \otimes (U_{ji}) \to (b_{ki}) \qquad \text{(13-27)}$$

Consider now Eq. (13-21) as an equation of type 3. The least upper bound to solutions to this equation [such that $(U_{ji}) \to (x^*_{ji})$] is

$$(x^*_{ji}) = \overline{(\overline{a_{jk}}) \otimes (b_{ki})} \qquad \text{(13-28)}\blacklozenge$$

Again from Eq. (13-15) it follows that, if $(U_{ji}) \to (x^*_{ji})$, then

$$(\overline{a_{kj}}) \otimes (U_{ji}) \to (\overline{b_{ki}}) \qquad \text{(13-29)}$$

Remembering that, if $P \to Q$, $\bar{Q} \to \bar{P}$, we obtain

$$(b_{ki}) \to \overline{(\overline{a_{kj}}) \otimes (U_{ji})}$$

If (U_{ji}) is unitary, Eq. (13-20) gives

$$(b_{ki}) \to (a_{kj}) \otimes (U_{ji}) \qquad \text{(13-30)}$$

Hence, by implications (13-27) and (13-30), every unitary matrix (U_{ji}) such that $(U_{ji}) \to (x'_{ji})$ and $(U_{ji}) \to (x^*_{ji})$, that is, such that

$$(U_{ji}) \to (x'_{ji}) \cdot (x^*_{ji}) \qquad \text{(13-31)}\blacklozenge$$

satisfies Eq. (13-19), namely, $(a_{kj}) \otimes (U_{ji}) = (b_{ki})$, as desired (recalling that $P \to Q$ and $Q \to P$ if and only if $P = Q$).

Conditions for Existence of Solutions. From the nature of the type 1 equations it is obvious that a unique solution always exists, which may be found from either Eq. (13-19) or Eq. (13-21), which are equivalent. In type 2 and 3 equations, however, no solutions need exist, and if a solution does exist, it need not be unique.

In the type 3 equations, even though there may be matrices that satisfy Eq. (13-19), none of these may be unitary; i.e., none may also satisfy Eq. (13-21). Recalling that the function of the unitary matrix (U_{ji}) in Eq. (13-19) is to permute the columns of (a_{kj}) into the columns of (b_{ki}), with perhaps omissions and duplications, it is obvious that a solution of an equation of type 3 will exist only if for every column of (b_{ki}) there is an identical column of (a_{kj}), though the reverse need not hold.

For example, in the equation

$$\begin{pmatrix} 0001 \\ 0011 \\ 0111 \\ 1111 \end{pmatrix} \otimes (U_{ji}) = \begin{pmatrix} 1000 \\ 1100 \\ 1110 \\ 1111 \end{pmatrix}$$

(U_{ji}) is obviously $\begin{pmatrix} 0001 \\ 0010 \\ 0100 \\ 1000 \end{pmatrix}$; but $\begin{pmatrix} 0001 \\ 0011 \\ 0110 \\ 1110 \end{pmatrix} \otimes (U_{ji}) = \begin{pmatrix} 1000 \\ 1100 \\ 1110 \\ 1111 \end{pmatrix}$ has no unitary solution, even though $(U_{ji}) = \begin{pmatrix} 0001 \\ 1010 \\ 0100 \\ 1000 \end{pmatrix}$ (among others) will satisfy it.

In the type 2 equations, if (U_{ji}) is a true permutation matrix, having 1 unit in each column *and* 1 *unit in each row*, then we see immediately that Eq. (13-20) can be applied to Eq. (13-22) to give

$$(a'_{kj}) = \overline{\overline{(b_{ki})}} \otimes (U_{ji})^T = (b_{ki}) \otimes (U_{ji})^T$$

or $$(a'_{jk}) = (U_{ji}) \otimes (b_{ik})$$

Comparing this with Eq. (13-24), we see that implication (13-25) reduces to an equality. Thus we see that, if (U_{ji}) is a true permutation matrix, then Eq. (13-19) has a unique solution for $(a_{jk}) = (a_{kj})^T$, given by Eq. (13-24). For example, for the equation

$$(a_{kj}) \otimes \begin{pmatrix} 0100 \\ 0001 \\ 1000 \\ 0010 \end{pmatrix} = \begin{pmatrix} 1000 \\ 1100 \\ 1110 \\ 1111 \end{pmatrix}$$

$$(a_{jk}) = (a'_{jk}) = (a^*_{jk}) = \begin{pmatrix} 0100 \\ 0001 \\ 1000 \\ 0010 \end{pmatrix} \otimes \begin{pmatrix} 1111 \\ 0111 \\ 0011 \\ 0001 \end{pmatrix} = \begin{pmatrix} 0111 \\ 0001 \\ 1111 \\ 0011 \end{pmatrix} \quad \text{or} \quad (a_{kj}) = \begin{pmatrix} 0010 \\ 1010 \\ 1011 \\ 1111 \end{pmatrix}$$

If (U_{ji}) is not a true permutation matrix, then, as we have defined it, it will have some columns which are duplicates of others. In any product $(a_{kj}) \otimes (U_{ji}) = (b_{ki})$ the columns of (b_{ki}) will show the same duplications as those of (U_{ji}). An equation of type 2 involving such a (U_{ji}) can have no solution unless this condition on the duplication of the columns of (b_{ki}) holds. It is obvious that, since (U_{ji}) merely rearranges the columns of (a_{kj}) in forming (b_{ki}), a solution (a_{kj}) can always be formed by rearranging the columns of (b_{ki}), provided only that (b_{ki}) has the same duplications of columns as does (U_{ji}). Further, if the jth row of (U_{ji}) is all zeros, then the jth column of (a_{kj}) never appears in (b_{ki}) so that this column of (a_{kj}) is entirely arbitrary in the solution of a type 2 equation. For example, the equation

$$(a_{kj}) \otimes \begin{pmatrix} 0001 \\ 1010 \\ 0000 \\ 0100 \end{pmatrix} = \begin{pmatrix} 0000 \\ 0001 \\ 0101 \\ 1111 \end{pmatrix}$$

has a solution,

$$(a_{kj}) = \begin{pmatrix} 00\Phi0 \\ 10\Phi0 \\ 10\Phi1 \\ 11\Phi1 \end{pmatrix}$$

(where the Φ symbols indicate that the third column is arbitrary), while the equation

$$(a_{kj}) \otimes \begin{pmatrix} 0001 \\ 1010 \\ 0000 \\ 0100 \end{pmatrix} = \begin{pmatrix} 0000 \\ 0001 \\ 1001 \\ 1111 \end{pmatrix}$$

has no solution.

EXERCISES

Find all solutions for the following equations:

(*a*) $(x_{kj}) \otimes \begin{pmatrix} 0100 \\ 1001 \\ 0000 \\ 0010 \end{pmatrix} = \begin{pmatrix} 1011 \\ 1001 \\ 1101 \end{pmatrix}.$

Solution

$$(x'_{kj}) = \overline{\begin{pmatrix} 0100 \\ 0110 \\ 0010 \end{pmatrix} \otimes \begin{pmatrix} 0100 \\ 1000 \\ 0001 \\ 0100 \end{pmatrix}} = \overline{\begin{pmatrix} 1000 \\ 1001 \\ 0001 \end{pmatrix}} = \begin{pmatrix} 0111 \\ 0110 \\ 1110 \end{pmatrix}$$

$$(x^*_{kj}) = \begin{pmatrix} 1011 \\ 1001 \\ 1101 \end{pmatrix} \otimes \begin{pmatrix} 0100 \\ 1000 \\ 0001 \\ 0100 \end{pmatrix} = \begin{pmatrix} 0101 \\ 0100 \\ 1100 \end{pmatrix}$$

Hence the solutions are

$$\begin{tikzcd}
 & & \begin{pmatrix} 0111 \\ 0100 \\ 1100 \end{pmatrix} \arrow[r] \arrow[dr] & \begin{pmatrix} 0111 \\ 0110 \\ 1100 \end{pmatrix} \arrow[dr] & & \\
\begin{array}{l}\text{Greatest}\\\text{lower}\\\text{bound}\end{array} & \begin{pmatrix} 0101 \\ 0100 \\ 1100 \end{pmatrix} \arrow[ur] \arrow[r] \arrow[dr] & \begin{pmatrix} 0101 \\ 0110 \\ 1100 \end{pmatrix} \arrow[ur] \arrow[dr] & \begin{pmatrix} 0111 \\ 0100 \\ 1110 \end{pmatrix} \arrow[r] & \begin{pmatrix} 0111 \\ 0110 \\ 1110 \end{pmatrix} & \begin{array}{l}\text{Least}\\\text{upper}\\\text{bound}\end{array} \\
 & & \begin{pmatrix} 0101 \\ 0100 \\ 1110 \end{pmatrix} \arrow[ur] \arrow[r] & \begin{pmatrix} 0101 \\ 0110 \\ 1110 \end{pmatrix} \arrow[ur] & &
\end{tikzcd}$$

(*b*) $\begin{pmatrix} 1011 \\ 0100 \end{pmatrix} \otimes (x_{jk}) = \begin{pmatrix} 10 \\ 11 \end{pmatrix}.$

(*c*) $(x_{kj}) \otimes \begin{pmatrix} 0100 \\ 1011 \end{pmatrix} = \begin{pmatrix} 1011 \\ 1100 \end{pmatrix}.$

(*d*) $(x_{kj}) \otimes \begin{pmatrix} 0100 \\ 0001 \\ 0000 \\ 1010 \end{pmatrix} = \begin{pmatrix} 0001 \\ 0001 \\ 0101 \\ 1011 \end{pmatrix}.$

(*e*) Derive equations corresponding to the numbered equations of this section for the matrix equation $(V_{kj}) \otimes (a_{ji}) = (b_{ki})$, where (V_{kj}) is a matrix having just 1 unit in each *row* (cf. footnote on page 434).

13-6. Derivation of the Fundamental Matrix Equation and the Fundamental Formulas

The Pseudopermutation Transformation. The derivations of the fundamental formulas are based on the antecedence formula for problems of type 1, which is

$$(F_{kj}) \otimes (R_{ji}) = (E_{ki}) \qquad (13\text{-}32)\blacklozenge$$

For this reason Eq. (13-32) is called the *fundamental matrix equation* relating Boolean matrices with digital-circuit design. To see how this formula arises, let us first consider F to be of the form $F = F(f_1,f_2,f_3)$, where we have omitted the $X_1, \ldots, X_K$ of the more general formulation. In this case the formula for type 1 reduced to $(F_{0j}) \otimes (R_{ji}) = (E_{0i})$, where (F_{0j}) and (E_{0i}) are matrices with only one row. More specifically they are the designation numbers themselves; that is, $(F_{0j}) = \#F$, and $(E_{0i}) = \#E$. Then we want to show that $\#F \otimes (R_{ji}) = \#E$.

This equation follows clearly from two simple observations. The first observation is merely a restatement of the meaning of a designation number with respect to the standard basis. Whenever the signal values (that is, 1 or 0) in the input wires to a circuit correspond to column p of the basis, then the signal value (1 or 0) in the output wire is the value of the pth position of the designation number of the output wire. For example, if $F = \bar{f}_1 \cdot \bar{f}_2 + f_1 \cdot f_3$, then we have with respect to $b[f_1,f_2,f_3]$

j	0	1	2	3	4	5	6	7
$\#f_1 =$	0	1	0	1	0	1	0	1
$\#f_2 =$	0	0	1	1	0	0	1	1
$\#f_3 =$	0	0	0	0	1	1	1	1
$\#F =$	1	1	0	0	0	1	0	1

Then if the input values correspond to column zero of the basis, i.e., $\begin{matrix}0\\0\\0\end{matrix}$, then the output value on wire F will be a unit, and so forth.

The second observation is essentially the same, except that we are considering more than one output wire, i.e., more than one circuit with the same inputs. For example, let $f_1 = A_2 + \bar{A}_3$, $f_2 = (A_1 + A_2) \cdot \bar{A}_3$, and $f_3 = A_1 \cdot A_2 + (A_1 + A_2) \cdot A_3$. Then we have with respect to $b[A_1,A_2,A_3]$ the array for f_1, f_2, f_3 shown in Fig. 13-9. In this case, for example, if the input values of A_1, A_2, A_3 correspond to column 4 of $b[A_1,A_2,A_3]$, i.e., to $\begin{matrix}0\\0\\1\end{matrix}$, then the set of output values of f_1, f_2, and f_3 will be found in position 4 of each of $\#f_1$, $\#f_2$, and $\#f_3$, namely, $\begin{matrix}0\\0\\0\end{matrix}$. In other words, column 4 of $b[A_1,A_2,A_3]$ corresponds to column 0 of $b[f_1,f_2,f_3]$. To find the value of the fourth position of the $\#F$ with respect to $b[A_1, A_2,A_3]$, observe that column 4 of $b[A_1,A_2,A_3]$ corresponds to column 0 of $b[f_1,f_2,f_3]$, which in our example corresponds to a unit in the zeroth position of $\#F$ with respect to $b[f_1,f_2,f_3]$. Hence we conclude that the value of the fourth position of $\#F$ with respect to $b[A_1,A_2,A_3]$ (that is, the fourth position of $\#E$) is a unit.

Summarizing, $\#E$ (that is, $\#F$ with respect to $b[A_1,A_2,A_3]$) can be found from $\#F$ (with respect to $b[f_1,f_2,f_3]$) by *rearranging the positions* of $\#F$. For our example we have found above that the value of position 0 now becomes the value of position 4 in the rearrangement. Similarly the rest of the positions are rearranged as in Fig. 13-10. Observe that such a rearrangement can be made with a unitary or pseudopermutation matrix [which we call (R_{ji})], having just 1 unit in each column (see Secs. 11-10 and 13-5). For our example we have

i	0	1	2	3	4	5	6	7	
$\#A_1$	0	1	0	1	0	1	0	1	
$\#A_2$	0	0	1	1	0	0	1	1	
$\#A_3$	0	0	0	0	1	1	1	1	
$\#f_1$	1	1	1	1	0	0	1	1	
$\#f_2$	0	1	1	1	0	0	0	0	
$\#f_3$	0	0	0	1	0	1	1	1	
$\#F$	-	-	-	-	1	-	-	-	$=\#E$

FIG. 13-9. Array for f_1, f_2, and f_3.

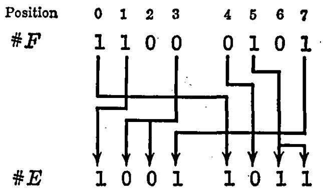


FIG. 13-10. Rearrangement of the positions of $\#F$.

$$\#F \otimes (R_{ji}) = (1100 \quad 0101) \otimes \begin{pmatrix} 0000 & 1000 \\ 1000 & 0000 \\ 0000 & 0000 \\ 0110 & 0000 \\ & \\ 0000 & 0100 \\ 0000 & 0011 \\ 0000 & 0000 \\ 0001 & 0000 \end{pmatrix} = (1001 \quad 1011) = \#E$$

Such an R matrix can obviously be found for any other rearrangement which might be required and depends only on the form of the functions $f_1(A_1,A_2,A_3, \ldots)$, $f_2(A_1,A_2,A_3, \ldots)$, etc.

The Fundamental Equation. In the more general case, where

$$F = F(f_1, \ldots ,f_J,X_1, \ldots ,X_K)$$

only the variables $f_1, \ldots , f_J$ are affected. Observe that in a standard basis $b[f_1, \ldots ,f_J,X_1, \ldots ,X_K]$ the first J rows consist of the standard basis $b[f_1, \ldots ,f_J]$ repeated 2^K times. Thus we can subdivide the basis $b[f_1, \ldots ,f_J,X_1, \ldots ,X_K]$, and consequently $\#F_1$, into 2^K groups. A rearrangement of the columns of $b[f_1, \ldots ,f_J]$ to correspond to an array of the designation numbers of the $f_1, \ldots , f_J$, such as is shown in Fig. 13-9, has the effect of rearranging the bits of $\#F$ only within each of these groups. Furthermore, the bits with every group undergo the *same* rearrangement. We recall that a unitary matrix such as (R_{ji}) has the effect of rearranging the columns of a matrix which it multiplies on the right; i.e., multiplying by (R_{ji}) rearranges the bits within each row of a matrix. Thus we form a matrix, called (F_{kj}), by folding #*F* so that its group becomes individual rows, multiply by the proper (R_{ji}) to rearrange the bits within the rows, and unfold the resulting matrix to form a new number. This number is the required #*E*, since it represents the necessary rearrangement of #*F*; thus Eq. (13-32) is justified (see page 420).

The Fundamental Formulas. The fundamental equation is of course a unitary equation, and hence all the results of Sec. 13-5 apply. Substituting the matrices (F_{kj}), (R_{ji}), and (E_{ki}) for matrices (a_{kj}), (U_{ji}), and (b_{ki}), respectively, in Eqs. (13-19), (13-21), (13-23), (13-24), (13-26), and (13-28), we find the six fundamental formulas of Eq. (13-4). Comparison of Secs. 13-3 and 13-5 will show that the solutions to Eq. (13-32) are the antecedence solutions, and solutions to $(\bar{F}_{kj}) \otimes (R_{ji}) = (\bar{E}_{ki})$ [see, for example, Eq. (13-21)] are consequence solutions. In the fundamental formulas, Eq. (13-4), we have used the subscripts a and c to distinguish between the determinations of matrices corresponding to antecedence and consequence solutions, which were denoted by the prime (′) and asterisk (*) in Sec. 13-5.

EXERCISES

(*a*) Describe the meaning of Eq. (13-25) in terms of the type 2 circuit-design problem.

(*b*) Describe the meaning of Eq. (13-31) in terms of the type 3 circuit-design problem.

(*c*) Using the methods of Secs. 13-5 and 13-6, prove the assertions made in Sec. 13-3 under Properties of the Solutions.

13-7. Additional Topics†

a. Operations and Properties of Boolean Matrices. Besides the operation of Boolean matrix multiplication defined in Sec. 11-11, we utilize in these chapters two matrix relationships (equality and implication), four matrix operations (*and, or, not,* and transposition), and three matrix properties (associativity, product transposition, and invariance of implication under multiplication):

Equality. $(A_{ij}) = (B_{ij})$ if $A_{ij} = B_{ij}$ in every position. That is, (A_{ij}) and (B_{ij}) have units and zeros in exactly the same positions.

Implication. $(A_{ij}) \rightarrow (B_{ij})$ if $A_{ij} \rightarrow B_{ij}$ in every position. That is, (B_{ij}) has units in at least every position in which (A_{ij}) has units, for example, $\begin{pmatrix}00\\01\end{pmatrix} \rightarrow \begin{pmatrix}11\\01\end{pmatrix}$. As with the designation numbers $(A_{ij}) \rightarrow (B_{ij})$ and $(B_{ij}) \rightarrow (A_{ij})$ only if $(A_{ij}) = (B_{ij})$.

And (logical multiplication). $(A_{ij}) \cdot (B_{ij}) = (C_{ij})$ means that $C_{ij} = A_{ij} \cdot B_{ij}$ for every position. That is, the logical product has units only where (A_{ij}) and (B_{ij}) both have units; for example, $\begin{pmatrix}01\\11\end{pmatrix} \cdot \begin{pmatrix}11\\10\end{pmatrix} = \begin{pmatrix}01\\10\end{pmatrix}$.

† The material of Additional Topic *b* was developed by William Ruthven Smith and James Bruce Wilson, and that of Additional Topics *c, d,* and *e* by William Ruthven Smith, when they were graduate students of the author's at the George Washington University (see W. R. Smith, Master's thesis, June, 1959).

Or (logical addition). $(A_{ij}) + (B_{ij}) = (C_{ij})$ means that $C_{ij} = A_{ij} + B_{ij}$ for every position. That is, the logical sum has units wherever either (A_{ij}) or (B_{ij}) or both have units; for example, $\begin{pmatrix}01\\01\end{pmatrix} + \begin{pmatrix}01\\10\end{pmatrix} = \begin{pmatrix}01\\11\end{pmatrix}$.

Not (complementation). $(B_{ij}) = \overline{(A_{ij})}$ means $B_{ij} = \bar{A}_{ij}$ for every position. That is, $\overline{(A_{ij})}$ has zeros where (A_{ij}) has units, units where (A_{ij}) has zeros, for example, $\overline{\begin{pmatrix}01\\00\end{pmatrix}} = \begin{pmatrix}10\\11\end{pmatrix}$. Because of the definition we may write $\overline{(A_{ij})} = (\bar{A}_{ij})$ (that is, the complement of the matrix equals the matrix of the complements).

Transposition (denoted here by a superscript T). $(A_{ij})^T = (B_{ij})$ means $B_{ij} = A_{ji}$ for each value of i and j. That is, the transpose of a matrix is formed by interchanging its rows with its columns, for example, $\begin{pmatrix}11\\01\\10\end{pmatrix}^T = \begin{pmatrix}101\\110\end{pmatrix}$.

Notice that equality, implication, and logical multiplication and addition are defined only between matrices of the same dimensions. Complementation of a matrix results in a matrix of the same dimension, while transposition results in a matrix whose width is the former depth and whose depth is the former width.

The following three properties are easily verified from the definitions of the operations involved:

Associativity

$$((A_{ij}) \cdot (B_{ij})) \cdot (C_{ij}) = (A_{ij}) \cdot ((B_{ij}) \cdot (C_{ij})) = (A_{ij}) \cdot (B_{ij}) \cdot (C_{ij})$$
$$((A_{ij}) + (B_{ij})) + (C_{ij}) = (A_{ij}) + ((B_{ij}) + (C_{ij})) = (A_{ij}) + (B_{ij}) + (C_{ij})$$
$$((A_{hi}) \otimes (B_{ij})) \otimes (C_{jk}) = (A_{hi}) \otimes ((B_{ij}) \otimes (C_{jk})) = (A_{hi}) \otimes (B_{ij}) \otimes (C_{jk})$$

Product transposition

$$((A_{hi}) \otimes (B_{ij}) \otimes (C_{jk}))^T = (C_{jk})^T \otimes (B_{ij})^T \otimes (A_{hi})^T$$

That is, the transpose of the product of matrices is equal to the product of the transposes of the factor matrices, taken in reverse order.

Invariance of implication under multiplication. If

$$(A_{ij}) \rightarrow (C_{ij})$$

and
$$(B_{jk}) \rightarrow (D_{jk})$$

then
$$(A_{ij}) \otimes (B_{jk}) \rightarrow (C_{ij}) \otimes (D_{jk})$$

Obviously the implication of the conclusion holds if either implication of the hypotheses is equality.

References

Hohn, F. E., and R. L. Schissler: Boolean Matrices and the Design of Combinational Relay Switching Circuits, *Bell System Tech. J.*, vol. 34, pp. 201–202, 1955.

Luce, R. D.: Note on Boolean Matrix Theory, *Proc. Am. Math. Soc.*, vol. 3, pp. 382–388, 1952.

b. Generation of E and F Matrices by Direct Matrix Multiplication. Consider an E function of the form

$$E(A_1, \ldots ,A_I,X_1, \ldots ,X_K) = P(X_1 \ldots ,X_K) \cdot Q(A_1, \ldots ,A_I)$$

It is clear from the manner in which (E_{ki}) is formed that in such a case

$$(E_{ki}) = (\#P)^T \otimes (\#Q)$$

For example, if $P = X_1 + \bar{X}_2$ and $Q = \bar{A}_1 \cdot A_2 + A_1 \cdot \bar{A}_2$, then

$$(1101)^T \otimes (0110) = \begin{pmatrix} 0110 \\ 0110 \\ 0000 \\ 0110 \end{pmatrix} = (E_{ki})$$

In fact, if $E = P_1Q_1 + P_2Q_2 + \cdots$, where $P_m = P_m(X_1, \ldots, X_K)$ and $Q_m = Q_m(A_1, \ldots, A_I)$, then it follows that

$$(E_{ki}) = \begin{pmatrix} \#P_1 \\ \#P_2 \\ \cdots \end{pmatrix}^T \otimes \begin{pmatrix} \#Q_1 \\ \#Q_2 \\ \cdots \end{pmatrix}$$

For example, if $E = A_1 \cdot A_2 + X_1 \cdot \bar{A}_2 \cdot A_3 + X_2 \cdot (A_1 \cdot \bar{A}_3 + A_2)$, then

$$(E_{ki}) = \begin{pmatrix} \#I \\ \#X_1 \\ \#X_2 \end{pmatrix}^T \otimes \begin{pmatrix} \#A_1 \cdot A_2 \\ \#\bar{A}_2 \cdot A_3 \\ \#(A_1 \cdot \bar{A}_3 + A_2) \end{pmatrix} = \begin{pmatrix} 1111 \\ 0101 \\ 0011 \end{pmatrix}^T \otimes \begin{pmatrix} 0001 & 0001 \\ 0000 & 1100 \\ 0111 & 0011 \end{pmatrix}$$

$$= \begin{pmatrix} 0001 & 0001 \\ 0001 & 1101 \\ 0111 & 0011 \\ 0111 & 1111 \end{pmatrix}$$

as in Sec. 13-2. (Note that the numbers for the functions are formed from $b[X_1, \ldots, X_K]$ and $b[A_1, \ldots, A_I]$, respectively.)

We can generalize these results. If $E = W[S(A,X), T(A,X), \ldots]$, then $(E_{ki}) = W[(S_{ki}), (T_{ki}), \ldots]$. That is, if the function E is expressed as a function of subfunctions $S(A,X)$, $T(A,X)$, etc., then the matrices for the subfunctions can be formed as above and these combined according to the operations of the main function to form the matrix for E. For example, consider

$$E = \overline{(A_1 + X_1 \cdot \bar{X}_2) \cdot (A_2 \cdot (\bar{X}_1 + X_2))}$$

We have

$$(E_{ki}) = \overline{\{[(\#I)^T \otimes (\#A_1)] + [(\#X_1\bar{X}_2)^T \otimes (\#I)]\} \cdot \{(\#(\bar{X}_1 + X_2))^T \otimes (\#A_2)\}}$$

On a different grouping of the terms, we could have equivalently

$$(E_{ki}) = \overline{\left[\begin{pmatrix} \#I \\ \#X_1 \cdot X_2 \end{pmatrix}^T \otimes \begin{pmatrix} \#A_1 \\ \#I \end{pmatrix}\right] \cdot \left[(\#(\bar{X}_1 + X_2))^T \otimes (\#A_2)\right]}$$

and so forth.

These results of course also hold for the generation of the F matrix. For example, consider $F = \bar{f}_1 \cdot f_3 + X_1 \cdot f_1 \cdot \bar{f}_2 + X_2 \cdot f_2 \cdot \bar{f}_3$. Here

$$(F_{kj}) = \begin{pmatrix} \#I \\ \#X_1 \\ \#X_2 \end{pmatrix}^T \otimes \begin{pmatrix} \#\bar{f}_1 \cdot f_3 \\ \#f_1 \cdot \bar{f}_2 \\ \#f_2 \cdot \bar{f}_3 \end{pmatrix} = \begin{pmatrix} 1111 \\ 0101 \\ 0011 \end{pmatrix}^T \otimes \begin{pmatrix} 0000 & 1010 \\ 0100 & 0100 \\ 0011 & 0000 \end{pmatrix}$$

$$= \begin{pmatrix} 0000 & 1010 \\ 0100 & 1110 \\ 0011 & 1010 \\ 0111 & 1110 \end{pmatrix}$$

as in Sec. 13-2.

c. Finding (R_{ji}) Directly in Type 3 *Problems.* From Eq. (13-31) we observe that $(R_{ji}) = (R_{ji})_a \cdot (R_{ji})_c$ for problems of type 3. However, the same (R_{ji}) can be obtained directly from (F_{kj}) and (E_{ki}) provided that we introduce a new kind of matrix-multiplication operation, called the *theta* product, defined as follows: $(S_{ji}) = (P_{jk}) \ominus (Q_{ki})$ if and only if for each position

$$\begin{aligned} S_{ji} &= \prod_k (P_{jk} = Q_{ki}) \\ &= \prod_k (P_{jk} \cdot Q_{ki} + \bar{P}_{jk} \cdot \bar{Q}_{ki}) \end{aligned}$$

For observe that

$$(R_{ji}) = (R_{ji})_a \cdot (R_{ji})_c = \overline{(F_{kj})^T \otimes (\bar{E}_{ki})} \cdot \overline{(\bar{F}_{kj})^T \otimes (E_{ki})}$$

or

$$R_{ji} = \overline{\sum_k F_{jk} \cdot \bar{E}_{ki}} \cdot \overline{\sum_k \bar{F}_{jk} \cdot E_{ki}}$$

which by De Morgan's rules becomes

$$\begin{aligned} R_{ji} &= \prod_k (\bar{F}_{jk} + E_{ki}) \cdot \prod_k (F_{jk} + \bar{E}_{ki}) \\ &= \prod_k (\bar{F}_{jk} + E_{ki}) \cdot (F_{jk} + \bar{E}_{ki}) \\ &= \prod_k (F_{jk} \cdot E_{ki} + \bar{F}_{jk} \cdot \bar{E}_{ki}) \\ &= \prod_k (F_{jk} = E_{ki}) \end{aligned}$$

Thus in terms of our theta matrix product

$$(R_{ji}) = (F_{kj})^T \ominus (E_{ki})$$

which is the desired result.

d. Finding (R_{ji}) for a Set of Given Functions $f_1, \ldots, f_j$ by Theta Matrix Multiplication. From the concept of type 1 problems observe that

$$\#f_1(b[f_1, \ldots ,f_J]) \otimes (R_{ji}) = \#f_1(A_1, \ldots ,A_I)$$

and likewise for $f_2, \ldots, f_J$. Hence

$$(b[f_1, \ldots ,f_J]) \otimes (R_{ji}) = \begin{pmatrix} \#f_1(A_1, \ldots ,A_I) \\ \#f_2(A_1, \ldots ,A_I) \\ \ldots\ldots\ldots\ldots \\ \#f_J(A_1, \ldots ,A_I) \end{pmatrix}$$

Now if $b[f_1, \ldots ,f_J]$ and $\#f_1(A_1, \ldots ,A_I), \ldots, \#f_J(A_1, \ldots ,A_I)$ are known, finding (R_{ji}) is a type 3 problem. Hence

$$(R_{ji}) = (b[f_1, \ldots ,f_J])^T \ominus \begin{pmatrix} \#f_1(A_1, \ldots ,A_I) \\ \#f_2(A_1, \ldots ,A_I) \\ \ldots\ldots\ldots\ldots \\ \#f_J(A_1, \ldots ,A_I) \end{pmatrix}$$

For example, suppose that

$$\begin{aligned} f_1 &= A_1 \cdot \bar{A}_2 + \bar{A}_1 \cdot A_3 \\ f_2 &= \bar{A}_1 \cdot A_2 + A_1 \cdot \bar{A}_3 \\ f_3 &= A_1 \cdot A_2 + \bar{A}_2 \cdot A_3 \end{aligned}$$

Then with respect to $b[A_1,A_2,A_3]$ we have

$$\#f_1 = 0100 \quad 1110 \qquad \#f_2 = 0111 \quad 0010 \qquad \text{and} \qquad \#f_3 = 0001 \quad 1101$$

Substituting into our formula,

$$(R_{ji}) = \begin{pmatrix}0101 & 0101\\ 0011 & 0011\\ 0000 & 1111\end{pmatrix}^T \Theta \begin{pmatrix}0100 & 1110\\ 0111 & 0010\\ 0001 & 1101\end{pmatrix}$$

$$= \begin{pmatrix}000\\ 100\\ 010\\ 110\\ \\ 001\\ 101\\ 011\\ 111\end{pmatrix} \Theta \begin{pmatrix}0100 & 1110\\ 0111 & 0010\\ 0001 & 1101\end{pmatrix} = \begin{pmatrix}1000 & 0000\\ 0000 & 0000\\ 0010 & 0000\\ 0100 & 0010\\ & \\ 0000 & 0001\\ 0000 & 1100\\ 0001 & 0000\\ 0000 & 0000\end{pmatrix}$$

as in Sec. 13-2.

e. All Solutions to the Boolean Matrix Equation. Given the matrix equation

$$(a_{ik}) \otimes (x_{kj}) = (b_{ij})$$

—where $i = 0, 1, \ldots, I$; $j = 0, 1, \ldots, J$; and $k = 0, 1, \ldots, K$—first form

$$(S_{mj}) = \{(a_{ik}) \otimes (b[X_0,X_1, \ldots ,X_K])\}^T \Theta (b_{ij})$$

When (S_{mj}) is interpreted as an R-type matrix, all solutions to the matrix equation can be found. For example, consider

$$\begin{pmatrix}101\\ 011\end{pmatrix} \otimes (x_{kj}) = \begin{pmatrix}10\\ 11\end{pmatrix}$$

Here $k = 0, 1, 2$, and hence $(b[X_0,X_1,X_2]) = \begin{pmatrix}0101 & 0101\\ 0011 & 0011\\ 0000 & 1111\end{pmatrix}$, that is, just the basis for $K + 1 = 3$ variables. Thus we have from our formula

$$(S_{mj}) = \left\{\begin{pmatrix}101\\ 011\end{pmatrix} \otimes \begin{pmatrix}0101 & 0101\\ 0011 & 0011\\ 0000 & 1111\end{pmatrix}\right\}^T \Theta \begin{pmatrix}10\\ 11\end{pmatrix}$$

$$= \begin{pmatrix}0101 & 1111\\ 0011 & 1111\end{pmatrix}^T \Theta \begin{pmatrix}10\\ 11\end{pmatrix}$$

$$= \begin{pmatrix}00\\ 10\\ 01\\ 11\\ \\ 11\\ 11\\ 11\\ 11\end{pmatrix} \Theta \begin{pmatrix}10\\ 11\end{pmatrix} = \begin{array}{cc} & j\ 01 \\ \begin{matrix}m\ 0\\ 1\\ 2\\ 3\\ \\ 4\\ 5\\ 6\\ 7\end{matrix} & \begin{pmatrix}00\\ 00\\ 01\\ 10\\ \\ 10\\ 10\\ 10\\ 10\end{pmatrix}\end{array}$$

From (S_{mj}) we write a result array for (x_{kj}), using the basis columns indicated by m:

	$j = 0$	$j = 1$
$k = 0$	1, 0, 1, 0, 1	0
$k = 1$	1, 0, 0, 1, 1	1
$k = 2$	0, 1, 1, 1, 1	0

This gives rise to five matrices,

$$\begin{pmatrix}10\\11\\00\end{pmatrix} \quad \begin{pmatrix}00\\01\\10\end{pmatrix} \quad \begin{pmatrix}10\\01\\10\end{pmatrix} \quad \begin{pmatrix}00\\11\\10\end{pmatrix} \quad \text{and} \quad \begin{pmatrix}10\\11\\10\end{pmatrix}$$

Note that in general $m = 0, 1, \ldots, 2^K - 1$.

To see why the formula holds, observe that the matrix equation is really equivalent to $J + 1$ sets each of $I + 1$ equations. The equations for $j = 0$ are, for example,

$$\begin{aligned} a_{00} \cdot x_{00} + a_{01} \cdot x_{10} + \cdots + a_{0K} \cdot x_{K0} &= b_{00} \\ a_{10} \cdot x_{00} + a_{11} \cdot x_{10} + \cdots + a_{1K} \cdot x_{K0} &= b_{10} \\ \cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots \\ a_{I0} \cdot x_{00} + a_{I1} \cdot x_{10} + \cdots + a_{IK} \cdot x_{K0} &= b_{I0} \end{aligned}$$

Now note that the columns of the basis $b[X_0, \ldots, X_K]$ are all possible combinations of values that the $x_{00}, x_{10}, \ldots, x_{K0}$ could possibly take. Thus

$$(a_{00}a_{01} \cdots a_{0K}) \otimes (b[x_0, \ldots, x_K]) = (r_{00}r_{01} \cdots r_{0m})$$

gives the values r_{0k} which the left-hand side of the zeroth equation has when the $x_{00}, x_{10}, \ldots, x_{K0}$ take on the combination of values given by the kth column of the basis. Similarly the values of the left-hand sides of all the equations for each column of the basis can be determined from

$$(a_{ik}) \otimes (b[x_0, \ldots, x_K]) = (R_{im})$$

Consider the matrix equation

$$(a_{ik}) \otimes (x_{k0}) = (b_{i0})$$

(i.e., just the first set of equations written out explicitly above). A column vector (x_{k0}) that satisfies this equation is simply that basis column corresponding to a column of (R_{im}) identical to the column (b_{i0}). Another way of putting this is to form

$$(S_{m0}) = (R_{im})^T \ominus (b_{i0})$$

and then interpret the row number of a unit in (S_{m0}) as the position of the basis column that satisfies the equation and hence is (x_{k0}). Clearly this argument can be extended to include all the columns of (b_{ij}), and our formula is proved.

If the Boolean matrix equation is

$$(x_{ik}) \otimes (a_{kj}) = (b_{ij})$$

then $$(S_{mi}) = \{(b[X_1, \ldots, X_K])^T \otimes (a_{kj})\} \ominus (b_{ij})^T$$

where the direct interpretation of (S_{mi}) gives $(x_{ki}) = (x_{ik})^T$. For example, let

$$(x_{ik}) \otimes \begin{pmatrix}010\\010\end{pmatrix} = \begin{pmatrix}000\\000\\010\end{pmatrix}$$

Then

$$(S_{mi}) = \left\{\begin{pmatrix}0101\\0011\end{pmatrix}^T \otimes \begin{pmatrix}010\\010\end{pmatrix}\right\} \ominus \begin{pmatrix}000\\000\\010\end{pmatrix}^T$$

$$= \left\{\begin{pmatrix}00\\10\\01\\11\end{pmatrix} \otimes \begin{pmatrix}010\\010\end{pmatrix}\right\} \ominus \begin{pmatrix}000\\001\\000\end{pmatrix}$$

$$= \begin{pmatrix}000\\010\\010\\010\end{pmatrix} \ominus \begin{pmatrix}000\\001\\000\end{pmatrix} = \begin{array}{cc} & i\ 012 \\ m\ \begin{matrix}0\\1\\2\\3\end{matrix} & \begin{pmatrix}110\\001\\001\\001\end{pmatrix}\end{array}$$

Thus $(x_{ki}) = \begin{pmatrix}001\\000\end{pmatrix}, \begin{pmatrix}000\\001\end{pmatrix}, \begin{pmatrix}001\\001\end{pmatrix}$, or the solutions are

$$(x_{ik}) = \begin{pmatrix}00\\00\\10\end{pmatrix} \quad \begin{pmatrix}00\\00\\01\end{pmatrix} \quad \begin{pmatrix}00\\00\\11\end{pmatrix}$$

CHAPTER 14

APPLICATIONS OF MATRIX EQUATIONS IN CIRCUIT DESIGN

14-1. Type 1 Problems in Circuitry

Introduction. In the previous chapter we have described a general problem in the logical design of digital circuitry and have shown how to solve it by means of Boolean matrices. In this present chapter we give examples of the applications of that problem under a variety of circumstances.

Substitution. We shall consider first the solution to the type 1 problem of Sec. 13-1, with $f_1 = A_1 \cdot \bar{A}_2 + \bar{A}_1 \cdot A_3$, $f_2 = \bar{A}_1 \cdot A_2 + A_1 \cdot \bar{A}_3$, and $f_3 = A_1 \cdot A_2 + \bar{A}_2 \cdot A_3$ and $F = \bar{f}_1 \cdot f_3 + X_1 \cdot f_1 \cdot \bar{f}_2 + X_2 \cdot f_2 \cdot \bar{f}_3$. If we did not know about our computational method, we would substitute for f_1, f_2, and f_3 in F, obtaining E,

$$\begin{aligned}\overline{(A_1 \cdot \bar{A}_2 + A_1 \cdot A_3)} \cdot (A_1 \cdot A_2 + \bar{A}_2 \cdot A_3) \\ + X_1 \cdot (A_1 \cdot \bar{A}_2 + \bar{A}_1 \cdot A_3) \cdot \overline{(\bar{A}_1 \cdot A_2 + A_1 \cdot \bar{A}_3)} \\ + X_2 \cdot (\bar{A}_1 \cdot A_2 + A_1 \cdot \bar{A}_3) \cdot \overline{(A_1 \cdot A_2 + \bar{A}_2 \cdot A_3)}\end{aligned}$$

and try to simplify this result. Or, better yet, we might find the designation number of E with respect to $b[A_1,A_2,A_3,X_1,X_2]$ and from this find a simple form. However, we can find the designation number of F directly without performing the substitution, by using the formula for problems of type 1. To illustrate the process, we first find (R_{ji}) corresponding to f_1, f_2, and f_3. This was done in Sec. 13-2 for the given f_1, f_2, and f_3:

$$(R_{ji}) = \begin{pmatrix} 1000 & 0000 \\ 0000 & 0000 \\ 0010 & 0000 \\ 0100 & 0010 \\ \\ 0000 & 0001 \\ 0000 & 1100 \\ 0001 & 0000 \\ 0000 & 0000 \end{pmatrix}$$

The matrix (F_{kj}) for our F was also found in Sec. 13-2 and is

$$(F_{kj}) = \begin{pmatrix} 0000 & 1010 \\ 0100 & 1110 \\ 0011 & 1010 \\ 0111 & 1110 \end{pmatrix}$$

whence, to find (E_{ki}), we use the formula for type 1 problems,

$$(F_{kj}) \otimes (R_{ji}) = \begin{pmatrix} 0000 & 1010 \\ 0100 & 1110 \\ 0011 & 1010 \\ 0111 & 1110 \end{pmatrix} \otimes \begin{pmatrix} 1000 & 0000 \\ 0000 & 0000 \\ 0010 & 0000 \\ 0100 & 0010 \\ \\ 0000 & 0001 \\ 0000 & 1100 \\ 0001 & 0000 \\ 0000 & 0000 \end{pmatrix}$$

$$= \begin{pmatrix} 0001 & 0001 \\ 0001 & 1101 \\ 0111 & 0011 \\ 0111 & 1111 \end{pmatrix} = (E_{ki})$$

From this matrix,

$$\#E = 0001 \quad 0001 \quad 0001 \quad 1101 \quad 0111 \quad 0011 \quad 0111 \quad 1111$$

which in Sec. 13-2 was seen to correspond to

$$E = A_1 \cdot A_2 + X_1 \cdot \bar{A}_2 \cdot A_3 + X_2 \cdot (A_1 \cdot \bar{A}_3 + A_2)$$

Often, when it is desired to check E in problems of types 2 and 3, the matrices (R_{ji}) and (F_{kj}) already have been obtained. The formula for type 1 problems then becomes a valuable tool, as will be illustrated in subsequent sections.

EXERCISES

(*a*) Let

$$\begin{aligned} F(f_1,f_2,f_3,X_1,X_2) &= (f_1 \cdot \bar{X}_2 + \bar{f}_2 \cdot X_1) \cdot (f_3 \cdot \bar{X}_1 + \bar{f}_2 \cdot \bar{X}_2) \\ f_1 &= (A_1 + A_2) \cdot A_3 \\ f_2 &= (A_1 \cdot \bar{A}_2 \cdot A_3 + \bar{A}_1 \cdot A_2 \cdot \bar{A}_3) \\ f_3 &= (A_2 \cdot \bar{A}_3 + A_1 \cdot \bar{A}_2) \end{aligned}$$

Find $E = F(A_1,A_2,A_3,X_1,X_2)$.

(*b*) Let

$$\begin{aligned} F &= f_1 \cdot \bar{f}_2 + f_2 \cdot \bar{f}_3 + f_3 \cdot \bar{f}_1 \\ f_1 &= A_1 \cdot \bar{A}_2 + A_2 \cdot \bar{A}_3 \\ f_2 &= A_2 \cdot \bar{A}_3 + A_3 \cdot \bar{A}_1 \\ f_3 &= A_3 \cdot \bar{A}_1 + A_1 \cdot \bar{A}_2 \end{aligned}$$

Find $E = F(A_1,A_2,A_3,X_1,X_2)$.

14-2. Type 2 Problems in Circuitry

Constraints. The determination of logical dependence or constraints between given elements has been mentioned as a special case of the determination of consequence solutions to a type 2 problem. If $f_1(A_1, \ldots ,A_I)$, $f_2(A_1, \ldots ,A_I)$, . . . , $f_J(A_1, \ldots ,A_I)$ are J given functions, the constraint between them is a function $C(f_1,f_2, \ldots ,f_n)$ such that with respect to $b[A_1, \ldots ,A_I]$ we have

$$C[f_1(A_1, \ldots ,A_I),f_2(A_1, \ldots ,A_I), \ldots ,f_J(A_1, \ldots ,A_I)] = I$$

Observe the elementary fact that if $I \rightarrow W(A_1, \ldots ,A_I)$ then $I = W$. Hence we can write the C as a constraint if

$$I \rightarrow C[f_1(A_1, \ldots ,A_I),f_2(A_1, \ldots ,A_I), \ldots ,f_J(A_1, \ldots ,A_I)]$$

Thus *the constraint C is found as a consequence solution of type 2 in the special case where E is I;* it is therefore determined by

$$(R_{ji}) \otimes (I_{ik}) = (C_{jk})$$

where (C_{jk}) is used instead of (F_{jk}) to emphasize the special nature of the formula. For example, suppose that

$$\begin{aligned} f_1 &= A_1 \cdot \bar{A}_2 + \bar{A}_1 \cdot A_3 \\ f_2 &= \bar{A}_1 \cdot A_2 + A_1 \cdot \bar{A}_3 \\ f_3 &= A_1 \cdot A_2 + \bar{A}_2 \cdot A_3 \end{aligned}$$

To determine the constraint between f_1, f_2, and f_3, first form (R_{ji}), which in this case has been determined in Sec. 13-2. The range of i and j is given by the number of elements A_i and functions f_j, but the range of k depends on the particular problem under consideration. In our case let $k = 0$ only, whence we have

$$(R_{ji}) \otimes (I_{ik}) = \begin{pmatrix} 1000 & 0000 \\ 0000 & 0000 \\ 0010 & 0000 \\ 0100 & 0010 \\ 0000 & 0001 \\ 0000 & 1100 \\ 0001 & 0000 \\ 0000 & 0000 \end{pmatrix} \otimes \begin{pmatrix} 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \end{pmatrix} = \begin{pmatrix} 1 \\ 0 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 0 \end{pmatrix} = (C_{jk})$$

Therefore $\# C = 1011\ 1110$, and $C = (\bar{f}_1 + f_2) \cdot \bar{f}_3 + (\bar{f}_1 + \bar{f}_2) \cdot f_3$. This method becomes especially useful in problems where only $\# C$ is desired and where (R_{ji}) may have been determined previously.

Circuit-design Example. We now proceed with general problems of type 2, which often make good use of the special case just described.

Suppose three circuits with outputs, respectively, $A_1 \cdot \bar{A}_2 + \bar{A}_1 \cdot A_3$,

$\bar{A}_1 \cdot A_2 + A_1 \cdot \bar{A}_3$, and $A_1 \cdot A_2 + \bar{A}_2 \cdot A_3$ have already been constructed. Suppose that it is desired to construct a new circuit with output

$$A_1 \cdot A_2 + X_1 \cdot \bar{A}_2 \cdot A_3 + X_2 \cdot (A_1 \cdot \bar{A}_3 + A_2)$$

The problem is to construct the new circuit whose inputs are X_1 and X_2 and the outputs of each of the three circuits already constructed. This problem is essentially one of finding the form $F(f_1,f_2,f_3,X_1,X_2)$, where $f_1 = A_1 \cdot \bar{A}_2 + \bar{A}_1 \cdot A_3$, $f_2 = \bar{A}_1 \cdot A_2 + A_1 \cdot \bar{A}_3$, $f_3 = A_1 \cdot A_2 + \bar{A}_2 \cdot A_3$; and $E(A_1,A_2,A_3,X_1,X_2) = A_1 \cdot A_2 + X_1 \cdot \bar{A}_2 \cdot A_3 + X_2 \cdot (A_1 \cdot \bar{A}_3 + A_2)$ are given. For antecedence solutions we have, by the formula for problems of type 2,

$$(R_{ji}) \otimes (\bar{E}_{ik}) = (\bar{F}_{jk})_a$$

or, from the examples of Sec. 13-2,

$$(\bar{F}_{jk})_a = \begin{pmatrix} 1000 & 0000 \\ 0000 & 0000 \\ 0010 & 0000 \\ 0100 & 0010 \\ & \\ 0000 & 0001 \\ 0000 & 1100 \\ 0001 & 0000 \\ 0000 & 0000 \end{pmatrix} \otimes \overline{\begin{pmatrix} 0000 \\ 0011 \\ 0011 \\ 1111 \\ \\ 0101 \\ 0101 \\ 0011 \\ 1111 \end{pmatrix}} = \begin{pmatrix} 1111 \\ 0000 \\ 1100 \\ 1100 \\ \\ 0000 \\ 1010 \\ 0000 \\ 0000 \end{pmatrix}$$

[where we recall that (E_{ik}) is the same as (E_{ki}) with the rows and columns interchanged]. Hence

$$(F_{kj})_a = \begin{pmatrix} 0100 & 1011 \\ 0100 & 1111 \\ 0111 & 1011 \\ 0111 & 1111 \end{pmatrix}$$

or

$$\#F_a = 0100 \quad 1011 \quad 0100 \quad 1111 \qquad 0111 \quad 1011 \quad 0111 \quad 1111$$

We could now find F with respect to $b[f_1,f_2,f_3,X_1,X_2]$, except that doing this would imply that f_1, f_2, f_3, X_1, and X_2 are all independent, i.e., that no constraint exists between them. However, this is not necessarily true: most often f_1, f_2, and f_3 are *not* independent. This constraint with respect to $b[f_1,f_2,f_3,X_1,X_2]$ is determined by

$$(R_{ji}) \otimes (I_{ik}) = \begin{pmatrix} 1000 & 0000 \\ 0000 & 0000 \\ 0010 & 0000 \\ 0100 & 0010 \\ & \\ 0000 & 0001 \\ 0000 & 1100 \\ 0001 & 0000 \\ 0000 & 0000 \end{pmatrix} \otimes \begin{pmatrix} 1111 \\ 1111 \\ 1111 \\ 1111 \\ \\ 1111 \\ 1111 \\ 1111 \\ 1111 \end{pmatrix} = \begin{pmatrix} 1111 \\ 0000 \\ 1111 \\ 1111 \\ \\ 1111 \\ 1111 \\ 1111 \\ 0000 \end{pmatrix} = (C_{jk})$$

Hence

$$\#C = 1011 \quad 1110 \quad 1011 \quad 1110 \qquad 1011 \quad 1110 \quad 1011 \quad 1110$$

Thus we must find F_a with respect to the constrained basis $b_c[f_1,f_2,f_3,X_1,X_2]$,

$$\begin{array}{rl}
\#f_1 = & 001 \quad 010 \quad 001 \quad 010 \qquad 001 \quad 010 \quad 001 \quad 010 \\
\#f_2 = & 011 \quad 001 \quad 011 \quad 001 \qquad 011 \quad 001 \quad 011 \quad 001 \\
\#f_3 = & 000 \quad 111 \quad 000 \quad 111 \qquad 000 \quad 111 \quad 000 \quad 111 \\
\#X_1 = & 000 \quad 000 \quad 111 \quad 111 \qquad 000 \quad 000 \quad 111 \quad 111 \\
\#X_2 = & 000 \quad 000 \quad 000 \quad 000 \qquad 111 \quad 111 \quad 111 \quad 111 \\
\hline
\#F_a = & 000 \quad 101 \quad 000 \quad 111 \qquad 011 \quad 101 \quad 011 \quad 111
\end{array}$$

This gives $\qquad F_a = \bar{f}_1 \cdot f_3 + X_1 \cdot f_1 \cdot \bar{f}_2 + X_2 \cdot f_2 \cdot \bar{f}_3$

However, we are not yet finished, for all we now have is an F_a such that $F_a \rightarrow E$. To see whether or not $F_a(f_1,f_2,f_3,X_1,X_2)$ is E, we must substitute $f_1(A_1,A_2,A_3)$, $f_2(A_1,A_2,A_3)$, and $f_3(A_1,A_2,A_3)$ into F_a and see whether or not $\#F_a = \#E$ with respect to $b[A_1,A_2,A_3,X_1,X_2]$. Hence we form

$$(F_{kj}) \otimes (R_{ji}) = \begin{pmatrix} 0100 & 1011 \\ 0100 & 1111 \\ 0111 & 1011 \\ 0111 & 1111 \end{pmatrix} \otimes \begin{pmatrix} 1000 & 0000 \\ 0000 & 0000 \\ 0010 & 0000 \\ 0100 & 0010 \\ 0000 & 0001 \\ 0000 & 1100 \\ 0001 & 0000 \\ 0000 & 0000 \end{pmatrix} = \begin{pmatrix} 0001 & 0001 \\ 0001 & 1101 \\ 0111 & 0011 \\ 0111 & 1111 \end{pmatrix} = (F_{ki})_a$$

Hence $$(F_{ki})_a = \begin{pmatrix} 0001 & 0001 \\ 0001 & 1101 \\ 0111 & 0011 \\ 0111 & 1111 \end{pmatrix} = (E_{ki})$$

as desired. In this example F_a is truly the desired solution, that is, $F_a = E$.

Further Discussion. Whether $F_a = E$ or not is determined in most cases by means of the type 1 formulas. Consider the circumstances where $F_a \neq E$. In such a case, since $F_a \rightarrow E$, we can always find by inspection a function $W(A_1, \ldots ,A_I,X_1, \ldots ,X_K)$ such that $F + W = E$. Similarly if $F_c \neq E$, then since $E \rightarrow F_c$, we can always find by inspection a function $V(A_1, \ldots ,A_I,X_1, \ldots ,X_K)$ such that $F_c \cdot V = E$. The problem in each of these cases reduces to choosing from all possible functions W (or V) the one that best suits the particular situation in question. In any case, before determining F explicitly, any constraint that exists between the f_s must be calculated by means of the formula given at the beginning of this section; then F is determined with respect to the constrained basis $b_c[f_1, \ldots ,f_J,X_1, \ldots ,X_K]$.

EXERCISES

Draw the logical circuit diagrams for all results.

(*a*) Suppose that two circuits have already been constructed with outputs $A_1 \cdot A_2 + \bar{A}_1 \cdot \bar{A}_2$ and $A_1 \cdot \bar{A}_2 + \bar{A}_1 \cdot A_2$. The problem is to connect these circuits so that the following output will result:

$$E = A_1 \cdot A_2 \cdot (X_1 + X_2) + \bar{A}_1 \cdot (\bar{A}_2 \cdot X_1 + A_2 \cdot \bar{X}_1) + A_1 \cdot \bar{A}_2 \cdot \bar{X}_1 \cdot \bar{X}_2$$

Solution. $\#F_a(A_1,A_2,X_1,X_2) = 0110\ 1001\ 0000\ 1001$, while $\#E(A_1,A_2,X_1,X_2) = 0110\ 1001\ 0011\ 1001$, whence $F_a \neq E$. A function W can be found so that $F_a + W = E$. One W evidently is such that $\#W = 0000\ 0000\ 0011\ 0000$. To determine F_a explicitly as $F(f_1,f_2,X_1,X_2)$, find the constraints between f_1 and f_2, and then with respect to the constrained basis $b_c[f_1,f_2,X_1,X_2]$ we have

$$F_a + W = \underbrace{f_1 \cdot X_1 + f_2 \cdot \bar{X}_1 \cdot \bar{X}_2}_{F_a} + \underbrace{A_2 \cdot \bar{X}_1 \cdot X_2}_{W} = E$$

(*b*) Consider Exercise *a*, but this time find the consequence solution F_c.

Solution. $\#F_c(A_1,A_2,X_1,X_2) = 0110\ 1001\ 1111\ 1001$, whence $F_c \neq E$. An appropriate V is $V = A_2 + X_1 + \bar{X}_2$ (that is, $\#V = 1111\ 1111\ 0011\ 1111$), whence $F \cdot V = (f_1 \cdot (X_1 + X_2) + f_2 \cdot \bar{X}_1) \cdot (A_2 + X_1 + \bar{X}_2) = E$.

14-3. Type 3 Problems in Circuitry

Five Examples. In this section five examples will be given. The first example illustrates a normally complicated situation; the second considers the case when solutions *cannot* be obtained such that $F = E$; the third illustrates the situation in which many solutions turn up and additional criteria are required; the fourth considers the case when *no solutions exist;* the fifth illustrates how solutions may be obtained that involve only certain specified elementary elements.

The First Example. Suppose that a circuit F has been constructed with inputs f_1, f_2, X_1, and X_2 such that its output Boolean function is given by

$$F = \bar{f}_1 \cdot \bar{f}_2 \cdot \bar{X}_2 + f_1 \cdot \bar{f}_2 \cdot X_2 + \bar{f}_1 \cdot f_2(X_1 \cdot \bar{X}_2 + \bar{X}_1 \cdot X_2) + f_1 \cdot f_2 \cdot \bar{X}_1 \cdot \bar{X}_2$$

It is desired to produce, by attaching additional circuits to the f_1 and f_2 inputs of this F circuit, the following output result:

$$\begin{aligned} E = (\bar{A}_1 \cdot \bar{A}_3 + \bar{A}_2 \cdot A_3) \cdot \bar{X}_1 \cdot \bar{X}_2 + (\bar{A}_1 \cdot \bar{A}_3 + A_1 \cdot A_2) \cdot X_1 \cdot \bar{X}_2 \\ + (A_1 \cdot \bar{A}_3 + A_2 \cdot A_3) \cdot \bar{X}_1 \cdot X_2 \\ + (A_1 \cdot \bar{A}_2 \cdot \bar{A}_3 + \bar{A}_1 \cdot A_2 \cdot A_3) \cdot X_1 \cdot X_2 \end{aligned}$$

The problem is to design the two circuits, with inputs A_1, A_2, and A_3, whose outputs should be connected to f_1 and f_2, respectively. The first step in the solution is to find the sets of antecedence solutions $f_1(A_1,A_2,A_3)$, $f_2(A_1,A_2,A_3)$. Then $F_a(f_1(A_1,A_2,A_3),f_2(A_1,A_2,A_3),X_1,X_2) \rightarrow E$. The next step is to find the sets of consequence solutions; then $E \rightarrow F_c$. Now, for those sets of solutions that are *both* antecedence and consequence solutions, $F = E$, and hence these will be the desired solutions from which the circuits can be designed.

With respect to $b[A_1,A_2,A_3,X_1,X_2]$,

$$\begin{array}{rcccccccc}
\#E = & 1010 & 1100 & 1011 & 0001 & 0101 & 0011 & 0100 & 0010 \\
i & 0123 & 4567 & 0123 & 4567 & 0123 & 4567 & 0123 & 4567 \\
k & \multicolumn{2}{c}{0} & \multicolumn{2}{c}{1} & \multicolumn{2}{c}{2} & \multicolumn{2}{c}{3}
\end{array}$$

whence

$$(E_{ki}) = \begin{pmatrix} 1010 & 1100 \\ 1011 & 0001 \\ 0101 & 0011 \\ 0100 & 0010 \end{pmatrix}$$

Next, with respect to $b[f_1,f_2,X_1,X_2]$,

$$\begin{array}{rcccc}
\#F = & 1001 & 1010 & 0110 & 0100 \\
j & 0123 & 0123 & 0123 & 0123 \\
k & 0 & 1 & 2 & 3
\end{array}$$

whence

$$(F_{jk}) = \begin{pmatrix} 1100 \\ 0011 \\ 0110 \\ 1000 \end{pmatrix}$$

For antecedence solutions

$$(F_{jk}) \otimes (\bar{E}_{ki}) = \begin{pmatrix} 1100 \\ 0011 \\ 0110 \\ 1000 \end{pmatrix} \otimes \begin{pmatrix} 0101 & 0011 \\ 0100 & 1110 \\ 1010 & 1100 \\ 1011 & 1101 \end{pmatrix} = \begin{pmatrix} 0101 & 1111 \\ 1011 & 1101 \\ 1110 & 1110 \\ 0101 & 0011 \end{pmatrix} = (\bar{R}_{ji})_a$$

and therefore

$$(R_{ji})_a = \begin{array}{c} \begin{array}{cc} 0123 & 4567 \end{array} \\ \begin{pmatrix} 1010 & 0000 \\ 0100 & 0010 \\ 0001 & 0001 \\ 1010 & 1100 \end{pmatrix} \end{array}$$

Hence

RESULT ARRAY

i	0	1	2	3	4	5	6	7
$\#f_1$ $\#f_2$	0, 1 0, 1	1 0	0, 1 0, 1	0 1	1 1	1 1	1 0	0 1

There are four antecedence solutions:

$f_1 = A_1 \cdot \bar{A}_2 + \bar{A}_1 \cdot A_3$ $\qquad f_2 = A_1 \cdot A_2 + \bar{A}_2 \cdot A_3$
$f_1 = A_1 \cdot \bar{A}_2 + \bar{A}_1 \cdot A_3 + \bar{A}_1 \cdot A_2$ $\qquad f_2 = A_1 \cdot A_2 + \bar{A}_2 \cdot A_3 + A_2 \cdot \bar{A}_3$
$f_1 = \bar{A}_2 + \bar{A}_1 \cdot A_3$ $\qquad f_2 = A_1 \cdot A_2 + \bar{A}_2 \cdot A_3 + \bar{A}_1 \cdot \bar{A}_2$
$f_1 = \bar{A}_1 + \bar{A}_2$ $\qquad f_2 = A_1 \cdot A_2 + \bar{A}_2 \cdot A_3 + \bar{A}_1 \cdot \bar{A}_2 + A_2 \cdot \bar{A}_3$

For the consequence solutions we have

$$(\bar{F}_{jk}) \otimes (E_{ki}) = \begin{pmatrix} 0011 \\ 1100 \\ 1001 \\ 0111 \end{pmatrix} \otimes \begin{pmatrix} 1010 & 1100 \\ 1011 & 0001 \\ 0101 & 0011 \\ 0100 & 0010 \end{pmatrix} = \begin{pmatrix} 0101 & 0011 \\ 1011 & 1101 \\ 1110 & 1110 \\ 1111 & 0011 \end{pmatrix} = (\bar{R}_{ji})_c$$

$$(R_{ji})_c = \begin{pmatrix} \textcircled{1}\,0\,\textcircled{1}\,0 & 1\;1\;0\;0 \\ 0\,\textcircled{1}\,0\;0 & 0\;0\,\textcircled{1}\,0 \\ 0\;0\;0\,\textcircled{1} & 0\;0\;0\,\textcircled{1} \\ 0\;0\;0\;0 & \textcircled{1}\textcircled{1}\,0\;0 \end{pmatrix}$$

In order to obtain those solutions which are both consequence and antecedence solutions, we encircle those units which are common with the antecedence $(R_{ji})_a$ matrix. This solution is the common one:

RESULT ARRAY

i	0	1	2	3	4	5	6	7
$\#f_1$	0	1	0	0	1	1	1	0
$\#f_2$	0	0	0	1	1	1	0	1

or

$$f_1 = A_1 \cdot \bar{A}_2 + \bar{A}_1 \cdot A_3$$
$$f_2 = A_1 \cdot A_2 + \bar{A}_2 \cdot A_3$$

The desired circuits are therefore as in Fig. 14-1.

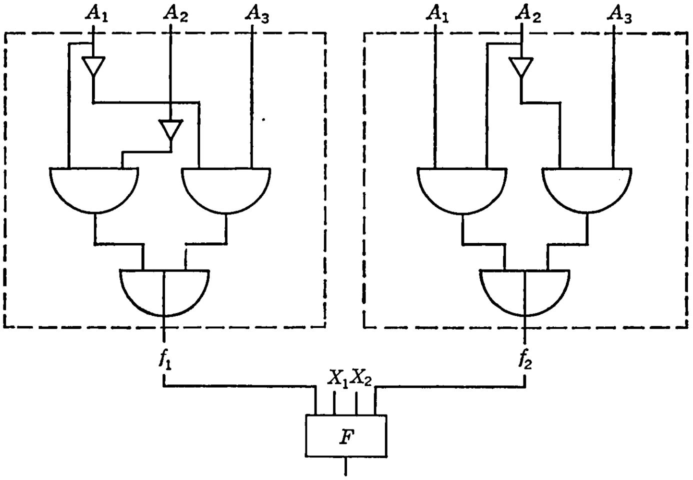


FIG. 14-1. Solution to the first example.

The Second Example. Suppose that E were the same as in the previous example, but that the circuit F had been constructed slightly different, as follows:

$$F = \bar{f}_1 \cdot \bar{f}_2 \cdot (\bar{X}_1 + \bar{X}_2) + f_1 \cdot \bar{f}_2 \cdot X_2 + \bar{f}_1 \cdot f_2 \cdot (X_1 \cdot \bar{X}_2 + \bar{X}_1 \cdot X_2) \\ + f_1 \cdot f_2 \cdot \bar{X}_1 \cdot \bar{X}_2$$

We are to construct an f_1 and an f_2, as in the previous example. The antecedence solutions would be

$$(R_{ji})_a = \begin{pmatrix} 0000 & 0000 \\ 0100 & 0010 \\ 0001 & 0001 \\ 1010 & 1100 \end{pmatrix}$$

And for consequence solutions we would have

$$(R_{ji})_c = \begin{pmatrix} 1011 & 1101 \\ 0100 & 0010 \\ 0001 & 0001 \\ 0000 & 1100 \end{pmatrix}$$

Comparing the units of the antecedence and consequence R matrices, we see that since there are no common units in columns 0 and 2 there are no solutions that are both antecedence and consequence solutions. This means that no set of input functions $f_1(A_1,A_2,A_3)$ and $f_2(A_1,A_2,A_3)$ will make the output of $F = E$. We can only have $F_a \rightarrow E$ for antecedence sets of solutions and $E \rightarrow F_c$ for consequence sets of solutions. However, we can always find a W such that, if $F_a \rightarrow E$, then $F_a + W = E$, or some V such that, if $E \rightarrow F_c$, then $F_c \cdot V = E$.

Since there is only one set of antecedence solutions, let us consider this case first. By the formula for type 1 problems we find $\#F_a$ with respect to $b[A_1,A_2,A_3,X_1,X_2]$:

$$\#F_a = 1010 \quad 1100 \quad \overset{8\ \ 10}{0001} \quad 0001 \quad 0101 \quad 0011 \quad 0100 \quad 0010$$

We compare this with $\#E$:

$$\#E = 1010 \quad 1100 \quad 1011 \quad 0001 \quad 0101 \quad 0011 \quad 0100 \quad 0010$$

We note first that $\#F_a$ does indeed imply $\#E$, as expected, and that $\#F_a$ differs from $\#E$ in that $\#F_a$ is missing 2 units, in positions 8 and 10. In order that $F_a + W = E$, W must be chosen so as to contribute these units. We could solve for W, but it can be easily obtained by inspection. Let $W = \bar{A}_1 \cdot \bar{A}_3 \cdot \bar{X}_2$,

$$1010 \quad 0000 \quad 1010 \quad 0000 \quad 0000 \quad 0000 \quad 0000 \quad 0000$$

Now $\#F_a + \#W = \#E$, as is easily seen. Hence our final circuits will be as in Fig. 14-2.

Similarly, we can choose one of the nontrivial consequence solutions f_1, f_2 and determine some V such that $F \cdot V = E$. [Note from the consequence $(R_{ji})_c$ that there are 16 possible solutions, i.e., four columns have 2 units each.] Suppose that it is desired to choose a nontrivial consequence solution and determine the circuits for f_1, f_2, and V. For exam-

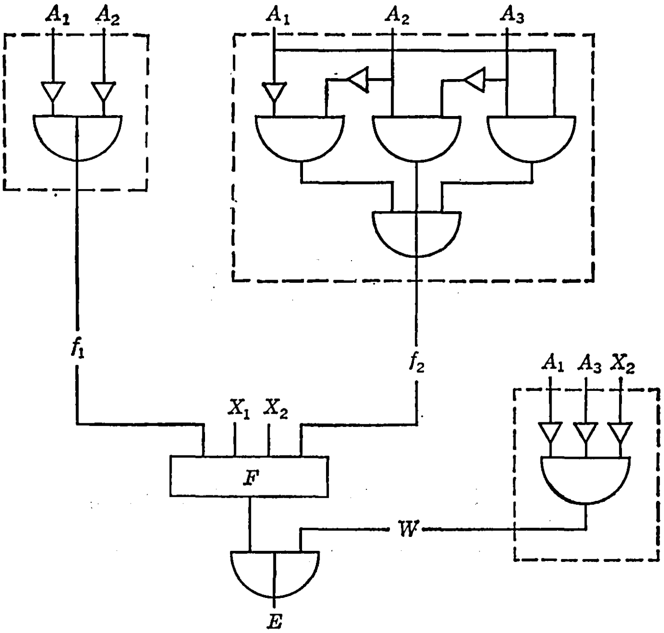


FIG. 14-2. Solution to the second example.

ple, choose the nontrivial solution corresponding to

$$(R_{ji}) = \begin{pmatrix} 1011 & 0001 \\ 0100 & 0010 \\ 0000 & 0000 \\ 0000 & 1100 \end{pmatrix}$$

This gives

$$\#F(A_1,A_2,A_3,X_1,X_2)$$
$$= 101\overset{3}{1} \quad 110\overset{7}{1} \quad 1011 \quad 0001 \quad \overset{16}{1}1\overset{18}{1}1 \quad 0011 \quad 0100 \quad 0010$$

On comparing this with $\#E$ we see that V must change the units in positions 3, 7, 16, and 18 to zeros. Hence we can have

$$V = (\bar{A}_1 + \bar{A}_2 + X_1 + X_2) \cdot (A_1 + A_3 + X_1 + \bar{X}_2)$$

by inspection.

Too Many Solutions. It sometimes happens that there are many antecedence or consequence solutions. This merely means that additional criteria are needed to choose the desired solution. Such criteria are often that the desired solutions be both antecedence and consequence solutions (so that $E = F$), that the set of functions of the solution be as simple as possible, that the set be independent, that the set satisfy some given constraints, etc. These additional criteria will quickly reduce the number of allowable solutions. This is actually an exceedingly interesting point, for here the mathematics is essentially disclosing whether or not the original formulation of the problem was adequately precise.

In order to illustrate these points, let us first solve the type 3 problem that was given in Sec. 13-1. [We determined (E_{ki}) and (F_{jk}) in Sec. 13-2.] To find the antecedence solutions using the formula for type 3 problems, find

$$(R_{ji})_a = \begin{array}{r} \\ j\ 0 \\ 1 \\ 2 \\ 3 \\ \\ 4 \\ 5 \\ 6 \\ 7 \end{array} \begin{array}{c} i\ 0123 \quad 4567 \\ \left(\begin{array}{cc} 1111 & 1111 \\ 0001 & 1101 \\ 0111 & 0011 \\ 0111 & 0011 \\ & \\ 0001 & 0001 \\ 0001 & 1101 \\ 0001 & 0001 \\ 1111 & 1111 \end{array}\right) \end{array}$$

Hence there are $2 \times 4 \times 4 \times 8 \times 4 \times 4 \times 4 \times 8 = 131{,}072$ sets of solutions.

This seems to be a large number of possible solutions, but after all it was narrowed down from the $8^8 = 16{,}777{,}216$ total possible sets of three functions of three variables! However, we are interested in solutions such that $F = E$, and our antecedence solutions assure us only that $F_a \rightarrow E$. Therefore let us find the consequence solutions such that $E \rightarrow F_c$, and then any solutions that turn out to be *both* antecedence and consequence solutions will satisfy $F = E$. For consequence solutions we have, upon substitution in the formula for type 3 problems,

$$(R_{ji})_c = \left(\begin{array}{cccc cccc} \textcircled{1} & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & \textcircled{1} & \textcircled{1} & 0 & 0 \\ 1 & \textcircled{1} & \textcircled{1} & 0 & 0 & 0 & \textcircled{1} & 0 \\ 1 & \textcircled{1} & \textcircled{1} & 0 & 0 & 0 & \textcircled{1} & 0 \\ &&&&&&& \\ 1 & 1 & 1 & \textcircled{1} & 1 & 1 & 1 & \textcircled{1} \\ 1 & 0 & 0 & 0 & \textcircled{1} & \textcircled{1} & 0 & 0 \\ 1 & 1 & 1 & \textcircled{1} & 1 & 1 & 1 & \textcircled{1} \\ \textcircled{1} & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{array}\right)$$

Comparing the antecedence $(R_{ji})_a$ with this consequence $(R_{ji})_c$, we note that they have in common only the units circled above; hence only solutions corresponding to these units will be both antecedence and consequence solutions as desired. Filling in for the common units only, we find the result array:

i	0	1	2	3	4	5	6	7
$\#f_1(A_1,A_2,A_3)$	0, 1	0, 1	0, 1	0, 0	1, 1	1, 1	0, 1	0, 0
$\#f_2(A_1,A_2,A_3)$	0, 1	1, 1	1, 1	0, 1	0, 0	0, 0	1, 1	0, 1
$\#f_3(A_1,A_2,A_3)$	0, 1	0, 0	0, 0	1, 1	0, 1	0, 1	0, 0	1, 1

Thus the number of desired solutions has been reduced from 131,072 to $2 \times 2 \times 2 \times 2 \times 2 \times 2 \times 2 \times 2 = 256$ solutions, but there are still too many. Hence we can add another criterion to our desired solution, e.g., that it be as simple as possible. In passing, note that one set of valid solutions is that given in Sec. 13-1 [see the discussion of matrix (R_{ji}) in Sec. 13-2].

To choose a simple solution from the 256 possible solutions, let us see whether or not any set of solutions has an elementary element as one of the functions. It is clear that f_2 cannot be expressed as an elementary element or its bar, for in $\#f_2(A_1,A_2,A_3)$ positions 1 and 2 must both be units and therefore f_2 cannot be A_1, $\bar{A}_1$, A_2, or $\bar{A}_2$; and since position 5 must be a zero and 6 must be a unit, f_2 cannot be A_3 or $\bar{A}_3$. Similar reasoning shows also that f_3 cannot be so expressed. However, note that f_1 can be chosen to be an elementary element, namely, $\bar{A}_2$. Throwing away solutions not involving $f_1 = \bar{A}_2$, we have the result array:

i	0	1	2	3	4	5	6	7	
$\#f_1(A_1,A_2,A_3)$	1	1	0	0	1	1	0	0	$= \#\bar{A}_2$
$\#f_2(A_1,A_2,A_3)$	1	1	1	0, 1	0	0	1	0, 1	
$\#f_3(A_1,A_2,A_3)$	1	0	0	1	0, 1	0, 1	0	1	

But this array still gives a choice of $2 \times 2 \times 2 \times 2 = 16$ sets of solutions, because of columns 3, 4, 5, and 7. Let us try to choose f_2 as simple as possible. From the elimination pairs of A_1 it is clear that f_2 can be chosen not to involve A_1. Then

$$\#f_2(A_1,A_2,A_3) = 1111 \quad 0011 = \#[A_2 + \bar{A}_3]$$

Note also that f_3 can be chosen so as to eliminate A_3:

$$\#f_3 = 1001 \quad 1001 = \#[A_1 \cdot A_2 + \bar{A}_1 \cdot \bar{A}_2]$$

Hence a simple set of solutions is given by the array:

i	0	1	2	3	4	5	6	7	
$\#f_1(A_1,A_2,A_3)$	1	1	0	0	1	1	0	0	$= \#\bar{A}_2$
$\#f_2(A_1,A_2,A_3)$	1	1	1	1	0	0	1	1	$= \#[A_2 + \bar{A}_3]$
$\#f_3(A_1,A_2,A_3)$	1	0	0	1	1	0	0	1	$= \#[A_1 \cdot A_2 + \bar{A}_1 \cdot \bar{A}_2]$

The three desired circuits are shown in Fig. 14-3.

No Solutions. It can happen that antecedence or consequence solutions do not exist. For example, suppose that it is desired to find a function $f_1(A_1,A_2)$ such that, if the output of its circuit is connected to the input of another circuit F whose output is $f_1 \cdot X_1 + \bar{f}_1 \cdot \bar{X}_1$, the total resulting output E will be $\bar{A}_1 \cdot \bar{A}_2 + \bar{A}_1 \cdot X_1 + \bar{A}_2 \cdot \bar{X}_1$. Let us start by

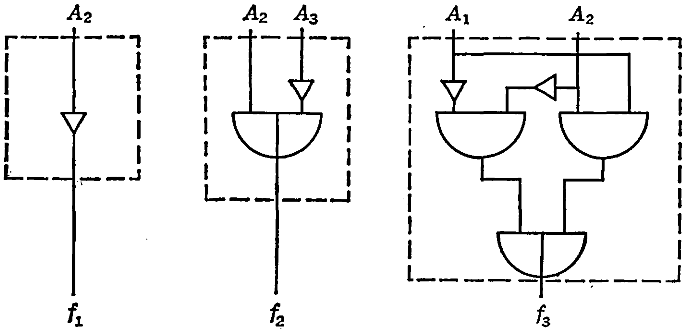


FIG. 14-3. Solution to the third example.

trying for antecedence solutions:

$$\begin{array}{r}
\#f_1 = 0101 \\
\#X_1 = 0011 \\
\hline
\#F = \#[f_1 \cdot X_1 + \bar{f}_1 \cdot \bar{X}_1] = 1001
\end{array}
\qquad
\begin{array}{rcc}
\#A_1 = & 0101 & 0101 \\
\#A_2 = & 0011 & 0011 \\
\#X_1 = & 0000 & 1111 \\
\hline
\#E = \#[\bar{A}_1 \cdot \bar{A}_2 + \bar{A}_1 \cdot X_1 + \bar{A}_2 \cdot \bar{X}_1] = & 1100 & 1010
\end{array}$$

whence $(F_{kj}) = \begin{pmatrix} 10 \\ 01 \end{pmatrix}$ $\quad (E_{ki}) = \begin{pmatrix} 1100 \\ 1010 \end{pmatrix}$

Substituting in the antecedence formulas,

$$(F_{jk}) \otimes (\bar{E}_{ki}) = \begin{pmatrix} 10 \\ 01 \end{pmatrix} \otimes \begin{pmatrix} 0011 \\ 0101 \end{pmatrix} = \begin{pmatrix} 0011 \\ 0101 \end{pmatrix} = (\bar{R}_{ji})_a$$

whence $$(R_{ji})_a = \begin{pmatrix} 1100 \\ 1010 \end{pmatrix}$$

To interpret (R_{ji}), we have

RESULT ARRAY

i	0	1	2	3
$\#f_1$	0, 1	0	1	None

Hence *no* antecedence solution exists. The condition for the existence of a solution is computed (see "Type 3 Problems" in Sec. 13-3) by

$$(V_j) \otimes (R_{ji}) = (11) \otimes \begin{pmatrix} 1100 \\ 1010 \end{pmatrix} = (1110)$$

to be interpreted with respect to $b[A_1,A_2]$: $\#A_1 = 0101$, $\#A_2 = 0011$. Hence $1110 = \#[\bar{A}_1 + \bar{A}_2]$. In order to have an antecedence solution, the signals on wires A_1 and A_2 must satisfy $A_1 \rightarrow \bar{A}_2$; that is, if a unit signal level appears on wire A_1, then a zero signal level must appear on wire A_2. Suppose that this were actually the case. To find the antecedence solution, we note that $b_c[A_1,A_2,X_1]$ becomes

$$\begin{array}{lr}
 & \#A_1 = 010 \quad 010 \\
 & \#A_2 = 001 \quad 001 \\
 & \#X_1 = 000 \quad 111 \\
\hline
\text{whence} & \#E = \#[\bar{A}_1 \cdot \bar{A}_2 + \bar{A}_1 \cdot X_1 + \bar{A}_2 \cdot \bar{X}_1] = 110 \quad 101
\end{array}$$

and
$$(E_{ki}) = \begin{pmatrix} 110 \\ 101 \end{pmatrix}$$

Then

$$\begin{pmatrix} 10 \\ 01 \end{pmatrix} \otimes \begin{pmatrix} 001 \\ 010 \end{pmatrix} = \begin{pmatrix} 001 \\ 010 \end{pmatrix} = (\bar{R}_{ji})_a \qquad \text{and} \qquad (R_{ji})_a = \begin{pmatrix} 110 \\ 101 \end{pmatrix}$$

This gives the result array:

i	0	1	2
$\#f$	0, 1	0	1

which, interpreted in terms of the constrained basis $b_c[A_1,A_2]$

$$\begin{array}{ll}
 & \#A_1 = 010 \\
 & \#A_2 = 001 \\
\hline
\text{becomes} & \#f_1 = 001 = \#A_2 \\
\text{or} & \#f_1 = 101 = \#\bar{A}_1
\end{array}$$

which gives two alternatives for the desired solution. However, it can be observed that these solutions make $F_a \rightarrow E$, but $F \neq E$.

Specifying the Variables. Suppose that it is desired to find consequence solutions of

$$E = A \cdot (\bar{B} \cdot C + B \cdot D) + \bar{A} \cdot \bar{C} \cdot (B \cdot \bar{D} + \bar{B} \cdot D)$$

that do not involve C. In other words, we want to determine all F such that $E \rightarrow F$ does not involve C. To do this, we rename our elementary elements A, B, C, and D as A_1, A_2, X_1, and A_3, respectively. Then we find consequence solutions of type 3 for E, where F is simply f_1. Thus

$$\begin{aligned}
\#E &= \#[A_1 \cdot (\bar{A}_2 \cdot X_1 + A_2 \cdot A_3) + \bar{A}_1 \cdot \bar{X}_1 \cdot (A_2 \cdot \bar{A}_3 + \bar{A}_2 \cdot A_3)] \\
&= 0010 \quad 1001 \quad 0100 \quad 0101
\end{aligned}$$

and $\#F = \#f_1 = 0101$ with respect to $b[f_1,X_1]$. Hence

$$(\bar{F}_{jk}) \otimes (E_{ki}) = \begin{pmatrix} 11 \\ 00 \end{pmatrix} \otimes \begin{pmatrix} 0010 & 1001 \\ 0100 & 0101 \end{pmatrix} = \begin{pmatrix} 0110 & 1101 \\ 0000 & 0000 \end{pmatrix} = (\bar{R}_{ji})$$

and
$$(R_{ji}) = \begin{pmatrix} 1001 & 0010 \\ 1111 & 1111 \end{pmatrix}$$

and hence there are eight solutions:

RESULT ARRAY

i	0	1	2	3	4	5	6	7
f	0, 1	1	1	0, 1	1	1	0, 1	1

Three simple nontrivial solutions are

$$A_1 + A_2 + A_3 = A + B + D$$
$$\bar{A}_1 + \bar{A}_2 + A_3 = \bar{A} + \bar{B} + D$$

and

$$A_1 + \bar{A}_2 + \bar{A}_3 = A + \bar{B} + \bar{D}$$

EXERCISES

Suppose that the circuit F has already been constructed and that it is desired to construct a circuit with output E. How should the circuits f_1, f_2, and f_3 be constructed?

(a) $F = (f_1 + f_3) \cdot \bar{X}_1 \cdot \bar{X}_2 \cdot \bar{X}_3 + f_3 \cdot X_1 \cdot \bar{X}_2 \cdot \bar{X}_3 + (f_2 + f_3) \cdot \bar{X}_1 \cdot X_2 \cdot \bar{X}_3$
$+ f_1 \cdot X_1 \cdot X_2 \cdot \bar{X}_3 + f_3 \cdot \bar{X}_1 \cdot \bar{X}_2 \cdot X_3 + f_1 \cdot f_2 \cdot X_1 \cdot \bar{X}_2 \cdot X_3$
$+ f_2 \cdot \bar{X}_1 \cdot X_2 \cdot X_3 + (f_3 + f_2 \cdot f_1) \cdot X_1 \cdot X_2 \cdot X_3.$

$E = ((\bar{A}_1 + A_2) \cdot \bar{A}_3 + A_3 \cdot (A_1 \cdot \bar{A}_2 + \bar{A}_1 \cdot A_2)) \cdot \bar{X}_1 \cdot \bar{X}_2 \cdot \bar{X}_3$
$+ (A_2 \cdot \bar{A}_3 + A_1 \cdot A_2 \cdot A_3) \cdot X_1 \cdot \bar{X}_2 \cdot \bar{X}_3$
$+ ((A_1 + A_2) \cdot \bar{A}_3 + A_3 \cdot \bar{A}_1) \cdot \bar{X}_1 \cdot X_2 \cdot \bar{X}_3$
$+ (\bar{A}_1 \cdot \bar{A}_2 \cdot \bar{A}_3 + A_3 \cdot (A_1 \cdot \bar{A}_2 + \bar{A}_1 \cdot A_2)) \cdot X_1 \cdot X_2 \cdot \bar{X}_3$
$+ (A_2 \cdot \bar{A}_3 + A_1 \cdot A_2 \cdot A_3) \cdot \bar{X}_1 \cdot \bar{X}_2 \cdot X_3 + \bar{A}_1 \cdot A_2 \cdot A_3 \cdot X_1 \cdot \bar{X}_2 \cdot X_3$
$+ (A_1 \cdot \bar{A}_2 \cdot \bar{A}_3 + \bar{A}_1 \cdot A_3) \cdot \bar{X}_1 \cdot X_2 \cdot X_3 + A_2 \cdot X_1 \cdot X_2 \cdot X_3.$

Solution. $f_1 = \bar{A}_1 \cdot \bar{A}_2 \cdot \bar{A}_3 + A_3 \cdot (\bar{A}_1 \cdot A_2 + A_1 \cdot \bar{A}_2), f_2 = A_1 \cdot \bar{A}_2 \cdot A_3 + \bar{A}_1 \cdot A_3,$ $f_3 = A_2 \cdot \bar{A}_3.$

(b) $F = \bar{X}_1 \cdot f_1 \cdot f_2 \cdot \bar{f}_3 + X_1 \cdot f_1 \cdot \bar{f}_2 \cdot \bar{f}_3 + \bar{f}_1 \cdot f_2 \cdot f_3.$

$E = \bar{X}_1 \cdot \bar{X}_2 \cdot (\bar{A}_3 \cdot (\bar{A}_1 \cdot \bar{A}_2 + A_1 \cdot A_2) + A_3 \cdot (A_1 \cdot \bar{A}_2 + \bar{A}_1 \cdot A_2)) + X_1 \cdot \bar{X}_2 \cdot \bar{A}_1$
$+ \bar{X}_1 \cdot X_2 \cdot (\bar{A}_3 \cdot (A_1 + A_2) + A_3 \cdot (\bar{A}_1 + A_2))$
$+ X_1 \cdot X_2 \cdot (\bar{A}_3 \cdot (\bar{A}_1 \cdot \bar{A}_2 + A_1 \cdot A_2) + A_1 \cdot \bar{A}_2 \cdot A_3).$

(c) $F = \bar{X}_1 \cdot \bar{X}_2 \cdot (f_2 \cdot \bar{f}_3 + f_1 \cdot \bar{f}_2 \cdot f_3) + X_1 \cdot \bar{X}_2 \cdot (\bar{f}_1 \cdot f_2 \cdot \bar{f}_3 + f_3 \cdot (\bar{f}_1 + f_2))$
$+ \bar{X}_1 \cdot X_2 \cdot (f_1 \cdot \bar{f}_2 \cdot \bar{f}_3 + \bar{f}_2 \cdot f_3) + X_1 \cdot X_2 \cdot (\bar{f}_1 \cdot \bar{f}_2 \cdot \bar{f}_3 + f_1 \cdot f_2).$

$E = \bar{X}_1 \cdot \bar{X}_2 \cdot (\bar{A}_1 \cdot A_2 \cdot \bar{A}_3 + A_2 \cdot A_3)$
$+ X_1 \cdot \bar{X}_2 \cdot (A_1 \cdot \bar{A}_2 \cdot \bar{A}_3 + A_3 \cdot (\bar{A}_1 \cdot \bar{A}_2 + A_1 \cdot A_2))$
$+ \bar{X}_1 \cdot X_2 \cdot (\bar{A}_3 \cdot (\bar{A}_1 \cdot \bar{A}_2 + A_1 \cdot A_2) + A_3 \cdot \bar{A}_2 \cdot \bar{A}_1)$
$+ X_1 \cdot X_2 \cdot (\bar{A}_3 \cdot A_2 \cdot A_1 + A_3 \cdot (\bar{A}_1 \cdot A_2 + A_1 \cdot \bar{A}_2)).$

14-4. Constraints in the Problems

The Introduction of Constraints. There are four possible ways in which constraints can appear in antecedence and consequence problems. There may be logical dependence among the $A_1, \ldots, A_I$ themselves, among the $X_1, \ldots, X_K$, among the $f_1, \ldots, f_J$, or between the $A_1, \ldots, A_I$ and the $X_1, \ldots, X_K$. In this section we consider how the first three such constraints affect the formation of the matrices; the fourth will be considered in the next section.

Constraints in the E and F Matrices. If constraints occur between the $A_1, \ldots, A_I$ themselves, then in $b[A_1, \ldots, A_I, X_1, \ldots, X_K]$ some columns of index i will be omitted. The corresponding columns of (E_{ki}) will also be omitted. Similarly, if some constraints occur between the $X_1, \ldots, X_K$, in $b[A_1, \ldots, A_I, X_1, \ldots, X_K]$ some columns of index k will be omitted; the corresponding rows of (E_{ki}) will also be omitted. For example, suppose that the logical dependence between A_1, A_2, and A_3 is given by $\bar{A}_1 = A_2 + A_3$ and $\bar{A}_2 = A_1 + A_3$. Suppose also that the logical dependence between X_1 and X_2 is such that $X_1 \rightarrow X_2$. Then from the normal $b[A_1,A_2,A_3,X_1,X_2]$ we have

$$\begin{array}{rllllllll}
\#A_1 = & 0101 & 0101 & 0101 & 0101 & 0101 & 0101 & 0101 & 0101 \\
\#A_2 = & 0011 & 0011 & 0011 & 0011 & 0011 & 0011 & 0011 & 0111 \\
\#A_3 = & 0000 & 1111 & 0000 & 1111 & 0000 & 1111 & 0000 & 1111 \\
\#X_1 = & 0000 & 0000 & 1111 & 1111 & 0000 & 0000 & 1111 & 1111 \\
\#X_2 = & 0000 & 0000 & 0000 & 0000 & 1111 & 1111 & 1111 & 1111 \\
\hline
\multicolumn{9}{l}{\#[(\bar{A}_1 = A_2 + A_3) \cdot (\bar{A}_2 = A_1 + A_3)]} \\
= & 0110 & 1000 & 0110 & 1000 & 0110 & 1000 & 0110 & 1000 \\
\#(\bar{X}_1 + X_2) = & 1111 & 1111 & 0000 & 0000 & 1111 & 1111 & 1111 & 1111 \\
\hline
\Pi\,\#(\text{constraints}) = & 0110 & 1000 & 0000 & 0000 & 0110 & 1000 & 0110 & 1000 \\
i & 0123 & 4567 & 0123 & 4567 & 0123 & 4567 & 0123 & 4567 \\
k & \multicolumn{2}{c}{0} & \multicolumn{2}{c}{1} & \multicolumn{2}{c}{2} & \multicolumn{2}{c}{3}
\end{array}$$

Hence the constrained basis $b_c[A_1,A_2,A_3,X_1,X_2]$ becomes

$$\begin{array}{rlll}
\#A_1 = & 100 & 100 & 100 \\
\#A_2 = & 010 & 010 & 010 \\
\#A_3 = & 001 & 001 & 001 \\
\#X_1 = & 000 & 000 & 111 \\
\#X_2 = & 000 & 111 & 111 \\
\hline
i & 124 & 124 & 124 \\
k & 0 & 2 & 3
\end{array}$$

Now suppose that $E = A_1 \cdot A_2 + X_1 \cdot \bar{A}_2 \cdot A_3 + X_2 \cdot (A_1 \cdot \bar{A}_3 + A_2)$, as in Sec. 13-2. Then with respect to $b_c[A_1,A_2,A_3,X_1,X_2]$

$$\#E = 000 \quad 110 \quad 111 \qquad \text{and} \qquad (E_{ki}) = \begin{pmatrix} 000 \\ 110 \\ 111 \end{pmatrix}$$

In general we can say that, if the constraints between $A_1, \ldots, A_I$ are such that $b_c[A_1, \ldots, A_I]$ has u columns and between $X_1, \ldots, X_K$ such that $b_c[X_1, \ldots, X_K]$ has v columns, then the designation number of E with respect to $b_c[A_1, \ldots, A_I, X_1, \ldots, X_K]$ is divided into v parts each of u positions; the successive parts form the successive rows of (E_{ki}).

We have a similar situation when considering (F_{kj}). For example, suppose that the logical dependence between f_1, f_2, and f_3 is such that $\bar{f}_1 = f_2 + f_3$ and $\bar{f}_2 = f_1 + f_3$ and between X_1 and X_2 such that $X_1 \rightarrow X_2$.

Then $b_c[f_1,f_2,f_3,X_1,X_2]$ is

$$\begin{array}{rl} \#f_1 = & 100 \quad 100 \quad 100 \\ \#f_2 = & 010 \quad 010 \quad 010 \\ \#f_3 = & 001 \quad 001 \quad 001 \\ \#X_1 = & 000 \quad 000 \quad 111 \\ \#X_2 = & 000 \quad 111 \quad 111 \end{array}$$

If $F = \bar{f}_1 \cdot f_3 + X_1 \cdot f_1 \cdot \bar{f}_2 + X_2 \cdot f_2 \cdot \bar{f}_3$ as in Sec. 13-2, then with respect to $b_c[f_1,f_2,f_3,X_1,X_2]$ we have

$$\#F = 001 \quad 011 \quad 111 \qquad \text{and} \qquad (F_{kj}) = \begin{pmatrix} 001 \\ 011 \\ 111 \end{pmatrix}$$

Constraints in the R Matrix. Consider next the effect of constraints on matrix (R_{ji}). Suppose $f_1 = A_1 \cdot \bar{A}_2 + \bar{A}_1 \cdot A_3$, $f_2 = \bar{A}_1 \cdot A_2 + A_1 \cdot \bar{A}_3$, and $f_3 = A_1 \cdot A_2 + \bar{A}_2 \cdot A_3$, as in Sec. 13-2. Then by Sec. 14-2 the constraint between f_1, f_2, and f_3 is given (for $k = 0, 1, 2,$ and 3) by

$$(C_j) = (R_{ji}) \otimes (I_i) = \begin{pmatrix} 1 \\ 0 \\ 1 \\ 1 \\ \\ 1 \\ 1 \\ 1 \\ 0 \end{pmatrix}$$

With respect to $b[f_1,f_2,f_3]$ we have $\#C = 1011 \ 1110$, and $b_c[f_1,f_2,f_3]$ becomes

$$\begin{array}{rl} \#f_1 = & 001 \quad 010 \\ \#f_2 = & 011 \quad 001 \\ \#f_3 = & 000 \quad 111 \end{array}$$

Now suppose that in addition $\bar{A}_1 = A_2 + A_3$ and $\bar{A}_2 = A_1 + A_3$; then $b_c[A_1,A_2,A_3]$ becomes

$$\begin{array}{rl} \#A_1 = & 100 \\ \#A_2 = & 010 \\ \#A_3 = & 001 \end{array}$$

Now we can determine (R_{ji}) with the constraints, for with respect to $b_c[A_1,A_2,A_3]$ we have

$$\begin{array}{rl} \#f_1 = & 101 \\ \#f_2 = & 110 \\ \#f_3 = & 001 \end{array} \qquad \text{whence } (R_{ji}) = \begin{array}{cc} & i \ 124 \\ \begin{array}{r} j \ 0 \\ 2 \\ 3 \\ \\ 4 \\ 5 \\ 6 \end{array} & \begin{pmatrix} 000 \\ 010 \\ 100 \\ \\ 000 \\ 001 \\ 000 \end{pmatrix} \end{array}$$

Note that introducing a constraint on the A_1, A_2, A_3 in turn introduced an *additional* constraint on f_1, f_2, f_3, for

$$(C_j) = (R_{ji}) \otimes (I_i) = \begin{bmatrix} 000 \\ 010 \\ 100 \\ \\ 000 \\ 001 \\ 000 \end{bmatrix} \otimes \begin{pmatrix} 1 \\ 1 \\ 1 \end{pmatrix} = \begin{bmatrix} 0 \\ 1 \\ 1 \\ \\ 0 \\ 1 \\ 0 \end{bmatrix}$$

or with respect to $b_c[f_1,f_2,f_3]$ we have $\#C = 011\ 010$. Hence $b_{cc}[f_1,f_2,f_3]$ (where the subscript cc indicates the double constraint) becomes

$$\begin{aligned} \#f_1 &= 101 \\ \#f_2 &= 110 \\ \#f_3 &= 001 \end{aligned} \qquad \text{whence } (R_{ji}) = \begin{array}{cc} & i\ 124 \\ j\ 2 & \\ 3 & \\ 5 & \end{array}\!\!\!\! \begin{array}{c} \\ \begin{pmatrix} 010 \\ 100 \\ 001 \end{pmatrix} \end{array}$$

where the indices given for i and j refer to the unconstrained bases.

EXERCISES

Consider the situation where the constraints between A_1, A_2, and A_3 are $\bar{A}_1 = A_2 + A_3$ and $\bar{A}_2 = A_1 + A_3$. Let the constraint between X_1 and X_2 be $X_1 \rightarrow X_2$. Let $f_1(A_1,A_2,A_3)$, $f_2(A_1,A_2,A_3)$, and $f_3(A_1,A_2,A_3)$ be the same as used above (i.e., the same as in Sec. 13-2). Under these circumstances:

(*a*) Find $b_{cc}[f_1,f_2,f_3,X_1,X_2]$.

(*b*) Find $b_c[A_1,A_2,A_3,X_1,X_2]$ and $b_c[A_1,A_2,A_3]$.

(*c*) Let $F = \bar{f}_1 \cdot f_3 + X_1 \cdot f_1 \cdot \bar{f}_2 + X_2 \cdot f_2 \cdot \bar{f}_3$, and find E, using the type 1 formula.

(*d*) Let $E = A_1 \cdot A_2 + X_1 \cdot \bar{A}_2 \cdot A_3 + X_2 \cdot (A_1 \cdot \bar{A}_3 + A_2)$ as in Sec. 13-2. Using the f_s given above, find $F(f_1,f_2,f_3,X_1,X_2)$ such that $F = E$.

(*e*) Let F be as in Exercise *c* and E be as in Exercise *d*; determine f_1, f_2, and f_3 such that $F = E$.

14-5. Ordinary Equations

The Problem. The methods discussed in the previous section for taking account of constraints in circuit-design problems are relatively straightforward. In all the cases discussed the $A_1, \ldots, A_I$ and the $X_1, \ldots, X_K$ were independent of each other even though there may have been some logical dependence among the $A_1, \ldots, A_I$ themselves and among the $X_1, \ldots, X_K$ themselves. In this section we shall investigate problems where there is a logical dependence between the $A_1, \ldots, A_I$ and the $X_1, \ldots, X_K$.

First it must be noticed that if we tried to form (E_{ki}) with respect to $b_c[A_1, \ldots ,A_I,X_1, \ldots ,X_K]$, where the constraint is between the A's and X's, the matrix would not be rectangular, i.e., some of the elements would be missing—and we shall not consider nonrectangular matrices.

For example, suppose that $A_1 \rightarrow X_1$ and $X_1 \cdot X_2 \rightarrow A_2$. Let

$$E = A_1 \cdot A_2 + X_1 \cdot \bar{A}_2 + X_2 \cdot (A_1 + A_2)$$

with respect to $b_c[A_1,A_2,X_1,X_2]$,

$$\begin{array}{rcccc}
\#A_1 = & 00 & 0101 & 00 & 01 \\
\#A_2 = & 01 & 0011 & 01 & 11 \\
\#X_1 = & 00 & 1111 & 00 & 11 \\
\#X_2 = & 00 & 0000 & 11 & 11 \\
\hline
\#E = & 00 & 1101 & 01 & 11 \\
 & & & & \\
i & 01 & 0123 & 01 & 23 \\
k & 0 & 1 & 2 & 3
\end{array}$$

$$(E_{ki}) = \begin{pmatrix} 0 & 0 & & \\ 1 & 1 & 0 & 1 \\ 0 & 1 & & \\ & & 1 & 1 \end{pmatrix}$$

which is not rectangular and has no meaning for us. In fact a little reflection will show that the whole *meaning* of problems of types 1, 2, and 3 vanishes if we let $X_s = X_s(A_1, \ldots ,A_I)$ directly. Hence let us consider in more detail the kinds of problems that can occur where $X_s = X_s(A_1, \ldots ,A_I)$.

If there is some relation between the $A_1, \ldots , A_I$ and $X_1, \ldots , X_K$, then the problem can be interpreted in terms of the problem of the *solution to ordinary equations* as considered in Chap. 12. In general these problems involve a function $E'(A_1, \ldots ,A_I,X_1, \ldots ,X_K)$ which is to be a tautology, i.e., always true, when appropriate functions

$$X_s = X_s(A_1, \ldots ,A_I)$$

are substituted into E'. Two questions may arise concerning these problems: (1) Can $X_1(A_1, \ldots ,A_I)$ be found which when substituted into a given E' will make E' a tautology? (2) Can an E' be found such that, when given functions $X_1(A_1, \ldots ,A_I), \ldots , X_K(A_1, \ldots ,A_I)$ are substituted into it, it becomes a tautology? These are essentially inverse problems to each other. The first question was really considered first in Chap. 12, where the function F was usually of the form of a Boolean equation,

$$H(A_1, \ldots ,A_I,X_1, \ldots ,X_K) = G(A_1, \ldots ,A_I,X_1, \ldots ,X_K)$$

For simultaneous equations the function E' is simply the product of the given Boolean equations. The second question, however, was not considered previously. In this section we shall present methods for the solution of both of these equations in terms of Boolean matrices.

Ordinary Equations as an Antecedence-solution Problem. In order to recast the solution-to-equations problem into the types of problems considered in this chapter, note that we either are given or desire to find functions

$$X_s = X_s(A_1, \ldots ,A_I)$$

Hence let $X_s = f_s(A_1, \ldots, A_I)$. Then we desire that these solutions make E a tautology, i.e., make $E = I$. In other words, we want

$$\begin{aligned}\{[X_1 = f_1(A_1, \ldots, A_I)] \cdot [X_2 = f_2(A_1, \ldots, A_I)] \cdots \\ [X_K = f_K(A_1, \ldots, A_I)]\} \\ \rightarrow \{E'(A_1, \ldots, A_I, X_1, \ldots, X_K) = I\}\end{aligned}$$

Hence our questions simply relate to *antecedence solutions of the form* $F = [(X_1 = f_1) \cdot (X_2 = f_1) \cdots (X_K = f_K)]$ of $E = [E' = I]$. Question 1 becomes an antecedence-solution problem of type 3, and question 2 becomes an antecedence-solution problem of type 1.

Two observations should be made: First recall that $\#(E' = I) = \#E'$, and hence the Boolean equation E' itself is the function E. Second, observe that the form of F is known. In this case it turns out that (F_{kj}) is a matrix with units on the diagonal and zeros elsewhere. For consider $F = [(X_1 = f_1) \cdot (X_2 = f_2)]$,

$$\begin{array}{rl} \#(X_1 = f_1) = & 1010 \quad 0101 \quad 1010 \quad 0101 \\ \#(X_2 = f_2) = & \underline{1000 \quad 0100 \quad 0010 \quad 0001} \\ \#F = & 1000 \quad 0100 \quad 0010 \quad 0001 \end{array} \qquad \text{and} \qquad (F_{jk}) = \begin{pmatrix} 1000 \\ 0100 \\ 0010 \\ 0001 \end{pmatrix}$$

This is an important observation, for if we denote such a diagonal matrix by (I_{jk}), it is easy to see that for any Boolean matrix (a_{kj})

$$(I_{jk}) \otimes (a_{kj}) = (a_{kj})$$

In other words, multiplying a matrix by (I_{jk}) results in the same original matrix. Thus, in finding antecedence solutions for type 3 problems with this F,

$$(I_{jk}) \otimes (\bar{E}_{ki}) = (\bar{E}_{ki}) = (\bar{R}_{ji}) \qquad \text{or} \qquad (R_{ji}) = (E_{ki})$$

Hence the matrix multiplication need never be performed, the (R_{ji}) being found directly as the (E_{ki}). A similar situation occurs for antecedence solutions for type 1 problems with this F, where the (E_{ki}) matrix is found directly as the (R_{ji}) matrix.

1. *Solution to Equations.* Here the equation E is given, and the problem is to find solutions $X_s = X_s(A_1, \ldots, A_I)$. As an example, consider the equation E to be

$$(A_1 + A_2) \cdot X_1 \cdot \bar{X}_2 + (\bar{A}_1 + \bar{A}_2) \cdot X_2 = A_1 \cdot A_2 + \bar{A}_1 \cdot \bar{A}_2$$

A quick method for finding $\#E$ is first to find $\#H$ and $\#G$,† and then to note that $\#E$ has units in those positions where the corresponding bits of $\#H$ and $\#G$ are both zero or both units; the $\#E$ has zeros elsewhere. Thus

$$\begin{array}{rl} \#(A_1 + A_2) \cdot X_1 \cdot \bar{X}_2 + (\bar{A}_1 + \bar{A}_2) \cdot X_2 = & 0000 \quad 0111 \quad 1110 \quad 1110 \\ \#A_1 \cdot A_2 + \bar{A}_1 \cdot \bar{A}_2 = & \underline{1001 \quad 1001 \quad 1001 \quad 1001} \\ \#E = & 0110 \quad 0001 \quad 1000 \quad 1000 \end{array}$$

† Recall that H and G are the left- and right-hand sides of the equation, respectively.

Thus $(E_{ki}) = (R_{ji}) = \begin{pmatrix} 0110 \\ 0001 \\ 1000 \\ 1000 \end{pmatrix}$, and the result array becomes

i	0	1	2	3
$\#f_1$	0, 1	0	0	1
$\#f_2$	1, 1	0	0	0

whence $X_1 = f_1 = A_1 \cdot A_2$ $\qquad X_2 = f_2 = \bar{A}_1 \cdot \bar{A}_2$
or $X_1 = f_1 = A_1 \cdot A_2 + \bar{A}_1 \cdot \bar{A}_2$ $\qquad X_2 = f_2 = \bar{A}_1 \cdot \bar{A}_2$

Summarizing, to find solutions to the equation E, (1) form $\#E$ and then (E_{ki}), and (2) note that $(E_{ki}) = (R_{ji})$, whence the solutions follow from (R_{ji}).

2. *Inverse Solution to Equations.* Here the functions

$$X_s = f_s(A_1, \ . \ . \ . \ ,A_I)$$

are given, and the problem is to find an equation E. For example, given

$$\begin{aligned} X_1 &= \bar{A}_1 \cdot (A_2 \cdot A_3 \cdot + \bar{A}_2 \cdot \bar{A}_3) + A_1 \cdot \bar{A}_2 \cdot A_3 = f_1 \\ X_2 &= \bar{A}_3 \cdot (A_1 + A_2) + \bar{A}_1 \cdot \bar{A}_2 \cdot A_3 \qquad\qquad = f_2 \end{aligned}$$

what single equation has X_1 and X_2 as solutions? To determine (R_{ji}), find, with respect to $b[A_1,A_2,A_3]$,

$$\begin{aligned} \#f_1 &= 1000 \quad 0110 \\ \#f_2 &= 0111 \quad 1000 \end{aligned} \qquad \text{whence } (R_{ji}) = (E_{ki}) = \begin{pmatrix} 0000 & 0001 \\ 1000 & 0110 \\ 0111 & 1000 \\ 0000 & 0000 \end{pmatrix}$$

and $\#E = 0000 \quad 0001 \quad 1000 \quad 0110 \quad 0111 \quad 1000 \quad 0000 \quad 0000$

However, note that, if E' is an equation such that $E \rightarrow E'$, then the solutions to E are also solutions to E'. This allows us the opportunity of choosing a more desirable E' for our equation.

To determine a good E', let us choose E' of the form

$$H(A_1,A_2,A_3,X_1,X_2) = G(A_1,A_2,A_3)$$

which is a form often found in circuit-design problems. Let us choose G explicitly to be $A_1 \cdot A_2 + A_1 \cdot \bar{A}_3 + A_2 \cdot \bar{A}_3 + \bar{A}_2 \cdot A_3$. With respect to $b[A_1,A_2,A_3,X_1,X_2]$ we have

$$\begin{aligned} \#E &= 0000 \quad 0001 \quad 1000 \quad 0110 \qquad 0111 \quad 1000 \quad 0000 \quad 0000 \\ \#G &= 0111 \quad 1101 \quad 0111 \quad 1101 \qquad 0111 \quad 1101 \quad 0111 \quad 1101 \end{aligned}$$

We determine H so that $\#E = \#(G = H)$:

$$\#H = 1000 \quad 0011 \quad 0000 \quad 0100 \qquad 1111 \quad 1010 \quad 1000 \quad 0010$$

That is, the bit of $\#H$ is the same as the corresponding bit in $\#G$ if $\#E$ has a unit in that position; in positions where $\#E$ has a zero, the corresponding bit in $\#H$ is different from that in $\#G$. However, this H is too complicated; perhaps we can find an E' such that $E \rightarrow E'$, for which H will be simpler. Since $E \rightarrow E'$, $\#E'$ can be formed only by changing some zeros of $\#E$ to units, and this has a corresponding effect on $\#H$. If Φ means that we can fix $\#E'$ so that $\#H$ can have either a 0 or a 1 in that position, the choices for $\#H$ are seen to be

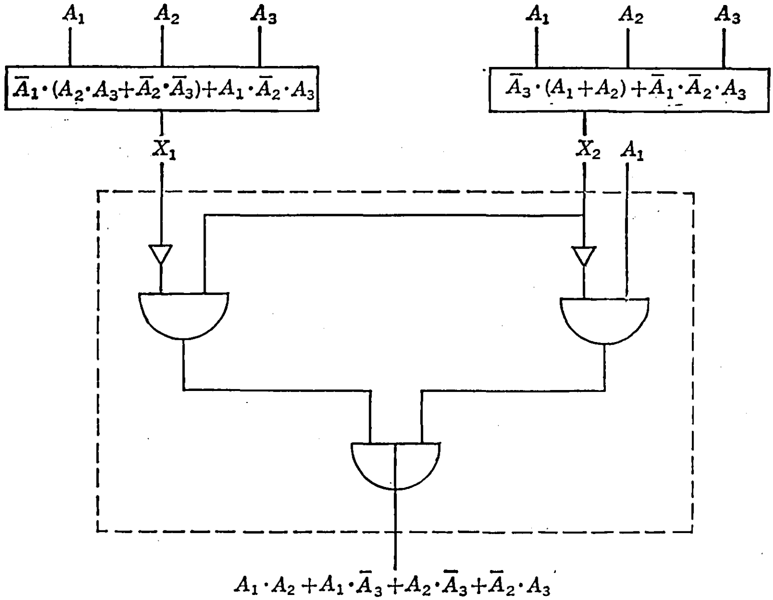

A_1
A_2
A_3
$\bar{A}_1\cdot(A_2\cdot A_3+\bar{A}_2\cdot\bar{A}_3)+A_1\cdot\bar{A}_2\cdot A_3$
X_1
A_1
A_2
A_3
$\bar{A}_3\cdot(A_1+A_2)+\bar{A}_1\cdot\bar{A}_2\cdot A_3$
X_2
A_1
$A_1\cdot A_2+A_1\cdot\bar{A}_3+A_2\cdot\bar{A}_3+\bar{A}_2\cdot A_3$

FIG. 14-4. Example of inverse solution to equations.

$$\Phi\Phi\Phi\Phi \quad \Phi\Phi\Phi 1 \quad 0\Phi\Phi\Phi \quad \Phi 10\Phi \quad \Phi 111 \quad 1\Phi\Phi\Phi \quad \Phi\Phi\Phi\Phi \quad \Phi\Phi\Phi\Phi$$

One simple $\#H$ that is included in these possibilities is

$$0101 \quad 0101 \quad 0101 \quad 0101 \quad 1111 \quad 1111 \quad 0000 \quad 0000 = \#(A_1 \cdot \bar{X}_2 + \bar{X}_1 \cdot X_2)$$

Hence we can have for the equation

$$A_1 \cdot \bar{X}_2 + \bar{X}_1 \cdot X_2 = A_1 \cdot A_2 + A_1 \cdot \bar{A}_3 + A_2 \cdot \bar{A}_3 + \bar{A}_2 \cdot A_3$$

Summarizing the technique, first (1) use $(R_{ji}) = (E_{ki})$ to determine $\#E$ from $f_1, f_2, \ldots, f_n$; then (2) let $E = (H = G)$, where G is given by the problem, and determine a simple H, making sure that E' is such that $E \rightarrow E'$.

As an application of this procedure to circuit design, suppose that two circuits with inputs A_1, A_2, and A_3 have been constructed such that the output of one is $X_1 = \bar{A}_1 \cdot (A_2 \cdot A_3 + \bar{A}_2 \cdot \bar{A}_3) + A_1 \cdot \bar{A}_2 \cdot A_3$ and of the other $X_2 = \bar{A}_3 \cdot (A_1 + A_2) + \bar{A}_1 \cdot \bar{A}_2 \cdot A_3$. Suppose that it is now desired to gate the outputs of these circuits to form a new circuit whose output is $A_1 \cdot A_2 + A_1 \cdot \bar{A}_3 + A_2 \cdot \bar{A}_3 + \bar{A}_2 \cdot A_3$. The problem is how to design the gating. The solution, as found above, appears in Fig. 14-4.

EXERCISES

(*a*) Suppose that two circuits with outputs $X_1 = (A_1 \cdot \bar{A}_2 + A_2 \cdot \bar{A}_3)$ and $X_2 = (\bar{A}_1 \cdot A_2 + \bar{A}_2 \cdot A_3)$ have been built. How can the outputs of these circuits be gated so that the final output is the following G?

$$G = A_1 \cdot \bar{A}_3 + A_1 \cdot \bar{A}_2 + A_2 \cdot \bar{A}_3 + \bar{A}_1 \cdot A_3$$

One solution is $X_1 + X_2 = G$.

(*b*) Suppose that three circuits have already been built with outputs $X_1 = A_1 \cdot \bar{A}_2 + \bar{A}_1 \cdot A_2 \cdot A_3$, $X_2 = \bar{A}_1 \cdot A_2 + (\bar{A}_1 + \bar{A}_2) \cdot A_3$, $X_3 = \bar{A}_1 \cdot A_2 + (\bar{A}_1 + A_2) \cdot \bar{A}_3$. It is desired to form from these a circuit with output

$$G = (A_1 \cdot A_2 + \bar{A}_1 \cdot \bar{A}_2) \cdot \bar{A}_3 + (A_1 \cdot \bar{A}_2 + \bar{A}_1 \cdot A_2) \cdot A_3$$

The problem is to form the desired circuit. One solution is

$$X_1 \cdot X_2 + \bar{X}_2 \cdot X_3 = G$$

14-6. More General Circuit-design Problems

Properties of the General Net. Until now the functions $f_1, f_2, \ldots, f_J$ were considered to be functions of $A_1, A_2, \ldots, A_I$ directly, that is, $f_s = f_s(A_1, \ldots, A_I)$. However, in the present section we extend our results to the more general situation where $f_1, f_2, \ldots, f_J$ are functions of other functions, say $g_1, g_2, \ldots, g_P$, that is, $f_s = f_s(g_1, \ldots, g_P)$; the functions $g_1, g_2, \ldots, g_P$ may themselves be considered in turn as functions of still other functions, and so forth, until finally a set of functions $h_1, h_2, \ldots, h_Q$ are direct functions of the $A_1, A_2, \ldots, A_I$. In Fig. 14-5 we give an example of three levels of such functions f_s, g_s, and h_s, although in general there can be as many levels as necessary, and each level can contain as many functions as necessary. In the most general case each function will involve all the functions of the preceding level.

The most important observation for our generalization is concerned with the R matrices. If the functions $f_1, f_2, \ldots, f_J$ are given explicitly, then in the usual fashion the R matrix can be derived (where $g_1, g_2, \ldots, g_P$ replaces the previous use of $A_1, A_2, \ldots, A_I$). Let us denote this matrix by $(R_{jp}{}^f)$, where the superscript f indicates that the R matrix refers to the set of functions $f_1, f_2, \ldots, f_J$. Similarly, if the functions $g_1, g_2, \ldots, g_P$ are given explicitly, we can derive $(R_{pq}{}^g)$, and so forth, until finally from an explicit form of $h_1, h_2, \ldots, h_Q$, we can derive $(R_{qi}{}^h)$.

Now the function $F(f_1, \ldots, f_J, X_1, \ldots, X_K)$ has, in a manner of speaking, no way of knowing that the input wires $f_1, f_2, \ldots, f_J$ involve such a complicated net; it can still consider f_s as an "eventual" function of $A_1, \ldots, A_I$. Hence F "sees" a total matrix (R_{ji}), as in the problem of Chap. 13. The important observation that we can make is that†

† See Sec. 13-7, Additional Topic *a*.

$$(R_{jp}{}^f) \otimes (R_{pq}{}^g) \otimes \cdots \otimes (R_{qi}{}^h) = (R_{ji}) \qquad (14\text{-}1)♦$$

In other words, we are stating that for the general case the fundamental matrix equation (13-32) becomes

$$(F_{kj}) \otimes [(R_{jp}{}^f) \otimes (R_{pq}{}^g) \otimes \cdots \otimes (R_{qi}{}^h)] = (E_{ki}) \qquad (14\text{-}2)♦$$

Similarly it is clear that if in the situation of Fig. 14-5 we let

$$(R_{jq}{}^G) = (R_{jp}{}^f) \otimes (R_{pq}{}^g) \qquad \text{and} \qquad (R_{pi}{}^H) = (R_{pq}{}^g) \otimes (R_{qi}{}^h)$$

we have

$$(R_{jq}{}^G) \otimes (R_{qi}{}^h) = (R_{ji}) \qquad \text{and} \qquad (R_{jp}{}^f) \otimes (R_{pi}{}^H) = (R_{ji})$$

and so forth. The general rule of thumb is that the R matrix corresponding to a set of functions nearer F appears nearer the left in the product, while the R matrix corresponding to a set of functions farther from F appears nearer the right in the product.

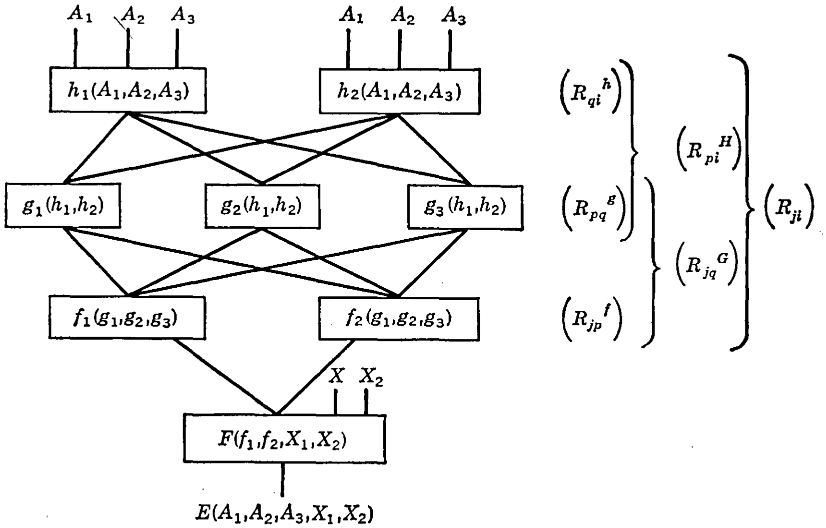


FIG. 14-5. The general circuit-design problem.

As an example, suppose that in Fig. 14-5 we had

$$h_1 = A_2 + A_3 \qquad h_2 = A_1 + A_3$$

whence

$$(R_{qi}{}^h) = \begin{pmatrix} 1000 & 0000 \\ 0010 & 0000 \\ 0100 & 0000 \\ 0001 & 1111 \end{pmatrix}$$

and

$$g_1 = h_1 \cdot h_2 \qquad g_2 = \bar{h}_1 \cdot \bar{h}_2 \qquad g_3 = h_1 \cdot \bar{h}_2$$

whence
$$(R_{pq}{}^{g}) = \begin{pmatrix} 0010 \\ 0001 \\ 1000 \\ 0000 \\ \\ 0100 \\ 0000 \\ 0000 \\ 0000 \end{pmatrix}$$

and
$$f_1 = g_1 + g_2 \qquad f_2 = g_2 + g_3$$

whence
$$(R_{jp}{}^{f}) = \begin{pmatrix} 1000 & 0000 \\ 0100 & 0000 \\ 0000 & 1000 \\ 0011 & 0111 \end{pmatrix}$$

By algebraic means we have

$$f_1 = (A_2 + A_3) \cdot (A_1 + A_3) + (\overline{A_2 + A_3}) \cdot (\overline{A_1 + A_3})$$

and
$$f_2 = (\overline{A_2 + A_3}) \cdot (\overline{A_1 + A_3}) + (A_2 + A_3) \cdot (\overline{A_1 + A_3})$$

whence
$$(R_{ji}) = \begin{pmatrix} 0100 & 0000 \\ 0001 & 1111 \\ 0010 & 0000 \\ 1000 & 0000 \end{pmatrix}$$

To check this by our formula, we compute

$$\begin{pmatrix} 1000\ 0000 \\ 0100\ 0000 \\ 0000\ 1000 \\ 0011\ 0111 \end{pmatrix} \otimes \begin{pmatrix} 0010 \\ 0001 \\ 1000 \\ 0000 \\ \\ 0100 \\ 0000 \\ 0000 \\ 0000 \end{pmatrix} \otimes \begin{pmatrix} 1000\ 0000 \\ 0010\ 0000 \\ 0100\ 0000 \\ 0001\ 1111 \end{pmatrix} = \begin{pmatrix} 0010 \\ 0001 \\ 0100 \\ 1000 \end{pmatrix} \otimes \begin{pmatrix} 1000\ 0000 \\ 0010\ 0000 \\ 0100\ 0000 \\ 0001\ 1111 \end{pmatrix} = \begin{pmatrix} 0100\ 0000 \\ 0001\ 1111 \\ 0010\ 0000 \\ 1000\ 0000 \end{pmatrix}$$

as expected.

Solving Circuit-design Problems. For circuit-design problems of type 1 the extension of the problem presents nothing new, except that the over-all (R_{ji}) matrix is computed from the individual-level R matrices according to the method just described. Similarly, for circuit-design problems of type 2 little change is required in the method of solution, (R_{ji}) being calculated from the individual-level R matrices by Eq. (14-1). However, consider problems of type 3. Here as before, given (F_{kj}) and (E_{ki}), all possible over-all (R_{ji}) can be computed. However, suppose that it is known, for example, that there are three levels of functions, namely, f_s, g_s, and h_s, and that the function sets g_s and h_s are known explicitly; then it might be required to find the function set f_s. In this case from Eq. (14-1) we have

$$(R_{jp}{}^{f}) \otimes [(R_{pq}{}^{g}) \otimes (R_{qi}{}^{h})] = (R_{ji})$$

where (R_{ji}) has just been calculated and the matrices in the square brackets are known. Thus, to find $(R_{jp}{}^f)$, that is, the function set f_s, simply solve the equation for the matrix $(R_{jp}{}^f)$, using Eqs. (13-19) and their correspondent from Exercise *e* of Sec. 13-5. This technique can be used, for the product in the square brackets will be of the form (U_{pi}), and $(R_{jp}{}^f)$ corresponds to (a_{jp}) considered in that section. If on the other hand $(R_{jp}{}^f)$ and $(R_{qi}{}^h)$ are known explicitly and it is desired to find $(R_{pq}{}^g)$, we would write

$$(R_{jp}{}^f) \otimes (R_{pq}{}^g) \otimes (R_{qi}{}^h) = (R_{jp}{}^f) \otimes (R_{pi}{}^{gh}) = (R_{ji})$$

and solve for $(R_{pi}{}^{gh})$, by the methods of Sec. 13-5. When $(R_{pi}{}^{gh})$ has been determined, we would solve

$$(R_{pq}{}^g) \otimes (R_{qi}{}^h) = (R_{pi}{}^{gh})$$

for the desired $(R_{pq}{}^g)$.

Summarizing, to solve circuit-design problems of type 3, the over-all (R_{ji}) matrix is first computed as usual from the appropriate fundamental formulas. Next by Eqs. (13-15) and (13-16), Eq. (14-1) relating the level R matrices with (R_{ji}) is solved for the desired results. As is immediately clear, there are in general many ways of tackling the same problems, using different variations of the techniques presented.

Examples. As a first example, consider a *circuit synthesis problem*, and suppose that the situation is as in Fig. 14-5, where $h_1 = A_2 + A_3$, $h_2 = A_1 + A_3$, $g_1 = h_1 \cdot h_2$, $g_2 = \bar{h}_1 \cdot \bar{h}_2$, $g_3 = h_1 \cdot \bar{h}_2$,

$F = (\bar{f}_1 + f_2) \cdot X_1 \cdot \bar{X}_2 + (f_1 + f_2) \cdot \bar{X}_1 \cdot X_2 + (f_1 \cdot f_2 + \bar{f}_1 \cdot \bar{f}_2) \cdot \bar{X}_1 \cdot \bar{X}_2$ and $E = (\bar{A}_2 \cdot \bar{A}_3 + \bar{A}_1 \cdot \bar{A}_3 \cdot X_1) \cdot \bar{X}_2 + (\bar{A}_1 + A_2 + A_3) \cdot \bar{X}_1 \cdot X_2$ are all given. That is to say, the circuits h_1, h_2, g_1, g_2, and g_3 have already been constructed and wired together; the circuit F has already been built. It is desired to determine how to build circuits f_1 and f_2, with inputs g_1, g_2, and g_3, so that the output of the entire system in terms of A_1, A_2, A_3, X_1, and X_2 will be the function E.

First we determine (R_{ji}) by the type 3 formulas, finding

$$(R_{ji}) = \begin{pmatrix} 0100 & 0000 \\ 0001 & 1111 \\ 0010 & 0000 \\ 1000 & 0000 \end{pmatrix}$$

as the only common antecedence and consequence solution. Next we observe that

$$(R_{pq}{}^g) \otimes (R_{qi}{}^h) = \begin{pmatrix} 0100 & 0000 \\ 0001 & 1111 \\ 1000 & 0000 \\ 0000 & 0000 \\[1em] 0010 & 0000 \\ 0000 & 0000 \\ 0000 & 0000 \\ 0000 & 0000 \end{pmatrix} = (R_{pi}{}^H)$$

whence we desire to solve $(R_{jp}{}^{f}) \otimes (R_{pi}{}^{H}) = (R_{ji})$ for the matrix $(R_{jp}{}^{f})$. It turns out [from Eqs. (13-23) and (13-24)] that

$$(R_{jp}{}^{f\prime}) = \begin{pmatrix} 1001 & 0111 \\ 0101 & 0111 \\ 0001 & 1111 \\ 0011 & 0111 \end{pmatrix} \qquad (R_{jp}{}^{f*}) = \begin{pmatrix} 1000 & 0000 \\ 0100 & 0000 \\ 0000 & 1000 \\ 0010 & 0000 \end{pmatrix}$$

whence by Eq. (13-25) $$(R_{jp}{}^{f}) = \begin{pmatrix} 100\Phi & 0\Phi\Phi\Phi \\ 010\Phi & 0\Phi\Phi\Phi \\ 000\Phi & 1\Phi\Phi\Phi \\ 001\Phi & 0\Phi\Phi\Phi \end{pmatrix}$$

Columns 0, 1, and 2 indicate that f_1 must involve g_1 and g_2, and columns 0, 2, and 4 indicate that f_2 must involve g_2 and g_3 (by the elimination pairs of positions). Hence a simple solution is seen to be:

	0	1	2	3	4	5	6	7	
$\#f_1$	0	1	1	1	0	1	1	1	$= g_1 + g_2$
$\#f_2$	0	0	1	1	1	1	1	1	$= g_2 + g_3$

Another set of valid solutions is:

$\#f_1$	0	1	1	0	0	1	1	0	$= g_1 \cdot \bar{g}_2 + \bar{g}_1 \cdot g_2$
$\#f_2$	0	0	1	1	1	1	0	0	$= g_2 \cdot \bar{g}_3 + \bar{g}_2 \cdot g_3$

As the second example, consider a *circuit-maintenance problem.* Suppose that a circuit has been constructed such that

$$g_1 = A_1 \cdot \bar{A}_2 + A_2 \cdot \bar{A}_3 \qquad g_2 = A_2 \cdot \bar{A}_3 + \bar{A}_1 \cdot A_3$$
$$f_1 = g_1 + \bar{g}_2 \qquad f_2 = \bar{g}_1 + g_2$$

and $$F = f_1 \cdot \bar{f}_2 + \bar{f}_1 \cdot f_2$$

The output of this circuit is seen to be $E = A_1 \cdot \bar{A}_2 + (\bar{A}_1 + \bar{A}_2) \cdot A_3$, where $\#E = 0100\ 1110$. Now suppose that the circuit breaks down, that it does not produce the proper output. The problem is to determine which of the circuits g_1, g_2, f_1, f_2, or F is not operating correctly. The information at our disposal for this investigation is (1) the original circuit design, i.e., the explicit (correct) functions g_1, g_2, f_1, f_2, and F, and (2) how the circuit is malfunctioning. This latter information is obtained by putting into the A_1, A_2, and A_3 inputs each of the eight possible input states and recording the output bit for each state. Suppose that these output states are 0000 1010 for the usual successive input-state order as given by the standard basis. This malfunctioning-output result is of course actually the designation number of the output of the malfunctioning circuit with respect to $b[A_1,A_2,A_3]$, that is,

$$\#E_m(A_1,A_2,A_3) = 0000 \quad 1010$$

Our purpose is to use this information to explore all possible ways in which the circuits f_1, f_2, g_1, g_2, and F can malfunction to give the output result $\#E_m$.

Let us begin by first making the reasonable assumption that only one of these circuits is malfunctioning. To be specific, let us see whether f_1 or f_2 can malfunction to produce $\#E_m$. Observe that $\#F = 0110$, whence

$$(F_{j0}) \otimes (\bar{E}_{0i}) = \begin{pmatrix} 0 \\ 1 \\ 1 \\ 0 \end{pmatrix} \otimes (1111 \quad 0101) = \begin{pmatrix} 0000 & 0000 \\ 1111 & 0101 \\ 1111 & 0101 \\ 0000 & 0000 \end{pmatrix} \qquad (R_{ji})_a = \begin{pmatrix} 1111 & 1111 \\ 0000 & 1010 \\ 0000 & 1010 \\ 1111 & 0101 \end{pmatrix}$$

$$(\bar{F}_{j0}) \otimes (E_{0i}) = \begin{pmatrix} 1 \\ 0 \\ 0 \\ 1 \end{pmatrix} \otimes (0000 \quad 1010) = \begin{pmatrix} 0000 & 1010 \\ 0000 & 0000 \\ 0000 & 0000 \\ 0000 & 1010 \end{pmatrix} \qquad (R_{ji})_c = \begin{pmatrix} 1111 & 0101 \\ 1111 & 1111 \\ 1111 & 1111 \\ 1111 & 0101 \end{pmatrix}$$

and $(R_{ji}) = \begin{pmatrix} 1111 & 0101 \\ 0000 & 1010 \\ 0000 & 1010 \\ 1111 & 0101 \end{pmatrix}$. Next we observe that $(R_{pi}{}^{g}) = \begin{pmatrix} 1000 & 0001 \\ 0100 & 0100 \\ 0000 & 1010 \\ 0011 & 0000 \end{pmatrix}$.

Hence, solving $(R_{jp}{}^{f}) \otimes (R_{pi}{}^{g}) = (R_{ji})$ for $(R_{jp}{}^{f})$, find

$$(R_{jp}{}^{f\prime}) = \begin{pmatrix} 0000 & 1010 \\ 1111 & 0101 \\ 1111 & 0101 \\ 0000 & 1010 \end{pmatrix} \otimes \begin{pmatrix} 1000 \\ 0100 \\ 0001 \\ 0001 \\ \\ 0010 \\ 0100 \\ 0010 \\ 1000 \end{pmatrix} = \begin{pmatrix} 1101 \\ 0010 \\ 0010 \\ 1101 \end{pmatrix}$$

$$(R_{jp}{}^{f*}) = \begin{pmatrix} 1111 & 0101 \\ 0000 & 1010 \\ 0000 & 1010 \\ 1111 & 0101 \end{pmatrix} \otimes \begin{pmatrix} 1000 \\ 0100 \\ 0001 \\ 0001 \\ \\ 0010 \\ 0100 \\ 0010 \\ 1000 \end{pmatrix} = \begin{pmatrix} 1101 \\ 0010 \\ 0010 \\ 1101 \end{pmatrix}$$

Hence $(R_{pi}{}^{f}) = \begin{pmatrix} 1101 \\ 0010 \\ 0010 \\ 1101 \end{pmatrix}$ with the result array:

	0	1	2	3
f_1	0, 1	0, 1	0, 1	0, 1
f_2	0, 1	0, 1	1, 0	0, 1

giving all possible pairs of functions that will result in the malfunctioning. Suppose that f_1 is good; then since $\#f_1 = \#(g_1 + \bar{g}_2) = 1101$, we find that $\#f_2 = 1111 = \#I$, which is certainly a reasonable way for a malfunctioning circuit to act. Suppose instead that we assume that f_2 is good; then since $\#f_2 = \#(\bar{g}_1 + g_2) = 1011$, we find that

$$\#f_1 = 1001 = \#(g_1 \cdot g_2 + \bar{g}_1 \cdot \bar{g}_2)$$

which is rather an unreasonable way for a circuit to break. Of course the evaluation of alternative malfunctioning explanations ultimately depends on the *electronic design* of the circuit. Similarly the circuits g_1, g_2, and F could be examined.

EXERCISES

(*a*) For the circuit-maintenance problem discussed above explore the possibilities of malfunctioning circuits g_1, g_2, and F that can explain the erroneous output $\#E_m$.

(*b*) For the circuit-synthesis problem discussed above suppose that the following circuits have already been built: $h_1 = A_2 + A_3$, $h_2 = A_1 + A_3$, $f_1 = g_1 + g_2$, and $f_2 = g_2 + g_3$, with F as described above. How can the circuits g_1, g_2, and g_3 be constructed so that when wired in as shown in Fig. 14-5 the function E described above will result?

14-7. Additional Topics†

a. Logical Interpretation of Antecedence and Consequence Solutions, and Examples. The majority of logical problems involve given or accepted premises, hypotheses, rules, or other logical relationships which are the given equations and essentially comprise the statement of the problems. There are two types of solutions to a set of given equations, the *antecedence* solutions and the *consequence* solutions. Antecedence solutions are hypotheses or theories from which the given equations can be deduced; consequence solutions can be deduced from the given equations. In other words, the truth of the antecedence solutions is sufficient for the truth of the given equations, but the truth of the consequence solutions is necessary for the truth of the given equations. If the given equations are true, then the consequence solutions are true, while the antecedence solutions may or may not be true; but the truth of the given equations can be deduced from the hypotheses embodied in the antecedence solutions, this latter being the method for theory construction. In terms of our above notation the function E represents the given equations and the functions F_a and F_c the antecedence and consequence solutions, respectively.

First Example. Suppose that the Lower Slobbovian army had three kinds of companies: rifle companies A_1, machine-gun companies A_2, and submachine-gun companies A_3. Consider, in addition, the following pieces of equipment: a light machine gun X_1, a scout car X_2, and a submachine gun X_3. Now suppose that

† The material of Additional Topics *b*, *c*, and *d* was developed by Neely F. J. Matthews and that of Additional Topics *e* and *f* by William Ruthven Smith when they were graduate students of the author's at The George Washington University (see W. R. Smith, Master's thesis, June, 1959, and N. F. J. Matthews, Master's thesis, June, 1959).

military intelligence has determined that the distinguishing features that identify the three types of companies in terms of the equipment they carry are as follows:

$$\begin{aligned} A_1 &= X_1 \cdot \bar{X}_3 + X_2 \cdot X_3 \\ A_2 &= X_2 \cdot \bar{X}_3 + \bar{X}_1 \cdot X_3 \\ A_3 &= \bar{X}_1 \cdot \bar{X}_2 + X_2 \cdot X_3 \end{aligned}$$

It happened that in a certain field an observer noticed that soldiers were carrying either machine guns or submachine guns, or both, he could not be sure, and there was no scout car—i.e., the observer reported $(X_1 + X_3) \cdot \bar{X}_2$. Suppose in addition that it was known that a submachine-gun company never accompanies a machine-gun company, that is, $A_3 \rightarrow \bar{A}_2$. The problem then is: Assuming that at least one is present in the field, what companies of the Lower Slobbovian army are in the field? In more precise form the problem is: What are the consequence solutions to the given intelligence equations, of the form

$$(X_1 + X_3) \cdot \bar{X}_2 = f(A_1, A_2, A_3)$$

under the constraint $A_3 \rightarrow \bar{A}_2$? To solve this problem, find

$$(F_{jk}) = \begin{pmatrix} 1011 & 0011 \\ 0100 & 1100 \end{pmatrix} \qquad (E_{ki}) = \begin{pmatrix} 0000 & 1000 \\ 0100 & 0000 \\ 0010 & 0000 \\ 0001 & 0000 \\[6pt] 0000 & 0010 \\ 1000 & 0000 \\ 0000 & 0001 \\ 0000 & 0100 \end{pmatrix}$$

Hence
$$(R_{ji}) = \begin{pmatrix} 0011 & 1101 \\ 1100 & 0010 \end{pmatrix}$$

and
$$\#f = 1100 \quad 0010$$

However, at least one company must be in the field—that is, $A_1 + A_2 + A_3 = I$ is a second constraint. Since $\#(A_1 + A_2 + A_3 = I) = 0111\ 1111$ and $\#(A_3 \rightarrow \bar{A}_2) = 1111\ 1100$, we find that

$$\#f = \_100 \quad 00\_\_\_ = A_1 \cdot \bar{A}_2 \cdot \bar{A}_3$$

In other words, from the observer report, together with the given intelligence information, it can be deduced that only A_1, a rifle company, is in the field.

Second Example. In biochemistry an enzyme often cannot be isolated easily, and therefore experiments performed with one enzyme often will involve several others. Consequently a combination of reactions must be observed. In such a complex situation the ordinary, simple logical analysis usually employed in experimental sciences is found inadequate. In these cases our logical computational methods can be extremely useful for evaluating experimental results and planning future experiments to yield the maximum information.

Suppose that a chemist is studying enzymes A_1, A_2, A_3 in relation to reactions X_1, X_2, X_3. Suppose that he has done the following four experiments: (E_1) A solution containing none of A_1, A_2, A_3 produced reaction X_2, but not X_1 and not X_3. (E_2) The solution contained A_1 and either A_2 or A_3, or both; he could not be sure. The reaction was neither X_2 nor both X_1 and X_3. (E_3) The solution had A_2 but not A_1 or else did not have A_2 but had A_3. Reactions X_1 and X_2 occurred, or reaction X_3 occurred, but X_1 did not. (E_4) The solution was obtained from a source that had A_3, and A_1 or A_2 or both, or else had neither A_1 nor A_3; the solution turns color, which means that X_1 does not take place or that both X_2 and X_3 do.

The antecedence problem is: What theories about the enzymes associated with each reaction will explain the experimental results? The consequence problem is: What combinations of enzymes are necessary for each reaction to take place?

The experiments and their designation numbers with respect to $b[A_1,A_2,A_3,X_1,X_2,X_3]$ are as follows:

E_1: $\bar{A}_1 \cdot \bar{A}_2 \cdot \bar{A}_3 = \bar{X}_1 \cdot X_2 \cdot \bar{X}_3$

0111 1111 0111 1111 1000 0000 0111 1111 0111 1111 0111 1111 0111 1111 0111 1111

E_2: $A_1 \cdot (A_2 + A_3) = \bar{X}_2 \cdot \overline{(X_1 \cdot X_3)}$

0001 0101 0001 0101 1110 1010 1110 1010 0001 0101 1110 1010 1110 1010 1110 1010

E_3: $\bar{A}_1 \cdot A_2 + \bar{A}_2 \cdot A_3 = X_1 \cdot X_2 + \bar{X}_1 \cdot X_3$

1101 0001 1101 0001 1101 0001 0010 1110 0010 1110 1101 0001 0010 1110 0010 1110

E_4: $A_3 \cdot (A_1 + A_2) + \bar{A}_1 \cdot \bar{A}_3 = \bar{X}_1 + X_2 \cdot X_3$

1010 0111 0101 1000 1010 0111 0101 1000 1010 0111 0101 1000 1010 0111 1010 0111

Thus $\#(E_1 \cdot E_2 \cdot E_3 \cdot E_4) =$

0000 0001 0001 0000 1000 0000 0000 1000 0000 0100 0100 0000 0010 0010 0010 0010

For the antecedence problem the form of the solution is $X_1 = f_1$, $X_2 = f_2$, $X_3 = f_3$. Recall for antecedence solutions of this special form that $(E_{ki}) = (R_{ji})$. Hence

$$(E_{ki}) = (R_{ji}) = \begin{pmatrix} 0000 & 0001 \\ 0001 & 0000 \\ 1000 & 0000 \\ 0000 & 1000 \\ & \\ 0000 & 0100 \\ 0100 & 0000 \\ 0010 & 0010 \\ 0010 & 0010 \end{pmatrix}$$

from which the solutions are read.

RESULT ARRAY

$\#f_1$	0	1	1, 0	1	1	0	1, 0	0
$\#f_2$	1	0	1, 1	0	1	0	1, 1	0
$\#f_3$	0	1	1, 1	0	0	1	1, 1	0

Thus there are four sets of antecedence solutions of the form $X_s = f_s$, namely,

$$X_1 = f_1 = A_1 \cdot \bar{A}_3 + \bar{A}_1 \cdot A_3, \text{ or } \bar{A}_3 \cdot (A_1 + A_2) + \bar{A}_1 \cdot \bar{A}_2 \cdot A_3,$$
$$\text{or } \bar{A}_3 \cdot (A_1 + A_2) + \bar{A}_1 \cdot A_3, \text{ or } A_1 \cdot \bar{A}_3 + \bar{A}_1 \cdot \bar{A}_2 \cdot A_3$$

with $\quad X_2 = f_2 = \bar{A}_1 \quad$ and $\quad X_3 = f_3 = A_1 \cdot \bar{A}_2 + \bar{A}_1 \cdot A_2$

The first two sets are logically independent. The last two sets of solutions imply that the reactions are not independent but that $X_2 \cdot X_3 \rightarrow X_1$ and $X_1 \cdot X_2 \cdot X_3 = 0$, respectively. For the consequence problem, the form of the solution and corresponding designation numbers with respect to $b[f_1,f_2,f_3,X_1,X_2,X_3]$ are as follows:

$\#(X_1 \rightarrow f_1) =$
1111 1111 0101 0101 1111 1111 0101 0101 1111 1111 0101 0101 1111 1111 0101 0101
$\#(X_2 \rightarrow f_2) =$
1111 1111 1111 1111 0011 0011 0011 0011 1111 1111 1111 1111 0011 0011 0011 0011
$\#(X_3 \rightarrow f_3) =$
1111 1111 1111 1111 1111 1111 1111 1111 0000 1111 0000 1111 0000 1111 0000 1111
$\#[(X_1 \rightarrow f_1) \cdot (X_1 \rightarrow f_2) \cdot (X_3 \rightarrow f_3)] =$
1111 1111 0101 0101 0011 0011 0001 0001 0000 1111 0000 0101 0000 0011 0000 0001

Hence†

$$(F_{jk}) = \begin{pmatrix} 1000 & 0000 \\ 1100 & 0000 \\ 1010 & 0000 \\ 1111 & 0000 \\ & \\ 1000 & 1000 \\ 1100 & 1100 \\ 1010 & 1010 \\ 1111 & 1111 \end{pmatrix} \quad \text{and} \quad (R_{ji}) = \overline{(\bar{F}_{jk}) \otimes (E_{ki})} = \begin{pmatrix} 0000 & 0001 \\ 0001 & 0001 \\ 1000 & 0001 \\ 1001 & 1001 \\ & \\ 0000 & 0101 \\ 0101 & 0101 \\ 1000 & 0101 \\ 1111 & 1111 \end{pmatrix}$$

This gives the result array:

$\#f_1$	1010	11	1	1111	11	1010	1	Φ
$\#f_2$	1111	10	1	1010	11	1100	1	Φ
$\#f_3$	1100	11	1	1100	10	1111	1	Φ

The most significant set of solutions for this form is the minimum set, i.e., the solutions with the most zeros; we choose

$$\#f_1 = 0111 \quad 1010 = \#[\bar{A}_3 \cdot (A_1 + A_2) + \bar{A}_1 \cdot A_3]$$
$$\#f_2 = 1010 \quad 1010 = \#\bar{A}_1$$
$$\#f_3 = 0110 \quad 0110 = \#[A_1 \cdot \bar{A}_2 + \bar{A}_1 \cdot A_2]$$

or

$$X_1 \rightarrow \bar{A}_3 \cdot (A_1 + A_2) + \bar{A}_1 \cdot A_3$$
$$X_2 \rightarrow \bar{A}_1$$
$$X_3 \rightarrow A_1 \cdot \bar{A}_2 + \bar{A}_1 \cdot A_2$$

b. Boolean Matrix Equations in Three Values. Until now we have been considering Boolean matrices the elements of which can be either 0 or 1. Suppose now that the elements of the matrix can take on three values, say 0, 1, and 2. However, we must define our operations for these values. Suppose that we define the operations + and · as in Sec. 12-11*d*, namely,

+	0	1	2
0	0	1	2
1	1	1	2
2	2	2	2

and

·	0	1	2
0	0	0	0
1	0	1	1
2	0	1	2

with the $^{-}$ as $\bar{0} = 1$, $\bar{1} = 2$, and $\bar{2} = 0$. Then the matrix equation

$$(a_{ik}) \otimes (x_{kj}) = (b_{ij})$$

means

$$b_{ij} = \sum_k a_{ik} \cdot x_{kj}$$

†$(F_{jk}) = (F_{kj})^T$.

where the summation and product are computed as above. Using the analogy of the solution to the two-valued matrix equations, we shall find an (x'_{kj}) solution such that, if (x_{kj}) is also a solution, then $(x_{kj}) \rightarrow (x'_{kj})$. Here we use the → in the sense $0 \rightarrow 1 \rightarrow 2$, $0 \rightarrow 0$, $1 \rightarrow 1$, $2 \rightarrow 2$. Also $(p_{kj}) \rightarrow (q_{kj})$ if $p_{kj} \rightarrow q_{kj}$ for each k and j. In order to determine our (x'_{kj}), let us first observe that for a particular (a_{ik}) and (b_{ij}) the greatest value that x_{kj} can take so that $a_{ik} \cdot x_{kj} \rightarrow b_{ij}$ is as follows (where "greatest" is in terms of our → relation):

a_{ik}	012	012	012
b_{ij}	000	111	222
x_{kj}	200	221	222

For example, if $a_{ik} = 0$ and $b_{ij} = 0$, then $0 \cdot 2 = 0 \rightarrow 0$ so that x_{kj} can be 0, 1, 2; the remaining eight cases are treated similarly. In terms of our three-valued operations we may write (omitting the subscripts) $x = \overline{a + \bar{a}} + \overline{(b + \bar{\bar{b}})} \cdot \bar{a} + b$. Now consider a new operation, denoted by c as derived from this relation:

$$\begin{array}{cc|ccc} & & & b & \\ & \mathsf{c} & 0 & 1 & 2 \\ \hline & 0 & 2 & 2 & 2 \\ a & 1 & 0 & 2 & 2 \\ & 2 & 0 & 1 & 2 \end{array}$$

(Note that this operation is *not commutative.*) Based on this definition, consider the matrix operation

$$(a_{ki}) \copyright (b_{ij}) = (c_{kj})$$

defined by
$$c_{kj} = \prod_i (a_{ki} \mathsf{c}\, b_{ij}) = \min(a_{ki} \mathsf{c}\, b_{ij})$$

Then
$$(x'_{kj}) = (a_{ik})^T \copyright (b_{ij})$$

is the desired solution. If a solution to our original matrix equation exists, then

$$(a_{ik}) \otimes [(a_{ik})^T \copyright (b_{ij})] = (b_{ij})$$

Otherwise
$$(a_{ik}) \otimes [(a_{ik})^T \copyright (b_{ij})] \rightarrow (b_{ij})$$

c. Problems of Types 1 *and* 2 *in Three Values.* With the above in mind we can extend our matrix methods for antecedence and consequence solutions to this three-valued logic. The matrix (E_{ki}) is now formed from a designation number with three possible values for each position, and similarly for (F_{kj}). The matrix (R_{ji}) must now have *only a single* 2 *in each column,* in order to perform the necessary pseudo-permutation. Then the *fundamental equation*

$$(F_{kj}) \otimes (R_{ji}) = (E_{ki})$$

still holds. For example, let $F = (\overline{f_1 + \bar{f}_2}) \cdot X$, $f_1 = \overline{A_1 \cdot \bar{A}_2}$, and $f_2 = \overline{A_1 + \bar{A}_2}$. Then

$\#f_1 =$	012	012	012	012	012	012	012	012	012
$\#f_2 =$	000	111	222	000	111	222	000	111	222
$\#X =$	000	000	000	111	111	111	222	222	222
$\#F =$	000	000	000	110	000	110	220	000	120

	012	345	678
$\#A_1 =$	012	012	012
$\#A_2 =$	000	111	222
$\#f_1 =$	122	120	111
$\#f_2 =$	220	000	120

Hence $(F_{kj}) = \begin{pmatrix} 000 & 000 & 000 \\ 110 & 000 & 110 \\ 220 & 000 & 120 \end{pmatrix}$ and $(R_{ji}) = \begin{pmatrix} 000 & 002 & 000 \\ 000 & 200 & 002 \\ 002 & 020 & 000 \\ \\ 000 & 000 & 000 \\ 000 & 000 & 200 \\ 000 & 000 & 000 \\ \\ 000 & 000 & 000 \\ 200 & 000 & 020 \\ 020 & 000 & 000 \end{pmatrix}$

Then

$$(F_{kj}) \otimes (R_{ji}) = \begin{pmatrix} 000 & 000 & 000 \\ 100 & 101 & 011 \\ 200 & 202 & 022 \end{pmatrix} = (E_{ki})$$

or

$$\#E = 000 \quad 000 \quad 000 \quad 100 \quad 101 \quad 011 \quad 200 \quad 202 \quad 022$$

and

$$E = \{\overline{(A_1 + \bar{A}_1)} \cdot \overline{(A_2 + \bar{A}_2)} + \overline{\overline{(A_1 \cdot \bar{A}_1)}} \cdot \overline{(A_2 + \bar{\bar{A}}_2)} + \overline{\overline{(\bar{A}_1 \cdot \bar{\bar{A}}_1)}} \cdot \overline{(\bar{A}_2 + \bar{\bar{A}}_2)}\} \cdot X$$

Thus it is clear how to solve problems of type 1, that is, substitution problems. For type 2 problems we solve the fundamental equation by finding

$$(F_{jk}) = (R_{ji}) \circledcirc (E_{ik})$$

and then substitute into

$$(F_{kj}) \otimes (R_{ji}) = (E'_{ki})$$

to determine whether or not $(E'_{ki}) = (E_{ki})$, that is, whether or not a solution to the circuit-design problem actually exists.

Suppose that

$$f_1 = \overline{A_1 \cdot \bar{A}_2} \qquad f_2 = \overline{A_1 + \bar{A}_2}$$

and

$$E = \{\overline{(A_1 + \bar{A}_1)} \cdot \overline{(A_2 + \bar{A}_2)} + \overline{\overline{(A_1 \cdot \bar{A}_1)}} \cdot \overline{(A_2 + \bar{\bar{A}}_2)} + \overline{\overline{(\bar{A}_1 \cdot \bar{\bar{A}}_1)}} \cdot \overline{(\bar{A}_2 + \bar{\bar{A}}_2)}\} \cdot X$$

as above; then, to find F, we compute

$$\begin{pmatrix} 000 & 002 & 000 \\ 000 & 200 & 002 \\ 002 & 020 & 000 \\ \\ 000 & 000 & 000 \\ 000 & 000 & 200 \\ 000 & 000 & 000 \\ \\ 000 & 000 & 000 \\ 200 & 000 & 020 \\ 020 & 000 & 000 \end{pmatrix} \circledcirc \begin{pmatrix} 012 \\ 000 \\ 000 \\ \\ 012 \\ 000 \\ 012 \\ \\ 000 \\ 012 \\ 012 \end{pmatrix} = \begin{pmatrix} 012 \\ 012 \\ 000 \\ \\ 222 \\ 000 \\ 222 \\ \\ 222 \\ 012 \\ 000 \end{pmatrix} = (F_{jk})$$

Since f_1 and f_2 are constrained, we find F with respect to the constrained basis,

$$\begin{array}{rcccccccccccccccccccc}
\#f_1 = & 012 & \cancel{0} & 1 & \cancel{2} & \cancel{0} & 12 & & 012 & \cancel{0} & 1 & \cancel{2} & \cancel{0} & 12 & & 012 & \cancel{0} & 1 & \cancel{2} & \cancel{0} & 12 \\
\#f_2 = & 000 & \cancel{1} & 1 & \cancel{1} & \cancel{1} & 22 & & 000 & \cancel{1} & 1 & \cancel{1} & \cancel{2} & 22 & & 000 & \cancel{1} & 1 & \cancel{1} & \cancel{2} & 22 \\
\#X = & 000 & \cancel{0} & 0 & \cancel{0} & \cancel{0} & 00 & & 111 & \cancel{1} & 1 & \cancel{1} & \cancel{1} & 11 & & 222 & \cancel{2} & 2 & \cancel{2} & \cancel{2} & 22 \\
\hline
\#F = & 000 & & 0 & & & 00 & & 110 & & 0 & & & 10 & & 220 & & 0 & & & 20
\end{array}$$

which is now the same as the F used above.

There are *two* other equivalent forms to the fundamental equation which follow directly, since (R_{ji}) is a pseudopermutation:

$$(\bar{F}_{kj}) \otimes (R_{ji}) = (\bar{E}_{ki}) \qquad \text{and} \qquad (\bar{\bar{F}}_{kj}) \otimes (R_{ji}) = (\bar{\bar{E}}_{ki})$$

These give rise to three types of logical problems: *antecedence, intercedence,* and *consequence* problems. Can you find equations analogous to the fundamental formulas for type 2 problems corresponding to antecedence, intercedence, and consequence problems?

d. M-valued Post† *Logics.* The three-valued logic discussed above has the property that $+(a,b) = \max(a,b)$ and $\cdot(a,b) = \min(a,b)$, where $0 < 1 < 2$ as usual. Also the $^{-}$ permuted cyclically 0, 1, 2. We can generalize these operations to $(m + 1)$ values $0, 1, 2, \ldots, m$ and define $+(a,b) = \max(a,b)$, $\cdot(a,b) = \min(a,b)$, and $^{-}$ to permute cyclically. For example, for five-valued logic we would have

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

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

$\bar{0} = 1$
$\bar{1} = 2$
$\bar{2} = 3$
$\bar{3} = 4$
$\bar{4} = 0$

All the above results can be directly carried over, provided that the c operation becomes

c	0	1	2	3	· · ·	m
0	m	m	m	m	· · ·	m
1	0	m	m	m	· · ·	m
2	0	1	m	m	· · ·	m
3	0	1	2	m	· · ·	m
4	0	1	2	3	· · ·	m
· · ·	· · ·	· · ·	· · ·	· · ·	· · ·	· · ·
m	0	1	2	3	· · ·	m

or for five-valued logic

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

† From E. L. Post (see References below).

References. For discussions of the logical considerations associated with many-valued logics see, for example:

Post, E. L.: Introduction to a General Theory of Elementary Propositions, *Am. J. Math.*, vol. 43, pp. 163–185, 1921.

Rosenbloom, P.: "The Elements of Mathematical Logic," pp. 51–69, Dover Publications, New York, 1950.

Rosser, J. B., and A. R. Turquette: Axiom Schemes for M-valued Propositional Calculi, *J. Symbolic Logic*, vol. 10, pp. 61–82, 1945.

For other approaches to many-valued logic in circuit design see:

Berlin, R. D.: Synthesis of N-valued Switching Circuits, *IRE Trans. on Electronic Computers*, vol. EC-7, no. 1, pp. 52–57, March, 1958.

Gilbert, E. N.: N-terminal Switching Circuits, *Bell System Tech. J.*, vol. 30, pp. 668–688, 1951.

Lee, C. Y.: Switching Functions for an N-dimensional Cube, *Trans. AIEE*, vol. 73, pp. 289–291, September, 1954.

——— and W. H. Chen: Several Valued Combinational Switching Circuits, *Communs. and Electronics*, no. 25, pp. 278–283, July, 1956.

e. Problems of Type 3 *in an M-valued Post Logic.* The many-valued generalization of the Θ matrix product is particularly useful for problems of type 3 in an $(m + 1)$-valued Post logic. We define

$$(c_{kj}) = (a_{ki}) \,\Theta\, (b_{ij})$$

such that c_{kj} is m if the kth row of (a_{ki}) *is identical to* the jth column of (b_{ij}); otherwise c_{kj} is 0. With this in mind we have from the fundamental equation that

$$(R_{ji}) = (F_{kj})^T \,\Theta\, (E_{ki})$$

from which the solutions can be found that have a single m in each column. For example, if $m + 1 = 3$ (that is, three-valued logic) and $F = \overline{(f_1 + \bar{f}_2)} \cdot X$ and

$$E = \{\overline{(A_1 + \bar{A}_1)} \cdot \overline{(A_2 + \bar{A}_2)} + \overline{\overline{(A_1 \cdot \bar{A}_1)}} \cdot \overline{(A_2 + \bar{\bar{A}}_2)} + \overline{\overline{(A_1 \cdot \bar{\bar{A}}_1)}} \cdot \overline{(\bar{A}_2 + \bar{\bar{A}}_2)}\} \cdot X$$

then

$$(R_{ji}) = \begin{pmatrix} 012 \\ 012 \\ 000 \\[1ex] 000 \\ 000 \\ 000 \\[1ex] 011 \\ 012 \\ 000 \end{pmatrix} \Theta \begin{pmatrix} 000 & 000 & 000 \\ 100 & 101 & 011 \\ 200 & 202 & 022 \end{pmatrix} = \begin{pmatrix} 200 & 202 & 022 \\ 200 & 202 & 022 \\ 022 & 020 & 200 \\[1ex] 022 & 020 & 200 \\ 022 & 020 & 200 \\ 022 & 020 & 200 \\[1ex] 000 & 000 & 000 \\ 200 & 202 & 022 \\ 022 & 020 & 200 \end{pmatrix}$$

The $3 \times 5 \times 5 \times 3 \times 5 \times 3 \times 5 \times 3 \times 3 = 151{,}875$ sets of solutions include the set $f_1 = \overline{A_1 \cdot \bar{A}_2}$, $f_2 = \overline{A_1 + \bar{A}_2}$.

It is interesting to note the relation between the Θ and the © operations at this point, namely,

$$(a_{ki}) \,\Theta\, (b_{ij}) = \{(a_{ki}) © (b_{ij})\} \cdot \{(\overline{a_{ki}}) © (\overline{b_{ij}})\} \cdot \{(\overline{\overline{a_{ki}}}) © (\overline{\overline{b_{ij}}})\}$$

for our three-valued logic. Can you prove this? An analogous relation holds for an $(m + 1)$-valued Post logic.

f. All Solutions to Boolean Matrix Equations in a Many-valued Logic. Let $\textcircled{M}$ be *any matrix-type logical operation* [that is, for $(c_{pr}) = (a_{pq}) \textcircled{M} (b_{qr})$, each c_{pr} is some logical function of corresponding (qth) elements of the pth row of (a_{pq}) and the rth column of (b_{qr})]. Then if

$$(a_{ik}) \textcircled{M} (x_{kj}) = (b_{ij})$$

all matrix solutions, if any exist, can be found from

$$(S_{mj}) = \{(a_{ik}) \textcircled{M} (b[X_1, \ . \ . \ . \ ,X_\kappa])\}^T \Theta (b_{ij})$$

by the usual R-matrix type of procedure, generalized as required. The operations Θ and $\textcircled{c}$ can be considered as specialized examples of the general matrix operation $\textcircled{M}$.

PART 4

LOGICAL DESIGN OF DIGITAL-COMPUTER CIRCUITRY

CHAPTER 15

SERIAL ARITHMETIC OPERATIONS

15-1. Introduction

Generalized Concept of a Computer. As we have observed in previous chapters, only one word at a time can be transmitted between the computer's memory and its computing unit. In the computing unit are temporary storage locations or registers which receive the words from the memory; we have previously noted some of these registers, such as the accumulator and the instruction register. As a word is transmitted from a memory location to a register or back, it passes through logical *gating networks* of the types considered in Part 3. The computations take place in these logical gating networks; i.e., in these gating networks the numbers are transformed or combined according to the operation being executed. Thus we can conceive a computer as consisting of a temporary storage and a more permanent storage, with gating networks in between (see Fig. 15-1). Words are transmitted from the permanent storage, the computer memory, through the gating networks to the temporary storage, the computing-unit registers, and then back again through the gating networks. This chapter is concerned with the gating networks by means of which numbers are transformed according to the computations involved in arithmetic operations.

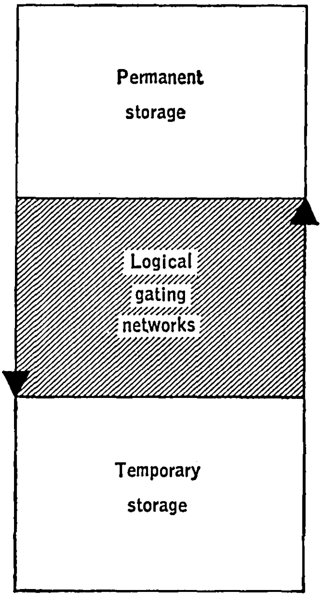


FIG. 15-1. Generalized concept of a computer.

Serial and Parallel Computers. In a *serial computer* a word is transmitted serially in a train, one bit at a time, to and from the computer's memory through the gating networks (see Fig. 15-2). In Chap. 2 such a sequence of events was described where the accumulator was the register involved and the adder was the gating network. *The length of time during which* 1 *bit is transmitted from the memory, through the gating network, and into the register is called a unit time interval.* Thus, if a word has *w* bits, it will take *w* unit time intervals to transmit it to a register, or back, in a serial computer.

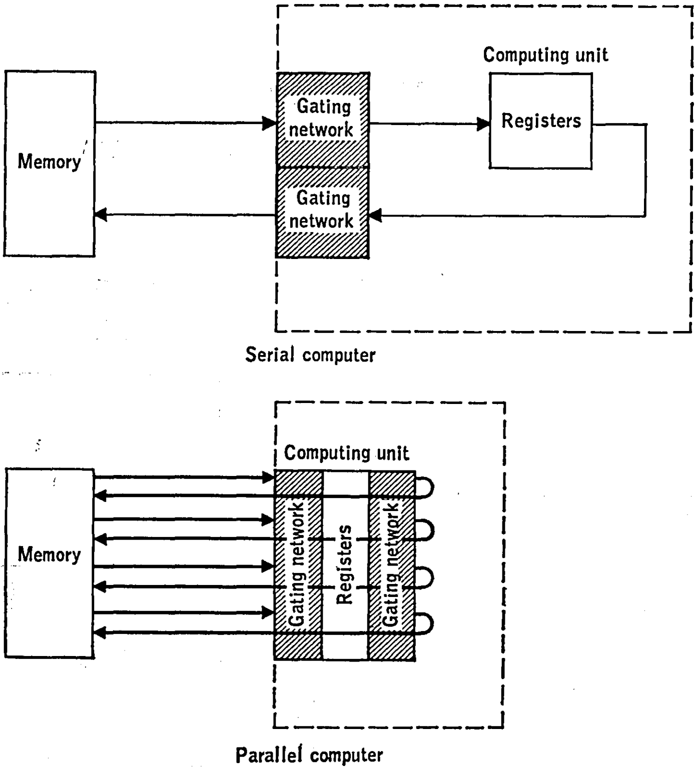


FIG. 15-2. Serial vs. parallel computer.

In a *parallel computer,* on the other hand, the bits of a word can be transmitted through the gating network from the memory to a register or back all at the same time, in parallel (see Fig. 15-2). Thus it takes only *one unit time interval* to transmit an entire word to the registers through the gating network. Thereby parallel computers can be many times faster than serial computers. On the other hand this increased speed is gained at a cost of additional hardware; for a serial computer need handle only 1 bit at a time, whereas a parallel computer must be able to handle all the bits of a word simultaneously. Thus a serial computer will in general be much less expensive and slower than a parallel computer.

Topics to Be Covered. In this chapter and the next we shall consider both serial and parallel arithmetic units, both binary and decimal. In general there are several ways for solving each of the problems posed. We take the point of view that it is better to consider one method in detail than to survey all methods known. It is felt that the specific method chosen in each case will contain all the essential features common to other methods.

This chapter starts with a section on simple, common computer components such as shift registers, counters, adders, and so forth. Next we discuss concepts involving the synchronous operation of a computer, which are basic to the subsequent material. The serial arithmetic unit is then discussed, where we have used a combination adder-subtractor unit rather than the more popular input-output complementary-adder technique. The approach is pedagogically sound, since the subtractor is almost identical to the adder. It is felt that this will help the student to grasp the fundamentals more rapidly. The exercises then proceed systematically to build up the input-output complementer technique, which now becomes easy to master. The excess-three decimal-coded binary scheme is used to illustrate the decimal arithmetic unit. Although there are other decimal techniques (see Additional Topics, Sec. 15-7), excess-three can be treated as a simple extension of binary methods. In addition the parallel decimal arithmetic unit can be treated as a variation of the parallel binary arithmetic unit. This is done in the following chapter.

15-2. Common Computer Components

Synchronous Dynamic Flip-Flop and Shift Register. As will be illustrated in Part 5 of this text, flip-flops as described in Chap. 2 can be constructed from electronic circuits with two stable states. However, they can also be constructed by means of *and* and *or* gates. Consider, for example, the *synchronous dynamic flip-flop.* This is called a dynamic flip-flop because it has a recirculation loop (as considered above); if a unit signal voltage is recirculating, the flip-flop is said to be in the unit state; otherwise it is in the zero state. Besides the recirculation C the input is a signal pulse A and a clock pulse P. Here the synchronous aspect enters; for a unit signal pulse in A occurring synchronously with the clock pulse will turn the flip-flop to its unit state; a zero signal pulse in A occurring synchronously with the clock pulse will turn the flip-flop to its zero state; when no clock pulse appears, the flip-flop will not change its state. Thus this flip-flop acts like a 1-bit storage cell, storing whatever signal pulse was last synchronized with the clock pulse. Thus the function table becomes

$$\begin{array}{ll} \#A & = 0101 \quad 0101 \\ \#C & = 0011 \quad 0011 \\ \#P & = 0000 \quad 1111 \\ \hline \#C' & = 0011 \quad 0101 = C \cdot \bar{P} + A \cdot P \end{array}$$

and the circuit diagram appears in Fig. 15-3. Note that the recirculation is called C on one side of the unit delay line, while it is called C' on the other (why?).

A set of such flip-flops *cascaded together*, i.e., the output of one becoming the input signal to the next, is called a *shift register*. When a clock pulse appears, each flip-flop takes on the state that the previous flip-flop had. Hence, if C_{i-1} is the state (output) of the previous flip-flop and C_i that of the flip-flop under consideration, then the function table for the new state of the flip-flop C_i' is

$$\begin{array}{lll}
\#C_{i-1} & = 0101 & 0101 \\
\#C_i & = 0011 & 0011 \\
\#P & = 0000 & 1111 \\
\hline
\#C_i' & = 0011 & 0101 = C_i \cdot \bar{P} + C_{i-1} \cdot P
\end{array}$$

Figure 15-4 illustrates such a shift register. Observe that we had to move the unit delay line round from its position in Fig. 15-3 (why?). The accumulator in a serial computer is an example of such a shift register, where, as each result leaves the adder and enters the accumulator, the contents of the latter are shifted each unit time period to make room for the entering result bit.

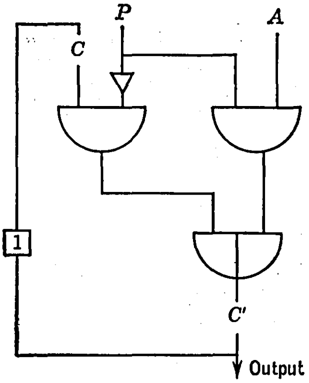


FIG. 15-3. Flip-flop circuit.

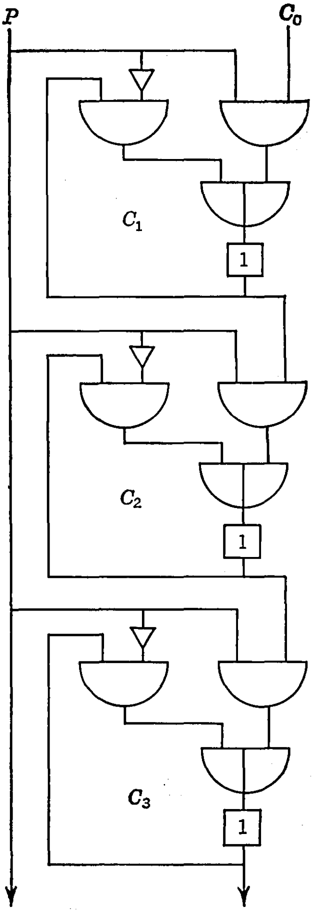


FIG. 15-4. Shift register.

Cascaded Counters. We have considered the design of counters previously. For example, we know how to design a counter that counts 0, 1, 2, . . . , p, 0, 1, 2, . . . , p, 0, 1, . . . , and so forth. Now suppose that we had two such counters and connected them together so that *whenever the first increased past* p *to* 0, *it would increase the count of the second by* 1. Then the second counter would register how many times the first counter passed p. Thus suppose that the first counter is counting clock pulses (i.e., increases by 1 for each unit signal voltage of its input clock pulse); then if the count on the first counter is a_0 and the count on the second is a_1, the total number of clock pulses counted will be $a_1q^1 + a_0q^0$ (where $q = p + 1$, $q^0 = 1$). Similarly, if the second counter is connected in this way to still a third counter of the same kind, and if the counts on the three counters are a_2, a_1, and a_0, then the number of clock pulses counted will be $a_2q^2 + a_1q^1 + a_0q^0$, and so forth. Thus it becomes clear that the counts of counters *cascaded* in this way represent the digits of a number to the base, or radix, q (see Fig. 15-5). Note, however, that when a component counter reaches the count of q, although it sends a signal to increase the count on the next counter, *its own count* will really be zero. This of course is necessary since $a_i < q$.

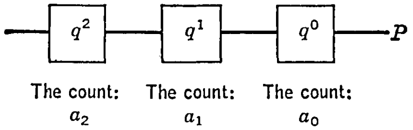


FIG. 15-5. Cascading counters of radix q.

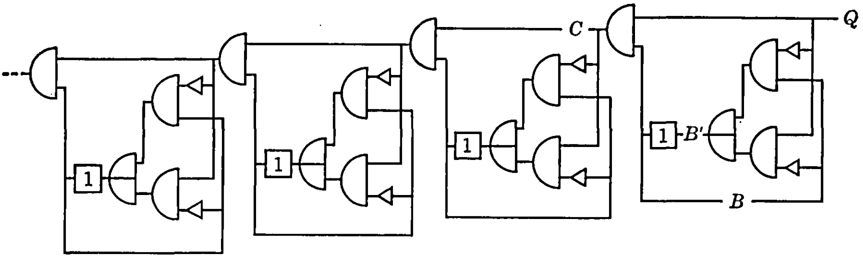


FIG. 15-6. Cascading counters for $q = 2$.

For instance, if $q = 4$, the counts of three cascaded counters would sequence as follows:

```
1st counter: 0 1 2 3 0 1 2 3 0 1 2 3 0 1 2 3 0 1 2 3 · · ·
                     ↓       ↓       ↓       ↓
2d counter:  0 0 0 0 1 1 1 1 2 2 2 2 3 3 3 3 0 0 0 0 · · ·
                                             ↓
3d counter:  0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 · · ·
```

As a first example, let us consider the simple case of $q = 2$. Let Q denote the signal that increases the count of a component counter, C the signal generated by the component counter to increase the next counter, and B the count of the component counter itself (where only 1 bit is required in this case). Thus we have for the function table

$$\begin{array}{ll} B & 0101 \\ \underline{Q} & \underline{0011} \\ B' & 0110 = B \cdot \bar{Q} + \bar{B} \cdot Q \\ C & 0001 = B \cdot Q \end{array}$$

Figure 15-6 is a diagram of four such counters cascaded together.

As the second example, let us consider $q = 4$. Then our component counter will be a 2-bit counter that counts 0, 1, 2, 3, 0, 1, 2, 3, 0, . . .

and that generates the triggering signal for the next cascaded counter. For instance, we would have

$$
\begin{array}{lll}
 & 0123 & 4567 \\
B_1 & 0101 & 0101 \\
B_2 & 0011 & 0011 \\
Q & 0000 & 1111 \\
\hline
B'_1 & 0101 & 1010 = B_1 \cdot \bar{Q} + \bar{B}_1 \cdot Q \\
B'_2 & 0011 & 0110 = B_2 \cdot \bar{Q} + B_1 \cdot \bar{B}_2 \cdot Q + \bar{B}_1 \cdot B_2 \\
C & 0000 & 0001 = B_1 \cdot B_2 \cdot Q
\end{array}
$$

This counter counts in binary, as shown in Table 15-1. However, there is no necessity for a counter to count in the usual binary sequence; so let us consider a counter that counts in the so-called *cyclic sequence* shown in Table 15-1. In this case we have

TABLE 15-1. BINARY AND CYCLIC COUNTING SEQUENCES

Decimal	Binary		Cyclic	
	B_2	B_1	B'_2	B'_1
0	0	0	0	0
1	0	1	0	1
2	1	0	1	1
3	1	1	1	0
0	0	0	0	0
. . .	. . .	. . .	. . .	. . .

$$
\begin{array}{lll}
B_1 & 0101 & 0101 \\
B_2 & 0011 & 0011 \\
Q & 0000 & 1111 \\
\hline
B'_1 & 0101 & 1100 = B_1 \cdot \bar{Q} + \bar{B}_2 \cdot Q \\
B'_2 & 0011 & 0101 = B_2 \cdot \bar{Q} + B_1 \cdot Q \\
C & 0000 & 0010 = \bar{B}_1 \cdot B_2 \cdot Q
\end{array}
$$

Two such counters cascaded together are shown in Fig. 15-7. Comparing this with Fig. 15-6, it is seen that our cyclic counter is somewhat simpler. If we compare our cyclic counter with a binary counter for $q = 4$, we see that the cyclic counter is much simpler.

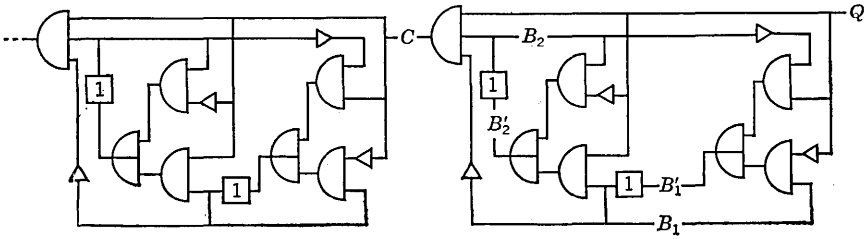


FIG. 15-7. Cascading cyclic counters with $q = 4$.

Parallel Counter. Observe that in cascaded counters, after the signal Q (which is to increase the count) enters the first counter stage, its effect must propagate sequentially through all the stages before the actual count finally appears. In practice this can consume a relatively large portion of time. There is, however, a method of arranging the cascaded counters so that the count appears immediately, with no propagation time lag. Consider, for example, a straight binary counter with $q = 2$. Observe that stage a_i should change (from 0 to 1 or 1 to 0) only when stages $a_{i-1}, a_{i-2}, \ldots, a_0$ *are all units* and a signal Q is received—for example,

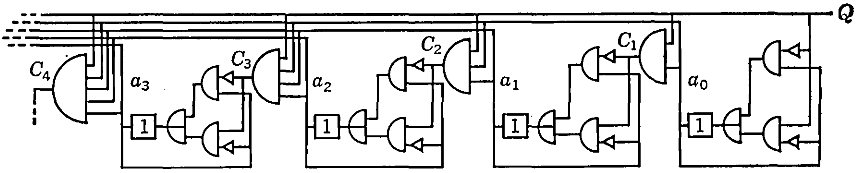


FIG. 15-8. Parallel counter composed of cascaded stages.

$$001111 + 1 = 010000$$

Hence, to have a parallel counter we simply make the signal C_i that changes the a_i stage, as follows:

$$C_i = a_{i-1} \cdot a_{i-2} \cdot \cdot \cdot a_0 \cdot Q$$

Figure 15-8 illustrates these connections. Figure 15-8 should be compared with Fig. 15-6.

EXERCISES

(*a*) Design a flip-flop circuit that has, besides its recirculation input and the clock input, two other inputs A and B such that a unit pulse in A synchronized with the clock pulse will put the circuit in its unit state, while a unit pulse in B synchronized with the clock pulse will put the circuit in its zero state. Assume that unit pulses in both A and B will never occur simultaneously (i.e., a constraint).

(*b*) Design a shift register using the flip-flops of Exercise *a*.

(*c*) Design a cascaded counter using units with $q = 8$ (that is, 3 bits) where each unit counts in the following sequence: 000, 001, 011, 010, 110, 111, 101, 100, 000, ... Compare 2 of these units with the circuit found by cascading 6 of the $q = 2$ units.

(*d*) Design a cascaded parallel counter with $q = 4$, where each of the counter stages counts in cyclic. (HINT: The solution to this problem will be related to Fig. 15-7 in the same way Fig. 15-8 is related to Fig. 15-6.)

15-3. Common Computer Components (*Continued*)

Simple Adders and Subtractors. Consider the addition of two binary numbers as ordinarily accomplished by hand. First the least significant bits are added, forming the least significant sum bit and a carry bit. Then the next least significant bits are added, with the carry, forming the next sum and carry bits, and so on. A *serial* adder is a circuit whose inputs are the 2 corresponding bits from the two numbers to be added, and the carry bit from the preceding sum; the outputs are the new sum and carry bits. The two numbers are fed serially and synchronously into the adder (in the manner illustrated in Chap. 2), and the carry recirculates. Figure 15-9 shows four phases in the addition of 10101 and 10011, where the last carry becomes the most significant bit of the completed sum. If the ith bit of each of the numbers is denoted by A_i and B_i, respectively, and the ith bit of the sum by R_i and the generated carry by C_i, then the function table for a serial adder becomes

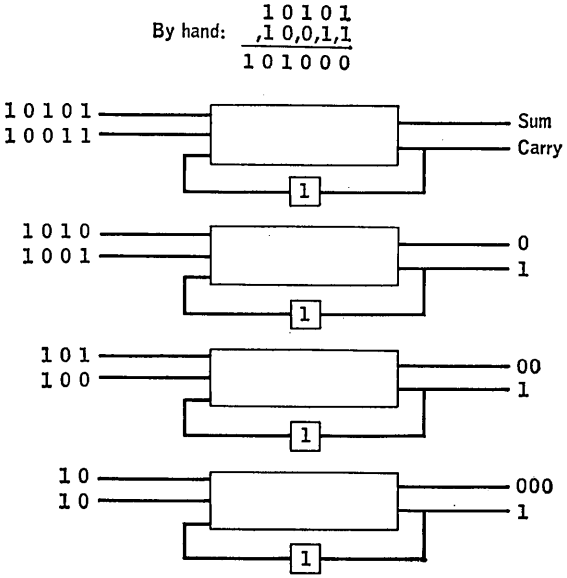


FIG. 15-9. Several stages in the serial addition of 10101 and 10011.

A_i	0101	0101	
B_i	0011	0011	
C_{i-1}	0000	1111	
R_i	0110	1001	$= (A_i \cdot \bar{B}_i + \bar{A}_i \cdot B_i) \cdot \bar{C}_{i-1} + (\bar{A}_i \cdot \bar{B}_i + A_i \cdot B_i) \cdot C_{i-1}$
C_i	0001	0111	$= A_i \cdot B_i + (A_i + B_i) \cdot C_{i-1}$

The design of a serial adder is shown in Fig. 15-10.

For a different design of the serial adder note that the designation numbers of R_i and $\bar{C}_i$ (1110 1000) differ in only two positions. This suggests that we might construct R_i from $\bar{C}_i$. To find all ways of doing this, let $R_i = \bar{C}_i \cdot X + Y$, and solve for X and Y. On performing the computations, find that

$$X = 011\Phi \quad 1\Phi\Phi\Phi \qquad \text{and} \qquad \#Y = 0\Phi\Phi0 \quad \Phi001$$

One simple solution is $X = A_i + B_i + C_{i-1}$ and $Y_i = A_i \cdot B_i \cdot C_{i-1}$.

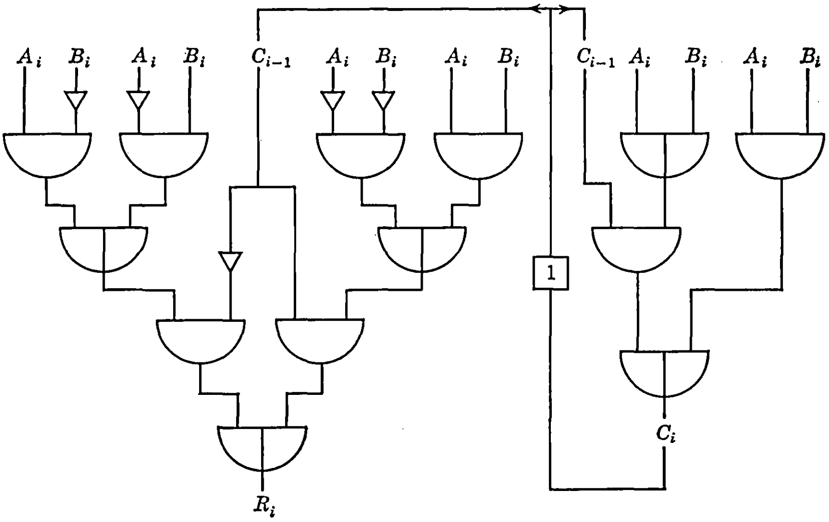


FIG. 15-10. Serial adder.

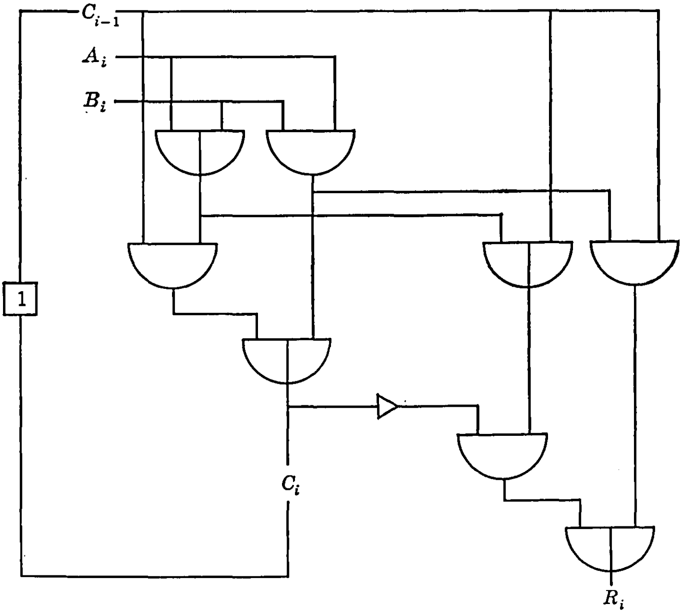


FIG. 15-11. Serial adder.

Hence we have the serial adder of Fig. 15-11. Perhaps an even simpler adder could be constructed if we note that between R_i and $\bar{C}_i$ only five elementary products† are needed. This observation results in the adder of Fig. 15-12.

† Or five elementary sums.

In a similar way let us consider a serial subtractor, where for the present we shall assume that the result of the subtraction is always positive. Our function table becomes (where D_i now represents the subtractive carry)

A_i	0101	0101
B_i	0011	0011
D_{i-1}	0000	1111
R_i	0110	1001
D_i	0010	1011

Note that the difference (R_i) in subtraction and the sum in addition are identical; only the carries differ. In the case of subtraction we have

$$D_i = \bar{A}_i \cdot B_i + (\bar{A}_i + B_i) \cdot D_{i-1}$$

Note that D_i has the same form as C_i, *except that $\bar{A}_i$ replaces A_i* (where the notation D_{i-1} replaces C_{i-1}). Hence, if we complemented A_i, we could *use the same circuitry* to generate both the subtraction and addition carries. Thus, if the sum were desired, we would send A_i into the carry-generating circuitry; if the difference were desired, we would send $\bar{A}_i$ into this circuitry. For example, suppose that, if the signal P is a unit, the sum is to be formed; if P is a zero, the difference is to be formed. This can be arranged as in Fig. 15-13.

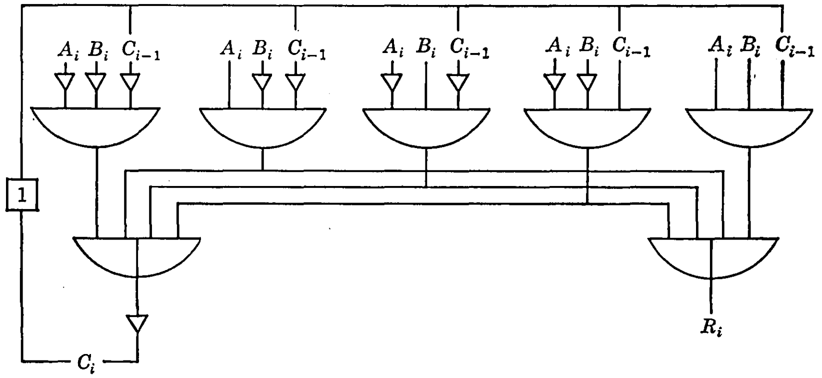


FIG. 15-12. Serial adder.

The subtractor can easily be formed in still another way from the adder. For if in the circuit of Fig. 15-12 we added an *and* gate forming $\bar{A}_i \cdot B_i \cdot C_{i-1}$, we could form D_i as well as C_i. Then either C_i or D_i would be recirculated depending on whether the sum or the difference is to be formed. Again this can be controlled by a signal P (see Exercise *f* of this section).

Logical Operations Derived from Adder. Consider the function tables for serial logical addition, logical multiplication, logical equalization, and logical nonequalization, respectively:

A_i	0101	A_i	0101	A_i	0101	A_i	0101
B_i	0011	B_i	0011	B_i	0011	B_i	0011
L_i	0111	M_i	0001	E_i	1001	F_i	0110

Circuits for these functions are easily formed. However, observe the addition carry of the last paragraph, namely, $C_i = 0001\ 0111$, and note that, when C_{i-1} is held at unity, C_i becomes simply L_i; and when C_{i-1} is held at zero, M_i is obtained. Similarly observe R_i of addition: when C_{i-1} is held at unity, E_i is obtained; when C_{i-1} is held at zero, F_i is obtained. Thus by fixing C_{i-1} we can enable our serial adder to double as a logical unit.

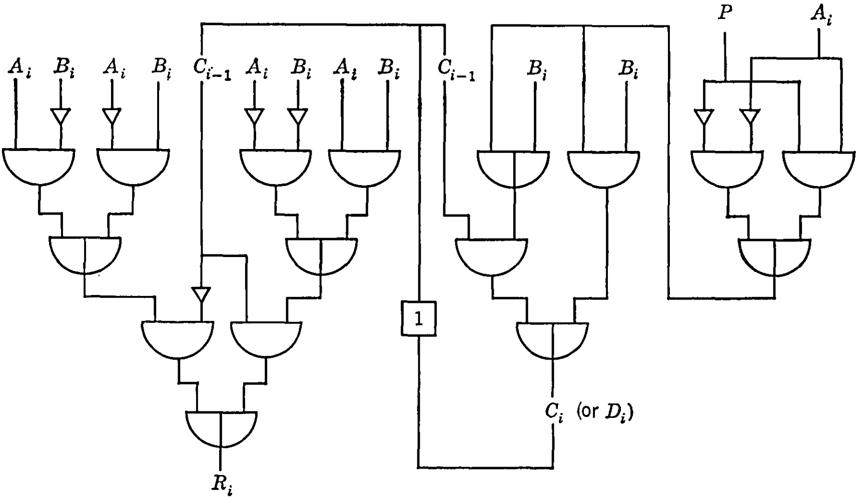


FIG. 15-13. Adder and subtractor combination forms sum for $P = 1$, forms difference for $P = 0$.

To see how the carry may be controlled, suppose that K_L, K_M, K_E, and K_F are control signals that take on unit signal voltage whenever the logical operation indicated by the respective subscript is to be accomplished. Let C_i^* represent the controlled carry. Then since only one of K_L, K_M, K_E, and K_F can be "on" at a particular time, we have the following function table of all possible input conditions for C_{i-1}^*:

K_L	01000	01000	
K_M	00100	00100	
K_E	00010	00010	
K_F	00001	00001	
C_{i-1}	00000	11111	
C_{i-1}^*	01010	11010	$= K_L + K_E + \bar{K}_M \cdot \bar{K}_F \cdot C_{i-1}$

On the other hand the R_i output of the adder must be modified so that it will produce the logical results as well. Calling this modified output R_i^*, we find the function table:

R_i	0101	0101	0101	0101	0101	
C_i	0011	0011	0011	0011	0011	
K_L	0000	1111	0000	0000	0000	
K_M	0000	0000	1111	0000	0000	
K_E	0000	0000	0000	1111	0000	
K_F	0000	0000	0000	0000	1111	
R_i^*	0101	0011	0011	0101	0101	$= R_i \cdot \bar{K}_L \cdot \bar{K}_M + C_i \cdot (K_L + K_M)$

Hence we have the modified input-output adder as in Fig. 15-14. When none of the signals K_L, K_M, K_E, or K_F is on, the adder will behave as an ordinary adder; but if any one of these signals is on, the adder will behave as a logical unit.

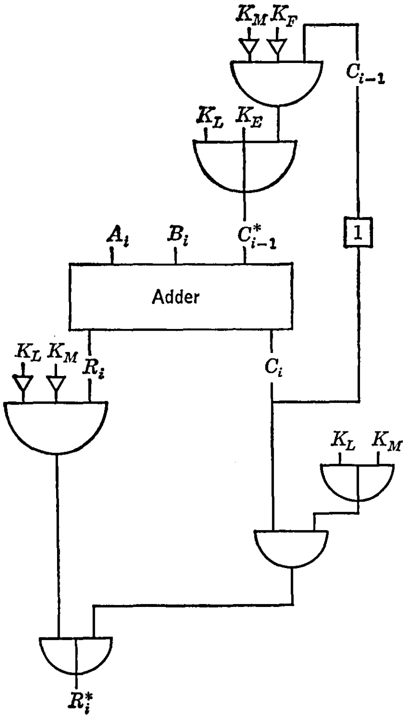


FIG. 15-14. Use of adder for logical operations.

Subtractive Complementer. The *subtractive complement* of an integer N to the base q and of p significant figures is defined as $C(N) = q^p - N$. For decimal numbers

$$C(N) = 10^p - N$$

for binary numbers

$$C(N) = 2^p - N$$

For example, if $N = 10110$ (base 2), then $p = 5$, whence we should have $C(N) = 100000 - 10110 = 01010$. If $N = 77$ (base 10), then $p = 2$, whence $C(N) = 100 - 77 = 23$.

The importance of the subtractive complement is related to the process of subtraction. Recall that, in the subtractor designed above, we assumed that the result would always be positive. Since our subtractor formed $A - B$, this was the same as assuming that $A > B$. But suppose that $B > A$, what would the circuit we designed do then? It would of course still form $A - B$. An illustration in terms of decimal numbers may clarify this situation. If we perform the subtraction 87 − 64, we find 23 as the result; on the other hand, if asked to perform 64 − 87, we could call −23 the result. But if we perform 64 − 87 digit by digit, we find

$$\begin{array}{r} 64 \\ -\ \underline{87} \\ \cdots\ 9977 \end{array}$$

or, on keeping only the two least significant figures, find 77. This is what our subtractor would do. However, notice that

$$C(77) = 100 - 77 = 23$$

which is the correct result when preceded by a minus sign.

Thus the importance of the subtractive complementer becomes clear: *for a subtractor designed to form $A - B$, the case of $B > A$ is treated by taking the subtractive complement of the result.* To see this more precisely, note that the "cutting off" of the 9s and retaining only the 77 is equivalent to forming $100 + 64 - 87$, that is, $q^p + A - B$. Thus, if $B > A$, then, as desired,

$$\begin{aligned} &C(q^p + A - B) \\ &\quad = q^p - (q^p + A - B) \\ &\quad = -(A - B) = B - A \end{aligned}$$

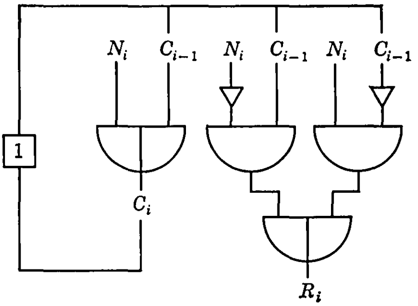


FIG. 15-15. Subtractive complementer.

To design a subtractive complementer, observe that we are merely subtracting N from all zeros; the 2^p in $C(N) = 2^p - N$ is taken care of by terminating the calculations. The function table therefore becomes (where C_i is the recirculating carry)

0	0000	
N_i	0101	
C_{i-1}	0011	
R_i	0110	$= N_i \cdot \bar{C}_{i-1} + \bar{N}_i \cdot C_{i-1}$
C_i	0111	$= N_i + C_{i-1}$

Hence the circuit appears as in Fig. 15-15.

EXERCISES

(*a*) Design a serial adder by forming C_i from $\bar{R}_i$; that is, let $C_i = \bar{R}_i \cdot X + Y$.

(*b*) How can an adder be made to subtract? [HINT: Note that $A + C(B) = A + 2^p - B = 2^p + (A - B)$. Thus, if the unit in the 2^p position is ignored, $A - B$ is formed.] Illustrate with an example.

(*c*) If subtraction is accomplished as in Exercise *b*, what happens when $B > A$? How can this be remedied? Illustrate with an example.

(*d*) Observe that $C(N)$ for binary N can be formed by complementing N (that is, forming $\bar{N}$) and adding 1. For instance if $N = 10110$, observe that $01001 + 1 = 01010$ (where + stands for ordinary addition), which is $C(10110)$. How can you explain this?

(*e*) Design a circuit which has as inputs two serial binary numbers A and B (like the adder) and whose output will be a unit if $A > B$, zero otherwise.

(*f*) Design an adder-subtractor combination analogous to the adder of Fig. 15-12, so that the combination forms the sum $A + B$ when a signal P is a unit and the difference $A - B$ when P is a zero.

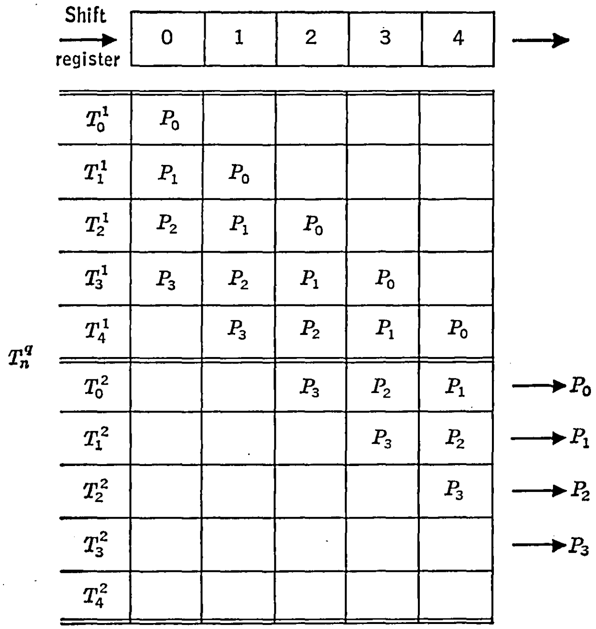
15-4. Preliminaries to Synchronous Operation

Phase Pulses. In Sec. 2-3 we distinguished four phases in the execution of an instruction. For example, consider the execution of an addition instruction that reads "Add (α) to (β), put the sum into γ, and take the next instruction from δ." During phase 1, (β) would usually be transmitted into the accumulator; during phase 2, (α) would be transmitted into the arithmetic unit and added to (acc) (see Figs. 2-8 to 2-13); during phase 3, (acc) would be transmitted into γ; during phase 4 the next instruction would be transmitted into the instruction register. In order that the different parts of the computer know which phase of operation is in progress, *phase pulses* are emitted by the control unit. We denote by $F1$, $F2$, $F3$, and $F4$ the phase pulse signals that are units only during phases 1, 2, 3, and 4, respectively. We shall consider the generation of these signals in detail in Chap. 17. It suffices here, however, to note that phase 4 occurs in three-, two-, and one-address systems, when the next instruction is brought into the instruction register. Phase 2 of course also occurs in the three-, two-, and one-address systems, where a modified definition of phase 2 must be made for certain instructions of the two- and one-address systems.

Unit Time Intervals and Minor Cycles. In Sec. 2-4 we discussed clock pulses and briefly described how the entire computer was synchronized by these pulses. These pulses define units of time that are indivisible in so far as the computer is concerned. Thus a clock pulse is said to be a *unit time interval* in width (for example, the interval between t_i and t_{i+1} in Fig. 11-18). *A minor cycle of a computer is a measure of time in terms of unit time intervals; its numerical value is equal to the number of positions in the accumulator shift register.* That is, once a word has been selected from the memory, a minor cycle is the number of unit time intervals required for the word to be propagated down the accumulator shift register, at the rate of one shift per unit time interval. As we shall see below, the number of positions in the accumulator is usually greater than the number of bits in a word.

Phases 1, 3, and 4 usually consist of one minor cycle each; phase 2 will consist of one minor cycle for addition and subtraction but more than one minor cycle for multiplication and division, as we shall see below. The minor cycles are numbered within each phase, starting from 1; the unit time intervals are numbered within each minor cycle, starting from 0. Thus, within a phase, a particular unit time interval can be indicated by denoting its number and minor cycle. The nth unit time interval of the qth minor cycle is denoted by $T_n{}^q$. This notation will be used extensively below. A signal pulse that is a unit only during the nth unit time interval of each minor cycle is denoted by T_n. Such pulses are generated by a counter that counts the clock pulses themselves. The synchronous operation of a computer is directed by these pulses.

The Continuously Shifting Register. A topic that will prove of importance in discussing the serial arithmetic unit is an analysis of the continuously shifting register. The accumulator is such a register, for, as we have seen in Sec. 2-4, as each bit of a word is transmitted from the memory into the accumulator the contents of the accumulator must shift one stage, to make room for the entering bit. Thus after each unit time interval the contents of the accumulator must shift. For example, consider a 5-bit continuously shifting register accepting a 4-bit word $P3\,P2\,P1\,P0$. Label the stages of the shift register by 0, 1, 2, 3, and 4, from left to right as indicated in Fig. 15-16; then the contents of this register as a function of time is shown, where it is assumed that $P0$ enters stage 0 at $T_0{}^1$.



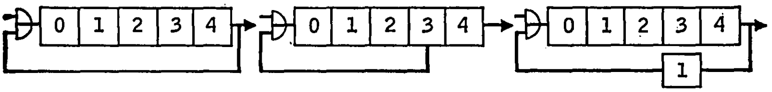
Shift register	0	1	2	3	4	
T_0^1	P_0					
T_1^1	P_1	P_0				
T_2^1	P_2	P_1	P_0			
T_3^1	P_3	P_2	P_1	P_0		
T_4^1		P_3	P_2	P_1	P_0	
T_0^2			P_3	P_2	P_1	→ P_0
T_1^2				P_3	P_2	→ P_1
T_2^2					P_3	→ P_2
T_3^2						→ P_3
T_4^2						

FIG. 15-16. A 4-bit word shifting through a 5-bit continuously shifting register.

Next consider the more complicated situation where the output of the shift register is recirculated, as in Fig. 15-17. In the timing rows we have omitted the P, and the numbers represent the respective bits of our 4-bit word. In (1), straight recirculation, note that each minor cycle (after the first) is identical. That is, at $T_0{}^q$, $P0$ is at stage 0, $P1$ is at stage 4, $P2$ is at stage 3, etc. On the other hand consider the case where the recirculation loop is shortened as in (2). Here our word "precesses," so to speak: e.g., at $T_0{}^1$, $P0$ is at position 0; at $T_0{}^2$, $P0$ is at position 1; at $T_0{}^3$, $P0$ is at position 2, etc.; at $T_0{}^q$, $P0$ is at position $q - 1$ (for $q \leq 5$). In (3) a lengthened precession is considered: $P0$ is at position 0 at $T_0{}^1$, at position 0 at $T_1{}^2$, at position 0 at $T_2{}^3$; so in general $P0$ is at position 0 at T^q_{q-1} for $q \leq 5$.

Let us generalize these observations. Let p represent a stage number of the shift register and s a bit number of a word. Then for the situation pictured in (3) of Fig. 15-17, that of lengthened precession, we find that $n - (q - 1) - p = s$. Thus, given $T_n{}^q$, we can determine s for a particular p or, conversely, p for a given s. For example, for $T_3{}^2$ and $p = 1$

(1) Straight recirculation (2) Shortened precession (3) Lengthened precession



T_0^1	0						0						0					
T_1^1	1	0					1	0					1	0				
T_2^1	2	1	0				2	1	0				2	1	0			
T_3^1	3	2	1	0			3	2	1	0			3	2	1	0		
T_4^1	-	3	2	1	0		0	3	2	1	0		-	3	2	1	0	
T_0^2	0	-	3	2	1	→ 0	1	0	3	2	1	→ 0	-	-	3	2	1	→ 0
T_1^2	1	0	-	3	2	→ 1	2	1	0	3	2	→ 1	0	-	-	3	2	→ 1
T_2^2	2	1	0	-	3	→ 2	3	2	1	0	3	→ 2	1	0	-	-	3	→ 2
T_3^2	3	2	1	0	-	→ 3	0	3	2	1	0	→ 3	2	1	0	-	-	→ 3
T_4^2	-	3	2	1	0	-	1	0	3	2	1	→ 0	3	2	1	0	-	-
T_0^3	0	-	3	2	1	→ 0	2	1	0	3	2	→ 1	-	3	2	1	0	-
T_1^3	1	0	-	3	2	→ 1	3	2	1	0	3	→ 2	-	-	3	2	1	→ 0
T_2^3	2	1	0	-	3	→ 2	0	3	2	1	0	→ 3	0	-	-	3	2	→ 1
T_3^3	3	2	1	0	-	→ 3	1	0	3	2	1	→ 0	1	0	-	-	3	→ 2
T_4^3	-	3	2	1	0	-	2	1	0	3	2	→ 1	2	1	0	-	-	→ 3
T_0^4	0	-	3	2	1	→ 0	3	2	1	0	3	→ 2	3	2	1	0	-	-
T_1^4	1	0	-	3	2	→ 1	0	3	2	1	0	→ 3	-	3	2	1	0	-
T_2^4	2	1	0	-	3	→ 2	1	0	3	2	1	→ 0	-	-	3	2	1	→ 0
T_3^4	3	2	1	0	-	→ 3	2	1	0	3	2	→ 1	0	-	-	3	2	→ 1
T_4^4	-	3	2	1	0	-	3	2	1	0	3	→ 2	1	0	-	-	3	→ 2

FIG. 15-17. Continuously shifting register with various recirculation situations.

we find $3 - 1 - 1 = 1$, as can be easily confirmed from (3). However, for $T_2{}^3$ and $p = 3$ we find $2 - 2 - 3 = -3$, which clearly is not one of the values taken by s since $s = 0, 1, 2$, and 3. In such a situation let us note that if there were w stages to the shift register, that is, $p = 0, 1, 2, \ldots, w - 1$, then $T_n{}^q = T_{n+w}^{q-1}$; that is, $T_n{}^q$ and T_{n+w}^{q-1} represent the same time period. For our example, $w = 5$; hence we replace $T_2{}^3$ by $T_7{}^2$, whence we find $7 - 1 - 3 = 3$, which is correct.

Next consider the situation in (2) of Fig. 15-17. Here note that $n + (q - 1) - p = s$. For example, for $T_1{}^3$ and $p = 0$, $1 + 1 - 0 = 2$, which can be checked in the proper chart of Fig. 15-17. However, if $p = 4$, for example, we would have $1 + 1 - 4 = -2$. The solution to this dilemma is to note that $T_1{}^2 = T_6{}^1$, whence $6 + 0 - 4 = 2$, which is correct.

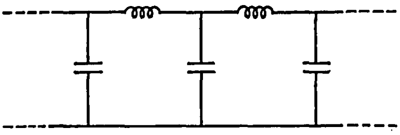


FIG. 15-18. Delay line.

In general it is now easy for us to see that if Δ is the number of stages lengthened (and $-\Delta$ is the number of stages shortened) then

$$n - \Delta(q - 1) - p = s \tag{15-1}$$

where we shall often have to use the fact that

$$T_n{}^q = T_{n+w}^{q-1} = T_{n-w}^{q+1} \tag{15-2}$$

to obtain the proper solution.

These formulas are primarily of use in discussing serial multiplication and division. The recirculation of the accumulator for addition and subtraction is analogous to (1) of Fig. 15-17, where the formulas are not really necessary.

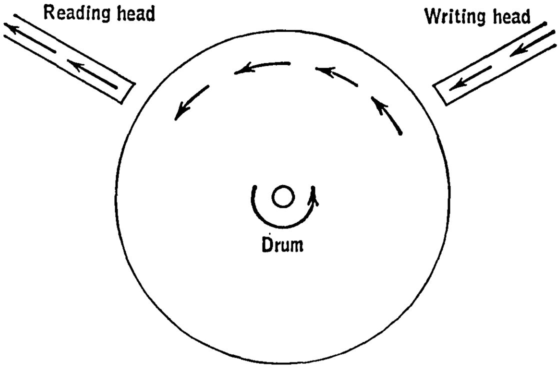


FIG. 15-19. Using a drum for a delay-line shift register.

The continuously shifting register is important, not only because the serial accumulator is one, but also because all *delay-line registers* are of this type. Until now we have considered registers as *shift* registers. However, delay lines can be used for the temporary storage required for registers. Previously we have used the concept of a *unit* delay when discussing recursive functions; delay lines that are used as part of a register usually afford *many time periods of delay.* Figure 15-18 represents a possible method of constructing a delay line. For the purposes of logical design the most important difference between a shift register and a delay-line register is that the pulses in a shift register may be shifted or not at any time, depending on the occurrence of a controlling shift pulse, but that, as is clear from Fig. 15-18, the pulses in a delay-line register keep moving all the time. A delay-line register is thus like a shift register that is *always* shifting. The advantage of a delay-line shift register is its often low cost as compared with a shift register. For example, a small computer with a drum could use part of the drum for a delay-line shift register (see Fig. 15-19). Here the pulses are put on the rotating drum in sequence and are picked up after the delay of rotation at the reading head. A great disadvantage of delay-line registers is that they are not appropriate for parallel computers, as will be seen below.

EXERCISES

(*a*) The Pedagac has a minor cycle of 23 unit time intervals. Design a cascaded counter that will generate T_0, T_1, . . . , T_{22}. (See Fig. 18-16, pages 612, 613.)

(*b*) Draw the detailed logical diagram for the shift register of Fig. 15-17, and indicate where the connections would be made for each of the recirculation situations of the figure. Show that the recirculation loops are connected correctly in each case.

(*c*) For $\Delta = -1$, $w = 5$, locate $P2$ at $T_2{}^2$; locate $P2$ at $T_0{}^2$ (that is, find p). Check results with the charts of Fig. 15-17.

(*d*) For $\Delta = +1$, $w = 5$, locate $P2$ at $T_4{}^3$; locate $P2$ at $T_0{}^3$. Check results with the charts of Fig. 15-17.

(*e*) For $\Delta = +1$, $w = 5$, what bit will be located at $p = 2$ at $T_2{}^3$? (That is, find s.) Can you explain the result obtained? Check result with the charts of Fig. 15-17.

(*f*) For the arrangement shown in Fig. 15-19, how can shortened and lengthened precession be obtained without using additional heads? (HINT: Use the drum for only part of the register.)

15-5. Serial Arithmetic Unit: Addition and Subtraction

The Addition of Words. We have shown in Sec. 15-3 how a serial adder is designed. Now consider the problem of incorporating such an adder into the arithmetic unit of a serial computer. In general only one word at a time can be transmitted from the memory to the arithmetic unit. Hence in adding two words, say A and B, the word B is usually transmitted to the accumulator first, during phase 1. Then during phase 2 this word is recirculated from the accumulator through the adder at the same time as A is being transmitted from the memory through the adder, and in this way A and B are added and the sum put into the accumulator. In the three- or four-address systems it is convenient during phase 1 to transmit into the accumulator through the adder; here B is added to all zeros, which is accomplished by gating the recirculation from the accumulator with $F2$, which is off during phase 1 (see Fig. 15-20). However, the other input to the adder must be open during both phase 1 and phase 2 and is therefore gated with $F1 + F2$. For one-address addition systems the contents of the accumulator is taken as one of the arguments directly, phase 1 being eliminated. The two-address system may work either way.

The sign bits of the words are not sent through the adder; they are recorded in flip-flops (not shown in Fig. 15-20) until needed. Therefore $P1$ is the first bit of a word to enter the accumulator, since the $P0$ bit is the sign bit. *It is customary to arrange the timing so that, when a word is transmitted from the memory to the accumulator, its* $P1$ *bit enters the zeroth stage at* $T_1{}^1$. For the present we shall assume that it takes no appreciable time for a pulse to go through the adder or, in fact, through any gating levels. Of course this is not actually so, as we shall see when we discuss the electronic design of the gates. Thus, in phase 1, bit $B1$ enters the zeroth stage of the accumulator shift register at $T_1{}^1$; in phase 2, the sum of $B1$ and $A1$, namely, $R1$, enters the zeroth stage at $T_1{}^1$.

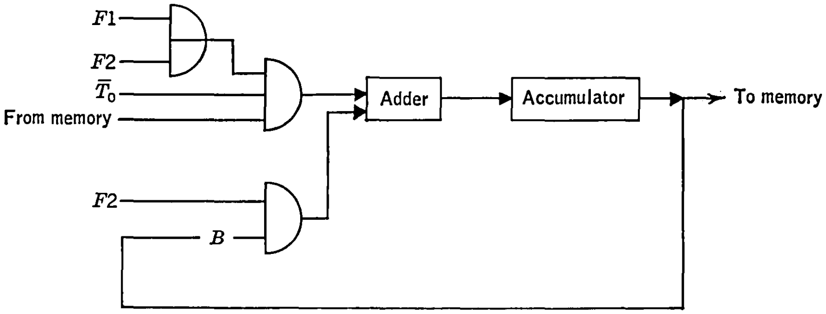


FIG. 15-20. The addition of words.

The Sum and the Difference. Since each word may be either positive or negative, sometimes even when subtraction is called for, the *sum* may be required, and when addition is called for, the *difference* may be required. Let us define some terms in order to facilitate the discussion. *Henceforth we shall mean by addition and subtraction the operation written into any instruction. On the other hand by sum and difference we shall mean that which is carried out by the arithmetic unit in any instance.* Table 15-2 summarizes all possibilities.

TABLE 15-2. SUM AND DIFFERENCE CONDITIONS

S_A	S_B	Operation W	Result P
+	+	Add	Sum
−	+	Add	Difference
+	−	Add	Difference
−	−	Add	Sum
+	+	Subtract	Difference
−	+	Subtract	Sum
+	−	Subtract	Sum
−	−	Subtract	Difference

Our adder-subtractor combination of the previous section forms the sum or difference of A and B depending on whether the sum carry C or the difference carry D is recirculated. The signal P that determines this must itself then depend on three signals: the sign of the number A, denoted by S_A, the sign of the number B, denoted by S_B, and the operation. Let S_A and S_B be 1 for a positive number and 0 for a negative number. Let W represent the operation, such that for addition W is 1 and for subtraction 0. Then from Table 15-2 we can form the following condition-function table for P, where it is recalled that when P is 1 the sum is to be formed, when P is 0 the difference is to be formed:

S_A	0101	0101
S_B	0011	0011
W	0000	1111
P	0110	1001

$$P = (S_A \cdot \bar{S}_B + \bar{S}_A \cdot S_B) \cdot \bar{W} + (S_A \cdot S_B + \bar{S}_A \cdot \bar{S}_B) \cdot W$$

The next problem is to determine the sign of the result. For this we need one additional piece of information if the difference was formed, namely, whether $|A| > |B|$ or $|B| > |A|$. Then the sign conditions are summarized in Table 15-3. Let us suppose for the moment that this can be determined and that, if $|B| > |A|$, then X is 1 and, if $|A| > |B|$, then X is 0. Let S be the sign (1 if positive, 0 if negative) of the sum or difference as the case may be; then the condition-function table for S becomes

TABLE 15-3. SIGN CONDITIONS FOR $A \pm B = C$†

$$\left.\begin{array}{l}(+A) + (+B)\\(+A) - (-B)\end{array}\right\} = (+C)$$

$$\left.\begin{array}{l}(-A) + (+B)\\(-A) - (-B)\end{array}\right\} = \left\{\begin{array}{l}(+C) \text{ for } |B| \geq |A|\\(-C) \text{ for } |B| < |A|\end{array}\right.$$

$$\left.\begin{array}{l}(+A) + (-B)\\(+A) - (+B)\end{array}\right\} = \left\{\begin{array}{l}(+C) \text{ for } |A| \geq |B|\\(-C) \text{ for } |A| < |B|\end{array}\right.$$

$$\left.\begin{array}{l}(-A) + (-B)\\(-A) - (+B)\end{array}\right\} = (-C)$$

† $(+A)$ means that A is positive, $(-A)$ means that A is negative, and similarly for B and C. Note that zero is positive.

										9	10		12			15
S_A	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1
S_B	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1
W	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1
X	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1
S	0	1	0	1	0	1	0	1	1	1	0	0	0	0	1	1

$$S = S_A \cdot \bar{X} + (\bar{S}_B \cdot \bar{W} + S_B \cdot W) \cdot X$$

Note that the case for $A = B$ was not covered. This must be taken care of separately, as we shall see below.

As can be seen from Table 15-3, the only time X is important is when a *difference* is being calculated by the adder-subtractor. For the adder-subtractor described above, the difference calculation always performed is $|A| - |B|$. When $|B| > |A|$, the result is $2^p + |A| - |B|$ and the subtractive complement of the result must be taken. Now if p is taken greater than the length of the word, units would appear to the left of the word. For example, consider 5-bit words with the difference carried to eight places,

$$\begin{array}{r|l} 000 & 01010 \\ -\ 000 & 10110 \\ \hline \cdots\ 111 & 10100 \\ \uparrow\ \ & \end{array}$$

The subtractive complement is 01100, and the result is negative. However, the important point to note is the propagation of units to the left in the result. This will occur whenever the difference is formed with $|B| > |A|$. Hence it is this propagation of units that tells us when $|B| > |A|$ or $|B| < |A|$. Thus a propagated unit can be taken for X. It is wisest to choose a unit in position $p + 2$ (as indicated by the arrow), where p is the number of positions in the word, for position $p + 1$ could possibly become a unit due to normal *overflow*, but not $p + 2$. When X is chosen as this bit, it is easily seen that conditions 9, 10, 12, and 15 will never occur, for here the *sum* of the numbers is being formed. Hence these columns can be dropped, forming a constrained basis. Under this constraint S reduces to

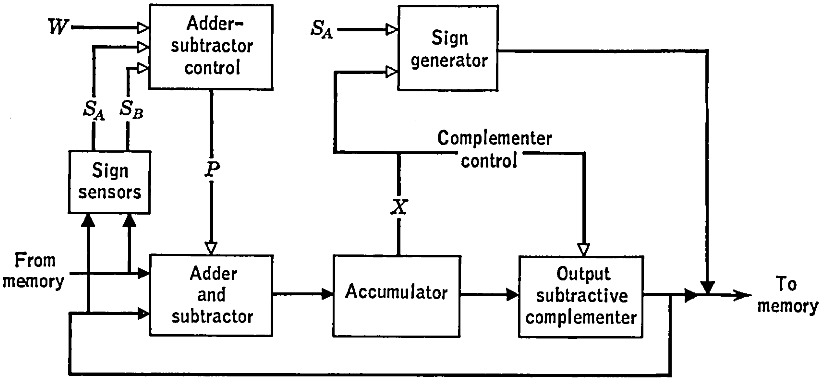


FIG. 15-21. Adder and subtractor combination with controls.

$$S = S_A \cdot \bar{X} + \bar{S}_A \cdot X$$

In Fig. 15-21 we have summarized the developments so far in this section. For the addition and subtraction of words we described an adder-subtractor combination (that formed $|A| + |B|$ and $|A| - |B|$) followed by an elongated accumulator. However, to take care of the case when $|B| > |A|$, we need an output subtractive complementer. Also we need controls to tell the adder combination when to form the sum and when to form the difference, controls to determine the sign of the result, and a control for the output subtractive complementer. This latter control is particularly simple: if a unit is propagated two positions more significant than the most significant (leftmost) position of a word, then the complementer is turned on; otherwise it is left off. The sign bits S_B and S_A that precede the entrance of B and A from the memory are sensed as they enter the arithmetic unit and are not used in the adder itself; the sign of the result is placed in the word just before it goes to the memory. Note that the accumulator must be at least 2 bits longer than the word length itself (why?).

Decimal Addition and Subtraction. In Sec. 3-10 we discussed the concept of decimal-coded binary computer design. Here we shall discuss a particular system of decimal-coded binary which contains the basic ingredients of any such system. The specific circuits used for decimal addition and subtraction of course depend on the particular decimal-coded binary code used. We shall choose a code called the *excess-three* code, in terms of which these operations are especially simple to perform. Table 15-4 gives the excess-three code. If V is a decimal digit, then its excess-three code is found by converting $V + 3$ to binary, which we shall represent by $(V + 3)_b$.

The first obvious advantage of this code concerns the decimal carry. In adding two decimal digits a carry is obtained when the sum of the digits is 10 or greater. In the excess-three code, if we add in the usual binary fashion the codes representing two such digits, we find†

TABLE 15-4. DECIMAL-CODED BINARY IN THE EXCESS-THREE CODE SYSTEM

Decimal	*Excess-three code*
0	0011
1	0100
2	0101
3	0110
4	0111
5	1000
6	1001
7	1010
8	1011
9	1100

$$(U + 3)_b + (V + 3)_b = (U + V)_b + (3 + 3)_b \geq 10_b + 6_b = 16_b$$

but binary 16 has a fifth bit, which corresponds to the decimal carry. This suggests a method for performing the addition of two excess-three binary-coded decimal words. Simply add the corresponding digits (groups of 4 bits) as if they were actually binary numbers; if there is a carry from the previous 4-bit addition result, add this in also. Of

† Recall that if U and V are decimal digits $(U + V)_b = U_b + V_b$, where the subscript b means "converted to binary."

course the binary sum will not be the proper excess-three code and hence must be adjusted accordingly, as will now be considered.

Suppose that we desire to form the following sum, where we use 4-digit numbers as an example:

$$\begin{array}{cccccc} & & N_4 & N_3 & N_2 & N_1 \\ + & & M_4 & M_3 & M_2 & M_1 \\ \hline & R_5 & R_4 & R_3 & R_2 & R_1 \end{array}$$

where N_i, M_i, and R_i are the decimal digits of the arguments and the result. Now, if $N_1 + M_1 \leq 9$, then $N_1 + M_1 = R_1$. However, we desire the digit R_1 in excess-three code, i.e., as $(R_1 + 3)_b$. Suppose that we added the excess-three codes of N_1 and M_1 directly; we would obtain $(N_1 + 3)_b + (M_1 + 3)_b = (N_1 + M_1 + 6)_b = (R_1 + 6)_b$. Thus, to obtain the correct excess-three result, we would form the sum of the excess-three codes of corresponding digits and *subtract* 3; that is,

$$(R_1 + 3)_b = (N_1 + 3)_b + (M_1 + 3)_b - 3_b$$

Next consider $N_1 + M_1 > 9$. In this case R is the least significant figure of the addition. The most significant digit, 1, becomes the carry to the next position (e.g., if $N_1 = 6$, $N_2 = 7$, then $N_1 + N_2 = 13$ and $R_1 = 3$ with the 1 becoming the carry). Thus $R_1 = N_1 + M_1 - 10$, and the desired excess-three code result is $(N_1 + M_1 - 10 + 3)_b$. We observed above that, under these conditions, the binary sum of the excess 3 digits will contain 5 significant bits; we use the fifth bit as a carry, retaining only the other 4. The 4-bit result of the excess-three addition becomes $(N_1 + 3)_b + (M_1 + 3)_b - 16_b = (N_1 + M_1 - 10)_b$. To this we must add 3 to obtain the correct excess-three result:

$$[(N_1 + M_1 - 10) + 3]_b = [(N_1 + 3)_b + (M_1 + 3)_b] - 16_b + 3_b$$

Summarizing, the rules for addition are as follows: *To form the excess-three result of adding (summing) the corresponding digits of two numbers, sum their excess-three codes and then subtract* 3 *if no carry results, or add* 3 *if a carry does result.* The carry increases M_{i+1} by 1, and hence also its excess-three code. Thus the carry is added to the addition of the least significant place of M_{i+1} just as in an ordinary binary adder. For example, consider the following addition:

Decimal		*Decimal-coded binary*		
274		0101	1010	0111
+465		+0111	1001	1000
	Binary sum.......	1101	0011	1111
	Carry record......	No	Yes	No
	Adjustment.......	−0011	+0011	−0011
Result 739		1010	0110	1100

Consider now subtraction. Forming

$$(N_1 + 3)_b - (M_1 + 3)_b = (N_1 - M_1)_b$$

we see immediately that, if $N_1 \geq M_1$, all we need do is add 3 to this result, that is, $R_1 + 3 = (N_1 - M_1 + 3)_b = (N_1 + 3)_b - (M_1 + 3)_b + 3_b$. On the other hand suppose that $M_1 > N_1$; then we desire

$$R_1 = 10 + N_1 - M_1$$

whence $(R_1 + 3)_b = (10 + N_1 - M_1 + 3)_b$. We form

$$\begin{aligned} 2^4{}_b + (N_1 + 3)_b - (M_1 + 3)_b - 3_b &= (16 + N_1 - M_1 - 3)_b \\ &= (10 + N_1 - M_1 + 3)_b \end{aligned}$$

Notice that in this latter case the binary subtraction will produce a carry from the most significant position (which is the reason for the $2^4{}_b$), and hence the carry indicates that the difference between $(N_1 + 3)_b$ and $(M_1 + 3)_b$ must be reduced by 3.

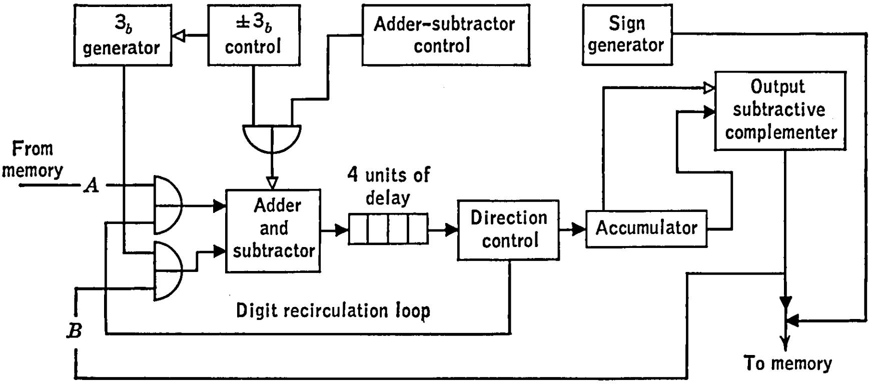


FIG. 15-22. Adder and subtractor combination for excess-three decimal-coded binary words, with controls.

Summarizing, the rules for subtraction are as follows: *To form the excess-three result of subtracting (differencing) corresponding digits, difference their excess-three codes and then add* 3 *if no carry results or subtract* 3 *if a carry does result.* Again any carry is added to $(M_{i+1} + 3)_b$, and the subtraction then takes place in these four bit positions, etc.

Figure 15-22 represents a block diagram of a serial excess-three adder and subtractor and associated control circuits. The 4 units of delay adjacent to the accumulator hold the 4 bits of the sum or difference of the excess-three codes of corresponding digits before the recirculation through the adder for ±3 adjustment. The 3_b generator generates 0011 to be added to or subtracted from the recirculating 4-bit result. During this recirculation no further bits of A or B enter the adder. When the recirculation and ±3 adjustment have been accomplished, the next digits, i.e., the next 4 bits each, of A and of B enter the adder. The process continues alternating: sum or difference next 4 bits, recirculate result adding or subtracting 3, sum or difference next 4 bits, recirculate result, etc. The carry must be recorded in a flip-flop, held during the recirculation, and then inserted just as the next digits of A and of B enter the adder. The $\pm 3_b$ control signal, denoted by P_r, determines whether the 3_b should be summed or differenced during the recirculation, while the adder-subtractor control signal P does this during the processing of the next digits. The signal P is generated in the same way as in the binary case of the previous paragraph; the signal P_r depends on P and on the carry C_D, as described by our above rules. Since P is 1 when the two numbers are being summed and 0 when they are being differenced, the function table for P_r becomes

$$\begin{array}{ll} C_D & 0101 \\ \underline{P} & \underline{0011} \\ P_r & 1001 = C_D \cdot P + \bar{C}_D \cdot \bar{P} \end{array}$$

The sign generator is as in the binary case, except that sometimes the sign itself is coded into 4 bits. The output subtractive complementer is actuated this time by a 9 in the overflow decimal position.

The subtractive complement of an integer N having p significant figures in base ten is $10^p - N$. If the rightmost digits of N are zeros, then $C(N)$ ends in the same number of zeros; thence the rightmost nonzero digit of N is subtracted from 10, and each succeeding leftward digit is subtracted from 9. For example, we have

$$\begin{array}{r} 100000 \\ \underline{55500} \\ 44500 \end{array}$$

Observe that the inverse $\bar{N}_i$ of the ith digit of N represented in excess-three is $15_b - (N_i + 3)_b = 12_b - (N_i)_b = (9 - N_i) + 3_b$, that is, the excess-three representation of $9 - N_i$. Note also that $10 - N_i = (9 - N_i) + 1$. Our subtractive complementer (Fig. 15-23) is designed to pass all rightmost zeros, form $10 - N_i = \bar{N}_i + 1$† for the rightmost nonzero digit, and then form $9 - N_i = \bar{N}_i$ for the leftward digits.

Referring to Fig. 15-23, if the least significant excess-three digit of N is zero (i.e., if $P4\ P3\ P2\ P1 = 0011$ at $T_0{}^q$, with $K = 0$), gate 1 sets the flip-flop to the unit state, through gate 2. As long as each later digit is a zero (for $K = 1, 2, \ldots$), the flip-flop will remain set. The first nonzero digit will reset the flip-flop to the zero state, through gate 3. As long as the flip-flop is set, $C = 1$ and gates 6 and 7 (and their associated *not* gates) form $R = \bar{\bar{N}}_i = N_i$ (that is, so long as $N_i = 0$). As the flip-flop is reset, its last unit bit is delayed, so that C will still be 1 as $(N_i)_0$

† According to the following function table for $(N_i)_j$, the jth *bit* of the ith *digit* of N:

$$\begin{array}{rl} (N_i)_j & 0101 \\ C & \underline{0011} \\ R & 1001 \\ C' & 0010 \end{array}$$

where C is the carry (C') delayed, and $j = 0 \rightarrow C = 1$, so that the 1 is added to N_i.

of the first nonzero digit of N enters the complementer, initiating the $10 - N_i$ function. If $N_1 \neq 0$, $T_0{}^q$ entering gate 4 will make $C = 1$ for $(N_1)_0$, to initiate $10 - N_1$. After $10 - N_i$ is formed for the first nonzero digit, no carry can result [unless by error $(N_i + 3)_b \doteq 1111$], and since the flip-flop will be reset, C will be 0 and gates 5 and 7 will form $R = \bar{N}_i \doteq 9 - N_i$ for all following digits.

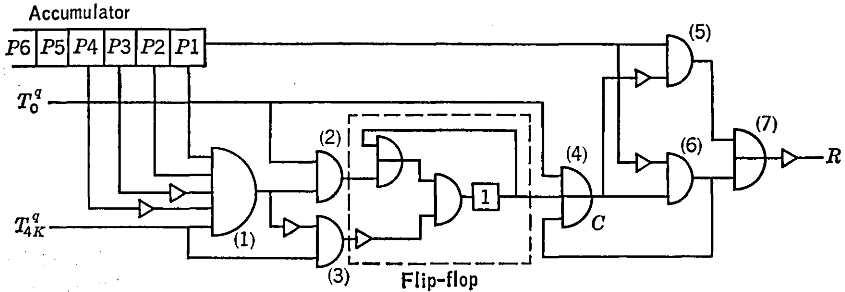


FIG. 15-23. Excess-three subtractive complementer. $T_{4K}{}^q$ marks the beginning of each successive digit as $K = 0, 1, 2, \ldots$.

EXERCISES

(*a*) Consider a word that is 15 bits plus a sign bit long. Suppose that the signal X as defined above is to be sensed from stage 1 of the accumulator at time $T_n{}^1$. What should be the value of n? (*Ans.* $T_{17}{}^1$.)

(*b*) What is the purpose of $\bar{T}_0$ in Fig. 15-20?

(*c*) In Exercise *b* of Sec. 15-3 it was described how an adder could subtract. This requires an *input subtractive complementer to the adder as well as an output subtractive complementer.* Suppose that it is decided always to form $C(B)$ if B is negative (where B enters the arithmetic unit first, before A enters). Then the decision on whether to form $C(A)$ is determined by the operation W and the signs S_A of A and S_B of B. Suppose that this complementation is not to take place when control signal Q is 0 and is to take place when Q is 1. Determine $Q = Q(S_A,S_B,W)$. (HINT: Recall from Exercise *b* of Sec. 15-3 that, if the difference of A and B is to be computed, only one of A or B, but not both, must be subtractive-complemented; if the sum of A and B is to be formed, either both A and B must not be subtractive-complemented or else both A and B must be subtractive-complemented.)

(*d*) If X is the output-subtractive-complementer control, how is it generated in the binary system of Exercise *c*? [HINT: The difficult case is when $C(A) + C(B)$ is formed in the accumulator. Here the sum is $2^p - A + 2^p - B = 2^{p+1} - (A + B)$. Hence we must form by means of the output complementer $2^{p+1} - (2^{p+1} - (A + B))$.]

(*e*) For the system of Exercise *c* design a circuit to generate the proper sign.

(*f*) How can advantage be taken of the adder to simplify the input-subtractive-complementer circuit of the system of Exercise *c*? (HINT: See Exercise *d* of Sec. 15-3.)

(*g*) Describe in detail a decimal system using excess-three decimal-coded binary words, where only an adder is available, together with input and output subtractive complementers.

(*h*) In a computer *zero is always considered to be positive.* Can a negative zero occur in the binary sign-generating system described in this section? (*Ans.* Yes.) Under precisely what conditions?

(*i*) A zero sensor circuit determines if the result of an addition or subtraction is zero. Design such a circuit for a binary system, and show how it can be made to override the sign generator when necessary.

(*j*) Design a zero sensor for the decimal system described in this section.

(*k*) Is a zero sensor necessary for a binary system as described in Exercise *c*?

(*l*) In Fig. 15-22 the 4 units of delay are followed by a direction-control box. This circuit is to recirculate the 4 bits for the $\pm 3_b$ adjustment and then send these 4 bits to the accumulator. Design a circuit that includes a 3-bit counter that will perform this task.

15-6. Serial Arithmetic Unit: Multiplication and Division

Multiplication. Binary multiplication as accomplished by a serial computer is similar to the ordinary hand method except that it is more convenient to form the partial sums directly instead of summing last. For example, consider the following multiplication $M \times N = P$:

By hand		*By serial computer*	
11101	Multiplicand N	11101	Multiplicand N
10011	Multiplier M	10011	Multiplier M
11101		1110¦1	1st partial sum
11101		11101¦	
00000		10101¦11	2d partial sum
00000		00000¦	
11101		1010¦111	3d partial sum
1000100111	Product P	00000¦	
		0101¦0111	4th partial sum
		11101¦	
		$\underbrace{10001}_{\text{Major}}¦\underbrace{00111}_{\text{Minor}}$	Product P

Hence, in a serial computer, the *product P* is formed by successively adding to the partial sum either all zeros or the appropriately shifted multiplicand N, as directed by the *multiplier M*. In the following discussion we assume that M and N are positive, since the sign of the product P is generated independently. Also recall that the sign bits never enter the register but are recorded for later use in the sign generator.

The method for implementing the multiplication process involves two additional shift registers besides the accumulator. One of these registers will record the multiplicand so that it will be readily available for forming the partial sums; this register is called the *icand register.* The other register records the multiplier and presents the appropriate bit to direct the partial-sum formation at each stage of the multiplication; this register is called the *ier register.* The ier register has another function also. Recall that a product can be twice the length of a word, and therefore major and minor multiplication is obtained (see above example). The ier register is used to store the minor product as it is formed. To see how this double function can be accomplished by the ier register, observe that, after a bit of the multiplier has directed the formation of its corresponding partial product, it can be discarded. Also note that the bits of the minor product are generated one at a time by each successive partial product (indicated by dashed line above). Hence, as a bit of the multiplier is discarded, it can be replaced in the ier register by the newly generated bit of the minor product at the end of the multiplication process. The major product is found in the accumulator.

Figure 15-24 is a simplified block diagram of the circuits used in serial multiplication for a three-address system. During phase 1 the multiplier enters the ier shift register (where the signal $F1$ is a unit during phase 1).

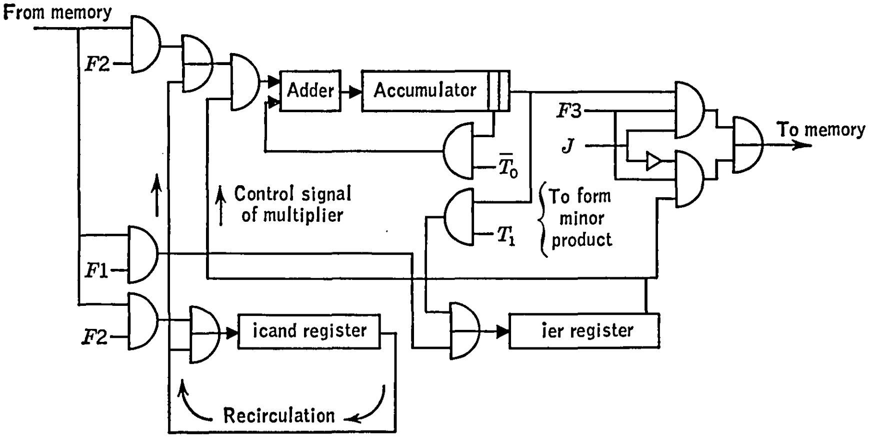


FIG. 15-24. Serial multiplication: arrangement shown for multiplication only.

During phase 2 the multiplicand is brought into the arithmetic unit (where the signal $F2$ is a unit during phase 2), and the multiplication is executed. In the first minor cycle of the phase the multiplicand enters the icand register to be stored; at the same time the first sum is formed, such that only if $P1$ of the multiplier is 1 will the multiplicand be allowed to enter the adder.

During the second minor cycle the icand register recirculates the multiplicand and at the same time sends it to the adder. At $T_0{}^2$ the multiplier is shifted right one position, putting $P2$ in the controlling position, and spilling $P1$, as $P1$ of the product is simultaneously inserted into the most significant position of the ier register. The multiplicand enters the adder or not depending on $P2$. At this time the first partial product is being recirculated through the adder, to be added to the multiplicand (or to all zeros, depending on $P2$). In the above example $P2$ is a unit. Note, however, that, when this addition is accomplished, the first partial product must be displaced one position *to the right* with respect to the multiplicand. This displacement is accomplished by recirculating the first partial product one position *before* it reaches the end of the accumulator. Thus $P2$ of the first partial product is added to $P1$ of the multiplicand, etc., to form the second partial product. Since $P1$ of the first partial product does not contribute to the second partial product and has already been stored in the register, it is not recirculated.

At $T_0{}^3$ the ier register is again shifted right as the next least significant bit of the minor product is inserted. This puts $P3$ of the multiplier in the controlling position and spills $P2$. Again the icand register recirculates the multiplicand, sending it as well to the adder. The second partial sum begins to recirculate one position before the end of the accumulator. And so the third and following partial products are generated, until the entire product has been formed. By this scheme a serial computer with n bits to a word will consume n minor cycles during multiplication.

In phase 3 either the major or the minor product is transmitted to the memory as indicated by the output gates from the accumulator and ier register as controlled by $F3$ and the signal J, which is 1 when the major product is desired, 0 when the minor product is desired in Fig. 15-24.

Let us now examine some of the timing factors involved in multiplication. The multiplier M is transmitted to the ier register so that $M1$ enters the zeroth stage at $T_1{}^1$ of phase 1 (since the sign bit is recorded elsewhere). M is shifted down the ier register so that $M1$ reaches the end and can be presented to control the adder input at $T_1{}^1$ of the first minor cycle of *phase* 2. During each subsequent minor cycle of phase 2 the ier register shifts at $T_1{}^q$, presenting the bit M_q of M. The multiplicand N is transmitted from the memory to both the icand register and adder during the first minor cycle of phase 2 so that $N1$ enters the zeroth stage at $T_1{}^1$. Of course, depending on $M1$, it may or may not get into the adder. However, it is shifted down the icand register and recirculated in it each minor cycle so that $N1$ is presented to the adder at $T_1{}^q$.

To describe the timing associated with the accumulator, we need some special notation. Let P^q be the qth partial product generated during the qth minor cycle of phase 2. Then the least significant bit of P^q, namely, P^q1 (since no sign is attached to P^q), is actually the Pq bit of the (minor) final product. We want to recirculate P^q so that P^q2 enters the adder during $T_1{}^{q+1}$. Thus we must stop the accumulator at the stage in which P^q2 resides at $T_0{}^{q+1}$. Using our formulas of Sec. 15-4, we have $T_0{}^{q+1} = T_w{}^q$, whence $w - 0(q - 1) - 2 = w - 2$ (since $\Delta = 0$; why?). Since stage $w - 1$ is the last stage, stage $w - 2$ is next to the last stage as shown in Fig. 15-16. Next we want P^q1 to be transmitted to the ier register at $T_1{}^{q+1}$, whence it must be at position $w - 1$ at $T_0{}^{q+1}$.

Division. Binary division is accomplished in a serial computer by a method quite different from the familiar hand technique. Let us consider the case in which the quantity format places the binary point to the left of a word, i.e., in which all numbers are less than 1. Let N be the dividend, D the divisor, and Q the computed quotient; assume that N and D are positive, the sign of Q being determined in a separate circuit. Thus we can write

$$\frac{N}{D} = Q = Q_1 2^{-1} + Q_2 2^{-2} + Q_3 2^{-3} + \cdots + Q_n 2^{-n} \qquad D > N$$

where Q_1 is the most significant, Q_n the least significant bit of the desired quotient. Now form

$$\begin{aligned} N' = 2N - D &= D(2Q - 1) \\ &= D(Q_1 - 1 + Q_2 2^{-1} + Q_3 2^{-2} + \cdots + Q_n 2^{-n+1}) \end{aligned}$$

Observe that, if $Q_1 = 1$, then $Q_1 - 1 = 0$ and since $Q_2 2^{-1} + Q_3 2^{-2} + \cdots + Q_n 2^{-n+1}$ is positive, $2N - D$ would be positive or zero. However, if $Q_1 = 0$, since $Q_2 2^{-1} + Q_3 2^{-2} + \cdots + Q_n 2^{-n+1} < 1$, we have $2N - D$ negative. Thus, depending on whether $2N - D$ is positive or negative, we can determine that Q_1 is 1 or 0.

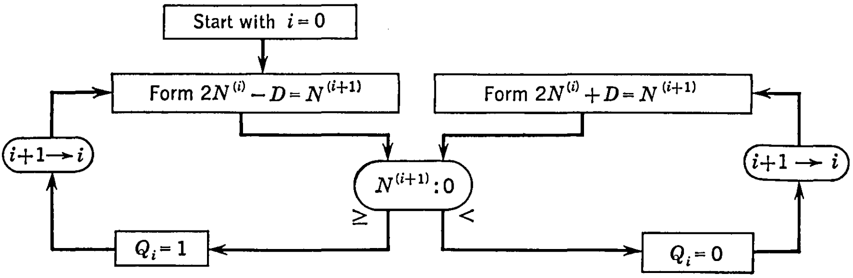


FIG. 15-25. Flow chart of serial-division process.

Suppose that $Q_1 = 1$ $(N' \geq 0)$; then

$$N' = 2N - D = 0 + D(Q_2 2^{-1} + Q_3 2^{-2} + \cdots + Q_n 2^{-n+1}) = DQ'$$

Next form

$$\begin{aligned} N'' = 2N' - D &= D(2Q' - 1) \\ &= D(Q_2 - 1 + Q_3 2^{-1} + Q_4 2^{-2} + \cdots + Q_n 2^{-n+2}) \end{aligned}$$

whence Q_2 is determined to be 1 or 0 depending on whether $2N' - D$ is positive or negative.

However, suppose that $Q_1 = 0$ $(N' < 0)$; then

$$N' = 2N - D = D(-1 + Q_2 2^{-1} + Q_3 2^{-2} + \cdots + Q_n 2^{-n+1}) = DQ'$$

This time form

$$\begin{aligned} N'' = 2N' + D &= D(2Q' + 1) \\ &= D(-2 + 1 + Q_2 + Q_3 2^{-1} + Q_4 2^{-2} + \cdots + Q_n 2^{-n+2}) \\ &= D(Q_2 - 1 + Q_3 2^{-1} + Q_4 2^{-2} + \cdots + Q_n 2^{-n+2}) \end{aligned}$$

which is exactly the same N'' as above. Hence we can flow-chart the division mechanism as in Fig. 15-25. The process ends when Q_n has been determined (this not being shown in our flow diagram of Fig. 15-25).

Of course, when $N^{(i)}$ becomes zero, the division has come out even. The computer, however, continues the process, but each succeeding $N^{(j)}$ will be negative, so that each succeeding Q_j will be zero.

As an example, suppose that $N = .10110110$ and $D = .11100000$. Then the calculations would proceed as follows:

Form $2N - D$

$$\begin{array}{r} 1.01101100 \\ -.11100000 \\ \hline +.10001100 \end{array} = N' > 0 \qquad \text{hence } Q_1 = 1$$

Form $2N' - D$

$$\begin{array}{r} 1.00011000 \\ -.11100000 \\ \hline +.00111000 \end{array} = N'' > 0 \qquad \text{hence } Q_2 = 1$$

Form $2N'' - D$

$$\begin{array}{r} 0.01110000 \\ -.11100000 \\ \hline -.01110000 \end{array} = N''' < 0 \qquad \text{hence } Q_3 = 0$$

Form $2N''' + D$

$$\begin{array}{r} -0.11100000 \\ +.11100000 \\ \hline +.00000000 \end{array} = N^{(iv)} = 0 \qquad \text{hence } Q_4 = 1$$

and the process is complete for this case, with $Q = .110100 \ldots.$

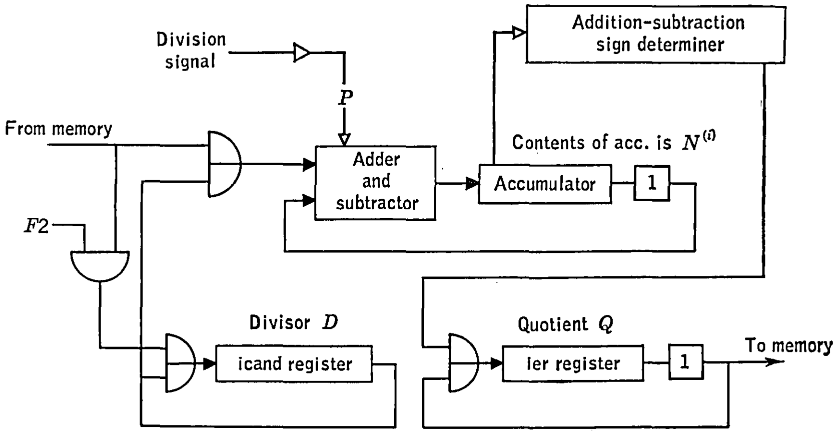


FIG. 15-26. Serial division: arrangement shown for division only.

Figure 15-26 is a simplified block diagram of the circuitry used in serial division. During phase 1 the dividend N enters the accumulator. During the first minor cycle of phase 2 the divisor D enters both the adder and the icand register. In the icand register D is recirculated each minor cycle, successively sending D to the adder. Now during the first minor cycle of phase 2 we wish to form $2N - D$. Forming $2N$ is simple, for all we need do is shift N to the left one position, or, equivalently, delay the entrance of N into the adder for 1 unit of time. Thus effectively

$2N$ enters the adder at the same time as D enters the adder. In this way $N' = 2N - D$ is formed in the adder during the first minor cycle. Next the addition-subtraction sign generator operates: if $N' = 2N - D \geq 0$, $Q_1 = 1$; if $N' < 0$, $Q_1 = 0$. Since a positive sign is a unit and a negative sign a zero, the sign bit itself is equal to the value of Q_1 and therefore is put into the ier register, where the quotient will be formed 1 bit at a time.

During the second minor cycle of phase 2 we wish to form N'' as either $2N' - D$ or $2N' + D$ depending on the value of Q_1 (see above). By delaying N' as it recirculates from the accumulator to the adder $2N'$ is effectively formed. The recirculating icand register sends D to the adder, where $N'' = 2N' - D$ (or $2N' + D$) is formed. The sign of N'' is determined and becomes Q_2. Now Q_2 is *less* significant than Q_1 and therefore must appear in the quotient adjacent to Q_1 on its *right.* To accomplish this, Q_1 has been recirculated around the ier register with an additional unit delay so that it will reenter the ier register just after Q_2 is placed there. Thus Q_2 will be to the right of Q_1 as desired. During the third minor cycle Q_3 is formed as usual, and the partial quotient Q_1Q_2 is recirculated again with a unit delay, enabling Q_3 to be placed to the right of Q_1Q_2, forming $Q_1Q_2Q_3$, and so forth.

Consider Table 15-2, Sum and Difference Conditions, as applied to the special case of division. The denominator D, which corresponds to the second operand B of the table, will always be positive. If a given $N^{(i)}$, which corresponds to the first operand A of the table, is positive, $N^{(i+1)}$ is formed by subtracting: $N^{(i+1)} = 2N^{(i)} - D$. From the table we see that *the result is formed by differencing.* If a given $N^{(i)}$ is negative, $N^{(i+1)}$ is formed by adding: $N^{(i+1)} = 2N^{(i)} + D$. Again the table shows that *the result is formed by differencing.* Hence in division the signal P of both the table and Fig. 15-26 can be forced off during the entire operation.

EXERCISES

(*a*) Design a sign generator for multiplication and division. (HINT: If the signs of the arguments are the same, the sign of the result is +; if different, the sign is −.)

(*b*) Suppose that a continuously shifting register were used for the ier register for multiplication. How should the recirculation loop be designed?

(*c*) How many minor cycles does multiplication take during phase 2?

(*d*) How many minor cycles does division take during phase 2?

(*e*) Suppose that $N > D$ in division. How can this be detected in the division method described in this section?

(*f*) Check the method of recirculation of the accumulator and the ier register by means of the formulas of Sec. 15-4.

(*g*) How would serial division take place in a computer in which the binary point is to the right (i.e., every number is an integer)?

(*h*) Discuss serial decimal multiplication and division based on a serial decimal adder and subtractor. [HINT: Multiplication is rather straightforward. For division one simple method is to let $N/D = Q$, whence $N = DQ = D(Q_1 10^{-1} + Q_2 10^{-2} + Q_3 10^{-3} + \cdots)$. Form $10N - tD = D(Q_1 - t + Q_2 10^{-1} + Q_3 10^{-2} + \cdots)$ for successive values of $t = 1, 2, \ldots, 10$, until $10N - tD$ becomes negative, whence

$Q_1 = t - 1$. Then we have $N' = 10N - tD = D(-1 + Q_2 10^{-1} + Q_3 10^{-2} + \cdots)$. Now form $10N' + tD$ for successive values of $t = 1, 2, \ldots, 10$, until $10N' + tD = ((t - 10) + Q_2 + Q_3 10^{-1} + \cdots)$ becomes positive, whence $Q_2 = 10 - t$. Then $N'' = 10N' + tD = (Q_3 10^{-1} + Q_4 10^{-2} + \cdots)$, and we are back to a situation analogous to that for Q_1.]

(*i*) What extra circuitry might be needed to detect "even" division, that is, $N^{(i)} = 0$, or $2N^{(i)} \pm D = \pm D$?

15-7. Additional Topics

a. Cyclic and Decimal-coded Binary Codes. We have alluded above to the cyclic code. The main property of this code is that successive numbers differ by only a single bit position, as can easily be seen from Table 15-5. The cyclic code given in this table can be derived from the corresponding binary number as follows: If the bit positions are indexed from right to left by i, then c_i, the ith bit of the cyclic code, can be found from the b_i and b_{i+1} bits of the binary code by means of the following formula:

$$c_i = b_i \cdot \bar{b}_{i+1} + \bar{b}_i \cdot b_{i+1}$$

The conversion would then be accomplished *from right to left.* The reverse conversion, from cyclic to binary, can be accomplished by solving the above formula for b_i in terms of c_i and b_{i+1},

$$b_i = c_i \cdot \bar{b}_{i+1} + \bar{c}_i \cdot b_{i+1}$$

The conversion in this case would be accomplished *from left to right.* The cyclic code is advantageous mainly in the use of relay circuits, for then a sticky relay will not give a false state as it is delayed in going from one cyclic number to the next. There are many other cyclic codes that have this property. Can you construct some of them?

TABLE 15-5. EXAMPLES OF CYCLIC AND DECIMAL-CODED BINARY CODES

Decimal	Binary	Cyclic	Excess-three	Biquinary	5-421	2-421	5-cyclic	Features
0	0000	0000	0011	0 0000	0 000	0000	0 000	
1	0001	0001	0100	0 0001	0 001	0001	0 001	
2	0010	0011	0101	0 0010	0 010	0010	0 011	
3	0011	0010	0110	0 0100	0 011	0011	0 010	
4	0100	0110	0111	0 1000	0 100	0100	0 110	
5	0101	0111	1000	1 0000	1 000	1011	1 000	
6	0110	0101	1001	1 0001	1 001	1100	1 001	
7	0111	0100	1010	1 0010	1 010	1101	1 011	
8	1000	1100	1011	1 0100	1 011	1110	1 010	
9	1001	1110	1110	1 1000	1 100	1111	1 110	
	✓		✓		¾			Binary arithmetic
			✓	✓	✓	✓	✓	Binary carry = decimal carry
			✓			✓		9s complement = inversion
	✓			✓	✓	✓		Constant weights
				✓	✓		✓	Sequence 0–4 like 5–9

The other codes shown in Table 15-5 are used for decimal-coded binary arithmetic. (We have already discussed such arithmetic for the excess-three codes.) The use of the biquinary code in this respect is typical. The binary part (i.e., the most significant bit) and the quinary part (the other 4 bits) are first added separately; then the quinary carry is added to the binary part. If a binary carry is generated, this is propagated to the quinary part of the next decimal digit to the left. For example, consider the following work for adding 26 and 34:

$$\begin{array}{rccccccl}
\textit{Decimal} & & \multicolumn{5}{c}{\textit{Biquinary}} & \\
36 & & 0 & 0100 & & 1 & 0001 & \\
+24 & & 0 & 0010 & & 0 & 1000 & \\
\cline{1-1}\cline{3-4}\cline{6-7}
 & & 0 & \swarrow 0000 & & 1 & \swarrow 0000 & \text{Quinary carry} \\
 & & 1 & & & 1 & & \\
\cline{3-4}\cline{6-7}
 & & 1 & 0000 & \swarrow & 0 & 0000 & \text{Decimal carry} \\
60 & & 1 & 0001 & & 0 & 0000 &
\end{array}$$

Similarly addition in the 5-421, 2-421, and 5-cyclic codes is accomplished in two parts.

Only two of the codes enable binary addition. Having the binary carry the same as the decimal carry from one digit to the next digit on the left is useful in designing an adder. If the code admits constant weights, it is more convenient for rapid decimal interpretation [i.e., in the 2-421 code 1011 means $(1 \times 2) + (0 \times 4) + (1 \times 2) + (1 \times 1) = 5$]. We have already seen the advantages of having the inversion of a binary number as its nines complement (nines complement of x is $9 - x$) when we discussed the excess-three code. Evidently there are many more possibilities for decimal-coded binary codes, which can be constructed to have various features. Can you construct a code that is 5-cyclic and for which the nines complement is the inversion [i.e., for which $9999 - (c_4c_3c_2c_1) = \overline{c_4c_3c_2c_1}$]? Further reading on this problem may be found in G. S. White, Coded Decimal Number Systems for Digital Computers, *Proc. IRE*, vol. 41, p. 1450, October, 1953; and in L. B. Wadel, Negative Base Number Systems, *IRE Trans. on Electronic Computers*, vol. EC-6, no. 2, p. 123, June, 1957.

b. Logically design the components of a digital differential analyzer, as described in Sec. 8-2.

c. Logically design the computing components of the small retrieval computer described in Sec. 8-4.

d. Logically design the functional part of the control computer described in Sec. 8-3.

CHAPTER 16

PARALLEL AND RAPID ARITHMETIC OPERATIONS

16-1. Introduction

In Chap. 15 we considered the basic ideas involved in the logical design methods and techniques of enabling a computer to perform arithmetic operations. In the present chapter we consider what may be referred to as refinements of those basic ideas that result in more rapid processing of these operations. Again we choose specific examples of parallel and rapid techniques, for it is felt that the student will gain more from a thorough discussion of one particular technique than from a cursory discussion of many methods. The addition and subtraction operations can be handled in parallel; i.e., all the bits of both arguments enter the adder at the same time, and all the bits of the result are formed simultaneously. However, complete parallel operation is not feasible for multiplication or division. Various observations will be made that allow short cuts eliminating some of the additions and subtraction required by the essentially serial character of the multiplication and division techniques. These short cuts for rapid multiplication and division can be used in conjunction with serial addition and subtraction, or with parallel operations for further speed in computation. At the end of the chapter is presented a discussion of how floating operation might be accomplished.

As will become quite apparent, the gain in speed arising from the parallel and rapid techniques is offset by the requirement of many more gates. In fact the great advantage of serial techniques is that a minimum of circuitry is necessary. In other words, speed costs money. On the other hand the ability to perform calculations rapidly is one of the primary purposes of computers, and hence the increased cost of building a more rapid computer may itself be offset by the greater potentialities the computer offers by its increased speed. Both serial and parallel computers have their places and their uses, and we believe that both types will continue to be designed.

16-2. The Parallel Adder:† Logical Design

Parallel Binary Carry Generation. In a parallel adder all the bits of both arguments enter the adder at once, and all the bits of the sum

† Sections 16-2 to 16-4 are based in part on A. Weinberger and J. L. Smith, Methods for High-speed Addition and Multiplication, *NBS Circ.* 591.

are formed at once. Recall that in Sec. 15-3 we found the R_i bit of the result to be $R_i = (A_i \cdot \bar{B}_i + \bar{A}_i \cdot B_i) \cdot \bar{C}_{i-1} + (\bar{A}_i \cdot \bar{B}_i + A_i \cdot B_i) \cdot C_{i-1}$. Hence, if A_i, B_i, and C_{i-1} were available, R_i could be determined. Since all the bits of the arguments enter a parallel adder at once, we certainly have A_i and B_i available for every bit position i. However, we must still obtain $C_{i-1} = C_{i-1}(A_1,B_1,A_2,B_2, \ . \ . \ . \ ,A_{i-1},B_{i-1})$ and this carry generation becomes the fundamental problem of parallel addition.

In forming the carries there are three limitations, which are essentially electronic in nature but nonetheless do affect the logical design of parallel adders. The electronic aspects will be discussed in detail in Part 5; here we shall consider only the resulting limitations on the logical design. The *first* limitation is that the output of a single gate can be the input to only a limited number of other gates. The *second* is that there can be only a limited number of inputs to a single gate. The *third* concerns our previous approximation that no time is consumed when a pulse travels through a gating array. This approximation holds when a single pulse travels through a chain of just a few gates from the memory to the accumulator or other register. But when a pulse travels through a chain of many gates, the delay through the gating circuitry can no longer be neglected. This limits the number of gates in the chain through which a single pulse can travel within a unit time interval and precludes the use of a propagated carry.

In a serial computer these three limitations are usually less stringent than those of the logical design, and they can often be neglected. Even in parallel circuitry some of the limitations may be disregarded, depending on the electronic design of the gates. For example, the first limitation for certain tube and core-transistor circuitry is so large that it need not be considered as a limitation at all. On the other hand, for a transistor gate at high speeds, the number of outputs a single gate can have is frequently severely limited by the electronic design.

The method for generating C_i is by direct substitution as follows:

$$C_1 = A_1 \cdot B_1$$

$$\begin{aligned} C_2 &= A_2 \cdot B_2 & &= A_2 \cdot B_2 \\ &\quad + (A_2 + B_2) \cdot C_1 & &\quad + (A_2 + B_2) \cdot A_1 \cdot B_1 \end{aligned}$$

$$\begin{aligned} C_3 &= A_3 \cdot B_3 & &= A_3 \cdot B_3 \\ &\quad + (A_3 + B_3) \cdot C_2 & &\quad + (A_3 + B_3) \cdot A_2 \cdot B_2 \\ & & &\quad + (A_3 + B_3) \cdot (A_2 + B_2) \cdot A_1 \cdot B_1 \end{aligned}$$

$$\begin{aligned} C_4 &= A_4 \cdot B_4 & &= A_4 \cdot B_4 \\ &\quad + (A_4 + B_4) \cdot C_3 & &\quad + (A_4 + B_4) \cdot A_3 \cdot B_3 \\ & & &\quad + (A_4 + B_4) \cdot (A_3 + B_3) \cdot A_2 \cdot B_2 \\ & & &\quad + (A_4 + B_4) \cdot (A_3 + B_3) \cdot (A_2 + B_2) \cdot A_1 \cdot B_1 \end{aligned}$$

$$\begin{aligned} C_5 &= A_5 \cdot B_5 & &= A_5 \cdot B_5 \\ &\quad + (A_5 + B_5) \cdot C_4 & &\quad + (A_5 + B_5) \cdot A_4 \cdot B_4 \\ & & &\quad + (A_5 + B_5) \cdot (A_4 + B_4) \cdot A_3 \cdot B_3 \\ & & &\quad + (A_5 + B_5) \cdot (A_4 + B_4) \cdot (A_3 + B_3) \cdot A_2 \cdot B_2 \\ & & &\quad + (A_5 + B_5) \cdot (A_4 + B_4) \cdot (A_3 + B_3) \cdot (A_2 + B_2) \cdot A_1 \cdot B_1 \end{aligned}$$

and so forth. Thus a 5-bit parallel adder can be constructed as in Fig. 16-1.

Let us examine this adder with respect to the above limitations. Consider the first, the loading of each output. Table 16-1 summarizes the number of outputs required for some of the gates of the adder. If any gate could have six outputs, we would be safe. However, suppose, for example, that a gate could have no more than three outputs. We can still squeeze through, however, by paralleling gates, as shown in Fig. 16-2 for $A_3 + B_3$. No gate has more than three outputs, and yet we can generate up to nine functions $A_3 + B_3$.

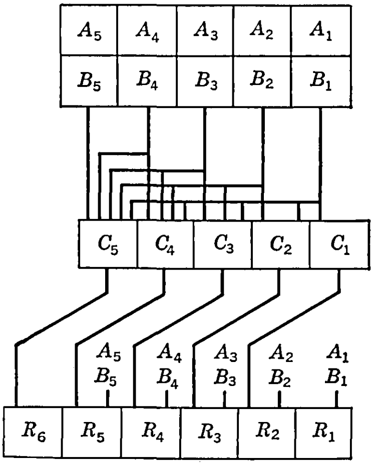


FIG. 16-1. Five-bit parallel adder.

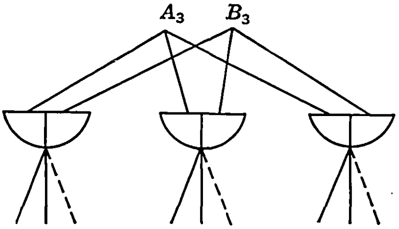


FIG. 16-2. Generating up to nine outputs $A_3 + B_3$.

Consider next the second limitation. The gate that has the largest number of inputs in our parallel adder is the *and* gate that forms $(A_5 + B_5) \cdot (A_4 + B_4) \cdot (A_3 + B_3) \cdot (A_2 + B_2) \cdot A_1 \cdot B_1$, which has six inputs. To see how this limitation can affect the design of our parallel adder, let us assume that a single gate can have no more than six inputs. This means that if we wanted to design, say, a parallel adder for more than 5 bits, the complete expressions for C_6, C_7, etc., analogous to the expression given above for C_5, can no longer be used, since to construct C_6 would require a seven-input *and* gate, and to construct C_7

TABLE 16-1. NUMBER OF OUTPUTS REQUIRED FOR 5-BIT PARALLEL ADDER

Function	*Number of outputs needed*
$A_1 \cdot B_1$	5
$A_2 \cdot B_2$	4
$A_3 \cdot B_3$	3
$A_4 \cdot B_4$	2
$A_5 \cdot B_5$	1
$A_2 + B_2$	4
$A_3 + B_3$	6
$A_4 + B_4$	6
$A_5 + B_5$	4

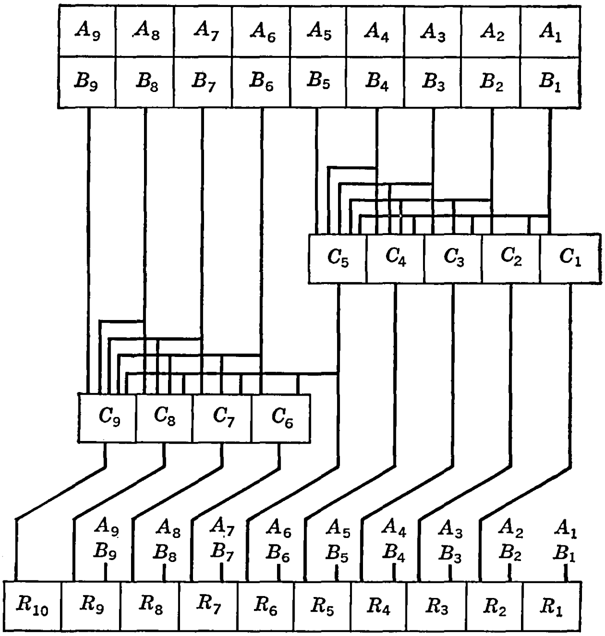

A_9 A_8 A_7 A_6 A_5 A_4 A_3 A_2 A_1
B_9 B_8 B_7 B_6 B_5 B_4 B_3 B_2 B_1
C_5 C_4 C_3 C_2 C_1
C_9 C_8 C_7 C_6
A_9 B_9 A_8 B_8 A_7 B_7 A_6 B_6 A_5 B_5 A_4 B_4 A_3 B_3 A_2 B_2 A_1 B_1
R_{10} R_9 R_8 R_7 R_6 R_5 R_4 R_3 R_2 R_1

FIG. 16-3. Nine-bit parallel adder.

would require a seven- and an eight-input *and* gate, and so forth [i.e., corresponding to the partial expressions: $+(A_6 + B_6) \cdot (A_5 + B_5) \cdot (A_4 + B_4) \cdot (A_3 + B_3) \cdot (A_2 + B_2) \cdot A_1 \cdot B_1$, etc.].

One way out of this dilemma is to note that

$$\begin{aligned}
C_6 &= A_6 \cdot B_6 \\
&\quad + (A_6 + B_6) \cdot C_5 \\
C_7 &= A_7 \cdot B_7 && = A_7 \cdot B_7 \\
&\quad + (A_7 + B_7) \cdot C_6 && \quad + (A_7 + B_7) \cdot A_6 \cdot B_6 \\
& && \quad + (A_7 + B_7) \cdot (A_6 + B_6) \cdot C_5 \\
C_8 &= A_8 \cdot B_8 && = A_8 \cdot B_8 \\
&\quad + (A_8 + B_8) \cdot C_7 && \quad + (A_8 + B_8) \cdot A_7 \cdot B_7 \\
& && \quad + (A_8 + B_8) \cdot (A_7 + B_7) \cdot A_6 \cdot B_6 \\
& && \quad + (A_8 + B_8) \cdot (A_7 + B_7) \cdot (A_6 + B_6) \cdot C_5
\end{aligned}$$

and so forth. By this method we can extend our adder to 9 bits added in parallel within this limitation, as in Fig. 16-3.

However, a pulse may now have to travel through a longer chain of gates, which brings up the third limitation. For example, consider a pulse arising from A_5 that goes into the formation of C_5 and then eventually of R_7. Figure 16-4 shows that it must travel through a chain of seven gates altogether.

Auxiliary Carry Functions. Suppose that we wanted to design an adder of more than 9 bits but could not allow any pulse to go through a chain longer than seven gates. This problem is solved by the use of *auxiliary carry functions* as follows: It is clear that C_{10} cannot be made based on C_5, because of the limitation on the number of inputs to a gate. It is also clear that we cannot base C_{10} directly on C_9, for this would lengthen the chain of gates that a pulse must go through. However, note that the formation of C_6, C_7, C_8, and C_9 is delayed until C_5 has been produced. Thus perhaps we could break up the construction of C_{10}, C_{11}, etc., into two parts, one part of which would be made concurrently with C_1, C_2, C_3, C_4, and C_5 and the final part of which would be made simultaneously with C_6, C_7, C_8, and C_9. To see how this can be accomplished, let us write out C_{10}.

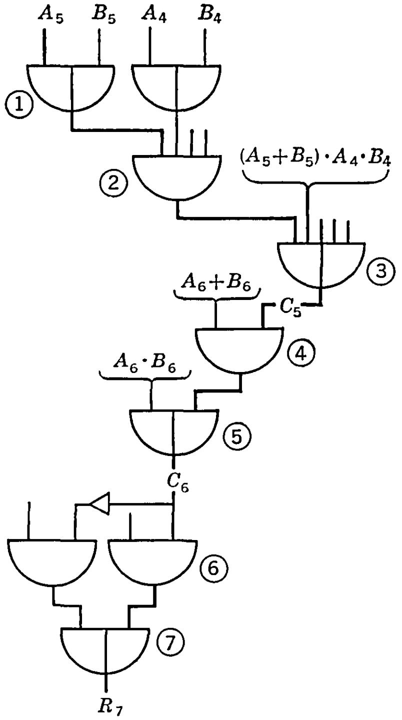


FIG. 16-4. Chain of seven gates through which pulse A_5 must travel to contribute to the formation of R_7.

$$\begin{aligned} C_{10} = {} & A_{10}\cdot B_{10} \\ & + (A_{10}+B_{10})\cdot A_9\cdot B_9 \\ & + (A_{10}+B_{10})\cdot(A_9+B_9)\cdot A_8\cdot B_8 \\ & + (A_{10}+B_{10})\cdot(A_9+B_9)\cdot(A_8+B_8)\cdot A_7\cdot B_7 \\ & + (A_{10}+B_{10})\cdot(A_9+B_9)\cdot(A_8+B_8)\cdot(A_7+B_7)\cdot A_6\cdot B_6 \\ & \boxed{+ (A_{10}+B_{10})\cdot(A_9+B_9)\cdot(A_8+B_8)\cdot(A_7+B_7)\cdot(A_6+B_6)}\cdot C_5 \end{aligned}$$

Call the function in the triangle X_{10} and in the rectangle below Y_{10}. Then

$$C_{10} = \begin{array}{|l|}\hline X_{10} \\ \hline + Y_{10} \\ \hline\end{array} \cdot C_5 = X_{10} + Y_{10} \cdot C_5$$

The *auxiliary carry function* X_{10} satisfies our requirements on the number of inputs to a single gate (no more than six), and so does Y_{10}. Thus we

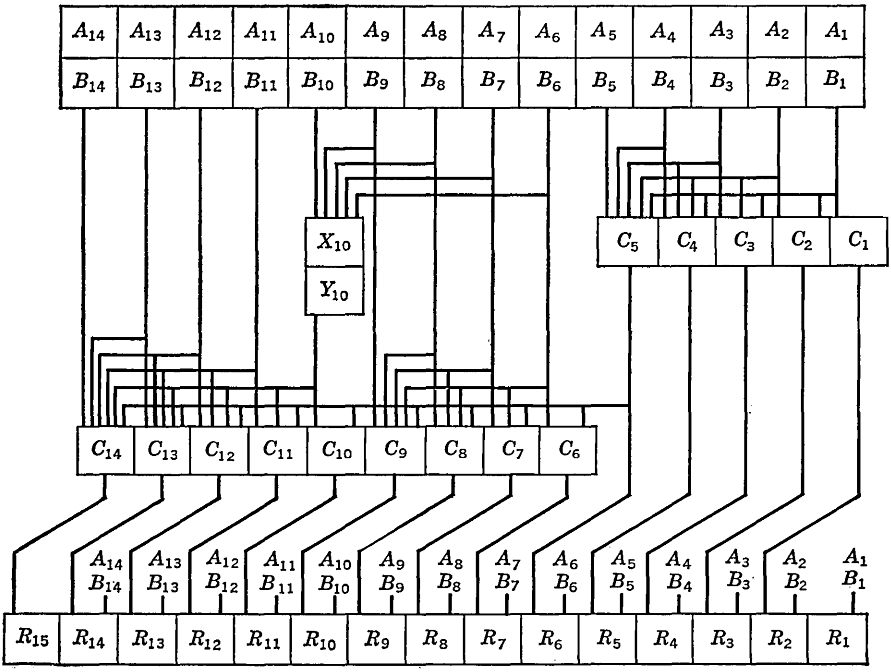

A_{14} A_{13} A_{12} A_{11} A_{10} A_9 A_8 A_7 A_6 A_5 A_4 A_3 A_2 A_1
B_{14} B_{13} B_{12} B_{11} B_{10} B_9 B_8 B_7 B_6 B_5 B_4 B_3 B_2 B_1
X_{10}
Y_{10}
C_5 C_4 C_3 C_2 C_1
C_{14} C_{13} C_{12} C_{11} C_{10} C_9 C_8 C_7 C_6
A_{14} B_{14} A_{13} B_{13} A_{12} B_{12} A_{11} B_{11} A_{10} B_{10} A_9 B_9 A_8 B_8 A_7 B_7 A_6 B_6 A_5 B_5 A_4 B_4 A_3 B_3 A_2 B_2 A_1 B_1
R_{15} R_{14} R_{13} R_{12} R_{11} R_{10} R_9 R_8 R_7 R_6 R_5 R_4 R_3 R_2 R_1

FIG. 16-5. Fourteen-bit parallel adder using auxiliary carry functions.

can form X_{10} and Y_{10} first, and then form C_{10} from these. Using these auxiliary carry functions, we can also form

$$\begin{aligned} C_{11} &= A_{11} \cdot B_{11} & &= A_{11} \cdot B_{11} \\ &\quad + (A_{11} + B_{11}) \cdot C_{10} & &\quad + (A_{11} + B_{11}) \cdot X_{10} \\ & & &\quad + (A_{11} + B_{11}) \cdot Y_{10} \cdot C_5 \end{aligned}$$

$$\begin{aligned} C_{12} &= A_{12} \cdot B_{12} & &= A_{12} \cdot B_{12} \\ &\quad + (A_{12} + B_{12}) \cdot C_{11} & &\quad + (A_{12} + B_{12}) \cdot A_{11} \cdot B_{11} \\ & & &\quad + (A_{12} + B_{12}) \cdot (A_{11} + B_{11}) \cdot X_{10} \\ & & &\quad + (A_{12} + B_{12}) \cdot (A_{11} + B_{11}) \cdot Y_{10} \cdot C_5 \end{aligned}$$

and similarly C_{13} and C_{14}. Hence we have the 14-bit parallel adder as shown in Fig. 16-5.

Generating C_{15} again raises a problem, for this would involve a seven-input gate even with the use of our auxiliary carry functions X_{10} and Y_{10}. However, we can construct, as follows, other carry functions, called X_{15} and Y_{15}, which will do the job. Writing out C_{15}, we find

$$C_{15} = \begin{array}{|l|l} A_{15}\cdot B_{15} & \\ + \ (A_{15}+B_{15})\cdot A_{14}\cdot B_{14} & \\ + \ (A_{15}+B_{15})\cdot (A_{14}+B_{14})\cdot A_{13}\cdot B_{13} & \\ + \ (A_{15}+B_{15})\cdot (A_{14}+B_{14})\cdot (A_{13}+B_{13})\cdot A_{12}\cdot B_{12} & \\ + \ (A_{15}+B_{15})\cdot (A_{14}+B_{14})\cdot (A_{13}+B_{13})\cdot (A_{12}+B_{12})\cdot A_{11}\cdot B_{11} & \\ \hline + \ (A_{15}+B_{15})\cdot (A_{14}+B_{14})\cdot (A_{13}+B_{13})\cdot (A_{12}+B_{12})\cdot (A_{11}+B_{11}) & \cdot X_{10} \\ + \ (A_{15}+B_{15})\cdot (A_{14}+B_{14})\cdot (A_{13}+B_{13})\cdot (A_{12}+B_{12})\cdot (A_{11}+B_{11}) & \cdot Y_{10}\cdot C_5 \\ \hline \end{array}$$

whence we write

$$C_{15} = \begin{array}{|l|l} X_{15} & \\ \hline + \ Y_{15} & \cdot X_{10} \\ + \ Y_{15} & \cdot Y_{10}\cdot C_5 \\ \hline \end{array} \qquad = X_{15} + Y_{15}\cdot (X_{10} + Y_{10}\cdot C_5)$$

where X_{15} and Y_{15} can be formed concurrently with X_{10}, Y_{10}, and C_5.

Summarizing, we observed three possible limitations on the logical design of the parallel adder and discussed methods and techniques for overcoming these difficulties. It is clear that the exact form of the logical design of a parallel adder depends on the specific values of the limitations involved and can be no more than the best possible compromise with these limitations.

EXERCISES

(*a*) Draw the logical details of the circuits represented by boxes C_1, C_2, C_3, C_4, and C_5 of Fig. 16-1.

(*b*) Draw the logical details of the circuit represented by box R_4 of Fig. 16-1.

(*c*) If C_6 had been made directly from A_5, B_5, A_4, B_4, . . . , A_1, B_1, how many more gates would be in the chain that the pulses A_5 would have to travel to contribute to the formation of R_6?

(*d*) By using additional auxiliary carry functions, show how Fig. 16-5 can be extended to add two 21-bit binary numbers in parallel under the limitations of no more than six inputs to a single gate. Explicitly display the additional auxiliary carry functions necessary.

16-3. Parallel Arithmetic Units

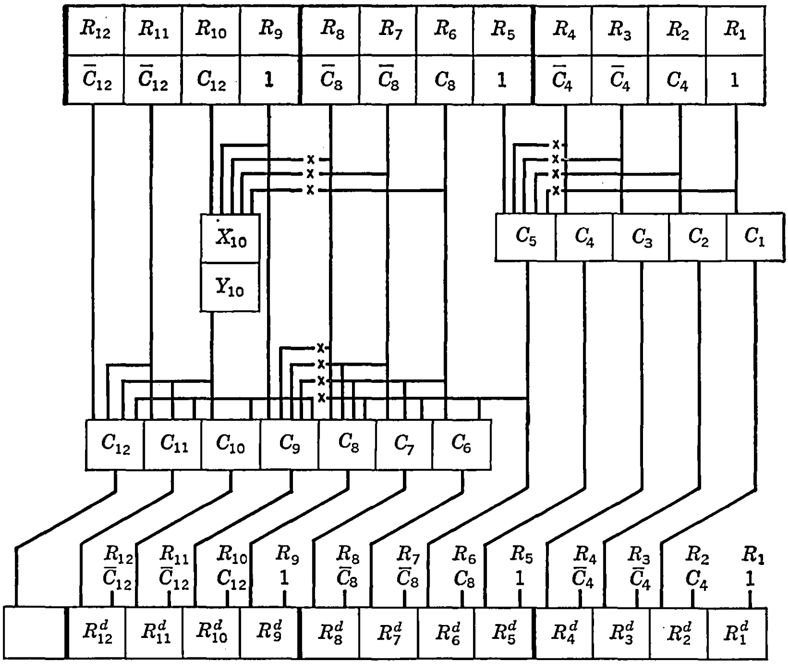
The Parallel Adder-Subtractor Combination. When forming the difference between two numbers, we have seen that R_i is the same as for the sum. However, consider the parallel generation of the carry for subtraction. We shall in this section continue to use C_i for the carry for subtraction as well as addition. We then have by successive substitution the carries for subtraction:

$$\begin{aligned}
C_1 &= \bar{A}_1 \cdot B_1 \\
C_2 &= \bar{A}_2 \cdot B_2 && = \bar{A}_2 \cdot B_2 \\
&\quad + (\bar{A}_2 + B_2) \cdot C_1 && \quad + (\bar{A}_2 + B_2) \cdot \bar{A}_1 \cdot B_1 \\
C_3 &= \bar{A}_3 \cdot B_3 && = \bar{A}_3 \cdot B_3 \\
&\quad + (\bar{A}_3 + B_3) \cdot C_2 && \quad + (\bar{A}_3 + B_3) \cdot \bar{A}_2 \cdot B_2 \\
& && \quad + (\bar{A}_3 + B_3) \cdot (\bar{A}_2 + B_2) \cdot \bar{A}_1 \cdot B_1
\end{aligned}$$

and so forth. It becomes immediately clear that if $\bar{A}_i$ is replaced by A_i these expressions become identical with those for addition given above. Thus, in order to subtract, all we need do is replace A_i by $\bar{A}_i$ in the carry and auxiliary carry functions and use the same circuitry as was used for parallel addition. Hence, with this added feature, the parallel adder designed above becomes a parallel subtractor also.

To incorporate our parallel adder-subtractor combination into an arithmetic unit, we must be able to take the subtractive complement of our result when necessary. This can easily be performed in parallel by making use of the parallel circuitry already available, for the subtractive complement of R is simply $2^p - R$. Thus we *send the result R back into the parallel adder via the route used previously by B and form the difference* $2^p - R$. Note that we did not reuse the serial adder for this purpose, because it would have required recirculating the contents of the accumulator back through the adder again, using up another minor cycle; it was more convenient to perform this operation in serial as the contents of the accumulator was being transferred to the memory. However, in our parallel arithmetic unit, it requires only an extra unit time interval to form the subtractive complement in the adder, and has the added advantage of using no further circuitry except for control purposes.

Parallel Decimal Addition and Subtraction. Again considering binary-coded decimal in the excess-three system, we shall show how the circuitry of our parallel binary adder can be extended to handle parallel decimal addition as well. As was shown in Sec. 15-5, the carry from one 4-bit coded decimal digit to the successive one during addition is the ordinary binary carry produced in the binary addition of the digit codes. Not only that, but the decimal sum was obtained by adding the arguments as if they were binary numbers, and then adjusting each 4-bit group of the result by adding or subtracting 3 to form the decimal-coded result. The method for extending our binary parallel adder to become a decimal parallel adder now becomes clear. First add the two arguments in parallel as if they were binary numbers, but recording the carries C_4, C_8, C_{12}, C_{16}, etc. Now send the result back to the adder, and this time operate only on the successive groups of 4 bits *independently;* subtract 3 if no carry *from* this group occurred; add 3 if a carry *from* this group did occur. The trick then is to operate in this way on the groups independently. This may be accomplished simply by cutting out any contribution to the result, carry, or auxiliary carry functions from any place other than these four positions. In Fig. 16-6, which is an adaptation of Fig. 16-5, we have indicated which pulses would be stopped from propagating for this $\pm 3_b$ adjustment phase of decimal addition. With a carry $C_i = 1$, $\bar{C}_i = 0$, and we add $\bar{C}_i\bar{C}_iC_i1 \equiv 0011$ to the corresponding 4 bits. With no carry $C_i = 0$, $\bar{C}_i = 1$, and we add $1101 = (2^4 - 3)_b$ (without carry). In adjusting the third decimal digit (R_9 to R_{12}), $C_{12} = 1$ indicates overflow in the addition.



R_{12} R_{11} R_{10} R_9 R_8 R_7 R_6 R_5 R_4 R_3 R_2 R_1
$\bar{C}_{12}$ $\bar{C}_{12}$ C_{12} 1 $\bar{C}_8$ $\bar{C}_8$ C_8 1 $\bar{C}_4$ $\bar{C}_4$ C_4 1
x
x
x
x
X_{10}
Y_{10}
C_5 C_4 C_3 C_2 C_1
x
x
x
x
C_{12} C_{11} C_{10} C_9 C_8 C_7 C_6
R_{12} $\bar{C}_{12}$
R_{11} $\bar{C}_{12}$
R_{10} C_{12}
R_9 1
R_8 $\bar{C}_8$
R_7 $\bar{C}_8$
R_6 C_8
R_5 1
R_4 $\bar{C}_4$
R_3 $\bar{C}_4$
R_2 C_4
R_1 1
R^d_{12} R^d_{11} R^d_{10} R^d_9 R^d_8 R^d_7 R^d_6 R^d_5 R^d_4 R^d_3 R^d_2 R^d_1

FIG. 16-6. Binary parallel adder illustrated during its use for $\pm 3_b$ adjustment when adapted for a 3-digit decimal parallel adder. The crosses indicate which pluses are to be stopped from propagation.

Parallel decimal subtraction can be accomplished by means of an adaptation of our binary parallel adder in a clearly straightforward manner, using the appropriate rules for the $\pm 3_b$ adjustment.

Another technique would be to add or subtract 3_b from each digit (as indicated by the carry in summing or differencing) by means of special circuitry for each group of 4 bits. It turns out that if R_4, R_3, R_2, R_1 represent an unadjusted group of 4 bits, then, whether differencing or summing, the only conditions when -3_b adjustment may be required are given by the following constrained basis (see Exercise *c*), where the adjusted values of the digits appear below:

$$\begin{array}{llll}
R_1 & 01 & 0101 & 0101 \\
R_2 & 11 & 0011 & 0011 \\
R_3 & 11 & 0000 & 1111 \\
R_4 & 00 & 1111 & 1111 \\
\hline
R_1{}^d & 10 & 1010 & 1010 = \bar{R}_1 \\
R_2{}^d & 10 & 0110 & 0110 = R_1 \cdot \bar{R}_2 + \bar{R}_1 \cdot R_2 \\
R_3{}^d & 01 & 1110 & 0001 = R_1 \cdot R_2 \cdot R_3 + (\bar{R}_1 + \bar{R}_2) \cdot \bar{R}_3 \\
R_4{}^d & 00 & 0001 & 1111 = R_1 \cdot R_2 \cdot \bar{R}_3 + R_3 \cdot R_4
\end{array}$$

This then is the circuitry required for the -3_b adjustment. Similarly it can be shown that, whether differencing or summing, the conditions when $+3_b$ adjustment may be required are the same for either case. Then (see Exercise *d*) we find for the circuitry that will form the $+3_b$ adjusted group of 4 bits:

$$\begin{aligned}
R_1{}^d &= \bar{R}_1 \\
R_2{}^d &= R_1 \cdot R_2 + \bar{R}_1 \cdot \bar{R}_2 \\
R_3{}^d &= (R_1 + R_2) \cdot \bar{R}_3 + \bar{R}_1 \cdot \bar{R}_2 \cdot R_3 \\
R_4{}^d &= (R_1 + R_2) \cdot R_3 + R_4
\end{aligned}$$

If $P = 1$ means sum, $P = 0$ means difference, and C is the carry, then the signal that tells which set of circuitry to use, i.e., whether to add 3 ($Z = 1$) or subtract 3 ($Z = 0$), becomes (according to Sec. 15-5)

$$Z = P \cdot C + \bar{P} \cdot \bar{C}$$

EXERCISES

(*a*) Draw a detailed schematic of a parallel arithmetic adder-subtractor unit, and discuss the function and operation of each of the necessary components.

(*b*) Describe in detail parallel decimal subtraction.

(*c*) Find the constrained basis representing all possible conditions on the unadjusted group of 4 bits $R_4R_3R_2R_1$ after summing two excess-three decimal digits that can require -3_b adjustment. Show that the conditions after differencing that can require -3_b adjustment are the same. (HINT: The smallest unadjusted sum of 2 digits is 6_b; the largest unadjusted sum of 2 digits that generates *no* carry is 15_b, etc.)

(*d*) Derive equations for circuitry that will form the $+3_b$ adjusted group of 4 bits after summing or differencing 2 decimal excess-three digits.

(*e*) What are the advantages and disadvantages of the two methods shown in the text for forming the $\pm 3_b$ adjustment?

16-4. Rapid Multiplication

Short Cuts for Rapid Multiplication. The methods for short-cut multiplication depend on the following observations: *First* note, for example, that the binary number $\overset{76543210}{00100100}$ can be written as $2^5 + 2^2$. Hence, to form $(00100100)(N)$, we need merely add $2^5N + 2^2N$. *Second* observe that, for example, the binary number $\overset{6543210}{0011111}$ can be written as $2^5 - 2^0$, that is, 0100000 − 1 = 0011111. In general it follows that

$$\begin{array}{ccccccccccccc} \beta+2 & \beta+1 & \beta & \beta-1 & & \alpha+1 & \alpha & \alpha-1 & \alpha-2 & & 2 & 1 & 0 \\ 0 & 0 & 1 & 1 & \cdots & 1 & 1 & 0 & 0 & \cdots & 0 & 0 & 0 \end{array} = 2^{\beta+1} - 2^{\alpha}$$

Thus, when a multiplier has a *string of units*, we need not form the partial product for each unit, but merely subtract (2^{α})(multiplicand) from $(2^{\beta+1})$(multiplicand). *Third* consider a binary number, say $\overset{6543210}{0011011}$; this can be written in the form $2^5 - 2^2 - 2^0$, since it is the same as 0011111 reduced by 2^2. Hence 0011011 is said to have a string of units with an *included zero*. Note that the effect of an included zero is a subtraction of the corresponding power of 2. Clearly this holds for more than one included zero: thus, for example, if a string of units from positions α to β has included zeros in positions γ and δ, then it can be written as $2^{\beta+1} - 2^{\gamma} - 2^{\delta} - 2^{\alpha}$.

Our three observations can be combined to write any binary number in terms of powers of 2, where the object is to have the least number of powers of 2 in the representation. For example,

$$\begin{array}{cccccccccccccccc} 15 & 14 & 13 & 12 & 11 & 10 & 9 & 8 & 7 & 6 & 5 & 4 & 3 & 2 & 1 & 0 \\ 0 & 1 & 1 & 0 & 1 & 1 & 1 & 0 & 0 & 1 & 0 & 0 & 1 & 1 & 1 & 1 \end{array}$$
$$= 2^{15} - 2^{12} - 2^{9} + 2^{6} + 2^{4} - 2^{0}$$

As an illustration, let us form the product

```
                              1111   1111   1111   1111   Multiplicand N
                              0110   1110   0100   1111   Multiplier M
                          −   1111   1111   1111   1111   −2^0 factor (partial product)
                      +1111   1111   1111   1111   0000   +2^4 factor
                   +   1110   1111   1111   1111   0001   Partial product
                 +11   1111   1111   1111   1100   0000   +2^6 factor
            +    100   1110   1111   1111   1011   0001   Partial product
           −1   1111   1111   1111   1110   0000   0000   −2^9 factor
   −        1   1011   0000   1111   1110   0100   1111   Partial product
        −1111   1111   1111   1111   0000   0000   0000   −2^12 factor
  − 1    0001   1011   0000   1110   1110   0100   1111   Partial product
 +111    1111   1111   1111   1000   0000   0000   0000   +2^15 factor
 +110    1110   0100   1110   1001   0001   1011   0001   Product
```

This may be checked by the long method.

Let us now discuss various problems that arise when choosing the appropriate powers of 2 for such a multiplier reduction. The problem is to determine when to add the multiplicand to the partial product, when to subtract it, and when to just shift the partial product and pass. A little notation can aid this discussion; rather than explicitly write out the powers of 2, let + above a bit position of the multiplier indicate the fact that the multiplicand should be added to the partial product from that position, let − above a bit position indicate a subtraction from that position, and · above a bit position indicate a pass. In this notation the multiplier of our above example becomes

$$\overset{+}{0}\overset{\cdot}{1}\overset{\cdot}{1}\overset{-}{0}\quad \overset{\cdot}{1}\overset{\cdot}{1}\overset{-}{1}\overset{\cdot}{0}\quad \overset{\cdot}{0}\overset{+}{1}\overset{\cdot}{0}\overset{+}{0}\quad \overset{\cdot}{1}\overset{\cdot}{1}\overset{\cdot}{1}\overset{-}{1}$$

The computer will sense the multiplier from right to left, i.e., from least toward successively more significant bits. At each bit it must decide whether to add, subtract, or pass. A good decision is one that will pass the largest number of bits, i.e., postpone or put off to the left as far as possible any addition or subtraction. For example, consider the number 01011. This may be interpreted as $\overset{\cdot}{0}\overset{\cdot}{0}\overset{+}{1}\overset{+}{0}\overset{\cdot}{1}\overset{-}{1}$ or as $\overset{\cdot}{0}\overset{+}{0}\overset{\cdot}{1}\overset{-}{0}\overset{\cdot}{1}\overset{-}{1}$. In the former case we considered the adjacent units to be a string, and in the latter case we considered all 3 units as belonging to a string with an included zero. However, the latter case was a better decision, for we postponed the addition farther to the left than in the former case (i.e., in the former case the addition was done at $P3$ and in the latter at $P4$). Similarly consider $\overset{+}{0}\overset{\cdot}{1}\overset{\cdot}{1}\overset{-}{0}\overset{-}{1}$ and $\overset{+}{0}\overset{\cdot}{1}\overset{-}{1}\overset{\cdot}{0}\overset{+}{1}$; the latter is considered better than the former.

These observations can best be summarized in terms of the concept of a *string of units.* Two or more adjacent units constitute a string of units, and as noted above, the least significant unit of a string is interpreted as $\overset{-}{1}$, while the zero to the left of the most significant unit is interpreted as $\overset{+}{0}$, for example as in $\overset{\cdot}{0}\overset{+}{0}\overset{\cdot}{1}\overset{\cdot}{1}\overset{\cdot}{1}\overset{\cdot}{1}\overset{-}{1}$. A string may include *isolated zeros,* i.e., zeros each of which is flanked by units; such zeros are interpreted as $\overset{-}{0}$, for example as in $\overset{\cdot}{0}\overset{+}{0}\overset{\cdot}{1}\overset{\cdot}{1}\overset{-}{0}\overset{\cdot}{1}\overset{-}{0}\overset{\cdot}{1}\overset{-}{1}$. *Isolated units,* i.e., units each of which is flanked by zeros and is not a member of a string, are interpreted as $\overset{+}{1}$, for example as in $\overset{\cdot}{0}\overset{\cdot}{0}\overset{+}{1}\overset{\cdot}{0}\overset{+}{1}\overset{\cdot}{0}\overset{\cdot}{0}$. Thus working from right to left, as a string of units is met, a subtraction must take place, and only subtraction can take place within the string; in exiting from a string, an addition takes place, and only additions can take place outside a string. Finally let us note that our above discussion showed that, when working from right to left, we should not initiate a string until at least two adjacent units occur, that is, $\cdots\ \overset{\cdot}{1}\overset{-}{1}\overset{\cdot}{0}\overset{+}{1}$ is better than $\cdots\ \overset{\cdot}{1}\overset{\cdot}{1}\overset{-}{0}\overset{-}{1}$, and we should not stop a string until at least two adjacent zeros occur, that is, $\overset{\cdot}{0}\overset{+}{0}\overset{\cdot}{1}\overset{-}{0}\overset{\cdot}{1}\overset{\cdot}{1}\ \cdots\ \overset{-}{1}$ is better than $\overset{\cdot}{0}\overset{\cdot}{0}\overset{+}{1}\overset{+}{0}\overset{\cdot}{1}\overset{\cdot}{1}\ \cdots\ \overset{-}{1}$. Based on the remarks of this paragraph, it is clear that Table 16-2 gives the rules determining whether the computer should add, subtract, or pass in considering the ith bit of the multiplicand.

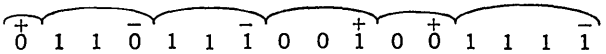
Rapid Multiplication in Serial and Parallel Computers. Rapid multiplication can be designed into a serial computer provided that the icand register is of the shift-register type, i.e., is not a delay line. The procedure would be to examine each bit of the multiplier, together with the next most significant bit and a record of the previous operation, to determine whether to recirculate the partial product in the accumulator for an addition or subtraction with the multiplicand, or just to shift. This process is clear, and no further description is necessary. However, let us consider the storing of the minor product in the ier register. When the partial product is negative, trouble arises, for what is effectively needed is the subtractive complement of those bits of the minor product formed in shifting a negative partial product (see Exercise *c*).

TABLE 16-2. CONDITIONS GOVERNING ADDITION OR SUBTRACTION OF THE MULTIPLICAND TO OR FROM THE PARTIAL PRODUCT OR PASSING WITHOUT AN OPERATION[†]

Condition			Interpretation	Present operation to be performed
P_{i+1}	P_i	Last operation performed		
1	1	Add	Initiating a string	Subtract
0	0	Subtract	Exiting a string	Add
1	0	Subtract	Within a string	Subtract
0	1	Add	Outside a string	Add
All other conditions				Pass

[†] Based on P_{i+1}, P_i of the multiplier and a knowledge of the last operation performed (which reflects whether P_i is in a string or not).

In a parallel computer an addition or subtraction takes only one unit time interval, and hence just the replacement of a shift for an addition or subtraction would not in itself save time. Thus, in order to take advantage of the rapid multiplication method in a parallel computer, the shift must be combined with the operation to be performed all in one unit time interval. For example, the computer may determine whether to add or subtract and then shift all the way to the bit where the next operation is to take place—during one unit time interval. As illustration, the arcs drawn below represent the processes completed during one unit time interval.



However, it is not always feasible to design a computer that can shift any number of positions all in one unit time interval, for to shift a word *n* positions in one unit time interval means that direct connections must exist between each stage and *all* stages up to *n* positions farther down the register. Thus the possible number of positions that can be shifted must be limited. J. L. Smith and A. Weinberger[†] have shown that a good choice for this limitation is to allow the shift register to shift one, two, three, or four stages. By "good" we mean that negligible additional savings in time would occur if shifts of five or more stages were allowed.

[†] Methods for High-speed Addition and Multiplication, *NBS Circ.* 591.

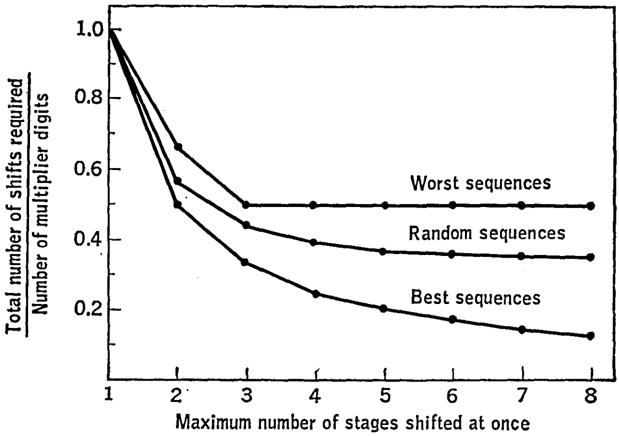


FIG. 16-7. Effectiveness of short-cut multiplication.

Figure 16-7 shows the effectiveness of short-cut multiplication as a function of the maximum number of stages allowed for a unit-time-interval shift.

We have seen above how the computer will decide when to add, subtract, or pass. The additional problem for parallel computers now is how far the computer should shift after the operation. As we have seen, there is good reason to allow the computer a choice of only one, two, three, or four positions. The first limitation on the number of positions is that, since the operation and the subsequent shift are to occur all in one unit time interval, the shift can only be up to the position where the next operation is required. If the next operation is required more than four positions away, then the shift is the maximum, or four positions; then, in the next unit time interval, the operation is passed, and the number of positions to shift recalculated and executed, etc. If the number of positions till the next operation is four, three, two, or one, then this is the number of positions to be shifted, of course. To determine the number of positions to be shifted *after* the operation is performed on P_i, the bits P_{i+1}, P_{i+2}, and P_{i+3} must be sensed. It can be shown from Table 16-2 that Table 16-3 gives the shift conditions (see Exercise *f*).

EXERCISES

(*a*) Design a circuit to determine whether to add, subtract, or pass during rapid multiplication (see Table 16-2). Suppose that only P_{i+1}, P_i, and the sign of the partial product are known; how can it be determined whether to add, subtract, or pass?

(*b*) Describe the functions and operation of the registers during rapid multiplication in a serial computer.

(*c*) Design the circuit that determines the proper bits of the minor product that are generated in serial operation when a negative partial product is shifted.

TABLE 16-3. RULES FOR SHIFTING

P_{i+3}	P_{i+2}	P_{i+1}	P_i †	Operation performed on P_i (and past condition other than shift)	Action
		$\overset{-}{1}$	$\dot{0}$	Pass (if last operation was add)	Shift 1 position
		$\overset{+}{0}$	$\dot{1}$	Pass (if last operation was subtract)	
	1	$\dot{0}$	$\overset{+}{1}$	Add	Shift 2 positions
	0	$\dot{1}$	$\overset{-}{0}$	Subtract	
	1	$\overset{:}{0}$	$\overset{+}{\dot{0}}$	Add; pass (if last operation was add)	
	0	$\overset{:}{1}$	$\overset{-}{\dot{1}}$	Subtract; pass (if last operation was subtract)	
1	$\dot{0}$	$\dot{0}$	$\overset{+}{1}$	Add	Shift 3 positions
0	$\dot{1}$	$\dot{1}$	$\overset{-}{0}$	Subtract	
1	$\overset{:}{0}$	$\overset{:}{0}$	$\overset{+}{\dot{0}}$	Add; pass (if last operation was add)	
0	$\overset{:}{1}$	$\overset{:}{1}$	$\overset{-}{\dot{1}}$	Subtract; pass (if last operation was subtract)	
$\dot{0}$	$\dot{0}$	$\dot{0}$	$\overset{+}{1}$	Add	Shift 4 positions
1	1	1	$\overset{-}{0}$	Subtract	
$\overset{:}{0}$	$\overset{:}{0}$	$\overset{:}{0}$	$\overset{+}{\dot{0}}$	Add; pass (if last operation was add)	
$\overset{:}{1}$	$\overset{:}{1}$	$\overset{:}{1}$	$\overset{-}{\dot{1}}$	Subtract; pass (if last operation was subtract)	

† The superscript plus or minus sign indicates the operation to be performed on P_i; the superscript dot indicates a shift.

(*d*) How are the rules of Table 16-2 to be interpreted when i is the least significant bit of a word and when i is the most significant bit of a word?

(*e*) Design a 12-bit shift register that can shift ahead one, two, or three positions, respectively, within one unit time interval.

(*f*) Derive the rules given by Table 16-3.

16-5. Rapid Division

The Method. We shall first present the method of rapid division and then discuss why it works. The method takes advantage of sequences of adjacent units that may occur in the quotient and generates them all at once. The flow chart of Fig. 16-8 summarizes the procedure, which ends when i equals the number of bits in a word.

We shall assume that the binary point of the computer word is to the left and that for $N/D = Q$, $D > N$, D is normalized (i.e., the most significant bit is a unit: $D = .1 \cdots$). The first step is to form

$$N' = N - D$$

Since in this first step N' is negative, we follow the negative loop. If N' has α zeros to the right of the binary point, then Q has at least $\alpha - 1$ units to the right of the point. In this initial step, $Q_i = 0$ for $i = 0$ merely means that $Q < 1$. Now we form $N'' = N' + D$ with our normalized N'. If the result is negative, we can complete $Q_\alpha = 0$ and place units for additional bits of Q. If the result is positive, we can complete $Q_\alpha = 1$ and place zeros for additional bits of Q. The procedure continues in this way, differencing and normalizing each time, and determining Q_i and Q_{i+1} through $Q_{(i+\alpha)-1}$ at each step. For example, consider

Dividend	$N =$	.1001	1111	0000	1100	Q quotient
Divisor	$-D =$	.1101	0000	0000	0000	
N'	−	×00.11	0000	1111	0100	0.1?
	+	.11	0100	0000	0000	↓
N''	+	×00	00.11	0000	1100	.11000?
	−		.11	0100	0000	↓
N'''	−		×00	00.11	0100	.110000111?
	+			.11	0100	↓
$N^{(iv)}$				.00	0000	.11000011110000 · · ·

As a further example consider

Dividend	$N =$	.1011	0000	0101	Q quotient
Divisor	$-D =$	.1101	0000	0000	
N'	−	×000.1	1111	1011	0.11?
	+	.1	1010	0000	↓
N''	−	×0	0.101	1011	.1101?
	+		.110	1000	↓
N'''	+		×000	.1101	.1101100?
	−			.1101	↓
$N^{(iv)}$	−			.0000	.11011001000 · · ·

Rapid Division in Computers. For a serial computer we again must have a shift register rather than a delay line for this rapid-division technique. The normalizing and differencing would occur in the same recirculation time. A circuit must sense the zeros of the result so as to form the proper bits of the quotient. For a parallel computer, again, as in rapid multiplication, the only way time can be saved is by shifting and adding or subtracting all in one unit time.

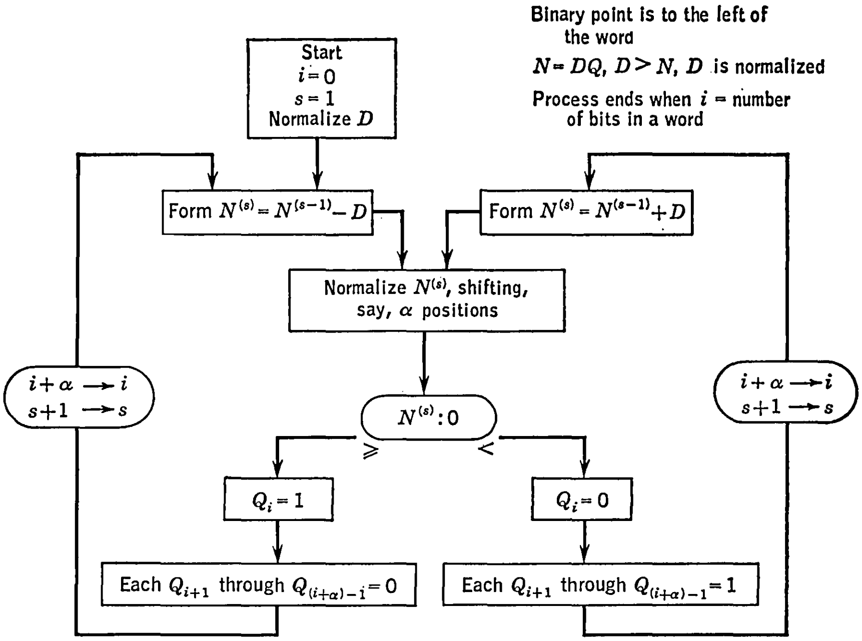

Binary point is to the left of the word
N = DQ, D > N, D is normalized
Process ends when i = number of bits in a word
Start
i = 0
s = 1
Normalize D
Form N^(s) = N^(s−1) − D
Form N^(s) = N^(s−1) + D
Normalize N^(s), shifting, say, α positions
i + α → i
s + 1 → s
N^(s) : 0
≥
<
Q_i = 1
Q_i = 0
Each Q_{i+1} through Q_{(i+α)−1} = 0
Each Q_{i+1} through Q_{(i+α)−1} = 1
i + α → i
s + 1 → s

FIG. 16-8. Flow chart for rapid division.

Rationale of Rapid Division. Some preliminary observations will aid the subsequent discussion of rapid division. First note that

$$2^{-\alpha} = .\overset{1}{0}\;\overset{2}{0}\cdots 0\;\overset{\alpha}{1}\;0\;0\cdots \qquad \text{and} \qquad 1-2^{-\alpha} = .\overset{1}{1}\;\overset{2}{1}\cdots 1\;\overset{\alpha}{1}\;0\;0$$

Next note that, if $(.1\cdots)\times(X) = .00\cdots\overset{\alpha}{0}1\cdots$, then

$$X = .00\cdots 0\begin{cases}\overset{\alpha}{0}\;1\cdots\\ 1\cdots\end{cases}$$

For example, if $(.1\cdots)\times(X) = .00001\cdots$, then

$$\begin{array}{rl} & .1\cdots \\ \times & .000\overset{\alpha}{1}\cdots \\ \hline & .00001\cdots \end{array}$$

or else

$$\begin{array}{rl} & .1\cdots \\ \times & .0000\overset{\alpha}{1} \\ \hline & .0000\underset{\curvearrowleft}{1}\ \text{Carry} \end{array}$$

Next observe that

$$\begin{aligned}1 - (.00 \cdots 0\overset{\alpha}{1} \cdots &\text{ followed by at least one other unit in any place)}\\ &= (.1\ 1 \cdots 1\ \overset{\alpha}{0} \cdots)\end{aligned}$$

whereas

$$\begin{aligned}1 - (.0\ 0 \cdots 0\ \overset{\alpha}{1}\ 0\ 0 \cdots &\text{ followed by all zeros)}\\ &= (.1\ 1 \cdots 1\ \overset{\alpha}{1}\ 0\ 0 \cdots)\end{aligned}$$

Our initial conditions are $N = DQ$, $D > N$, $2N > D$, $D = .1 \cdots$. The first step is to form $N' = N - D = D(Q - 1)$, where $N' < 0$. Suppose that $N' = -.\overset{1}{0}\ \overset{2}{0} \cdots \overset{\alpha}{0}\ 1 \cdots$; then we have

$$Q - 1 = -.\overset{1}{0}\ \overset{2}{0} \cdots 0 \left\{ \begin{matrix} \overset{\alpha}{0}\ 1 \cdots \\ 1 \cdots \end{matrix} \right.$$

whence $Q = .\overset{1}{1}\ \overset{2}{1} \cdots 1 \left\{ \begin{matrix} \overset{\alpha}{0} \\ 1 \end{matrix} \right. + 2^{-\alpha}Q'$, where $Q' < 1$. Thus we have determined the first $\alpha - 1$ bits of Q.

Next form $N'' = N' + 2^{-\alpha}D = D(Q - 1 + 2^{-\alpha})$, where $D > 0$. Three cases may arise:

1. $N'' = 0$. Then $Q = 1 - 2^{-\alpha}$, whence $Q = .1\ 1 \cdots 1\ \overset{\alpha}{1} + 2^{-\alpha}Q'$, where clearly Q' is zero and the division is concluded.

2. $N'' > 0$. Then $Q > 1 - 2^{-\alpha}$, or $Q > .1\ 1 \cdots \overset{\alpha}{1}$, whence

$$Q = .\overset{1}{1}\ \overset{2}{1} \cdots 1\ \overset{\alpha}{1} + 2^{-\alpha}Q'$$

Hence $Q - 1 + 2^{-\alpha} = .1\ 1 \cdots 1\ \overset{\alpha}{1} + 2^{-\alpha}Q' - 1 + 2^{-\alpha} = 2^{-\alpha}Q'$ and $2^{\alpha}N'' = DQ'$. If $2^{\alpha}N'' = .\overset{1}{0}\ \overset{2}{0} \cdots \overset{\beta}{0}\ 1 \cdots$, then

$$Q' = .\overset{1}{0}\ \overset{2}{0} \cdots 0 \left\{ \begin{matrix} \overset{\beta}{0}\ 1 \cdots \\ 1 \cdots \end{matrix} \right. = .\overset{1}{0}\ \overset{2}{0} \cdots \overset{\beta-1}{0} \left\{ \begin{matrix} 0 \\ 1 \end{matrix} \right. + 2^{-\beta}Q''$$

3. $N'' < 0$. Then $Q < 1 - 2^{-\alpha}$, or $Q < 0.\overset{1}{1}\ \overset{2}{1} \cdots \overset{\alpha}{1}$, whence

$$Q = .1\ 1 \cdots \overset{\alpha-1}{1}\ 0 + 2^{-\alpha}Q'$$

Hence

$$\begin{aligned}Q - 1 + 2^{-\alpha} &= .\overset{1}{1}\ \overset{2}{1} \cdots \overset{\alpha-1}{1}\ 0 + 2^{-\alpha}Q' - 1 + 2^{-\alpha}\\ &= 1 - 2^{-(\alpha-1)} + 2^{-\alpha}Q' - 1 + 2^{-\alpha}\\ &= 2^{-\alpha}(1 - 2 + Q') = 2^{-\alpha}(Q' - 1)\end{aligned}$$

whence $2^{\alpha}N'' = D(Q' - 1)$. If $2^{\alpha}N'' = -.\overset{1}{0}\,\overset{2}{0}\cdots\overset{\beta-1}{0}\,1\cdots$, then

$$Q' - 1 = -.\overset{1}{0}\,\overset{2}{0}\cdots\overset{\beta-1}{0}\left\{\begin{matrix}0\,1\cdots\\1\cdots\end{matrix}\right.$$

and

$$Q' = .\overset{1}{1}\,\overset{2}{1}\cdots\overset{\beta-1}{1}\left\{\begin{matrix}0\\1\end{matrix}\right. + 2^{-\beta}Q''$$

Thus we have now completely determined at least $\alpha + (\beta - 1)$ positions of Q. Consider $N'' > 0$, and form

$$N''' = N'' - 2^{-(\alpha+\beta)}D = 2^{-\alpha}D(Q' - 2^{-\beta})$$

Three cases may arise:

4. $N''' = 0$. Then $Q' = 2^{-\beta} = .\overset{1}{0}\,\overset{2}{0}\cdots\overset{\beta-1}{0}\,1 + 2^{-\beta}Q''$, where clearly Q'' is zero, and the division is concluded.

5. $N''' > 0$. Then $Q' > 2^{-\beta}$, or $Q' = +0.\overset{1}{0}\,\overset{2}{0}\cdots 0\,\overset{\beta}{1} + 2^{-\beta}Q''$. Hence, $Q' - 2^{-\beta} = .\overset{1}{0}\,\overset{2}{0}\cdots 0\,\overset{\beta}{1} + 2^{-\beta}Q'' - 2^{-\beta} = 2^{-\beta}Q''$, and

$$2^{\alpha+\beta}N''' = DQ''$$

which is just like case 2 over again.

6. $N''' < 0$. Then $Q' < 2^{-\beta}, Q' = +.\overset{1}{0}\,\overset{2}{0}\cdots 0\,\overset{\beta}{0} + 2^{-\beta}Q''$. Hence

$$Q' - 2^{-\beta} = .\overset{1}{0}\,\overset{2}{0}\cdots 0\,\overset{\beta}{0} + 2^{-\beta}Q'' - 2^{-\beta} = 2^{-\beta}(Q'' - 1)$$

and

$$2^{\alpha+\beta}N''' = D(Q'' - 1)$$

which is just like case 3 over again. Thus for $N'' \geq 0$ we have completely determined at least $\alpha + \beta + (\gamma - 1)$ positions of Q, where γ is the number of zeros beginning N'''; we have noted that case 4 refers to case 1 and case 6 refers to case 3 to complete the loops in these cases.

Consider $N'' < 0$, and form

$$N''' = N'' + 2^{-(\alpha+\beta)}D = 2^{-\alpha}D(Q' - 1 + 2^{-\beta})$$

Again three cases may arise, which are just like cases 1, 2, and 3. This completes the rationale for our rapid-division method.

EXERCISES

(*a*) Design a circuit that counts the zeros and generates the appropriate bits of the quotient during the rapid-division process.

(*b*) Describe in detail rapid division in serial and parallel computers.

(*c*) Construct an example where the rapid-division method presented in this example does not reduce the number of differences taken over that of the previous method given. Additional Topic *e* (Sec. 16-7) shows how to overcome this slowness.

16-6. Floating Operations

Addition and Subtraction. As described in Chap. 4, a word in a floating-point computer is divided into a "mantissa" and an exponent. In performing summation the exponents of the two arguments must be the same. In order that the least amount of significance be lost, the best procedure is to normalize both arguments first (i.e., adjust the exponent and shift the mantissa so that the most significant bit is a unit) and then shift the mantissa of the argument with the smallest exponent to the right until its exponent is the same as the largest of the previously normalized exponents. For example, for a mantissa having 8 bits, consider $0000\ 1101 \times 2^4 + 0110\ 1000 \times 2^{-3}$. On normalizing, these become $1101\ 0000 \times 2^0 + 1101\ 0000 \times 2^{-4}$; adjusting the smallest exponent to equal the largest, we have

$$1101\ 0000 \times 2^0 + 0000\ 1101 \times 2^0 = 1101\ 1101 \times 2^0$$

as the sum. In the event that there is an overflow, the exponent of the result must be increased by 1 and the resulting mantissa shifted right one position so that the overflow digit assumes the most significant position. Note that with this operation the result will always be normalized automatically. If an unnormalized result is desired, it is understood by convention (see Chap. 4) that the exponent of the result is to be that of the largest exponent of the *original* arguments. Thus an additional right shift would be necessary until the appropriate exponent is reached. Differencing is performed in the same way, normalizing first and then subtracting.

The process of normalizing a number consists in shifting the number to the left until a unit appears in the most significant position. The number of positions shifted is recorded on a counter, and the final count is subtracted from the exponent.

Multiplication and Division. In floating-point multiplication there is no need to normalize the numbers first, although there is no harm in doing so (why?). The exponents that are to be attached to the unnormalized major and minor products depend on the location of the binary point in the mantissa part of the word. For example, suppose that the point is to the left; then we would have, say, for a 5-bit mantissa:

$$\begin{array}{rl}
 & .01010 \times 2^a \\
 & \underline{.01101 \times 2^b} \\
 & 01010 \\
0 & 0000 \\
01 & 010 \\
010 & 10 \\
.0000 & 0 \\
\hline
\underbrace{.00100}_{\text{Major}} & \underbrace{00010}_{\text{Minor}} \times 2^{a+b}
\end{array}$$

Thus the major product would become $.00100 \times 2^{a+b}$, while the minor product would have to be written as $.00010 \times 2^{a+b-5}$. Hence the exponent of the major product is just the sum of the exponents of the arguments, while the exponent of the minor product is the sum of the exponents of the arguments minus the number of bits in the mantissa part of the word format. On the other hand suppose that the binary point was to the right; then we would find, for example,

$$\begin{array}{rl} & 01010. \times 2^a \\ & 01101. \times 2^b \\ \hline \underbrace{00100}_{\text{Major}} & \underbrace{00010.}_{\text{Minor}} \times 2^{a+b} \end{array}$$

Here the major product becomes $00100. \times 2^{a+b+5}$, while the minor product becomes $00010. \times 2^{a+b}$ (see Exercise *b*). When normalized major or minor products are desired, the exponent of the respective product is adjusted accordingly. For example, in the former example, the normalized major product would be $.10000 \times 2^{a+b-2}$, and the normalized minor product would be $.10000 \times 2^{a+b-5-3}$.

Division, on the other hand, requires more analysis in order to obtain the maximum significance in the result. Suppose, for example, that the binary point of the mantissa is always understood to be at the left; i.e., all mantissas are less than 1. Now, to obtain the maximum significance, we desire the quotient to have a unit in the most significant position. This can be arranged as follows: First normalize both arguments; then compute $2N - D$, which will be positive.† If $2N - D < 1$, proceed; if $2N - D \geq 1$, shift N to the right one position. Again compute $2N - D$, which is positive‡ and less than 1. With $2N - D > 0$, $Q_1 = 1$, and the division then proceeds as usual. The exponents and N and D must be adjusted after each shift. The quotient has thus been automatically normalized. The adjusted exponents are subtracted.

The Exponent Unit. In order to facilitate handling the exponents, it is sometimes desirable to have a separate *exponent unit* in which the exponents can be added and subtracted. Since the normalization and other shifting operations require addition (or subtraction) of a count to an exponent, this process may be accomplished in the exponent unit also. The exponent unit should also be capable of sensing an exponent overflow or underflow, which may only occur during multiplication and division. It is probably wisest to have an error halt in this event, since if the computer tries to fix up the mantissas to avoid the overflow, they may be cleared or otherwise undesirably altered. Such overflows or underflows would indicate an error in coding in any event.

† Normalized, $\frac{1}{2} \leq N < 1$, and $\frac{1}{2} \leq D < 1$, or $-1 < -D \leq -\frac{1}{2}$; thus $0 < 2N - D < \frac{3}{2}$.

‡ Shifted N is $N/2$; doubling gives N again. Thus $-\frac{1}{2} < N - D < \frac{1}{2}$. But $2N - D \geq 1$, or $N < \frac{1}{2} + D/2$. Hence $N - D \geq \frac{1}{2} + D/2 - D = \frac{1}{2} - D/2$. But $-D > -1$; so $-D/2 > -\frac{1}{2}$. Therefore $N - D > 0$.

EXERCISES

(*a*) Design a parallel 4-bit exponent unit.

(*b*) Formulate a general rule for determining the exponent of the major product and the minor product for words of w bits in the mantissa and with binary point p positions from the right.

(*c*) How might significance be lost if in addition or subtraction the arguments were not normalized first? Give specific examples of each.

(*d*) How might significance be lost in division if Q was not normalized as it was formed? Give examples.

(*e*) In unrounded addition and subtraction is it necessary to normalize the arguments first to ensure maximum significance?

(*f*) For unnormalized division could significance be lost by not first preparing N and D as described above? Give an example.

(*g*) In what common computing device are mantissas of the arguments of division prepared automatically as the division starts? Explain.

(*h*) Discuss the design of floating operations in a decimal-coded binary computer.

16-7. Additional Topics

a. Derive a formula giving the effectiveness of the rapid-multiplication scheme of Sec. 16-4 as summarized in Fig. 16-7. See, for example, J. L. Smith and A. Weinberger, Shortcut Multiplication for Binary Digital Computers, in *NBS Circ.* 591.

b. Design a 53-bit parallel binary adder so that no single pulse need travel through more than 12 gates, under the limitations given in Sec. 16-2. See, for example, A. Weinberger and J. L. Smith, A Logic for High-speed Addition, in *NBS Circ.* 591 (see also their A One-microsecond Adder Using One-megacycle Circuitry, *IRE Trans. on Electronic Computers,* vol. EC-5, no. 2, pp. 65–72, June, 1956).

c. Design a 12-bit parallel comparator.

d. Utilizing only *or-not* gates (that is, A *nor* $B = \overline{A + B}$), design a 36-bit parallel adder. Assume that each gate can drive at most six additional gates and that no gate can have more than six inputs (other than a clock pulse). (For notation see Sec. 18-2, Fig. 18-3.) Show that no pulse need travel through more than nine gates.

e. The method for rapid division given in Sec. 16-5 was developed by R. S. Ledley and J. B. Wilson. Modification of the techniques presented there is necessary if full advantage is to be taken of the possibilities of the method. To see this, work out

$$.011000001000000001 \div .10000001 = .110000001$$

and

$$.1000101010001 \div .1101 = .101010101$$

following the flow chart of Fig. 16-8 exactly.

Case 1. The first example above illustrates overshifting. Work the example again, but this time shift the arguments one place *less* before the second differencing. Observe that, if $D = .10 \cdots$, $N^{(i)} = .01 \cdots$ with $Q^{(i)} = .10 \cdots$ or $.11 \cdots$, for $N^{(i)} = Q^{(i)}D$. (Show this by multiplying maximum and minimum values.) The effect of this is that normalizing $N^{(i)}$ shifts it one place beyond the first bit of $Q^{(i)}$. If $Q^{(i)} = .10 \cdots$, this is desired (explain, considering the 1 as an isolated unit), but not if $Q^{(i)} = .11 \cdots$. Note, however, that if $\frac{2}{3} < Q^{(i)} < \frac{3}{4}$, the first 1 may be considered either as an isolated unit or as the first of a string of units with an isolated zero as the next bit (why?). Thus if it can be shown that $\frac{1}{2} \leq Q^{(i)} < \frac{3}{4}$, then $N^{(i)}$ should be normalized; if it can be shown that $\frac{2}{3} < Q^{(i)} < 1$, then $N^{(i)}$ should be shifted one place *less* than to normalize; but if it can be shown that $\frac{2}{3} < Q^{(i)} < 1$, then either shift is permissible. With this in mind, show (by computing the maximum and minimum possible values for $Q^{(i)}$ in each case) that for $D = .10 \cdots$ the following rules apply:

Shift one place *less* than to normalize

1. If $D = .1000 \cdots$ and $N^{(i)}$ begins $11 \cdots$
2. If $D = .100 \cdots$ and $N^{(i)}$ begins $111 \cdots$
3. If $D = .10\Phi0 \cdots$ and $N^{(i)}$ begins $1111 \cdots$
4. If $D = .100 \cdots$ and $N^{(i)}$ begins $11\Phi11 \cdots$ *or*
   If $D = .10010 \cdots$ and $N^{(i)}$ begins $11\Phi1\Phi \cdots$

Otherwise normalize

Case 2. The second example above illustrates undershifting. Work the example again, but this time shift the arguments one place *more* before *each* differencing. Note that, if $D = .11 \cdots$ and $N^{(i)} = .011 \cdots$, then $Q^{(i)} = .10 \cdots$; normalizing $N^{(i)}$ shifts beyond this first unit of $Q^{(i)}$, as is desired. If $D = .11 \cdots$ and $N^{(i)} = .11 \cdots$, then $Q^{(i)} = .11 \cdots$; normalizing $N^{(i)}$ shifts *to* the first unit of $Q^{(i)}$, as is desired. However, if $D = .11 \cdots$ and $N^{(i)} = .10 \cdots$, we desire to shift beyond the first unit of $Q^{(i)}$ whenever that unit is an isolated unit. Remembering that, if $\frac{2}{3} < Q^{(i)} < \frac{3}{4}$, the first unit of $Q^{(i)}$ may be considered isolated or not, as is convenient, show that the following rules apply for $D = .11 \cdots$:

Shift one place *more* than to normalize

1. If $D = .11 \cdots$ and $N^{(i)}$ begins $1000 \cdots$
2. If $D = .111 \cdots$ and $N^{(i)}$ begins $100 \cdots$
3. If $D = .1111 \cdots$ and $N^{(i)}$ begins $10\Phi0 \cdots$
4. If $D = .11\Phi11 \cdots$ and $N^{(i)}$ begins $100 \cdots$ *or*
   If $D = .11\Phi1\Phi \cdots$ and $N^{(i)}$ begins $100\Phi0 \cdots$

Otherwise normalize

Show that the rules apply equally well for negative $N^{(i)}$, and hence for $Q^{(i)}$ beginning with a zero. (The above modifications of the rapid-division method are the work of J. B. Wilson.)

f. References on Rapid Arithmetic Operations

Burtsev, V. S.: "Accelerating Multiplication and Division in High-speed Digital Computers," Institute of Exact Mechanics and Computing Technique, Academy of Science of USSR, Moscow, 1958.

Lehman, M.: High-speed Digital Multiplication, *IRE Trans. on Electronic Computers,* vol. EC-6, no. 3, pp. 204–205, September, 1957.

Leiner, A. L., A. Weinberger, and J. L. Smith: System Design of Digital Computers at the National Bureau of Standards: Methods for High-speed Addition and Multiplication, *NBS Circ.* 591.

Nadler, M.: Division in Digital Computers by the Radix Method, *Stroje na Zpracovani,* Sbornik IV, pp. 79–100, Prague, 1956. (In Czechoslovakian.)

Reitwiesner, G. W.: Summary Discussion of Performing Binary Multiplication with the Fewest Possible Additions, *BRL Tech. Note* 113, Ballistic Research Laboratories, Aberdeen, Md., February, 1957.

Robertson, J. E.: A New Class of Digital Division Methods, *IRE Trans. on Electronic Computers,* vol. EC-7, no. 3, p. 218, September, 1958. (The method for rapid division given in Sec. 16-5 can also be developed by extending Robertson's methods as applied to binary numbers.)

Tocker, K. D.: Techniques of Multiplication and Division for Automatic Binary Computers, *Quart. J. Mech. Appl. Math.*, August, 1958, pp. 364–384.

Univ. Ill., Digital Computer Lab., Tech. Progr. Rept., October, 1956.

g. Arithmetic Units. Further information concerning arithmetic units may be found in the following references:

Gillies, D. B., R. E. Meagher, D. E. Muller, R. W. McKay, J. P. Nash, J. E. Robertson, and A. H. Taub: On the Design of a Very-high Speed Computer, *Univ. Ill., Digital Computer Lab., Rept.* 80, October, 1957.

Greenwald, S., R. D. Heuter, S. N. Alexander: SEAC, *NBS Circ.* 551, January, 1955.

Grisamore, N. T., L. S. Rotolo, and G. Uyehara: Logical Design Using the Stroke Function, *IRE Trans. on Electronic Computers,* vol. EC-7, no. 2, p. 181, June, 1958.

Leiner, A. L., W. A. Notz, J. L. Smith, and A. Weinberger: System Design of the SEAC and DYSEAC, *IRE Trans. on Electronic Computers,* vol. EC-3, no. 2, June, 1954.

Lubkin, S.: Decimal Location in Computing Machines, *Math. Tables and Other Aids to Computation,* vol. 3, pp. 44–50, 1948.

Phister, M., Jr.: "Logical Design of Digital Computers," John Wiley & Sons, Inc., New York, 1958.

Richards, R. K.: "Arithmetic Operations in Digital Computers," D. Van Nostrand Company, Inc., Princeton, N.J., 1955.

Smith, C. V. L.: "Electronic Digital Computers," McGraw-Hill Book Company, Inc., New York, 1959.

Staff of Engineering Research Associates Inc.: "High-speed Computing Devices," McGraw-Hill Book Company, Inc., New York, 1950.

Staff of Harvard Computing Laboratory: "Synthesis of Electronic Computing and Control Circuits," Harvard University Press, Cambridge, Mass., 1951.

CHAPTER 17

CONTROL

17-1. Introduction

Orientation. In the previous two chapters we considered how arithmetic operations can be performed by a digital computer. Our discussions were limited solely to the examination of various *gating configurations* that can produce the sum of two arguments, the difference, the product, the quotient, and so forth. The present chapter is concerned with necessary *control* that enables a computer to perform those operations automatically, directed by a sequence of instructions.

There are two aspects to the control functions: *arithmetic operation control* and *instruction sequencing control.* The former is concerned with the arithmetic unit, which needs control signals to perform each of the various operations as directed by the instruction. Some of these signals are generated *within* the arithmetic unit itself as the operation is being performed; others are generated *external to* the arithmetic unit, in the so-called "operations signal generator." The instruction sequencing control is concerned with control signals that must be generated to direct the automatic execution and sequencing of instructions so that a complete program can be computed. The sequencing-control functions involve both the *sequencing of the chain of instructions that constitute a program* and the *sequencing of phases during the execution of a single instruction.*

Role of the Phases. The design of control circuitry takes place in two steps. In the first, control signals are designed that enable the computer to perform all the necessary functions that take place *within* each phase; the second step is to design controls for properly sequencing the phases. Before proceeding with the present chapter, let us review the functions performed in each phase, as defined for each of the four-, three-, two-, and one-address systems. Of course the naming of the phases is arbitrary; the assignment of functions to the different stages is also arbitrary to a large extent. However, to be specific, we have given in Table 17-1 a summary of the conventions to be used in this text. The table illustrates the function assigned to each phase for "typical" instructions, based on the instruction definitions given in Chap. 3.

The functions to be distributed among the phases are of five kinds. *First* there is the function of *determining* the address of the next instruction. For the four-address system this is δ unless the instruction was a comparison, when it might be γ. For the three-, two-, and one-address

TABLE 17-1. SUMMARY OF THE FUNCTIONS ASSIGNED TO THE PHASES OF THE FOUR-, THREE-, TWO-, AND ONE-ADDRESS SYSTEMS ILLUSTRATED FOR "TYPICAL" INSTRUCTIONS†

	Phase 1	Phase 2	Phase 3	Phase 4
Four-address:				
Add.........	Select α; read (α) into accumulator	Select β; read (β) into arithmetic unit, and perform operation	Select γ; put (acc) into γ	Select δ; read (δ) into instruction register; decode operation
Compare.....	Select α; read (α) into accumulator	Select β; read (β) into arithmetic unit, and perform operation; determine address of next instruction	....	Select next instruction from (δ) or (γ); read it into instruction register; decode operation
Three-address:				
Add.........	Select α; read (α) into accumulator	Select β; read (β) into arithmetic unit, and perform operation; update current-address counter	Select γ; put (acc) into γ	Select next instruction; read it into instruction register; decode operation
Compare.....	Select α; read (α) into accumulator	Select β; read (β) into arithmetic unit, and perform operation; determine address of next instruction	....	Select next instruction; read it into instruction register; decode operation
Two-address:				
Add.........	Select α; read (α) into accumulator	Select β; read (β) into arithmetic unit, and perform operation; update current-address counter	....	Select next instruction; read it into instruction register; decode operation
Compare.....	....	Select α; read (α) into arithmetic unit, and perform operation; determine address of next instruction	....	Select next instruction; read it into instruction register; decode operation
Transfer.....	....	....	Select α; put (acc) into α	Select β; read (β) into instruction register; decode operation

TABLE 17-1. SUMMARY OF THE FUNCTIONS ASSIGNED TO THE PHASES OF THE FOUR-, THREE-, TWO-, AND ONE-ADDRESS SYSTEMS ILLUSTRATED FOR "TYPICAL" INSTRUCTIONS† (*Continued*)

	Phase 1	Phase 2	Phase 3	Phase 4
One-address:				
Add.........	Select instruction; read it into instruction register; decode operation; update current-address counter for next instruction	Select α; read (α) into arithmetic unit, and perform operation		
Jump........	Select instruction; read it into instruction register; decode operation; determine address of next instruction			
Transfer.....	Select instruction; read it into instruction register; decode operation; update current-address counter for next instruction	....	Select α; put (acc) into α	

† Note that in the one-address system phase 1 really corresponds to what is called phase 4 in the four-, three-, and two-address systems.

systems this address depends on the operation: for all operations other than *compare* or *jump*, the address of the next instruction is obtained simply by adding 1 to the current-address counter, i.e., *updating* the current-address counter. On the other hand, for a *jump* or *compare* instruction in three-address, depending on the outcome of the operation, the address of the next instruction can be γ; for two-address it can be β; for one-address it can be α. *Second* there is the function of the *selection* of an address. Having determined an address, the computer must locate this address in the memory. The address selected may be the address of an argument, such as α or β; or it may be the address into which a word is to be transferred; or it may be the address of the next instruction. *Third* there is the function of actually transferring a word between the memory and the computing unit, e.g., reading the contents of the selected address into the computing unit, or writing the contents of the accumulator into the selected address. *Fourth* is the operation *decoding* function; and *fifth* is the *performance* of the operation.

In the four-, three-, and two-address systems the functions are generally assigned to the phases so that phase 1 is concerned with reading the first argument into the arithmetic unit; phase 3 is concerned with writing a word out of the arithmetic unit; phase 2 is concerned with reading the second argument from the memory into the arithmetic unit, performing the operation, and subsequently determining the address of the next instruction; and phase 4 is concerned with the next instruction and decoding its operation. In the one-address system phase 1 *never occurs* in the sense just mentioned. On the other hand, since the naming of the phases is arbitrary, *what corresponds to phase* 4 *in a multiaddress system* is usually called phase 1 for a one-address system; phases 2 and 3 still keep their general characterization.

The one-address system also presents another peculiarity. Since phase 2 does not always occur, it is necessary to relegate the determination of the address of the next instruction to phase 1. However, since the negative *jump* instruction does not require an operation, the determination of the address of the next instruction can occur immediately following the decoding of the operation. This means that, for a negative *jump* instruction, the address of the next instruction is prepared even before the operation of the present instruction is performed.

It will be of use to recall that transferring a word between the computing unit and the memory consumes one minor cycle. Only phase 2, wherein operations are performed, may take more than one minor cycle.†

Outline of This Chapter. Before considering the specific problems associated with control we first consider a type of logical gating structure that occurs frequently in control gating, namely, the decoding circuit (Sec. 17-2). Second, consideration is given to the various functions that control circuits must perform within the phases. The method of teaching is through illustration. The illustrations are chosen from the Pedagac design, which has a one-address format. It is felt that the general characteristics and problems involved in control-circuit design can best be obtained by examining in detail a few specific but typical examples (Secs. 17-3 to 17-5). The discussion of the functions that occur within the phases ends with the consideration of memory-selection techniques (Sec. 17-6). Next the sequencing control of the phases themselves is considered (Sec. 17-7). Finally the problems involved in designing the counters and timing generating signals are considered.

EXERCISES

(*a*) In Table 17-1 what is the difference between the three- and four-address system with regard to the functions of the phases?

† An exception to the rule occurs in the Pedagac, whose *shift* instruction takes two minor cycles in phase 3.

(*b*) Why does the *compare* operation have no phase 3?

(*c*) Could the decoding of the operation be accomplished as well in phase 1 in the four-address system? In the three-address system? In the two-address system?

(*d*) In the two-address system why is there no phase 1 or 3 for the comparison instruction? Why is there no phase 1 or 2 for the *transfer* instruction?

17-2. The Decoding Circuit†

Decoding Circuits. A *decoding circuit* has n independent input wires $A_1, A_2, \ldots, A_n$ and 2^n output wires; each output corresponds to one of the 2^n possible elementary products (see Sec. 11-3) of $A_1, A_2, \ldots, A_n$.

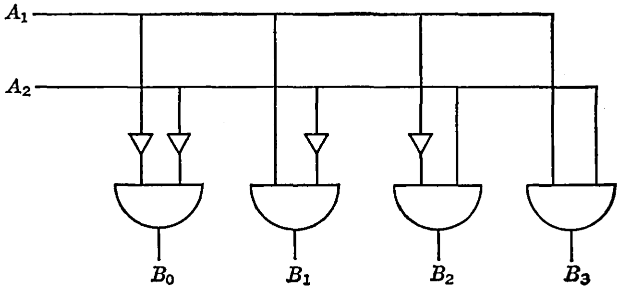


FIG. 17-1. The simplest nontrivial decoding circuit.

For example, the simplest nontrivial decoding circuit has 2 input wires and $2^2 = 4$ output wires as shown in Fig. 17-1. In general partial decoding circuits are used, where not all the 2^n possible outputs are actually produced. However, for the purposes of this section we shall assume throughout that all 2^n outputs always appear. Note that, by definition, the *outputs of a decoding circuit are mutually exclusive,* i.e., only one at a time can have a unit voltage.

When a decoding circuit has more than two inputs, it can be constructed with more than one stage, as shown, for example, in Fig. 17-2, where four inputs are used. As is clear from the figure, the number of gates used is greater when the gating is accomplished in more than one stage. However, the total number of *inputs to any gate* is less when more than one stage is used. In Fig. 17-2 the one-stage circuit requires $4 \times 2^4 = 64$ inputs, whereas the other circuit requires only

$$2 \times 2^2 + 2 \times 2^2 + 2 \times 2^4 = 48 \text{ inputs}$$

In this respect two situations can occur: the amount of electronic hardware needed to construct a circuit may be measured primarily by the number of gates or, on the other hand, by the number of gate inputs. In the former situation it is clearly more efficient to perform the decoding function all in one stage, while the reverse is true for the latter situation.

Diode Decoding Matrices. To see how this latter situation arises, consider diode gates, as discussed in Sec. 10-10. In Fig. 17-3 the top diagram

† This section is based on private communications with Chester Page of the National Bureau of Standards.

represents the circuit of Fig. 17-1 redrawn in terms of diode gates. It is easily seen that the positions of the diodes can be rearranged as in the bottom diagram of Fig. 17-3. In this arrangement the diodes are called *diode jumpers,* and such a circuit is called a *diode decoding matrix.* It is often found very convenient to place diodes at the crossings of the pairs of wires, and such diode matrices are used extensively. In diode matrices it is often important to *reduce to a minimum the number of diodes used,* which corresponds to reducing the number of inputs to the gates to a minimum.

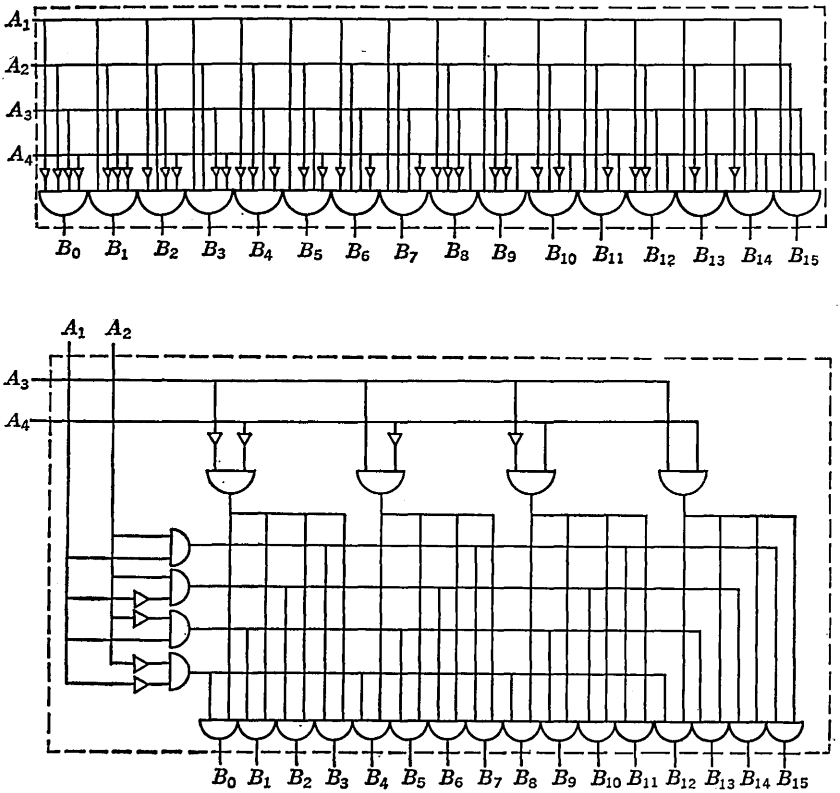

A_1
A_2
A_3
A_4
B_0 B_1 B_2 B_3 B_4 B_5 B_6 B_7 B_8 B_9 B_{10} B_{11} B_{12} B_{13} B_{14} B_{15}
A_1 A_2
A_3
A_4
B_0 B_1 B_2 B_3 B_4 B_5 B_6 B_7 B_8 B_9 B_{10} B_{11} B_{12} B_{13} B_{14} B_{15}

FIG. 17-2. Two ways of constructing a decoding circuit with four inputs A_1, A_2, A_3, and A_4. In our notation the top circuit is represented by $\{A_1A_2A_3A_4\}$, and the bottom circuit is represented by $\{\{A_1A_2\}\{A_3A_4\}\}$.

Computing the Number of Inputs to the Gates. The problem therefore arises as to what configuration of stages will result in the most efficient decoding circuit from the point of view of minimizing the total number of *inputs to the gates.* In order to facilitate this discussion, let us introduce some special notation. Let $\{A_1A_2 \cdot \cdot \cdot A_n\}$ represent a circuit with n inputs $A_1, A_2, \ldots, A_n$ and as many outputs as there are *possible input states.* The outputs are mutually exclusive, each having a

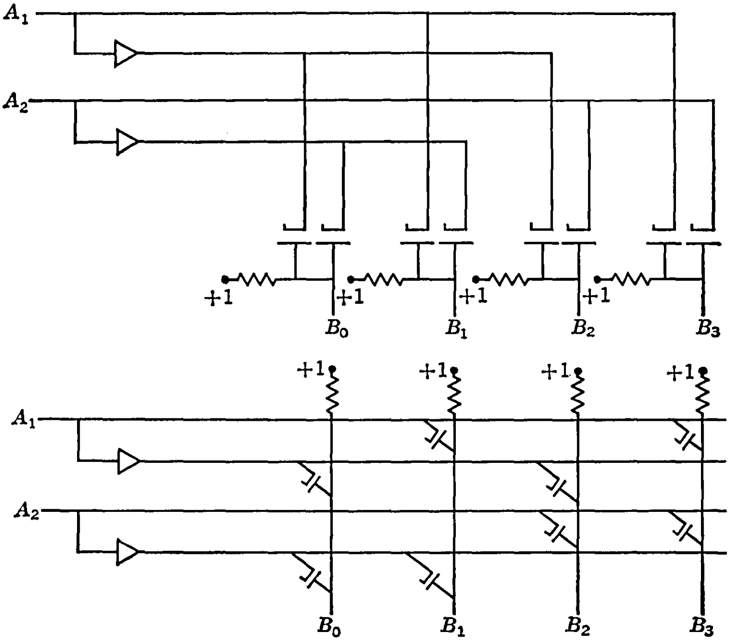


FIG. 17-3. The top diagram shows the use of diode gating in the circuit of Fig. 17-1. The bottom diagram shows the same circuit redrawn as a "jump" diode decoding matrix. (The crossed wires are not connected.)

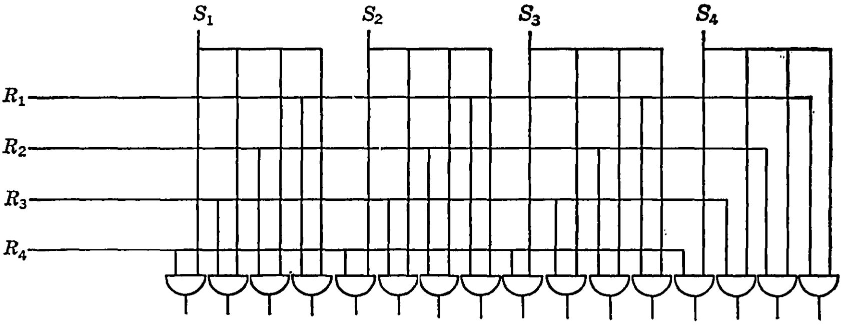


FIG. 17-4. Circuit corresponding to the expression $\{R_1R_2R_3R_4S_1S_2S_3S_4\}$, wherein only one at a time of the R_i and only one at a time of the S_i can have a unit signal voltage.

unit signal voltage for only one corresponding input state. For example, if $A_1, A_2, \ldots, A_n$ are *independent,* then $\{A_1A_2 \cdots A_n\}$ represents a one-stage decoding circuit, with 2^n outputs. As another example, if R_1, R_2, R_3, R_4 are *mutually exclusive* and S_1, S_2, S_3, S_4 are also mutually exclusive, then $\{R_1R_2R_3R_4S_1S_2S_3S_4\}$ will have $4 \times 4 = 16$ outputs (see Fig. 17-4).

Expressions with *multiple braces* are interpreted as circuits formed by stages "from inside out," similar to the use of parentheses in ordinary algebra. For example, observe that the circuit $\{A_1A_2\}$ (that is, the circuit of Fig. 17-1) and the circuit $\{A_3A_4\}$ each have four mutually exclusive outputs. Then the expression $\{\{A_1A_2\}\{A_3A_4\}\}$ represents a circuit similar to that of Fig. 17-4, but where $R_1R_2R_3R_4$ and $S_1S_2S_3S_4$ are replaced by the mutually exclusive outputs of $\{A_1A_2\}$ and $\{A_3A_4\}$. This is just the lower circuit of Fig. 17-2. Similarly the multiple-braced expression $\{\{A_1A_2\}A_3\}$ represents the circuit of Fig. 17-5. More complicated multiple-braced expressions can similarly be unambiguously interpreted as decoding circuits.

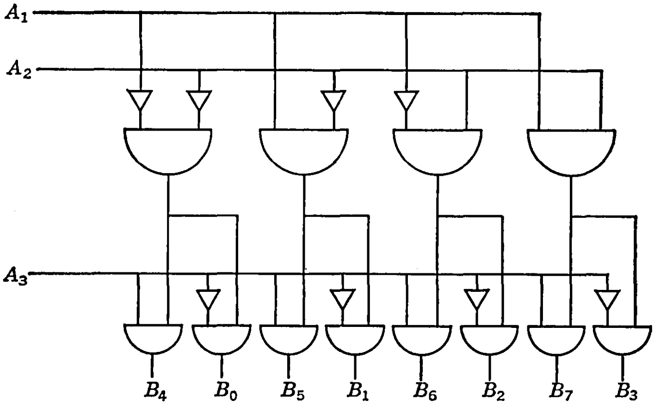


FIG. 17-5. The circuit corresponding to the expression $\{\{A_1A_2\}A_3\}$.

Now observe that the number of gates and hence the number of inputs to the gates can be easily calculated from the braced expression of a decoding circuit. For example, $\{A_1A_2 \cdots A_n\}$ clearly has 2^n gates and $n \times 2^n$ inputs to the gates (see Fig. 17-1 for $n = 2$ and Fig. 17-2 for $n = 4$). The circuit $\{R_1R_2 \cdots R_pS_1S_2 \cdots S_q\}$, where R_i and S_i are mutually exclusive within themselves, has pq gates and $2pq$ inputs to the gates (see Fig. 17-4 for $p = 4$, $q = 4$). For an expression with two levels of multiple bracing the *number of gates in the last stage* (and hence the number of mutually exclusive final outputs of the circuit) is given by

$$(\textit{Product of no. of mutually exclusive outputs of each of 1st-stage braced expressions}) \times 2^{(\textit{no. of unbraced terms})}$$

For example, the number of gates in the last stage of $\{\{A_1A_2\}\{A_3A_4\}\}$ is $2^2 \times 2^2 \times 2^0 = 16$ (see Fig. 17-4) and of $\{\{A_1A_2\}A_3\}$ is $2^2 \times 2^1 = 8$ (see Fig. 17-5); for the more complicated case $\{\{A_1A_2A_3\}\{A_4A_5\}A_6A_7A_8\}$ we have $2^3 \times 2^2 \times 2^3 = 256$. We can now compute the number of inputs to the gates of the last stage as

$$(\textit{No. of 1st-stage braced expressions} + \textit{no. of unbraced terms}) \times (\textit{no. of gates in last stage})$$

For example, for $\{\{A_1A_2\}\{A_3A_4\}\}$ the number of inputs to the gates in the last stage is $2 \times 16 = 32$, for $\{\{A_1A_2\}A_3\}$ we have $2 \times 8 = 16$, and for $\{\{A_1A_2A_3\}\{A_4A_5\}A_6A_7A_8\}$ we have $5 \times 256 = 1{,}280$.

Of course for the entire circuit of a multiply braced expression *the number of gates is the sum of the number of gates of each stage, and the number of inputs to the gates is the sum of the numbers for each stage.* For more than two levels of bracing the process is continued in the same manner. Thus for $\{\{A_1A_2\}\{A_1A_2\}\}$ we find a total of

$$2^2 + 2^2 + 2^2 \times 2^2 = 24 \text{ gates}$$

and $$2 \times 2^2 + 2 \times 2^2 + 2 \times 2^2 \times 2^2 = 48 \text{ inputs to the gates}$$

For $\{\{A_1A_2\}A_3\}$ we find a total of

$$2^2 + 2^2 \times 2^1 = 12 \text{ gates}$$

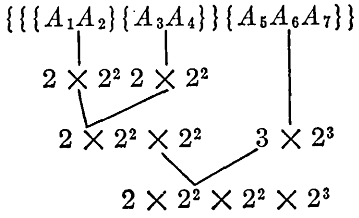
and $$2 \times 2^2 + 2 \times 2^2 \times 2^1 = 24 \text{ inputs to the gates}$$

For the expression $\{\{A_1A_2A_3\}\{A_4A_5\}A_6A_7A_8\}$ we have

$$2^3 + 2^2 + 2^3 \times 2^2 \times 2^3 = 268 \text{ gates}$$

and $$3 \times 2^3 + 2 \times 2^2 + 5 \times 2^3 \times 2^2 \times 2^3 = 1{,}312 \text{ inputs to the gates}$$

For a more complicated example including more than two levels of bracing consider the total number of inputs to the gates of the following expression (a more detailed discussion of brace or parenthesis interpretation can be found in Sec. 5-8):



whence $2 \times 2^2 + 2 \times 2^2 + 2 \times 2^2 \times 2^2 + 3 \times 2^3 + 2 \times 2^2 \times 2^2 \times 2^3 = 328$ inputs to the gates are needed.

Minimizing the Number of Inputs to the Gates. We are now ready to give a general *rule of thumb* that will enable the design of a decoding circuit with a minimum number of inputs to the gates. *The rule is as follows:* Start with a single braced expression that contains all the input wires: $\{A_1A_2 \cdots A_n\}$. Introduce another level of bracing that separates the input wires as closely as possible in half, thus:

$$\{\{A_1A_2 \cdots A_{n/2}\}\{A_{n/2+1}A_{n/2+2} \cdots A_n\}\}$$

if n is even; on the other hand if n is odd, then one of the two braces will have an extra wire. Continue the same process on each brace, until there remain only braces with either two or three wires. The reason it does not pay to break up a three-wire brace is that the number of inputs to the gates for $\{A_1A_2A_3\}$ is the same as for $\{\{A_1A_2\}A_3\}$, that is, $3 \times 2^3 = 2 \times 2^2 + 2 \times 2^2 \times 2^1 = 24$.

As an example, consider five input wires A_1, A_2, A_3, A_4, and A_5. The minimum decoding circuit is $\{\{A_1A_2A_3\}\{A_4A_5\}\}$ according to the above rule, and has $3 \times 2^3 + 2 \times 2^2 + 2 \times 2^3 \times 2^2 = 96$ inputs to the gates. For six inputs we have $\{\{A_1A_2A_3\}\{A_4A_5A_6\}\}$, which has

$$3 \times 2^3 + 3 \times 2^3 + 2 \times 2^3 \times 2^3 = 176 \text{ inputs to the gates}$$

For seven input wires we have $\{\{\{A_1A_2\}\{A_3A_4\}\}\{A_5A_6A_7\}\}$, which requires 328 inputs to the gates, as was computed above. It is instructive to display all the possible decoding circuits for five input wires. This is done in Table 17-2. The proof of our rule of thumb is beyond the scope of this text.

TABLE 17-2. ALL POSSIBLE (ESSENTIALLY DIFFERENT) DECODING CIRCUITS WITH FIVE INPUT WIRES

Circuit expressions	*Total number of inputs to the gates*
$\{A_1A_2A_3A_4A_5\}$	$5 \times 2^5 = 160$
$\{\{A_1A_2A_3A_4\}A_5\}$	$4 \times 2^4 + 2 \times 2^5 = 128$
$\{\{A_1A_2A_3\}A_4A_5\}$	$3 \times 2^3 + 3 \times 2^5 = 120$
$\{\{A_1A_2\}A_3A_4A_5\}$	$2 \times 2^2 + 4 \times 2^5 = 136$
$\{\{A_1A_2\}\{A_3A_4\}A_5\}$	$2 \times 2^2 + 2 \times 2^2 + 3 \times 2^5 = 112$
$\{\{A_1A_2A_3\}\{A_4A_5\}\}$	$3 \times 2^3 + 2 \times 2^2 + 2 \times 2^5 = 96$
$\{\{\{A_1A_2A_3\}A_4\}A_5\}$	$3 \times 2^3 + 2 \times 2^4 + 2 \times 2^5 = 120$
$\{\{\{A_1A_2\}A_3A_4\}A_5\}$	$2 \times 2^2 + 3 \times 2^4 + 2 \times 2^5 = 120$
$\{\{\{A_1A_2\}A_3\}A_4A_5\}$	$2 \times 2^2 + 2 \times 2^3 + 3 \times 2^5 = 120$
$\{\{\{A_1A_2\}A_3\}\{A_4A_5\}\}$	$2 \times 2^2 + 2 \times 2^3 + 2 \times 2^2 + 2 \times 2^5 = 96$
$\{\{\{A_1A_2\}\{A_3A_4\}\}A_5\}$	$2 \times 2^2 + 2 \times 2^2 + 2 \times 2^4 + 2 \times 2^5 = 112$
$\{\{\{\{A_1A_2\}A_3\}A_4\}A_5\}$	$2 \times 2^2 + 2 \times 2^3 + 2 \times 2^4 + 2 \times 2^5 = 120$

EXERCISES

(*a*) Draw the gating circuit for $\{\{\{A_1A_2\}\{A_3A_4\}\}A_5\}$.

(*b*) Draw the diode decoding matrix corresponding to $\{\{\{A_1A_2\}\{A_3A_4\}\}A_5\}$.

(*c*) Form all possible essentially different braced expressions for a decoding circuit with four input wires, and find the number of inputs to the gates of each.

(*d*) Why do the expressions $\{\{A_1A_2A_3\}\{A_4A_5\}\}$ and $\{\{\{A_1A_2\}A_3\}\{A_4A_5\}\}$ both have 96 inputs to the gates (see Table 17-2)?

(*e*) For a decoding circuit with six input wires form the braced expression that will minimize the number of inputs to the gates, and determine what this minimum number is.

(*f*) For a decoding circuit with nine input wires form the braced expression that will minimize the number of diodes necessary in the diode matrix decoding circuit, and determine how many diodes are required.

(*g*) List all possible (essentially different) braced expressions for a decoding circuit with seven input wires, and determine the total number of inputs to the gates for each expression.

(*h*) Every decoding circuit has associated with it an *encoding circuit* which when attached to the decoding circuit will yield the original inputs. Find the encoding circuit associated with the decoding circuit of Fig. 17-1. [HINT: Solve the equation $B_0 = \bar{A}_1 \cdot \bar{A}_2$, $B_1 = A_1 \cdot \bar{A}_2$, $B_2 = \bar{A}_1 \cdot A_2$, and $B_3 = A_1 \cdot A_2$ for $A_1 = f_1(B_0,B_1,B_2,B_3)$ and $A_2 = f_2(B_0,B_1,B_2,B_3)$.]

17-3. Arithmetic Control: Instruction Decoder and Operations Signal Generator

Instruction Decoder. Control of an arithmetic operation starts in the *instruction decoder,* where the operation is decoded. The bits stored in the operation positions of the instruction register are the inputs for the decoder, and the decoder has one output for each operation that can be performed by the computer. In the previous section we have discussed aspects of the design of such decoding circuits. The outputs of the instruction decoder are the inputs to the *operations signal generator.* Here the control signals necessary for each arithmetic operation are generated. These control signals then become inputs to the circuits of the arithmetic unit, together with the arguments of the operation (see Fig. 17-6).

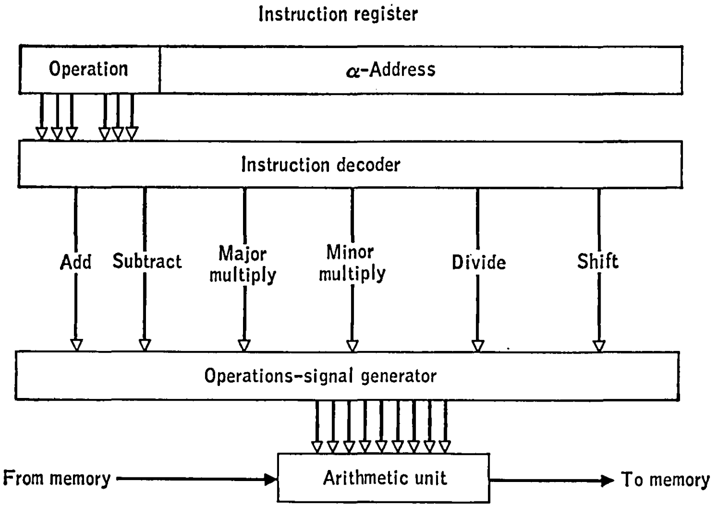


FIG. 17-6. Arithmetic control.

The word *detail* characterizes the essential nature of most control circuits. Their design depends on the circumstances of the computer characteristics and of the particular function to be controlled. The instruction decoder and operations signal generator are no exceptions to the generalization. Therefore it is felt that the nature of the fundamental problems involved can best be described in terms of a specific example of a particular control circuit. We choose for our illustration the control of certain arithmetic operations of the Pedagac (see Chap. 9).

The operations (and their codes) under consideration will be: addition (53), subtraction (54), major multiplication (42), minor multiplication (32), division (41), and circular shift (70).

Starting with the instruction decoder, recall that the leftmost 6 bits represent the operation of the instruction word. Calling these ω_6, ω_5, ω_4, ω_3, ω_2, and ω_1, respectively, a possible decoder appears in Fig. 17-7. The method used is to decode each octal position separately and then to combine the proper octal digits to form the operation signal wire output, labeled with the respective operation code followed by the letter D.

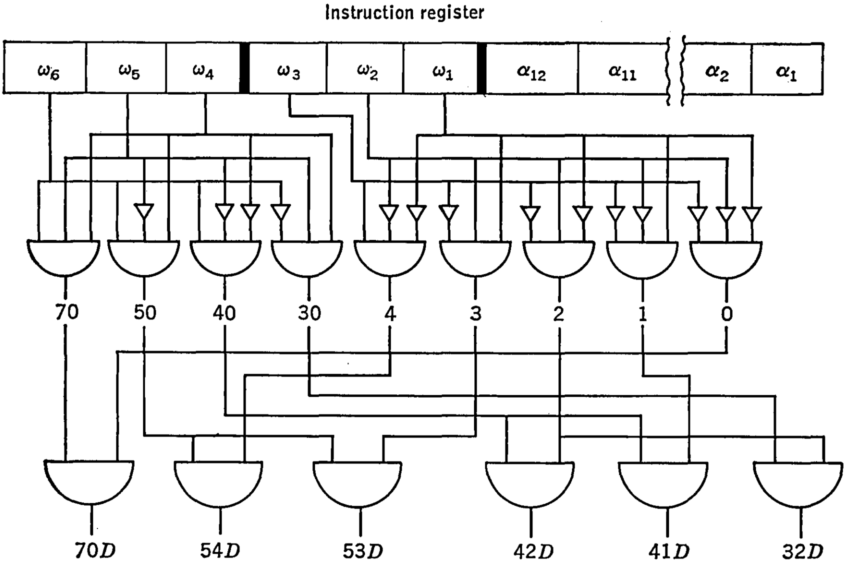


FIG. 17-7. Example of an instruction decoder.

The Operations Signal Generator. Continuing with our illustration, these six operation signals become inputs to the operations signal generator. Here they are gated with each other and with unit-time-interval pulses and with minor-cycle pulses to form the control signals that control the operations under consideration. The gating performed on the input signals in the operations signal generator varies in complexity from no gating at all to extensive gating combinations. The simplest of the signals in our illustration is the shift signal, called SHS, which is on whenever the *shift* instruction is called for, that is, [SHS] = [70D]†; thus no gating is necessary. Another simply generated control signal is required whenever major or minor multiplication is called for; let this be called the multiplication decode signal MDS; thus, [MDS] = [42D] + [32D]. A similar signal is on whenever multiplication or division is on, namely, [PQS] = [42D] + [41D] + [32D] (see Fig. 17-8).

† The brackets on the multiletter symbols are for clarity; they have no intrinsic meaning.

A more complicated type of control signal often generated is exemplified in the division control signal, called DCS. This signal is on during the computational part of the division process. Since there are 18 bits (plus a sign) in the word under consideration, it takes 18 minor cycles to form the quotient. On the other hand the computation proceeds only during T_1 through T_{18} of each minor cycle, even though the minor cycle is 23 unit time intervals in length. Thus we desire DCS to be off during T_0 and T_{19-22} of each of the 18 minor cycles. The one-address division instruction actually takes 19 minor cycles, the 19th minor cycle being utilized to transfer the quotient from the ier register where it was formed to the accumulator where the result is to be found. However, we do not want DCS to be on during this 19th minor cycle when no actual computing is done. Hence we have

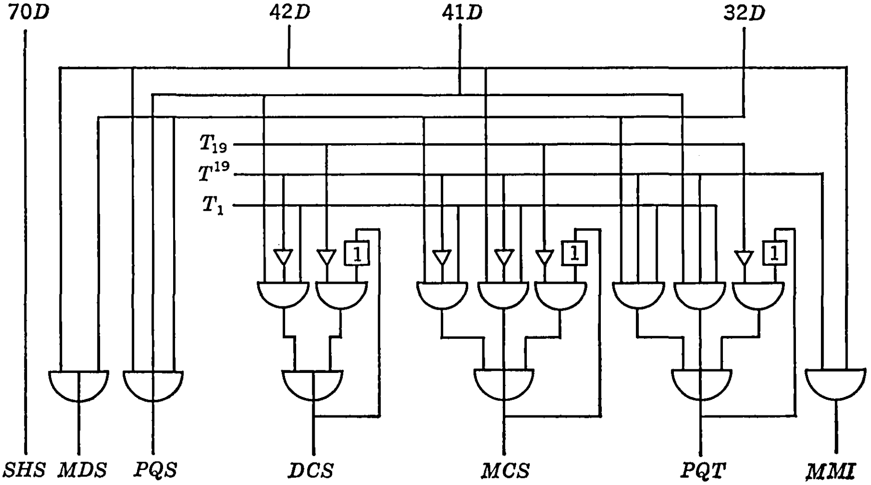


FIG. 17-8. Part of control signal generator.

$$[\text{DCS}] = [\overline{T^{19}}] \cdot [T_1] \cdot [41D] + [\bar{T}_{19}] \cdot [\text{DCS}]_d$$

where $[\text{DCS}]_d$ represents the recirculation of [DCS]. The left-hand product loads the circuit at T_1 of every minor cycle except T^{19} during the *divide* instruction (41); the right-hand product keeps this signal on from T_1 through T_{18} and turns it off at T_{19}. We have here of course a dynamic flip-flop (see Fig. 17-8).

A signal similar in nature to DCS is the multiply control signal MCS. This signal is to be on during the computational part of major or minor multiplication, i.e., from T_1 through T_{18} of minor cycle T^1 through T^{18}. Thus we have

$$[\text{MCS}] = [\overline{T^{19}}] \cdot [T_1] \cdot [32D] + [\overline{T^{19}}] \cdot [T_1] \cdot [42D] + [\bar{T}_{19}] \cdot [\text{MCS}]_d$$

The product-quotient transfer signal PQT controls the transfer of the quotient or the minor product from the ier register to the accumulator during T_{1-18} of the 19th minor cycle, thus:

$$[\text{PQT}] = [T^{19}] \cdot [T_1] \cdot [32D] + [T^{19}] \cdot [T_1] \cdot [41D] + [\bar{T}_{19}] \cdot [\text{PQT}]_d$$

For major multiplication the 19th minor cycle is used only to shift the accumulator to the right one position, i.e., at $T_0{}^{19}$, to transfer the most significant bit of the minor product which remains in the accumulator at the end of the 18th minor cycle. Hence for major multiplication a signal is needed that inhibits the accumulator from shifting after T_0 of the 19th minor cycle, called the major multiplication inhibit signal, MMI; whence [MMI] = $[T^{19}] \cdot [42D]$ (see also Fig. 17-8), which will later be inhibited by T_0.

One of the most complicated operation control signals generated is the shift control signal. Here the number of positions to be shifted is found in the α address of the instruction. In order to control the number of positions to be shifted, a "count-down" is made from α each time the accumulator is shifted; when the count reaches zero, a signal is generated that inhibits any further shifting of the accumulator. This inhibiting signal is called SCC (see Exercise *b*).

Initial Clearing of the Recirculation Loops. When a recirculation loop, or flip-flop, is to be used, the loop must be cleared to ensure that initially there is no bit recirculating in it. This can be accomplished in a manner similar to the clearing of a shift-register stage when it is being loaded, by gating the loading with a T_i signal and at the same time inhibiting any recirculation with a $\bar{T}_i$ signal. Often the clearing of a recirculation loop takes care of itself; i.e., it becomes cleared automatically because of the particular nature of the recirculation gates.

EXERCISES

(*a*) How are the dynamic flip-flops for the signals DCS, MCS, and PQT initially cleared? (HINT: Remember the sequencing of instructions.)

(*b*) Design a count-down circuit that starts with 5 bits, $\alpha 5$, $\alpha 4$, $\alpha 3$, $\alpha 2$, $\alpha 1$, for the shift control signal for an 18-bit word. Note that the initial count α can never be greater than 18, the maximum number of positions that can be shifted. Thus a constraint appears which can introduce significant simplifications. Assume that the count-down starts at T_5 (that is, since the length of the accumulator is 22 stages and the word length 18 bits, we have $22 - 18 + 1 = 5$). Hence at T_5 "load" each stage of the count-down circuit with αi, as the initial count. The circuit should then reduce the count by one each unit time interval. Call the count at any time $A_5A_4A_3A_2A_1$; then SCC, which is to come on when the count reaches zero, becomes SCC = $\bar{A}_5 \cdot \bar{A}_4 \cdot \bar{A}_3 \cdot \bar{A}_2 \cdot \bar{A}_1$. However, then SCC must remain on until the end of the minor cycle, whence SCC = $\bar{A}_5 \cdot \bar{A}_4 \cdot \bar{A}_3 \cdot \bar{A}_2 \cdot \bar{A}_1 + [\bar{T}_0] \cdot [\text{SCC}]_d$. One further remark must be made: The recirculation loops of the count-down circuit and of SCC *must be cleared* at T_5 to ensure that no miscellaneous pulses have preloaded the loops. Thus we have SCC = $\bar{A}_5 \cdot \bar{A}_4 \cdot \bar{A}_3 \cdot \bar{A}_2 \cdot \bar{A}_1 + [\bar{T}_5] \cdot [\bar{T}_0] \cdot [\text{SCC}]_d$. For the recirculation loops the "loading" gate is $[\alpha i] \cdot [T_5]$, while each recirculation gate must have a $\bar{T}_5$ in its *and* gates.

(*c*) Design a signal generator that will present an output for multiplication or division during T_1 through T_{18} of each minor cycle T^1 through T^{18}.

17-4. Arithmetic Control: Internally Generated Signals and Register Control

Internally Generated Signals. Some of the control signals used in the arithmetic unit are generated there during the execution of the operation. For example, as we have seen in Chap. 15, whether the adder-subtractor will form the sum or difference depends on a signal generated in the arithmetic unit from the signs of A and B and the operation signals for add or subtract. Also used to control processes within the arithmetic unit are the carry of the adder-subtractor, the overflow signal (which indicates that the subtractive complement of a number has been produced), the signal indicating that a result is zero (i.e., the zero signal), the multiply control signal (which controls whether or not the contents of the icand register or zero is to be added to the partial sum), and so forth, all generated within the arithmetic unit. The sign of the result may be considered as an internally generated signal; in a one-address system this sign will control a *negative jump* instruction.

As a specific example, consider the multiply control signal that determines whether to add zero or the multiplicand to the partial product. In the Pedagac this signal is called MUS. Because the Pedagac is a one-address computer, during the first minor cycle of a multiplication operation the multiplier is the recirculating contents of the accumulator, denoted by B. However, during this first minor cycle, B is also transferred to the ier register, and hence, during all minor cycles except the first, the multiplier is the contents of the multiplier register. Thus the first bit of the multiplier, sensed at $T_1{}^1$, appears as the least significant bit of B, whereas the ith bit of the multiplier, sensed at $T_1{}^i$ (that is, T_1 of the ith minor cycle), appears as the end bit of the (ier). If the bit of the multiplier sensed at $T_1{}^i$ is a unit, then signal MUS remains on for the rest of that minor cycle, opening the gate for the multiplicand to be added to the partial product; if the multiplier bit is a zero, then MUS remains off for the rest of that minor cycle. Hence we have for MUS

$$\begin{aligned}[\mathrm{MUS}] = {} & [B] \cdot [T^1] \cdot [T_1] \cdot [\mathrm{MDS}] \cdot [F2] \\ & + [\overline{T^1}] \cdot [T_1] \cdot (\mathrm{IER}) \cdot [\mathrm{MDS}] \cdot [F2] \\ & + [\bar{T}_0] \cdot [\mathrm{MUS}]_d \cdot [F2]\end{aligned}$$

The MDS signal is used to generate MUS during a multiplication instruction. The least significant bit of the ier register that keeps MUS on (or off) during a complete minor cycle is cleared at T_0 of every minor cycle so that a new control bit can be presented at T_1.

Register Shift Control. The shift registers of the arithmetic unit, such as the accumulator, icand, and ier registers, need two types of control, which are generated within the arithmetic unit. First there must be a control that shifts the register as required, called the register shift control; second, there must be control of the input to the register, called the register input control.

Continuing with our example started in Sec. 17-3, consider the accumulator shift control. Let us analyze all the circumstances under which the accumulator should be shifted. For addition, multiplication, and division (phase 2 operations) the accumulator should be shifted each unit time interval of a minor cycle, except at each T_0 when the signs of the arguments are sensed. Observe that for multiplication, *but not division*, the accumulator must be shifted at T_0 for minor cycles T^{2-19}, because at $T_0{}^{i+1}$ the ith bit of the minor product is transferred to the ier register. During the 19th minor cycle the shifting is continued throughout T^{19}_{1-22} for division and minor multiplication but not major multiplication, as observed in Sec. 17-3.

The *shift* instruction (a phase 3 operation) takes two minor cycles in the Pedagac. In the first minor cycle, (acc) is just recirculated through the output subtractive complementer; during the second minor cycle the shifting takes place. In this way the shifted result will always be complemented whenever necessary. For the *shift* instruction the accumulator should be shifted during T^1_{1-22}, and during the second minor cycle it should be shifted from T_5 until the count is zero.

We have collected in the previous paragraphs all the information needed to construct the accumulator shift control signal ASP. First observe that the term $[\bar{T}_0] \cdot [F2] \cdot [\overline{\text{MMI}}]$ will take care of addition and subtraction ($F2$ being on in this case for T^1 only) and multiplication and division during T^{1-19} (except for major multiplication, which is inhibited at T^{19}). However, we must add the term $[\overline{T^1}] \cdot [T_0] \cdot [F2] \cdot [\text{MDS}]$ so that T_0 will be on during T^{2-19} for multiplication and division. The term $[T^1] \cdot [\bar{T}_0] \cdot [F3] \cdot [\text{SHS}]$ shifts the accumulator for the first minor cycle of $F3$, and $[T^2] \cdot [\bar{T}_0] \cdot [F3] \cdot [\text{SHS}] \cdot [\overline{\text{SCC}}]$ shifts the accumulator appropriately during the second minor cycle, i.e., as directed by SCC. Thus we have all together

$$\begin{aligned}[\text{ASP}] = [\bar{T}_0] \cdot [F2] \cdot [\overline{\text{MMI}}] + [\overline{T^1}] \cdot [T_0] \cdot [F2] \cdot [\text{MDS}] \\ + [T^1] \cdot [\bar{T}_0] \cdot [F3] \cdot [\text{SHS}] + [T^2] \cdot [\bar{T}_0] \cdot [F3] \cdot [\text{SHS}] \cdot [\overline{\text{SCC}}]\end{aligned}$$

Adder Input Control. The direct input to the accumulator is particularly simple, being merely the output of the adder-subtractor. However, the input to the adder-subtractor is nontrivial and can be thought of as the virtual input to the accumulator. There are two inputs to the adder-subtractor, which we shall denote by A and B'.

Consider first the A input for our simplified example of six instructions of a one-address serial computer. Looking at the first minor cycle of phase 2 instructions, clearly A is (α) except for the multiplication instructions. For the multiplication instructions A is (α) provided that MUS is on, as was discussed above. Thus we have the following terms for A:

$$[T^1] \cdot [F2] \cdot (\alpha) \cdot [\overline{\text{MDS}}] + [T^1] \cdot [F2] \cdot (\alpha) \cdot [\text{MCS}] \cdot [\text{MUS}]$$

During both multiplication and division for the minor cycles after T^1,

A becomes the recirculating output of the icand register, except during multiplication when MUS is off. Hence we have the following terms:

$$[\overline{T^1}] \cdot [F2] \cdot (\text{ICN}) \cdot [\text{DCS}] + [\overline{T^1}] \cdot [F2] \cdot (\text{ICN}) \cdot [\text{MSC}] \cdot [\text{MUS}]$$

The *shift* instruction presents no input to A; that is, during phase 3, there is no A input. Hence

$$\begin{aligned} A = {} & [T^1] \cdot [F2] \cdot (\alpha) \cdot [\overline{\text{MDS}}] + [T^1] \cdot [F2] \cdot (\alpha) \cdot [\text{MCS}] \cdot [\text{MUS}] \\ & + [\overline{T^1}] \cdot [F2] \cdot (\text{ICN}) \cdot [\text{DCS}] + [\overline{T^1}] \cdot [F2] \cdot (\text{ICN}) \cdot [\text{MCS}] \cdot [\text{MUS}] \end{aligned}$$

For the B' input to the adder-subtractor, during addition or subtraction, we have the recirculating contents of the accumulator, called B: this gives B' the term $[F2] \cdot [T^1_{1-18}] \cdot [B] \cdot [\overline{\text{PQS}}]$. For division the B' input is the recirculating contents of the accumulator, which is the partial dividend delayed by 1, that is, B_d, during T^{1-18}; hence we have the term $[F2] \cdot [B]_d \cdot [\text{DCS}]$ (since DCS is only during T^{1-18}). For multiplication the B' input is the recirculating contents of the accumulator, i.e., the partial sum, and this holds for minor cycles T^{2-18}, but *not* for T^1; hence we have the term $[F2] \cdot [\overline{T^1}] \cdot [B] \cdot [\text{MSC}]$. Finally, for minor multiplication or division, during T^{19} the transfer from the ier to the accumulator takes place; hence the term $[F2] \cdot (\text{IER}) \cdot [\text{PQT}]$. The recirculation of the contents of the accumulator for the *shift* instruction also passes through the B' input; this gives the term $[F3] \cdot [B] \cdot [\bar{T}_0]$. Hence

$$\begin{aligned} B' = {} & [F2] \cdot [T^1_{1-18}] \cdot [B] \cdot [\overline{\text{PQS}}] + [F2] \cdot [B]_d \cdot [\text{DCS}] \\ & + [F2] \cdot [\overline{T^1}] \cdot [B] \cdot [\text{MCS}] + [F2] \cdot (\text{IER}) \cdot [\text{PQT}] + [F3] \cdot [B] \cdot [\bar{T}_0] \end{aligned}$$

EXERCISES

(*a*) Construct the signal that determines whether the adder-subtractor will form the sum or difference for a one-address serial computer with the instructions discussed in this section.

(*b*) Explain why no addition or subtraction control signal explicitly appears in the expression constructed for ASP.

(*c*) Why are signals MDS and MCS both necessary for the construction of A?

(*d*) In the construction of B' why is there no input during T^1 of multiplication?

(*e*) Describe the shift control of the icand register for a one-address serial computer with the operations considered in this section.

(*f*) Describe the icand-register shift control for a one-address serial computer with the operations considered in this section.

17-5. Control of Memory Address Selection and Instruction Sequencing

Instruction Register and Current-address Register. The control of memory selection and instruction sequencing takes place in the instruction register, current-address counter, and associated circuitry. Three basic functions are performed here: (1) determination is made of the addresses to be used during any phase; (2) the normal sequencing of instructions is controlled; and (3) the abnormal sequencing of instructions, the jump, is accomplished.

Consider the first of these functions. Memory addresses to and from which words are transmitted are found in only two locations: Addresses of operands or for results are found in the instruction register; addresses of instructions are found in the current-address counter. Whether an address is selected from the instruction register or from the current-address counter depends on the phase during which the selection is made. As an illustration, consider a one-address computer in which the next instruction is selected during phase 1, and an operand is obtained from the memory during the first minor cycle of phase 2 or returned to the memory during the first minor cycle of phase 3. We let αi be the ith bit of the α in the instruction register, and $\alpha' i$ be the ith bit of the current-address counter. Then, for $\alpha^* i$, the ith bit of the address upon which the selection is made, we have

$$[\alpha^* i] = [F1] \cdot [\alpha' i] + [F2] \cdot [T^1] \cdot [\alpha i] + [F3] \cdot [T^1] \cdot [\alpha i]$$

Now consider the second and third functions taken together. For a three-, two-, or one-address system the current-address register is an actual counter, so that, in the normal sequencing of instructions, the address of the next instruction is obtained by *adding* 1 *to the count.* For a jump an address recorded in the instruction register must be transferred to the current-address counter, to replace the previous contents of the counter.

As a specific illustration, let us consider a simplified version of a one-address instruction register, current-address register, and associated circuitry. In Fig. 17-9 the current-address register is a counter; when a jump is made, the signal J clears the counter and inserts the bits of the α address of the instruction register. N is the signal that increases the count by 1 for the normal sequencing of instructions. It can be shown (see Exercises a and b) that

$$[\alpha' 1] = [\alpha 1] \cdot J + [\overline{\alpha' 1}]_d \cdot N + [\alpha' 1]_d \cdot \bar{N} \cdot \bar{J}$$

The $\alpha^* i$ are formed as described above. For our simplified example the shift control for the instruction register is $[T_{1-18}] \cdot [F1]$, and the input control is simply $(\alpha) \cdot [F1]$.

Serial and Parallel Systems, and Relative Counters. The operation of the instruction register and current-address counter as described above is adequate for parallel operation provided that the counter is a parallel counter and that the instruction register is loaded in parallel.

The instruction register may be a recirculating delay line. In such a case the instruction decoder must end in flip-flops, which are set as the instruction enters the instruction register. The current-address counter need not be a counter at all, but merely a shift register, which may also recirculate. The recirculation must pass through an adder so that a unit can be added to the current address to determine the address of the next instruction in normal sequence and so that an address from the recirculating instruction can be inserted into the current address in case of a jump. The bits of α^*i would be gated with $F1$, $F2$, and $F3$ as above, except that now only a single set of gates would be necessary through which the αi and $\alpha' i$ pass.

The operation of a relative counter becomes clear at this point. For serial operation, as the instruction enters the instruction register it passes through a special single-address adder; if the appropriate control signals are present, this adder adds the contents of the relative counter to one or more addresses of the instruction. For parallel operation a parallel adder having the same length as the address is necessary. Occasionally the instruction format is so arranged that the relative count cannot be added to the instruction as it enters the instruction register. In such a case the addition of the relative count to the address would be accomplished as a separate function.

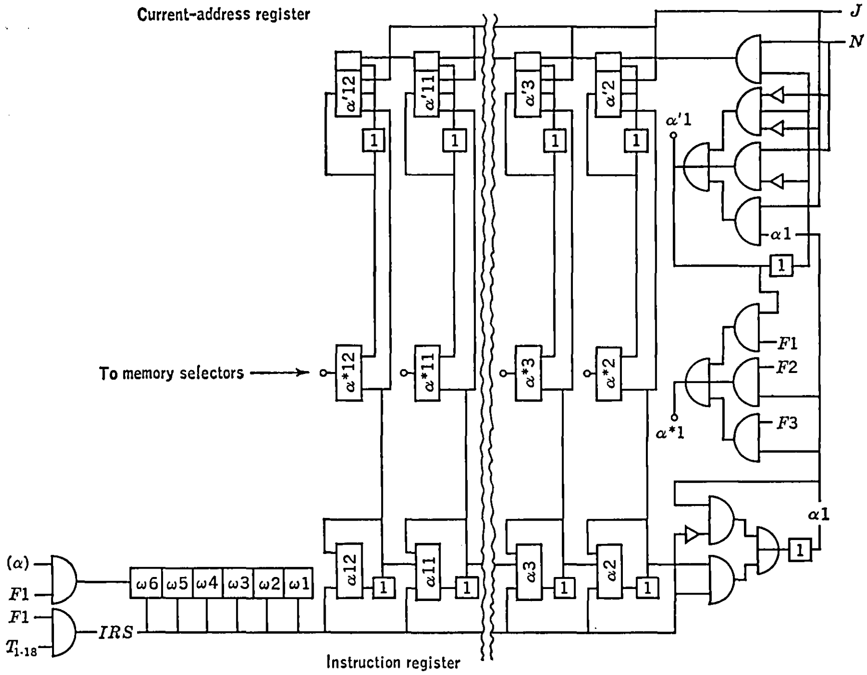


FIG. 17-9. The instruction register, current-address register, and associated circuitry.

EXERCISES

(*a*) Show that $[\alpha'1] = [\alpha 1] \cdot J + \overline{[\alpha'1]}_d \cdot N + [\alpha'1]_d \cdot \bar{N} \cdot \bar{J}$.

(*b*) Derive an expression for $\alpha' i$.

(*c*) Draw in detail two more stages of the current-address counter, instruction register, and α^*i generator, as indicated by the rectangles of Fig. 17-9.

(*d*) Draw a block diagram to show in detail all associated circuitry necessary for a recirculating delay-line serial instruction register and similar current-address counter. Describe its operation and the flow of words and addresses.

(*e*) Describe in detail the operation of a relative counter in a parallel computer. Draw a block diagram of the system.

17-6. Memory Selection

Obtaining a Word from the Memory. In the previous sections we have seen how the address to be selected, called α^*, is chosen either from the instruction register or from the current-address counter. The topic under consideration in this section is how this address is used to obtain a word from the memory or to replace a word into the memory. However, for a moment let us first consider the following question: Suppose that a word has been obtained from the memory; how does the computer know where to send it, to the arithmetic unit or to the instruction register? In essence we have already given the answer to this question in Secs. 17-4 and 17-5. In the arithmetic unit in our illustrations, (α) was always gated with $F2$. In other words, as (α) comes out of the memory it goes to both the arithmetic unit and the instruction register, but it is the phase that determines which location it enters.

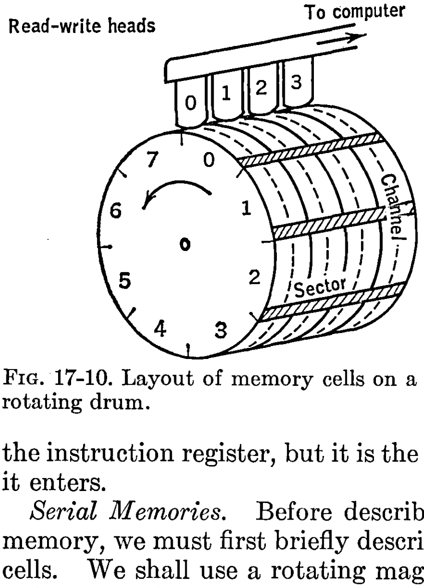
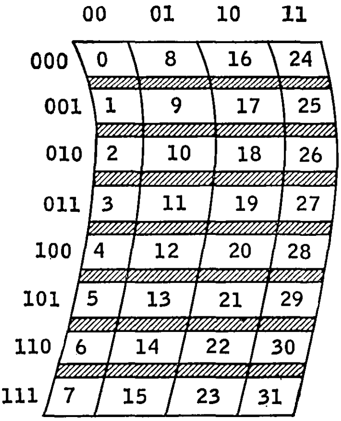


FIG. 17-10. Layout of memory cells on a rotating drum.

Serial Memories. Before describing how α^* selects a word from the memory, we must first briefly describe the physical layout of the memory cells. We shall use a rotating magnetic drum as an illustration, but all serial memories pose the same basic problems. As mentioned in Chap. 2, a rotating magnetic drum consists of a cylindrical surface with a coating of a magnetizable substance. We shall consider the electronic aspects of the drum in Part 5 of this book. The cylindrical surface of the drum is divided into bands called *channels*, and each channel is divided into lengths called *sectors.* The bits of a single word are recorded on this drum as a chain, or sequence, of small magnetized areas in one sector of a channel (see Fig. 17-10). Thus, by specifying its particular channel and sector, a word can be located, i.e., addressed. There is one *reading-and-writing head* for each channel which can sense the magnetic pulses, or bits, as they pass under the head (reading) or can place magnetic spots on the rotating-drum surface as it passes the head (writing).

As we shall describe in detail in a later section of this chapter, the computer is so synchronized with the drum that 1 bit passes underneath a head each unit time interval. The words (or sectors) are spaced so that precisely one minor cycle elapses between the passage of the first bit of one word under a head and the passage of the first bit of the next word. Ordinarily this means that there is some blank space between words (see crosshatched areas in Figs. 17-10 and 17-11). For example, the Pedagac has an 18-bit word plus a sign bit, but a 23-unit-time-interval minor cycle. Thus there will be four spaces after one word before the next word starts.

Consider the unrolled surface of a hypothetical drum, shown in Fig. 17-11. The columns represent the channels and the rows the sectors. The numbers appearing in each word location (each channel-sector location) are the addresses. Here there are four channels of eight sectors each. Any address can be formed by placing the binary column (channel) number adjacent to the binary row (sector) number, the most significant 2 bits of the address specifying the channel, the least significant 3 bits the sector. Thus address 5 becomes 00101, address 21, 10101, and so forth. This illustration is not realistic, for even the small drum used for the Pedagac has 32 channels with 128 sectors per channel, making a total of $2^5 \times 2^7 = 2^{12} = 4{,}096$ words. There are drums having as many as 100,000 words.



	00	01	10	11
000	0	8	16	24
001	1	9	17	25
010	2	10	18	26
011	3	11	19	27
100	4	12	20	28
101	5	13	21	29
110	6	14	22	30
111	7	15	23	31

FIG. 17-11. The organization of addresses on the drum memory.

Serial Memory Selection. Given an address α^*, the proper word is located on the drum as follows: The proper channel is selected by choosing the corresponding read-write head; the proper sector is selected by counting each sector as it passes under the head. (The former is a space, the latter a time, selection.) Let us consider the sector selector first. The count of a counter synchronized with the drum increases by 1 each time the first bit of a new word comes under the head; i.e., it increases by 1 at T_0 of every minor cycle. If we wished to read from or write into the fifth sector, we would compare 101 (of α^*) with the reading of this *sector counter.* When the count equals 101, the reading (or writing) would commence. The sector counter will be reset to zero when the start of the zero sector passes under the heads. In Fig. 17-12, α^*3, α^*2, α^*1 represent the sector-selector bits of α^* and $C3$, $C2$, $C1$ the count of the sector counter. The circuit shown will produce the *coincidence signal* COI only when $[\alpha^*1] = C1$, $[\alpha^*2] = C2$, and $[\alpha^*3] = C3$. The signal is needed only at T_0 and is therefore gated with T_0. As we shall see in the following section, this coincidence signal *chooses the sector by initiating the read or write process.*

Having shown how to choose the proper sector, we now turn our attention to how the proper channel is chosen. The method is simply to decode the channel-selecting bits of α^* (see Sec. 17-2 for a detailed discussion of decoding circuits). The output of each channel is gated with the corresponding output of the decoder, and only the signals to or from the selected channel will enter or leave the drum (see Fig. 17-13). The output (α) is gated with T^1_{0-18} since only during the first minor cycle of a phase is a word read out of the memory. This may occur only during phase 1 or 2 for a serial one-address computer. Writing into the memory occurs only during phase 3 during T_{0-18} of one minor cycle.†

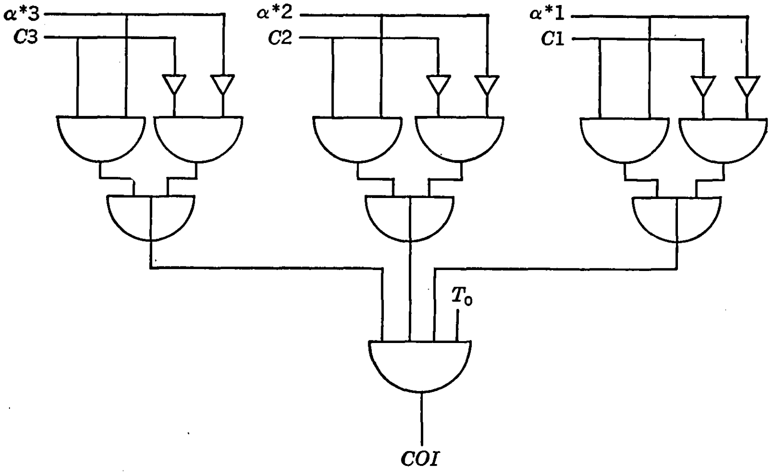


FIG. 17-12. Generation of the sector-selector coincidence signal.

The waiting time from when α^* is first produced to when the proper sector has been selected (i.e., to when the sector-counter bits agree with the α^* bits) is called the *memory access time.* For the drum we have described, the shortest access time is zero, i.e., when the first bit of the desired word is under the head at T_0 of the minor cycle in which α^* is formed. The longest access time occurs when the desired word has just been missed, since the drum must rotate a complete revolution minus one sector before the word comes round again. The average access time is often taken as half the longest access time.

Parallel Memories and Selection. In parallel memory selection all the bits of a word enter the instruction register or the accumulator at the same time. Similarly, in replacing a word in the memory, all the bits are replaced simultaneously. A drum memory can be used in a parallel fashion, when a word is recorded on the drum cylinder parallel to the axis instead of round the circumference. If only one word can be fitted along the length of the drum, then in this arrangement each of the drum heads reads (or writes) 1 bit of the word, and there must be as many heads as bits in the word format. This eliminates the need for channel selection. The need still remains for sector selection, but now there are

† Note that a write control signal must be fed to the write amplifiers to turn them on; absence of this signal will prevent writing onto the drum.

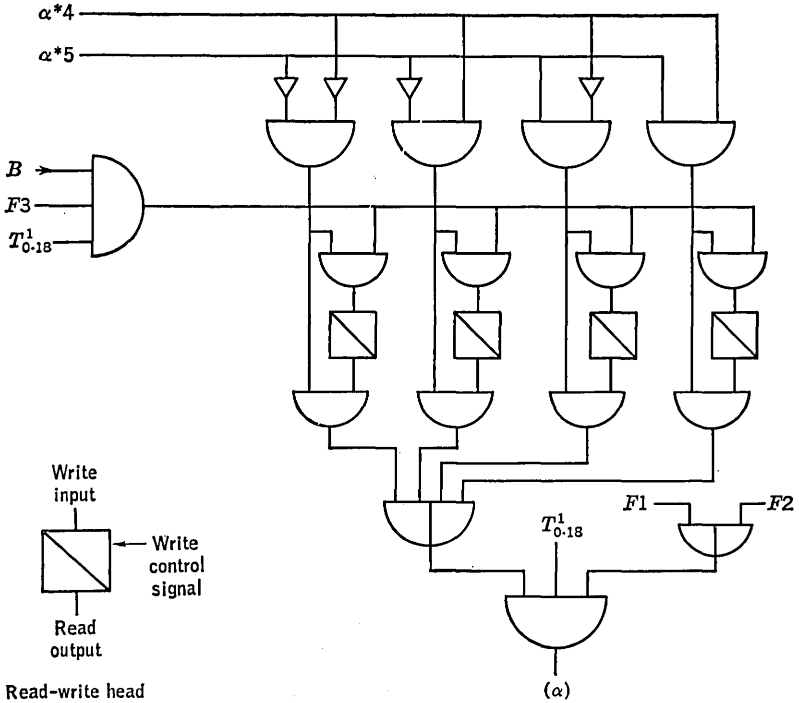


FIG. 17-13. The channel selector with associated read-in and -out circuitry.

as many sectors as words (see Fig. 17-14). Of course, if there are two or more words on the axial length of the drum, group selection is necessary.

The most common parallel memory in use is composed of magnetic cores; each core stores a single bit. Magnetic cores are made from ferrite, which has a relatively square hysteresis loop (see Fig. 17-15). They are usually formed in the shape of tiny rings which are magnetized in one direction to record, say, a unit, and in the other direction to record a zero. In Part 5 there are more detailed descriptions of the magnetic properties of cores and of the electrical-engineering aspects of their use. For our present purpose, namely, logical design, it suffices to observe that the direction of the magnetic flux in the core can be changed or detected by means of wires placed through the hole of the core.

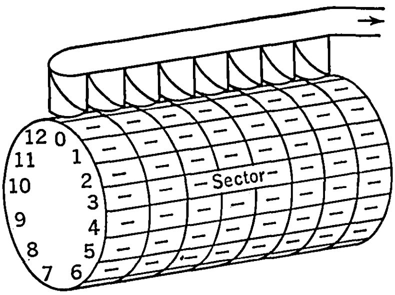


FIG. 17-14. Parallel access to a drum.

In Fig. 17-15 we have illustrated a core that requires a current I_c to be flipped (i.e., change of polarity); note that $\frac{1}{2}I_c$ or $\frac{2}{3}I_c$ *will not flip the core* if it is initially at zero. The simplest method of wiring a magnetic-core memory requires the use of three wires through each core (see Fig. 17-16). The cores are arranged as the points of a three-dimensional coordinate system; the X and Y coordinates of a core represent the address of the word of which that core is a member; the Z coordinate determines which bit of the word the core stores. Thus a single column of cores (i.e., all cores with the same X and Y coordinates) represents a single word. Through each core there is a wire representing its Z coordinate or bit number. The wiring is so arranged that all cores with the same X, Y, or Z coordinate have the same X, Y, or Z wire through them. The X-coordinate wires are the outputs of a decoding circuit (see Sec. 17-2) that decodes (i.e., has as inputs) half the bits of an address; the Y-coordinate wires are outputs of a decoding circuit for the other half. The Z wires are the parallel input to the registers of the computer.

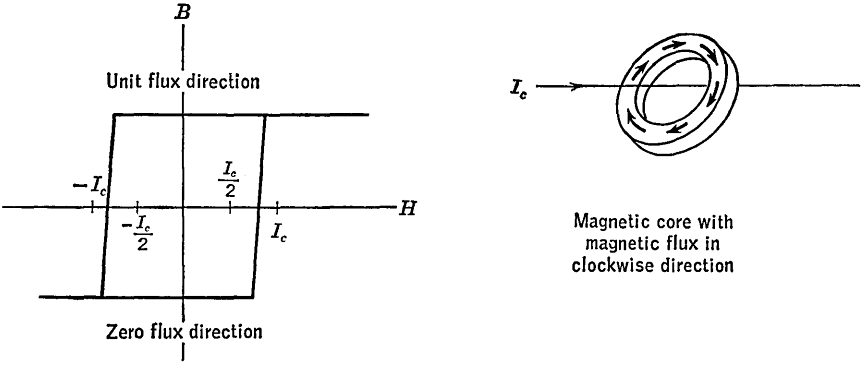


FIG. 17-15. Hysteresis loop and shape of a magnetic core.

To select a word with this system, a current of $-\frac{1}{2}I_c$ is passed through the X wire corresponding to the X half of the desired address; another current of $-\frac{1}{2}I_c$ is passed through the Y wire corresponding to the Y half of the address. *Only* cores in the proper X, Y column will then have the full current $-I_c$ passing through them; and of these cores only those which are recording units will flip to zero, while the zero cores will stay at zero. *Only* the Z wires through those cores which flip will have induced voltage to operate the registers, all the bits of the stored word being sensed at the same time (see Fig. 17-16). Since reading a word from the memory clears this word, the word must be immediately written back into the memory.

To write into the memory, the selected address must first be cleared as above, with $-\frac{1}{2}I_c$ in the corresponding X and Y wires. Then a current of $\frac{1}{3}I_c$ is passed through each of these wires, while $\frac{1}{3}I_c$ is also passed through only those Z wires which correspond to unit bits of the word being read in. The only cores flipped will be those receiving the current I_c, that is, those in the selected address corresponding to the unit bits in the word which was read in.

There are many variations to the wiring arrangements and the current values described above. For example, in writing into the memory all the X wires may carry $-\frac{1}{2}I_c$ except the selected X wire, which would not carry current. The selected Y wire would carry $\frac{1}{2}I_c$ and the other Y wires no current. A current of $\frac{1}{2}I_c$ through the Z wires corresponding to unit bits of the word being read in would cause a total current of I_c to pass through only the corresponding cores of the selected word (since outside the selected word the $-\frac{1}{2}I_c$ X currents will cancel the $+\frac{1}{2}I_c$ Y currents). Alternatively the bits of the address can be sep-

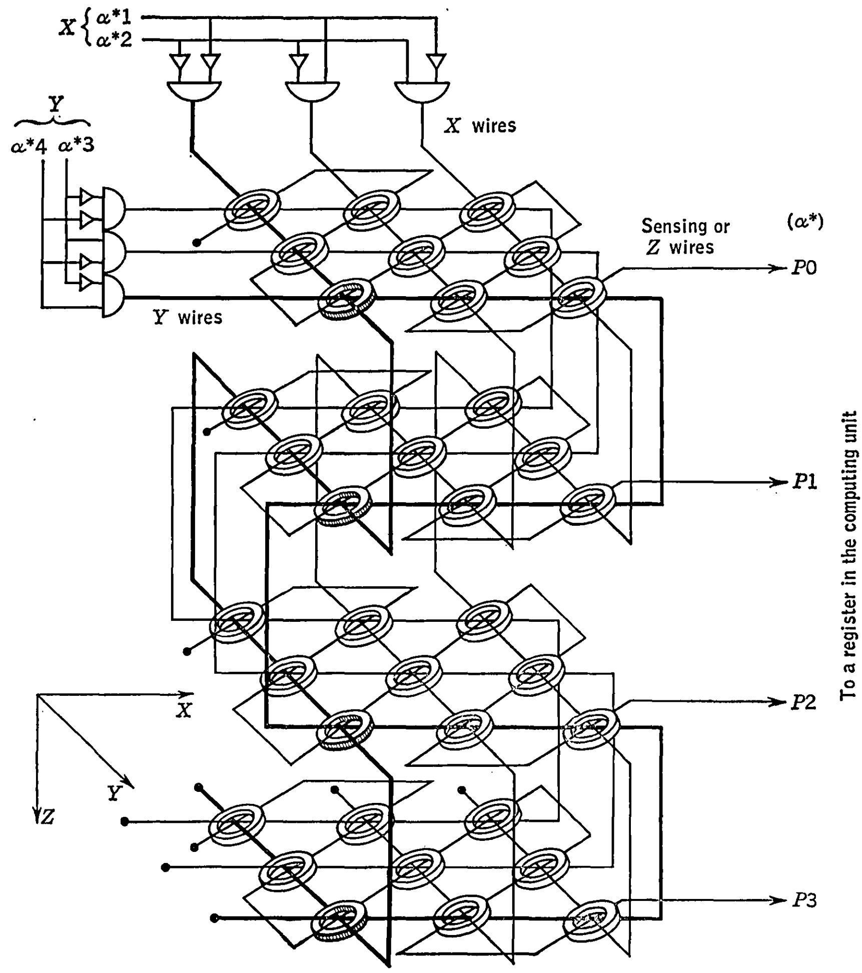


FIG. 17-16. Access and sensing wires of a core memory. The figure represents a memory with 4-bit addresses containing nine words each of 4 bits length. The dark wires indicate the selection of the word that is the contents of address $\overset{x}{10}\ \overset{y}{00}$.

arated into three sets instead of two. We then would have a four-dimensional array, with W, X, and Y wires selecting the word and Z the bits of the word and three decoding circuits instead of two.

Note that with a magnetic-core memory there is no time selection; only decoders are used. (Hence there are no idling phases in selecting a word—see next section.)

EXERCISES

(*a*) Diagram the control gates through which (α) must pass when being transmitted from the memory (1) to the instruction register and (2) to the arithmetic unit and (3) when being transmitted from the arithmetic unit to the memory; consider serial four-, three-, two-, and one-address computers. (HINT: During what phases are these gates open?)

(*b*) What would be the diameter of a drum that in a single channel stores 128 words of 23 bits each, if the bits are stored 100 to the inch?

(*c*) How long (in microseconds) would a unit time interval be if the drum described in Exercise *b* rotated at 1,800 rpm?

(*d*) What would be the average access time of the drum of Exercise *c*?

(*e*) In Fig. 17-12 a *parallel* comparator is illustrated for use in selecting the proper sector; i.e., all the bits of C and α^* are compared simultaneously. How would a *serial* comparator be constructed, to compare 1 bit of C and α^* at a time? (HINT: At what T_i would the sector counter have to be increased in order that the serial comparison be completed by the time T_0?)

(*f*) Compare the selection problems involved in serial-drum, parallel-drum, and magnetic-core memories with respect to time and space.

(*g*) How can the currents be arranged in a three-dimensional core memory so that the currents through the cores of nonselected words are only $\frac{1}{3}I_c$? (HINT· Try different directions of currents for selected and nonselected X and Y wires.)

(*h*) Repeat Exercise *g* for a four-dimensional core memory.

17-7. Control of Instruction Execution

Sequencing of Phases. Up to now we have studied the controls that act once we are in a particular phase. The subject of the present section is how we control the phases themselves; i.e., what are the controls that sequence the phases during the execution of an operation? The answer to this problem completes the control picture.

There are three aspects to the control of a phase signal: (1) it must be turned on; (2) it must be kept on for the duration of the phase; and (3) it must be turned off. Consider phase 1 for a four-address system (see Table 17-1). During this phase the address α is selected and read into the accumulator, in one minor cycle. In order to construct the phase 1 signal, we observe that phase 4, which precedes it, is only one minor cycle in length. Thus we have for phase 1

$$[F1] = [F4]_d \cdot [T_0] + [F1]_d \cdot [\bar{T}_0]$$

Here by delaying $F4$ by one unit time interval we initiate $F1$ at T_0 just as $F4$ goes off; the right-hand product keeps $F1$ on until the next T_0, which is the start of phase 2. Note that if $F4$ had been longer than one minor cycle this formula for $F1$ would not work but would make $F1$ go on during the second minor cycle of phase 4.

Remember that phase (2 and 3†), the operation computing phase, can last for more than one minor cycle. For these phases a special pulse called the operation ending pulse OEP is needed; it comes on only during the last unit time interval of the operation computing phases. Now consider $F1$ for a one-address system (see Table 17-1), which may follow either phase 2 or phase 3, depending on the previous instruction. Hence it must be initiated by OEP,

$$[F1] = [\mathrm{OEP}]_d + [F1]_d \cdot [\bar{T}_0]$$

In the case of phases, such as $F2$, that may last for more than one minor cycle, signal OEP must be used to end them,

$$[F2] = [F1]_d \cdot [T_0] + [F2]_d \cdot [\overline{\mathrm{OEP}}]_d$$

Signal OEP is delayed so that $F2$ remains on throughout its last minor cycle.

Thus the sequencing of phases rests with the construction of the signal OEP, produced in the operations signal generator. We shall illustrate the construction of this signal for the simplified one-address example that we were following in Secs. 17-3 and 17-4. Signal OEP is to come on at T_{22} of the last minor cycle of the operation being performed in phase 2 or 3. For addition and subtraction this is $T_{22}{}^1$; for multiplication and division this is $T_{22}{}^{19}$; for shift this is $T_{22}{}^2$. In our example we assumed that in phase 2 the only operations performed are the arithmetic; hence $\overline{\mathrm{PQS}}$ can be used for the addition and subtraction signal:

$$[\mathrm{OEP}] = [F2] \cdot [\overline{\mathrm{PQS}}] \cdot [T_{22}{}^1] + [F2] \cdot [\mathrm{PQS}] \cdot [T_{22}{}^{19}] + [F3] \cdot [\mathrm{SHS}] \cdot [T_{22}{}^2]$$

where PQS is on during multiplication and division and SHS is on during shift.

The Selection or Idling Phases. As we have seen in Sec. 17-6, when using a drum memory the computer must wait for the drum to rotate to the proper sector; this waiting period must be some integral number of minor cycles. It is convenient to denote such a period as an *idling* phase, because the primary components of the computer merely idle until the proper sector is selected. The coincidence signal COI comes on at T_0 of the minor cycle that coincidence is made; hence this signal can be used to end an idling phase and to initiate the next functional phase in sequence. Such an idling phase will precede any phase that requires drum memory access. For example, suppose that an idling phase preceded phase 2; call this phase IF2 (idling phase 2):

$$[\mathrm{IF2}] = [F1]_d \cdot [T_0] + [\mathrm{IF2}]_d \cdot [\overline{\mathrm{COI}}]$$

† See footnote on page 546.

Then we would have for $F2$ itself

$$[F2] = [\mathrm{IF2}]_d \cdot [\mathrm{COI}] + [F2]_d \cdot [\overline{\mathrm{OEP}}]_d$$

For a one-address system where an idling phase IF1 would precede phase 1, we have

$$[\mathrm{IF1}] = [\mathrm{OEP}]_d + [\mathrm{IF1}]_d \cdot [\overline{\mathrm{COI}}]$$

and also

$$[F1] = [\mathrm{IF1}]_d \cdot [\mathrm{COI}] + [F1]_d \cdot [\bar{T}_0]$$

By initiating the change of phase from an idling phase, COI selects the proper sector. For as soon as the computer changes from the idling phase, COI opens the gates that read B into the memory or (α) out of the memory, and the reading or writing proceeds.

The Input-Output Phases and Synchronization. Transmitting words between the outside world and the computer's memory by means of the input-output equipment presents a special problem, because the input-output external equipment generally operates at much slower speeds than does the computer. This problem is often solved by the use of an input-output buffer, which is a shift register designed in such a way that it may be controlled at times by the computer and at other times by the external equipment. For a computer with a drum memory three phases each for input and output can be distinguished as follows: (1) the IF phase during which an address is selected in the computer memory; (2) a phase FO during which a word is transmitted (in either direction) between the memory and the in-out buffer; and (3) a phase FB during which a word is transmitted between the in-out buffer and an external input or output unit.

Let us consider as an example the input and output instructions of the Pedagac. For the read-out instruction, phase IF Out (signal IFO) is initiated after $F1$ and lasts until COI comes on at coincidence. Then phase FO Out (signal FOO) is initiated. During this phase the buffer shift register is loaded by the computer proper, at its rapid speed, while the buffer is being shifted under the control of the computer; this phase lasts one minor cycle. The end of phase FO Out initiates FB Out. During this latter phase the word is read out of the buffer to the output unit, the buffer being shifted by the output equipment, at the slower speed. When the word has been completely read out, the output equipment notifies the computer; phase FB Out has ended, and the execution of the instruction "Read out one word from address α" has been completed. For phase signals IFO and FOO we have

$$[\mathrm{IFO}] = [F1] \cdot [21D] \cdot [T_0] + [\mathrm{IFO}]_d \cdot [\overline{\mathrm{COI}}]$$
$$[\mathrm{FOO}] = [\mathrm{IFO}]_d \cdot [\mathrm{COI}] + [\mathrm{FOO}]_d \cdot [\bar{T}_0]$$

where $21D$ is the instruction decode signal for read-out. For the generation of signal FBO let the signal EWO be initiated by the external output unit when the word has been read out of the buffer. Signal EWO need not come on at T_0, and so we must end phase FB Out only at [EWO] · $[T_0]$. Thus

$$\begin{aligned}[\text{FBO}] &= [\text{FOO}]_d \cdot [T_0] + [\text{FBO}]_d \cdot \overline{[\text{EWO}] \cdot [T_0]} \\ &= [\text{FOO}]_d \cdot [T_0] + [\text{FBO}]_d \cdot [\overline{\text{EWO}}] + [\text{FBO}]_d \cdot [\bar{T}_0]\end{aligned}$$

The problem remains of how to generate the signal EWO. One procedure is to have the output unit count each bit that it receives, presenting a signal to the computer when a complete word has been received. Let us assume that it is turned into a signal EWO′. This signal, which in general lasts several unit time intervals but less than a minor cycle, initiates EWO. Then EWO should stay on until the next T_0, at which time it stops FBO and initiates IF1, the next phase in sequence:

$$[\text{EWO}] = [\text{EWO}'] \cdot [\bar{T}_0] \cdot [\overline{\text{IF1}}]_d + [\text{EWO}]_d \cdot [\overline{\text{IF1}}]_d$$

The IF1 signal is used to inhibit EWO′ if it stays on past T_0; it is used to stop EWO. (See Exercise *c* below concerning the $\bar{T}_0$ in the left-hand product.)

This process for forming and using EWO is generally applicable to external signals from control-panel push buttons or from external in-out units. The original external signal is converted into one or more computer signals by electronic circuits called *synchronizers*. The synchronizer outputs initiate other signals to be used at specific times to initiate phases, etc. The signals so initiated can be used to turn off their initiating signal.

EXERCISES

(*a*) Suppose that, in the one-address drum computer that we have been using for illustration, phase 3 also included the *transfer* instruction as well as the *shift*. Construct the signals for IF1, $F1$, IF2, $F2$, IF3, and $F3$.

(*b*) Write expressions that for a one-address drum computer generate the input phase signals IF1, FOI, FBI, using the signal OOD, the instruction decode signal for "Read one word into address α," and the signal EWI′ produced by the input unit synchronizer when a complete word has been transferred into the buffer.

(*c*) Why must T_0 inhibit the initiation of EWO? (HINT: Suppose that it did not inhibit EWO, but that EWO were initiated at T_0; would FBO be turned off? What about IF1?)

17-8. Timing and Counters

Review. In Sec. 15-4 we introduced the ideas of minor cycles and unit time intervals, based on the clock pulses described in Sec. 2-4. In this section we shall describe the generation of clock pulses. Let us review again the over-all synchronous plan. The basic timing is kept by means of clock pulses which measure off the unit time intervals. A unit-time-interval counter counts these clock pulses, the count at any time being denoted by T_i. A certain number of these clock pulses constitute a minor cycle, and each time this number has been counted off the unit-time-interval counter is reset to zero. Another counter counts minor cycles, the count at any time being denoted by T^i. A signal T^i is on during every unit time interval of the ith minor cycle. A certain number of minor cycles constitutes a phase, the number depending on the particular phase and operation. A phase signal Fj is on during every unit time interval of every minor cycle of phase j. At the end of the phase the minor-cycle counter is reset to its initial count.

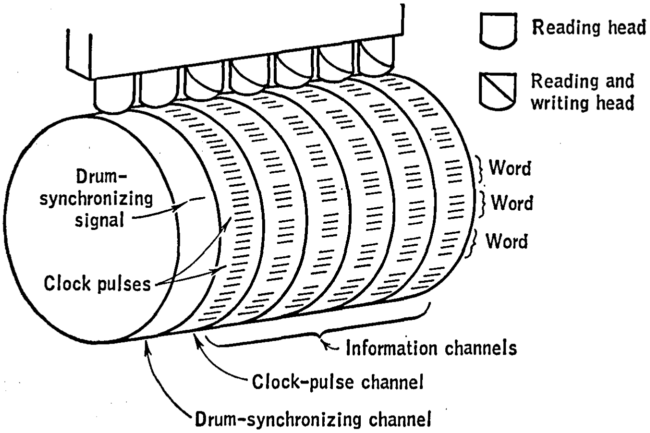


FIG. 17-17. Diagram of a drum showing drum synchronizing signal.

Serial and Parallel Clocking. Basic to all the counters and timekeeping signals are the clock pulses, denoted by *cp*. For a serial or parallel access magnetic-drum memory these clock pulses may arise from evenly spaced unit bits prerecorded on a special channel of the magnetic drum, called the *clock-pulse* drum channel (see Fig. 17-17). Then, while the drum is rotating, these bits are read from the drum by a separate head to form the clock pulses. In this way the synchronization of the entire computer is based directly on the rotating-drum speed, a far easier accomplishment than adjusting the speed of the drum to some outside clock frequency.

Consider next the situation that arises when the computer is turned off for the night with information from some previous computations recorded on the drum. When the power is cut, the drum stops rotating but the contents of the memory remains. The next morning, when the power is turned on and the drum starts rotating again, it is often desired to use the information that remained on the drum during the night.

However, to read a word out of the drum, T_0 must be synchronized with $P0$ (that is, the first bit of the word). The problem is: In starting the drum rotating how is synchronization reattained?

One common solution to this problem is to have on the drum a separate channel, called the drum-synchronizing channel, containing one single pulse (see Fig. 17-17). After the drum is started, the *very first time* this bit is recognized the unit-time-interval counter is started. Hence, as the unit-time-interval counter *starts*, the position of the reading-writing head is known to be at $P0$ of the zeroth word on each channel; and this will be the *same position* each time the drum is started. Also at this very first T_0 time the sector counter starts. In this way the address locations on the drum remain invariant for the life of the drum.

For a computer with a parallel magnetic-core memory the clock is simply an oscillator with a fixed frequency. Since memory selection does not depend on time, the difficulties encountered with the drum do not occur.

The Unit-time-interval Counter. The unit-time-interval counter counts the time intervals during a minor cycle. As an illustration, consider the Pedagac, with time intervals T_0 through T_{22}. The Pedagac counter does not start to count until the drum-synchronizing pulse appears for the first time when the drum is started rotating, and then with each clock pulse the count is increased by 1 until it reaches 22. Then after the T_{22} signal the count must return to T_0 again; i.e., the count should be cleared after T_{22}.

The Sector Counter. The sector counter counts the words within each channel. The Pedagac is designed to have $128 = 2^7$ words per channel. The sector counter therefore has seven stages, whose outputs are compared with the last 7 bits of the α address in generating the coincidence signal. Starting at zero when the drum-synchronizing pulse first appears, the count is increased by 1 with each succeeding T_0. Since the total number of sectors is an integral power of 2, the counter overflows, returning to zero, each time the drum-synchronizing pulse returns, and hence need not be specially reset.

The Minor-cycle Counter. As we have seen above, the minor-cycle counter presents signals for the operations signal generator, the arithmetic unit, and the phase generator. The minor-cycle count at which the counter is reset depends on the phase and operation being performed. The minor-cycle counter of the Pedagac starts with a count of 1 and can go up either to the count of 2 during the execution of the shift instruction or to the count of 19 during the execution of the multiplication and division instructions. Hence the main problem is to reset the counter at the proper time. This is done in phases 2 and 3 by using the OEP signal to reset the counter to 1. Since during other phases (including the idling phases) we need never count past 1, the count is forced to 1 at T_0 of every minor cycle of these phases as well as at T_0 of the first minor cycle of phases 2 and 3.

EXERCISES

Consider the Pedagac with 128 word channels, a 23-unit-time-interval minor cycle (T_{0-22}), a word containing 18 bits plus a sign bit, and instructions as described. Suppose that the following three-instruction program is being computed:

Address	Operation	α	Remarks
0010	53	0000	Add
0011	42	3725	Multiply
0012	70	0016	Shift

(*a*) What is the count on the sector counter when the *add* instruction has been completed?

(*b*) What is the count on the sector counter when the *multiply* instruction has been completed?

(*c*) How many minor cycles have passed from the time that coincidence was made on the sector selector with address 0010 until the *shift* instruction has been completed?

CHAPTER 18

PACKAGING AND THE LOGICAL DESIGN OF THE PEDAGAC

18-1. Introduction

Thread of Continuity. In the preface we mentioned that to maintain the thread of continuity throughout this book a complete digital computer would be designed. To this end we have considered in Chap. 9 the systems design of the Pedagac. We consider now the logical design of the machine.

In the preceding three chapters we have described the operation of various computer components, such as the adder, the sign generator, counters, multiplication and division controls, and the instruction decoder. We shall now consider an integrated example of how all these components are organized and assembled to form an operating system. New principles will arise that are associated with the complete organization of such a system, and it is felt that these can best be displayed by means of specific examples.

Point of View. The ability to design a complete computer system results to a large extent from experience. Probably the only way the student can gain any such experience is by following through a detailed example. Such is the point of view of this chapter. However, we must add that although we shall spend much time on the particular details of our illustration, this by no means implies that logical designs are limited to the methods and techniques here used. We intend that through his detailed study *the reader shall grasp the principles involved and the pitfalls encountered in a complete systems design.* The Pedagac has been designed to emphasize such principles. In presenting the logical design of the Pedagac we shall utilize methods which may aptly be termed *vertical* and *horizontal* description. The vertical descriptions relate to the specific function and purpose of each gate during the different operations; the horizontal descriptions follow the path of a bit as it is propagated through the computer during the execution of the various instructions. However, merely reading these descriptions will not be sufficient to give the student the sought-after experience. Therefore included in this chapter are an extraordinarily large number of exercises. These exercises differ from those of other chapters in that they are not intended to give the student practice in the methods considered; rather *their purpose is to guide the student* in his study of the functioning of the

Pedagac. These exercises will illustrate the intricate details involved in complete logical-systems design. They are essential to the teaching method of this chapter.

Outline of the Chapter. Computers are usually constructed by connecting together *preassembled combinations* of *and*, *or*, and *not* gates. Such combinations are called packages. Thus, preliminary to the discussions of the Pedagac we consider packaging, in Sec. 18-2. The remainder of the chapter is then concerned with the details of the Pedagac. First (in Sec. 18-3) the phases and block diagram of the Pedagac are covered. *This section is fundamental, since all that follows is based directly on its material.*

In Sec. 18-4 is considered the operations signal generator, whose output signals determine the mode of operation of the arithmetic unit, wherein most operations are performed. In Sec. 18-5 the vertical description of the arithmetic unit is given, and Sec. 18-6 integrates these two preceding sections in the horizontal descriptions of the arithmetic unit.

Two methods are commonly employed for recording the detailed logical design of a digital-computer or -control system. One method is by use of logical circuit diagrams, the other by use of the corresponding Boolean equations. Since, as we have seen previously (Secs. 10-11 and 10-12), the equations and their corresponding logical diagrams are equivalent, either method is completely satisfactory. It is felt, however, that the logical diagrams are probably easier for the student to grasp, and hence for the Pedagac these are given. Since for many purposes the Boolean equations may be more convenient, we have also included these in the text (see Appendix), using a symbolism consistent with the corresponding logical diagrams.

18-2. Packaging

The Package. The concept of a *package* must be discussed before considering the detailed logical design of the Pedagac. It has been found from experience that it is neither convenient nor economical to build a computer directly with the three basic building blocks, *and* gates, *or* gates, and *not* gates. Instead combinations of these building blocks are used as elementary modules. For example, a convenient package may consist of three *and* gates all feeding into an *or* gate (see Fig. 18-1). All these components are put in a single-circuit module, which in our example has nine inputs. Most often two outputs are used, one output being simply the negation of the other, as shown in Fig. 18-1. These packages are constructed in mass-production style, and the computer is made by wiring the packages together according to the logical design. Part 5 of this text will consider the electronic design of such package circuits. However, it is appropriate to point out here that, in addition to performing the logical gating of signals, the package also has the function of reshaping, amplifying, and resynchronizing the pulses.

The type of package shown in Fig. 18-1, called the *and-or* package, has the advantage that it produces a *sum-of-products* form, that is,

$$D = A_1 \cdot A_2 \cdot A_3 + B_1 \cdot B_2 \cdot B_3 + C_1 \cdot C_2 \cdot C_3$$

and also $\bar{D}$. Another type of package, particularly suited for parallel adders, is the *or-and-or* package, as shown in Fig. 18-2. Here the output

$$E = (A_1 + A_2) \cdot (B_1 + B_2) + (C_1 + C_2) \cdot (D_1 + D_2)$$

There are many variations in the kinds of packages. An important class of packages involves the use of a transistor. These packages in general only have a single output instead of the usual inverse outputs.

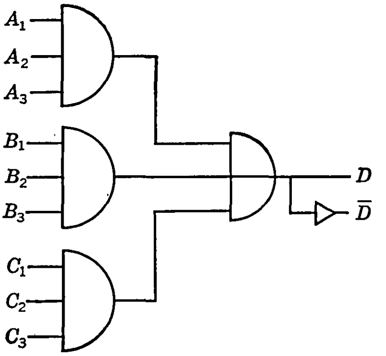


FIG. 18-1. Example of an *and-or* package with nine inputs and two outputs.

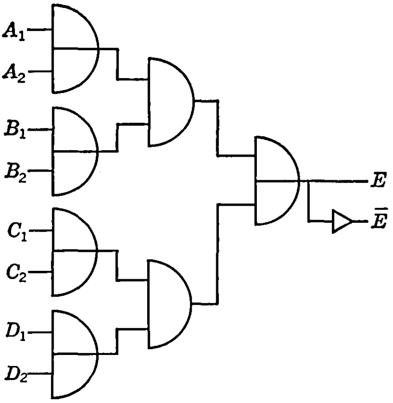


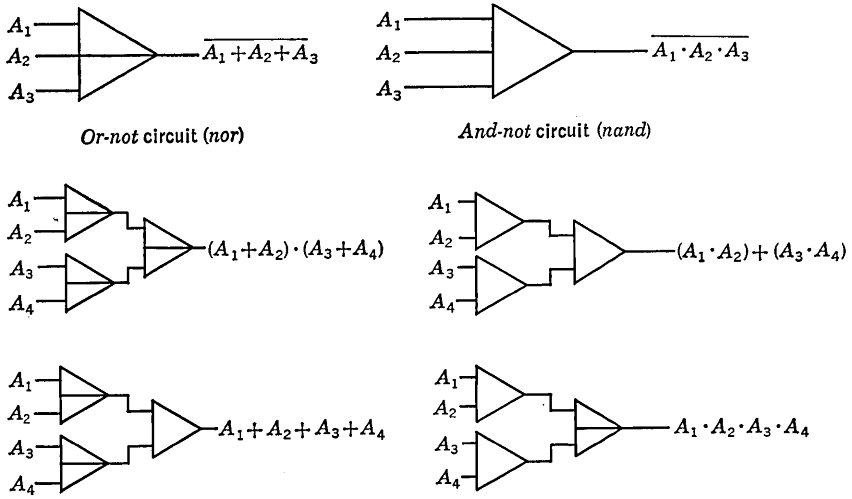
FIG. 18-2. The *or-and-or* package.

There are two basic types. In the first, called the *or-not* circuit (sometimes also referred to as a *nor* circuit), if the inputs are A_1, A_2, A_3, the output is $\overline{A_1 + A_2 + A_3}$, or equivalently $\bar{A}_1 \cdot \bar{A}_2 \cdot \bar{A}_3$ (see Fig. 18-3). For the second type, called the *and-not* circuit (sometimes also referred to as a *nand* circuit), if the inputs are A_1, A_2, A_3, the output is $\overline{A_1 \cdot A_2 \cdot A_3}$, or equivalently $\bar{A}_1 + \bar{A}_2 + \bar{A}_3$. Of course these definitions are easily generalized to any number of inputs (see symbolism of Fig. 18-3).

Besides being advantageous from an electronics point of view, the advantage from a logical point of view is seen when combinations of these packages are considered. The sum of products, product of sums, straight *and*, and straight *or* can all be obtained.

Packaging Limitations. As was remarked in the discussion of the parallel adder, certain electronic limitations can affect the logical design. Using the *and-or* package for illustration, there are (1) the limitation on the number of *and* gates in a single package, (2) the limitation on the number of inputs allowed to a single *and* gate, and (3) the limitation on the number of packages to which the output can be sent, i.e., limitation on the "drive" of the output.

There is in addition a fourth electronic constraint, the delay time through the package. For the most part *we have assumed, and will continue to assume throughout this chapter, that the gating of a signal occurs instantaneously.* The effect of the actual delay will be considered at the end of Part 5, when its electronic nature and causes can be more thoroughly understood. The alteration of the computer design to take account of these delays will then be straightforward.



Or-not circuit (*nor*)

And-not circuit (*nand*)

FIG. 18-3. The *or-not* and *and-not* circuits and their resulting combinations.

The Clock Pulse and Packaging. The concept of the *clock-pulse cp*, used for synchronization, was introduced in Sec. 2-5, where it was noted that it enters every *and* gate. Let us use the *and-or* package to illustrate how this might be accomplished. One input to each *and* gate of every package is reserved solely for the clock pulse. Since the clock pulse is on in every unit time interval and is included as an input to every *and* gate, *it is never indicated but always understood to be present.* For example, the package illustrated in Fig. 18-1 really is as in Fig. 18-4. The clock pulse is logically nonfunctional; the reason for its use will be given in Part 5.

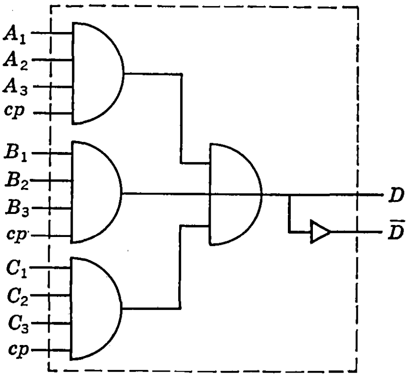


FIG. 18-4. *And-or* package showing the clock-pulse input to every *and* gate.

The Packages of the Pedagac. For the logical design of the Pedagac we shall choose the *and-or* package class. Five different types of packages will be used, having one *and* gate, two *and* gates, three *and* gates, four *and* gates, and five *and* gates, called types A, B, C, D, and E respectively. We assume that each *and* gate is limited to no more than *five* inputs (other than the clock pulse). There will be two outputs, normal and inverted. Rather than draw the entire circuit, we symbolize a package with only the *and* gates indicated, *it being understood that there is an* or *gate before the output* (see Fig. 18-5). The inverted output is not shown, however; when the inverted output is to be used, it is indicated where it is used in the usual way. This convention reduces the number of lines on the drawing. For example, a counter stage drawn with the package convention appears in Fig. 18-6. Here one package of type A and one of type B are used (cf. Fig. 15-6).

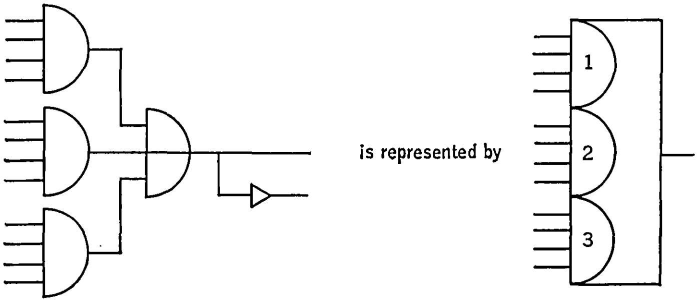


FIG. 18-5. Package representation.

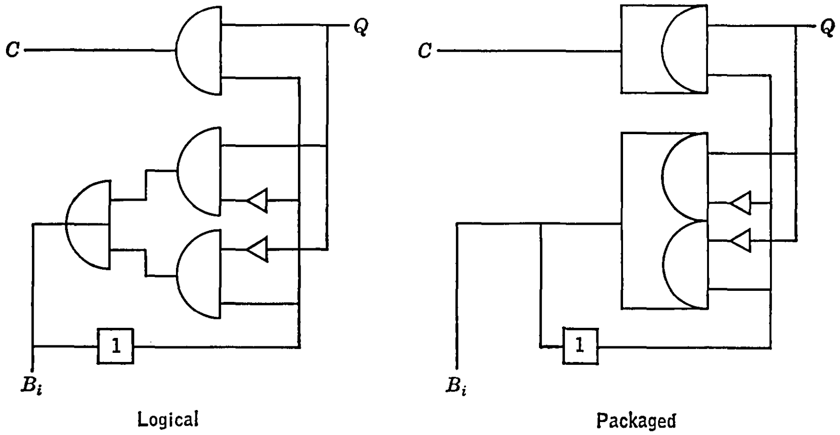


FIG. 18-6. Example of the packaging of a counter stage.

EXERCISES

(*a*) Package the circuits shown in Fig. 17-8, using *and-or* packages.

(*b*) Package a counter stage, using *and-not* and *or-not* packages. (HINT: Compare Figs. 15-6 and 18-6.)

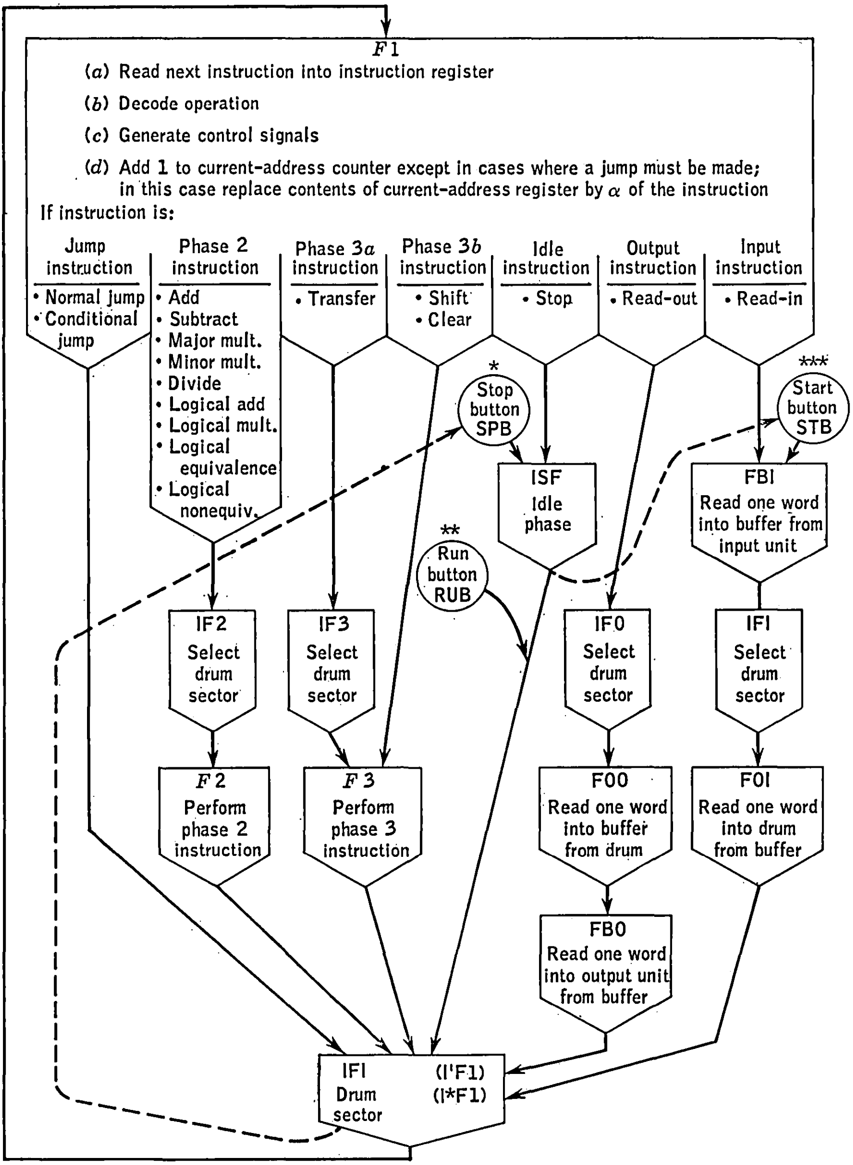
(*c*) Package a counter stage, using only *or-not* packages.
(*d*) Package the 5-bit parallel adder shown in Fig. 16-1, using *or-and-or* packages.
(*e*) Package the 5-bit parallel adder of Fig. 16-1, using *and-or* packages.
(*f*) Package the 5-bit parallel adder of Fig. 16-1, using *and-not* and *or-not* packages.
(*g*) Package the 5-bit parallel adder of Fig. 16-1, using only *or-not* packages.

18-3. Phases and Block Diagram of the Pedagac

Outline of Phase Sequencing. A thorough understanding of Fig. 18-7 is basic to an understanding of the logical design of the Pedagac. The general plan is as follows: During phase 1 the instruction to be executed is read into the instruction register, and the decoding of the operation is performed. In this phase also, after the operation has been decoded, the current-address register is set to the address of the next instruction, either by advancing it 1 or in the case of a *jump* instruction by inserting in it the α address from the instruction register. The arithmetic, logical, and input and output operations involve the address α in the memory, so that for these operations phase 1 will be followed by an idling phase, which ends when address α has been located on the drum. (For a *stop* instruction the idle stop phase ISF is initiated. This phase is analogous to the other idle phases in that the computer does not perform any functions except marking time; this phase will continue until the run or start buttons are pushed.) When the coincidence signal, which indicates that address α has been found, ends the idling phase, the operation called for by the instruction is performed, during phase 2 or phase 3. In phase 2 computation commences simultaneously with the reading of the second argument from address α into the adder-subtractor. Performing a typical arithmetic operation such as *add* consumes two phases: the memory selector phase IF2 and the computation phase $F2$. The input or output instructions, however, require three phases, for, besides the idling phase (IFO or IFI) and the phase required to transfer the word between the drum and the buffer (FOO or FOI), a third phase (FBI or FBO) is needed for transferring the word between the buffer and the input or output equipment. After an instruction has been executed, the next instruction must be selected. The proper drum address is selected during IF1 (for the current-address counter already contains the address of this next instruction); the instruction is read into the instruction register during $F1$, and the process continues as before.

PHASE 1. At the start of phase 1, (*a*) the instruction to be executed is read into the instruction register from the address on the drum held in the current-address counter (see Fig. 18-14). Thereafter during this phase the following functions are performed: (*b*) the operation is decoded in the operations decoder (see Fig. 18-14); (*c*) the control signals are generated by the operations signal generator (see Fig. 18-10); (*d*) the current-address counter is *set to the address of the next instruction* (see Fig. 18-14); and (*e*) the signal for the phase that should follow is produced in the phase generator (see Fig. 18-17).

The decoding and operations-signal generating functions take place



* *Stop button* forces computer to replace next phase with ISF

** *Run button* puts computer into phase IF1

*** *Start button* reads one word from external input into address 0000 and then takes (0000) as next instruction. (Only works when computer is in idling phase ISF.)

FIG. 18-7. Flow chart showing the sequencing of phases during the execution of aı instruction.

immediately after the instruction is read in. If the instruction is not a *jump*, the current-address counter is increased by 1. If a *jump* is to be made, then the contents of the current-address counter is replaced by the α part of the instruction register. In the Pedagac the functions performed during phase 1 after the instruction register is loaded are each essentially performed as parallel operations and therefore consume only a few unit time intervals. During T_{1-18} of phase 1 the instruction register is loaded; then during T_{19-22} the other functions are performed. Updating the current-address counter is left until T_{22} to ensure that a possible *jump* instruction will have been decoded.

When the instruction is a *jump*, normal or conditional, the entire operation (i.e., setting the current-address counter) is completed in phase 1 and $F1$ is followed directly by the idling phase IF1 for the next instruction. If the instruction is one of the arithmetic or logical operations, which involve another argument in addition to the contents of the accumulator, phase IF2 will follow $F1$. Of the phase 3 operations the computer need select an address only for the *transfer* instruction and in that case IF3 follows $F1$. For the *shift* or *clear* instructions phase 3 follows directly after phase 1. For a *stop* instruction the idling phase ISF follows phase 1. For a *read-out* instruction a memory address must be selected, and so phase IFO follows phase 1; for *read-in* there follows phase FBI, in which a word is read into the buffer from the input unit.

PHASES IF1, IF2, IF3, IFO, AND IFI. The appropriate drum sector is selected during these phases by the drum sector selector (see Fig. 18-15). The result of this selection process is the coincidence signal COI, which indicates that the drum has rotated to the proper word. Since it takes one minor cycle for each word in a channel to pass under the read-write head, the duration of these phases will be some integral number of minor cycles. For the Pedagac there are $2^7 = 128$ words in a channel, and the duration of such a phase can be as short as 1† or as long as 128 minor cycles. Then during the following phase ($F1$, $F2$, $F3$, FOO, and FOI, respectively) a channel is selected by the drum channel selector (see Fig. 18-15), and the word is transferred to or from the drum.

PHASES $F2$, $F3$, FOO, AND FOI. As we have seen, phase $F2$ can be from 1 to 19 minor cycles in duration, depending on the operation. During the first of these minor cycles the selected word is read into the adder-subtractor (see Figs. 18-11 to 18-13), and the operation is initiated. Addition, subtraction, and the four logical operations are completed in this first minor cycle; the others take 19 minor cycles each. At T_{22} of the final minor cycle a signal OEP appears, which is generated in the operations signal generator (see Fig. 18-10) to initiate the following phase, IF1 (see the phase generator, Fig. 18-17). For the *transfer* instruction the coincidence signal ends IF3, opening the writing gates to the proper channel; phase 3 then lasts one minor cycle. For the *clear* and *shift* instructions no memory reference is made, and $F3$ follows $F1$ immediately.

† The (delayed) idling-phase signals are used to initiate the following operating phases, and hence each must last at least one minor cycle.

The *clear* instruction requires only one minor cycle; the *shift* requires two, the first to ensure that the contents of the accumulator will be complemented, if necessary, and the second for the shifting itself. During phases FOO and FOI one word is transferred between the drum and the buffer, with the buffer under the control of the computer.

PHASES ISF, FBI, AND FBO. These are atypical phases. The ISF phase is an idling phase during which no functions are performed except time-interval and sector counting, i.e., during which the computer simply and literally marks time. It can be initiated by the *stop* instruction or by the stop button; it can be ended by the run button or the start button. (See the discussions of the stop, start, and run buttons in Sec. 18-8.) This phase is used in debugging a program or adding additional data to a program or to initiate a manual read-out, etc. The FBI (or FBO) phase takes place when a word is to be read from (to) the input (output) unit to (from) the buffer (see Fig. 18-18). During these phases the input or output unit has control of the buffer shifting pulse. When the shifting has been completed, the computer must wait for the next T_0 before beginning the next phase, in order that the machine will remain synchronized with the drum.

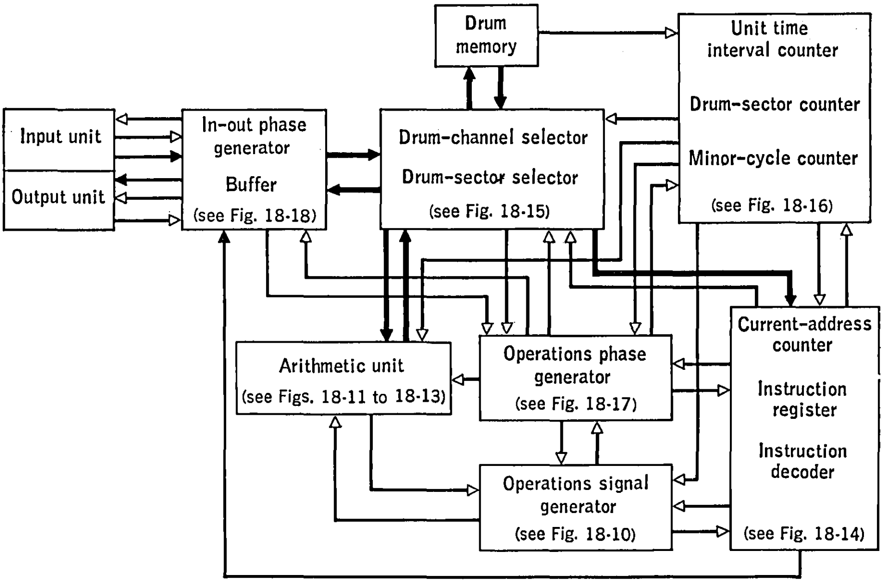


FIG. 18-8. Detailed block diagram of the Pedagac.

Block Diagram of the Pedagac (see Fig. 18-8). The heart of a computer is the arithmetic unit; directly or indirectly, all other parts of the computer service the arithmetic unit. The unit-time-interval and minor-cycle counters supply it directly with timing signals. The operations phase generator supplies it with phase pulses, in particular those of phases 2 and 3 during which it operates. The operations signal generator supplies it with signals specific to the operation being performed. The drum sector selector and drum channel selector deliver to and receive from it the words that are being operated on. The current-address counter, the instruction register, and the instruction decoder are all needed to generate their various· signals and may be thought of as indirectly servicing the arithmetic unit in this sense. Similarly the input-output unit, the in-out phase generator and buffer, and of course the memory are all necessary for the delivery and receipt of the words to be operated on and can therefore be considered as indirectly servicing the arithmetic unit.

The arithmetic unit generates one control signal that is used in the operations signal generator—the sign bit of the accumulator, used during a *conditional jump* instruction. Otherwise the operations signal generator receives most of its input signals from the instruction decoder and in a sense acts as the instruction-decoder output interpreter for the arithmetic unit. The operations phase generator also uses the outputs of the operations signal generator in determining phases. The operations phase generator in turn supplies the signal generator with signals used in forming the important instruction-ending pulse OEP and the *conditional jump* signal. The signal generator is also fed by the various counters.

The operations phase generator steps the computer through the proper phases. One phase signal, and only one, is on at all times when the computer is operating. A phase signal starts at T_0 of a given minor cycle and ends at T_{22} of the same or some later minor cycle. In general the phases essentially sequence themselves; i.e., each phase signal is initiated by the termination of the previous phase signal. Since a phase signal begins in the next unit time interval after the previous phase signal ended, a phase signal must be delayed 1 unit of time before it enters the gate generating the next phase signal. In addition two other important phase-ending signals are supplied to the operations phase generator: OEP, generated by the operations signal generator, and COI, generated by the drum sector selector. The former determines the end of phase 2 or 3, while the latter determines the end of IF1, IF2, IF3, IFO, or IFI.

The drum channel selector receives its input directly from the current-address counter in phase 1 or from the instruction register in other phases. In the Pedagac there are $2^5 = 32$ channels, the gates to only one of which can be open at a time. By opening the proper gates a channel is selected directly. However, the proper sector, or word within a channel, is selected indirectly. The drum sector selector compares the sector counter with the current-address counter in phase IF1 or with the α part of the instruction register in other idling phases. When coincidence occurs, the word can be read out of (or into) the drum, starting immediately at the T_0 of coincidence. The coincidence signal COI does not open any gates directly, as do the signals generated by the channel selector; rather *it simply steps the computer into the next phase,* in which it reads into or from the drum. It is the signal for this next phase that directly opens the proper gates—the gates to the arithmetic unit in $F2$, from the arithmetic unit in $F3$ (for transfer only), to the buffer in phase FOO, or from the buffer in phase FOI. In other words the channel selector opens the gate *to the proper place or channel* of the drum, but the sector selector, indirectly through the phase generator, opens the gates *to or from the arithmetic unit or buffer at the proper time.*

As noted above, besides supplying and receiving words the drum generates two signals, the *clock pulses* and the drum synchronizing pulse, which are considered again below in more detail.

Block Diagram of the Arithmetic Unit. In Fig. 18-9 the arithmetic unit has been divided arbitrarily into six sections. A combination adder-subtractor is used, and hence the adder-subtractor control generator is

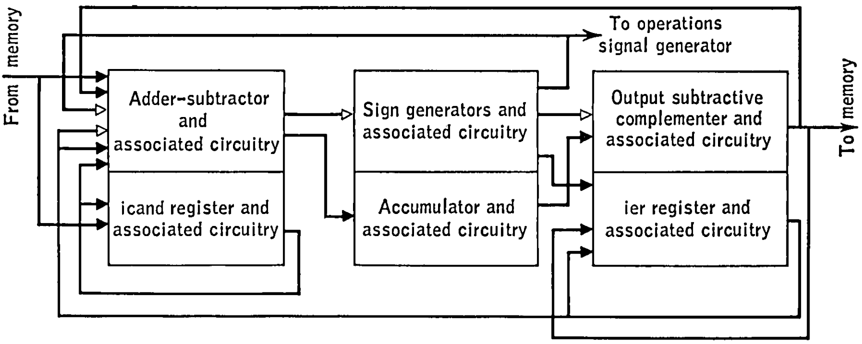

From memory
To operations signal generator
Adder-subtractor and associated circuitry
icand register and associated circuitry
Sign generators and associated circuitry
Accumulator and associated circuitry
Output subtractive complementer and associated circuitry
ier register and associated circuitry
To memory

FIG. 18-9. Block diagram of the arithmetic unit.

needed. During multiplication or division the icand register is intimately associated with the adder-subtractor, since it holds the multiplicand or the divisor. Besides receiving input words directly from the memory and the icand register, the adder-subtractor receives words from the recirculation of the accumulator (through the output subtractive complementer) and from the ier register. The word from the ier register is the minor product or the quotient; it is recirculated through the adder when it must be transferred to the accumulator as the final result. Similarly there is a recirculation through the adder during a *shift* instruction and a *transfer* instruction. Hence the input gating to the adder becomes somewhat formidable.

Further complexity is added by the different shifting requirements of the various instructions and by the necessity for overflow determination. The sign generators, one for addition and subtraction and one for multiplication and division, are complicated by the zero sensor and the sign holder, and by the need for the *sign-of-partial-dividend* generator.

The accumulator feeds directly into the output subtractive complementer. The output of the complementer may be sent to the memory through the drum-address-selector unit; or within the arithmetic unit it may go to the adder-subtractor or, when multiplication is initiated, to the ier register. During multiplication the partial-product bits are sent from the complementer to the ier register as they are generated, and the ier register supplies bits to the multiplier control-signal generator. During division the ier register receives the partial-quotient bits from the second stage of the accumulator.

Besides its receiving external control signals, some of the arithmetic unit's control signals are generated internally. These are indicated by the hollow-headed arrows between the boxes of the block diagram. The adder-subtractor receives the sign of the accumulator as a control signal during addition or subtraction; it also receives the multiplier control signal from the ier register (indirectly). The sign generators and the complementer receive the overflow as a control signal. In the zero sensor the result of an operation is used as a signal.

EXERCISES

(*a*) Trace on Fig. 18-7 the phases involved in the execution of each of the 17 instructions of the Pedagac.

(*b*) Trace the phases involved if the stop button is pushed, and then the run button; the stop button, and then the start button.

(*c*) Trace the phases involved if a *stop* instruction is executed and then the run button is pushed; if then the start button is pushed.

From the logical diagrams associated with each of the boxes of Fig. 18-8, determine:

(*d*) Precisely what pulses are received by the arithmetic unit from each source indicated.

(*e*) Precisely what pulses are received by the counters from each source indicated.

(*f*) Precisely what pulses are received by the current-address counter and instruction register.

(*g*) Precisely what pulses are received by the operations signal generator.

(*h*) Precisely what pulses are received by the operations phase generator.

(*i*) Precisely what pulses are received by the channel and sector selectors.

(*j*) Precisely what pulses are received by the in-out phase generator and buffer.

(*k*) Precisely what pulses are received by the input and output units.

(*l*) During what phases are each of the boxes of Fig. 18-8 functioning?

(*m*) The solid arrows in Fig. 18-9 indicate the flow of words between the boxes of the arithmetic unit. What word does each arrow represent, and during what operation does such a word flow?

18-4. Operations Signal Generator

The operations signal generator of the Pedagac generates 5 trivial and 19 nontrivial signals. In essence these signals tell the arithmetic unit what time period of a phase the computer is passing through and hence direct the unit in initiating, ending, and changing its various operations.

A description of each output signal of the generator is given in the Appendix. The composition of each signal is best shown by its logical diagram, all of which are given in Fig. 18-10. The exercises below are to be used as a guide to understanding the figure. A detailed analysis of the operations signal generator might be postponed until the arithmetic unit has been studied. However, during his study of that unit the student will need to know the general functions of the operations signals to understand why they are inputs to the various packages of the unit.

The operations signals can be classified broadly into four categories: First are signals which remain on throughout an operation and which serve to distinguish among the operations, namely, ADD, SUB, MDS, PQS, LDS, SHS, CLS, and TRS. Of these, three are on during more than one operation, MDS, PQS, and LDS. Second are signals that distinguish among the minor cycles within a phase, namely, MMI and DDS. Third are signals that distinguish among the unit time intervals within a minor cycle, namely, ASC, MCS, PQC, PQT, DRI, DCS, LMS, LAS, LES, LIS, SIC, SCC, and JCS. The five signals JCS, LMS, LAS, LES, and LIS also serve to distinguish among the operations, as did the signals of the first category (in Table 18-1 these signals are therefore entered

TABLE 18-1. OPERATIONS SIGNALS AND THEIR DURATIONS FOR EACH INSTRUCTION

Operation and decode signal	Category I, operation-determining signals	Category II, minor-cycle-determining signals		Category III, unit-time-interval-determining signals		Category IV, OEP	Phase
		Signal	Minor cycle	Signal	Time intervals	Time	
Addition, $53D$	ADD = $53D$	.....	†	.....	....	$T_{22}{}^{1}$	$F2$
Subtraction, $54D$	SUB = $54D$	.....	†	.....	....	$T_{22}{}^{1}$	$F2$
Major multiplication, $42D$	MDS	MMI	T^{19}	.....	....	$T_{22}{}^{19}$	$F2$
	PQS		T^{1-18}	MCS	T_{1-18}		
			T^{1-18}	PQC	T_{1-18}		
Minor multiplication, $32D$	MDS	.....	T^{19}	PQT	T_{1-18}	$T_{22}{}^{19}$	$F2$
	PQS		T^{1-18}	MCS	T_{1-18}		
			T^{1-18}	PQC	T_{1-18}		
Division, $42D$	PQS	DDS	T^{1-18}	.....	....	$T_{22}{}^{19}$	$F2$
				DRI	T_{21} only		
			T^{19}	PQT	T_{1-18}		
			T^{1-18}	DCS	T_{1-18}		
			T^{1-18}	PQC	T_{1-18}		
Logical multiplication, $71D$	LDS (LMS)	.....	†	LMS	T_{1-18}	$T_{22}{}^{1}$	$F2$
Logical addition, $72D$	LDS (LAS)	.....	†	LAS	T_{1-18}	$T_{22}{}^{1}$	$F2$
Logical equivalence, $73D$	LDS (LES)	.....	†	LES	T_{1-18}	$T_{22}{}^{1}$	$F2$
Logical nonequivalence, $74D$	LDS (LIS)	.....	†	LIS	T_{1-18}	$T_{22}{}^{1}$	$F2$
Shift, $70D$	SHS = $70D$	.....	T^{2}	SIC	As required	$T_{22}{}^{2}$	$F3$
			T^{2}	SCC	As required		
Clear, $50D$	CLS = $50D$	.....	†	.....	....	$T_{22}{}^{1}$	$F3$
Transfer, $52D$	TRS = $52D$	.....	†	.....	....	$T_{22}{}^{1}$	$F3$
Jump, $44D$	(JCS)	.....	†	JCS	T_{22}	....	$F1$
Conditional jump, $43D$	(JCS if SOB is 0)	.....	†	JCS	T_{22}	....	$F1$

† Single minor-cycle instructions.

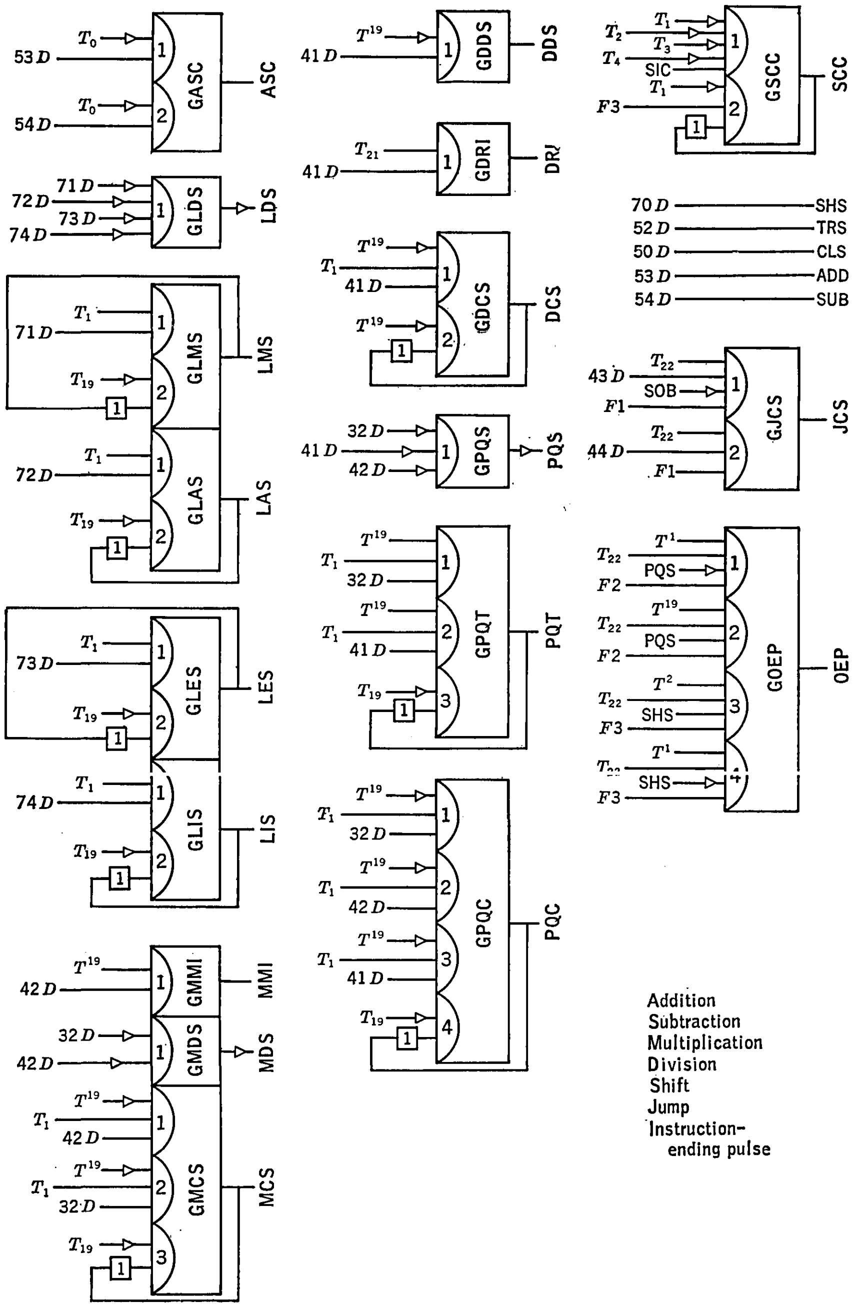
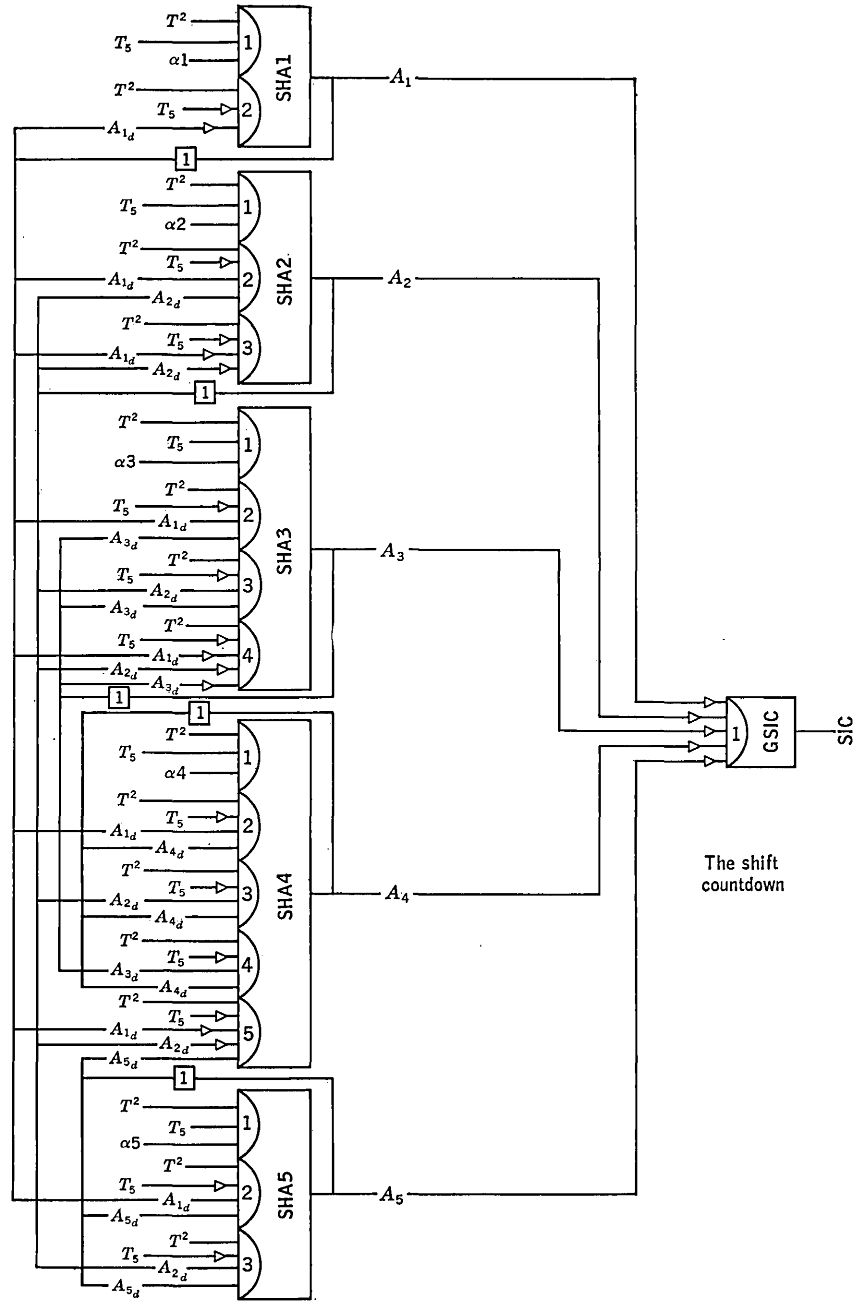


FIG. 18-10. Logical design of



the operations signal generator.

parenthetically in the "Category I" column). Signals MCS, PQC, PQT, DCS, SIC, and SCC are not on during every minor cycle of an operation and hence serve to distinguish among minor cycles, as did the signals of the second category (the minor cycles during which these signals are on are thus entered under the "Category II" column). Alone in the fourth category is the operations-ending pulse OEP. Table 18-1 shows the operations signals that are on during each instruction, and their durations.

The most complicated of the operations signals, the shift signals SIC and SCC, deserve a further word. During T^1 of the *shift* instruction while the contents of the accumulator is recirculating for complementation, the number of places to be shifted is held as the α address of the instruction register. During T^2, starting at $T_5{}^2$, the countdown is made until all zeros are obtained; then the shift inhibit control signal SIC is initiated. Signal SIC in turn initiates SCC (in package GSCC of Fig. 18-10), which performs the actual shift control function.

EXERCISES

(*a*) Discuss the functions of packages GASC, GMMI, GPQT, and GPQC.

(*b*) Describe the operation of the package GOEP. Give the function of each gate in detail.

(*c*) In package GMCS why is $\bar{T}_{19}$ used in the recirculation gate?

(*d*) Why does package GOEP include as inputs signals *F*2 and *F*3, whereas they do not appear as input signals to the other external signal-generating packages?

(*e*) The external control signals are used in the arithmetic unit only during phases 2 and 3. Why then are they generated during phase 1 in addition to phases 2 and 3?

(*f*) Explain the operation of packages GLDS, GPQS, and GMDS.

(*g*) How does SCC control the *shift* instruction in the arithmetic unit during T^1 and T^2?

18-5. Arithmetic Unit: Vertical Description

This section gives the function of each input gate of each package of the arithmetic unit, in the form of an extended outline. It is not specifically intended that this be read before the horizontal description. The vertical description is intended rather to be used with Table 18-1 and the logical diagrams of Figs. 18-11 to 18-13 as reference material as the student follows the paths of the bits in the horizontal description. The reader is cautioned, however, that he will not understand the arithmetic unit fully until he understands both this section and the next and that, though he might postpone working the exercises below until after the next section has been examined, he should not therefore neglect them entirely.

The *vertical description* of the *arithmetic unit* (see Figs. 18-11 to 18-13, pages 596 to 600) is as follows:

(α) Sign sensor SOAH

Gate 1 loads the sign of (α) at $T_0{}^1$ of F2.

Gate 2 holds the sign of (α) until OEP.

Adder-subtractor control generator ASCG

Gates 1, 2, 3, and 4 form the appropriate sum or difference control signal for the *add* and *subtract* instructions. For multiplication we sum all the time, and gate 5 forces OPS on. For division we difference all the time; so all gates are inhibited with DDS.

Adder-subtractor (α) input package ASAI

Gate 1 allows (α) to enter the adder during addition, subtraction, the four logical instructions, and the first minor cycle T^1 of division, i.e., not during multiplication.

Gate 2 allows (icand) to enter the adder for division T^{2-18}. ($\bar{T}_0$ is not needed since DCS can be on only during T_{1-18}, during division.)

Gate 3 allows (α) to enter the adder gated with MUS during T^1 of multiplication. (Like DCS above, MCS is off at T_0.)

Gate 4 allows (icand) to enter the adder gated with MUS during T^{2-18} of multiplication.

Adder-subtractor B input package ASBI

Gate 1 allows (acc) to enter the adder during phase 2 of addition, subtraction, and the four logical instructions (not during multiplication and division).

Gate 2 recirculates (acc) during T^1 of phase 3 of the shift and transfer; it forces (acc) to zero during phase 3 of clear. During T^2 of the shift, T^1's being off prevents the full-length recirculation of (acc).

Gate 3 allows $2N^i$ to go into adder during division T^{1-18}.

Gate 4 passes (acc) into the adder during T^{2-18} of multiplication [because T^1 in multiplication concerns only (α)].

Gate 5 passes the minor product or quotient from the ier register into the accumulator during T^{19} (the PQT signal).

Stage 1 of the icand register, S1IR

Gate 1 loads icand initially (during T^1 of $F2$) in division or multiplication.

Gate 2 recirculates (icand) during T^{2-18} of multiplication and division.

Gate 3 holds the bit until shifted, during T_{19}–T_0 of the appropriate minor cycles.

Stage n of the icand register, SnIR (S2IR is shown as an example)

Gate 1 receives bit from S($n - 1$)IR during shifting.

Gate 2 holds the bit until shifted.

Input to carry former ITCF (see Sec. 15-3)

Gate 1 passes B' to form sum (OPS = 1).

Gate 2 passes $\bar{B}'$ to form difference (OPS = 0).

Gate 3 passes $\bar{B}'$ during logical multiplication and addition, since for these operations the result is the additive carry C' (with the C input to the carry former appropriately constant).

Carry input to adder-subtractor CIAS (see Sec. 15-3)
Gate 1 forms the carry input ($C \equiv 1$) for logical addition and equivalence.
Gate 2 inhibits the delayed carry for logical multiplication and inequivalence ($C \equiv 0$) but otherwise passes the regular carry.

Adder-subtractor result former ASRF
Gates 1, 2, 3, and 4 form the sum-difference result (see Sec. 15-3).

Adder-subtractor carry former ASCF
Gates 1, 2, and 3 form the carry (see Sec. 15-3).

Result zero-detector generator RZDG
Gate 1 loads the result from the adder.
Gate 2 holds any unit found in the result until the next T_0.

Add-subtract sign generator ASSG
Gates 1 and 2 form the sign for addition and subtraction.
Gate 3 ensures a positive sign if the result is zero.

Product-quotient sign generator PQSG
Gates 1 and 2 form the sign of the product or quotient.
Gate 3 ensures a positive sign if the result is zero.

Sign of partial-dividend generator SPDG
Gate 1 forces SOD positive during T^1.
Gates 2 and 3 form the signs of the partial dividends during T^{1-18}.
Gate 4 ensures a positive sign if a partial dividend is zero.
Gate 5 holds SOD until the next sign is formed at T_{21}.

Sign of B holder SOBH
Gate 1 loads the sign of the *add* and *subtract* instruction result B.
Gate 2 loads the sign of the multiplication and division instruction result B.
Gate 3 holds the sign of B for the nonarithmetic instructions.
Gate 4 ensures a positive sign when the accumulator is cleared.
Gate 5 holds the sign of B during the arithmetic operations until the new sign is formed at OEP.

Accumulator-shift pulse signal generator ASPS
Gate 1 shifts the accumulator at each clock pulse except T_0 during every minor cycle of phase 2, except the 19th minor cycle of major multiplication.
Gate 2 shifts the accumulator during the *transfer* and *clear* instruction, except at T_0.
Gates 3 and 4 shift the accumulator during T^1 and T^2 of the *shift* instruction, as directed by SCC from shift sensor (see Exercise *t*).

Gate 5 shifts the accumulator during first pulse period (T_0) of each minor cycle of both multiplication instructions.

Stage 1 of the accumulator register S1AR
Gate 1 reads the result from the adder into the accumulator, except for logical multiplication and addition.
Gate 2 reads the additive carry into the accumulator as the result for the logical addition and multiplication instructions (see Sec. 15-3).
Gate 3 holds the bit until shifted.

Stage 5 of accumulator register S5AR
Gate 1 passes the shortened contents of the accumulator during T^2 of the *shift* instruction.
Gate 2 loads the signal from $P4$ during T^1 of the *shift* instruction.
Gate 3 loads the signal from $P4$ except during the *shift* instruction.
Gate 4 holds the bit until shifted.

Overflow holder OFLH
Gate 1 loads the overflow holder at each T_0.
Gate 2 holds the bit until the next T_0.

Output-complementer carry former OCCF (see Sec. 15-3)
Gates 1 and 2 form the carry for the complementer (see Exercise *cc*).

Output-complementer result former OCRF (see Sec. 15-3)
Gates 1, 2, and 3 form the result of the subtractive complementation (see Exercise *dd*).

Multiplier control signal generator MUSG
Gate 1 forms the multiplier control signal from $P1$ of (acc) during T^1.
Gate 2 forms multiplier control signal from $P1$ of (ier) during T^{2-18}.
Gate 3 holds the bit until the next minor cycle.

Ier-shift pulse signal generator ISPS
Gate 1 shifts the contents of accumulator into the ier register during T^1 of the multiplication instructions.
Gate 2 shifts in minor product bits during T^{2-19} of the multiplication instructions.
Gate 3 shifts for the recirculation of (ier) during the division instruction.
Gate 4 shifts (ier) out to the accumulator during T^{19} of minor multiplication and division.

Stage 1 of the ier register S1ER
Gate 1 loads the contents of accumulator into the ier register (see Exercise *hh*).
Gate 2 loads the least significant bit of the accumulator into the ier register (see Exercise *ii*).
Gate 3 recirculates (ier) during division.
Gate 4 holds the bit.

Stage 18 of the ier register S18ER

Gate 1 loads each quotient bit.

Gate 2 passes the normal output from stage 17 during normal shifting.

Gate 3 recirculates the contents of stage 18 during the division instruction.

Gate 4 recirculates the contents of stage 18 during instructions other than division.

EXERCISES

(*a*) In package ASAI gate 1, under what conditions will (α) enter the adder?

(*b*) For the *clear* instruction why is input $\overline{\text{CLS}}$ necessary only in gate 2 of package ASBI?

(*c*) How would the inputs to gate 2 of package ASAI be modified if DCS were replaced by DDS?

(*d*) Compare the functions of packages SOBH and SOAH.

(*e*) Explain the operation of the adder-subtractor control package ASCG during division.

(*f*) Explain the operation of package ASAI and ASBI during the *transfer* instruction.

(*g*) What occurs in each gate of packages ASAI and ASBI during phase 1?

(*h*) Explain the purpose of input $\overline{\text{PQS}}$ to gate 1 of package ASBI.

(*i*) What are the functions of the $\overline{T^1}$ inputs to gates 2 and 4 of package ASAI and to gate 2 of package S1IR?

(*j*) What is the function of gate 3, package S1IR? When does the package act to store a bit?

(*k*) How is B' forced through package ITCF for logical multiplication and addition? Why is $F2$ necessary at gate 2 but not at gate 1?

(*l*) How are the special cases of C formed for the logical instructions?

(*m*) Show one condition in which T_0 is essential to the proper operation of the adder-subtractor result former.

(*n*) Explain the purpose of T_0 in package ASCF.

(*o*) Describe the operation of packages ASRF and ASCF during the *transfer* and *shift* instructions.

(*p*) What is the purpose of the $\overline{\text{LES}}$ and $\overline{\text{LIS}}$ signals of gate 2, package S1AR?

(*q*) What is the purpose of the LDS and $\overline{\text{LDS}}$ signals of package CIAS?

(*r*) The countdown for the *shift* instruction starts at T_5. How is this arranged for in the logical design? (HINT: See packages ASPS and GSCC.) Describe the operation of package S5AR during a *shift* instruction.

(*s*) In package ASPS why are the inputs SHS and $F3$ used in gates 2, 3, and 4?

(*t*) Describe the functions of gates 3 and 4 of package ASPS.

(*u*) Why can the zero detector be on during every minor cycle?

(*v*) Under what conditions is gate 3 of package ASSG essential?

(*w*) Why is a separate package SPDG necessary, in addition to package ASSG?

(*x*) Why must SOD of package SPDG be forced positive during T^1 of the division instruction (gate 1)?

(*y*) Why is recirculation necessary for package SPDG and not for package ASSG?

(*z*) What are the functions of gates 3 and 5 of package SOBH, and why are the two used instead of one?

(*aa*) Why is the sign of B held?

(*bb*) Why must OF be held in package OFLH?

(*cc*) Construct a vertical description of package OCCF.

(*dd*) Construct a vertical description of package OCRT. Why is neither OF nor $\overline{\text{OF}}$ an input to gate 2?

(*ee*) In package MUSG why are both gates 1 and 2 necessary?

(*ff*) Why must signal MUS be held in package MUSG?

(*gg*) What is the purpose of the T_{1-17} input to gate 3 of package ISPS?

(*hh*) When is gate 1 of package S1ER used?

(*ii*) When is gate 2 of package S1ER used?

(*jj*) What happens to the least significant bit of the ier register during T^{2-18} of multiplication?

(*kk*) In gate 3 of package S18ER why are both $\bar{T}_{21}$ and $\overline{\text{SPI}}$ necessary?

(*ll*) Why are both gates 3 and 4 necessary in package S18ER?

(*mm*) Redesign the ier register so as to read the quotient bits into stage 1 instead of into stage 18. Include any additional changes in the computer, including generators for any additional control signals required.

18-6. Arithmetic Unit: Horizontal Description

Phase 1 *of All Instructions.* During phase 1 the next instruction is being read into the instruction register and decoded, etc. The arithmetic unit idles. The accumulator holds the result of the last operation; the icand register, the multiplicand of the last multiplication or the divisor of the last division; and the ier register, the minor product of the last major multiplication, unless it has been cleared by a minor multiplication or a division. (Since the icand and ier registers are cleared before they are reused, their contents are noted but hereafter ignored.) The sign of the accumulator is held in package SOBH, but package SOAH (holds the sign of the α operand) has been cleared at the end of the last operation. The zero detector was cleared after SOB was formed, but package OFLH will hold a bit if there was an overflow into the second stage of the accumulator (as the result of a differencing with a larger subtrahend). The input gates to the arithmetic unit will all be closed.

Addition and Subtraction Instructions. At $T_0{}^1$ of phase 2 the sign of (α) is fed from package GMOα in the memory selection unit to the sign-of-α holder (package SOAH). The signals SOA and SOB along with signal ADD for addition or SUB for subtraction form signal OPS in package ASCG. OPS is fed to package ITCF, where it determines whether the sum or difference will be generated (see Sec. 15-5).

During $T_1{}^1$ the least significant bit of (α) and the least significant bit of (acc), which will have been complemented if signal OF is present, are passed through the adder-subtractor input packages ASAI and ASBI. In package ITCF the input to the adder-subtractor carry generator is formed by gating signals B' (from package ASBI) and OPS as follows: $B' \cdot \text{OPS} + \bar{B}' \cdot \overline{\text{OPS}}$ (Sec. 15-3). The outputs of packages ITCF, ASAI, ASBI, and CIAS are fed to ASCF, where the carry is formed. The output of package ASCF is delayed one unit time interval and fed back to package CIAS (which passes it during all arithmetic operations).

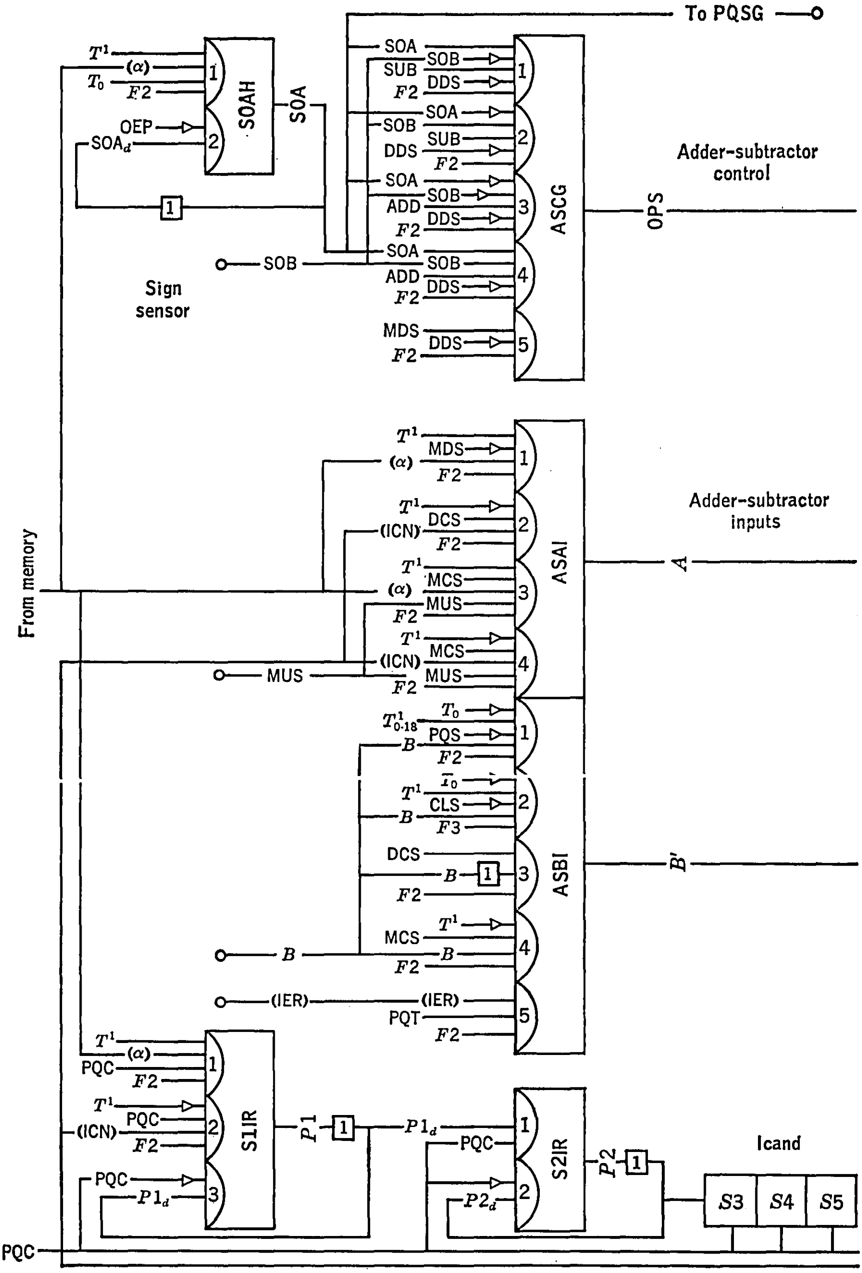
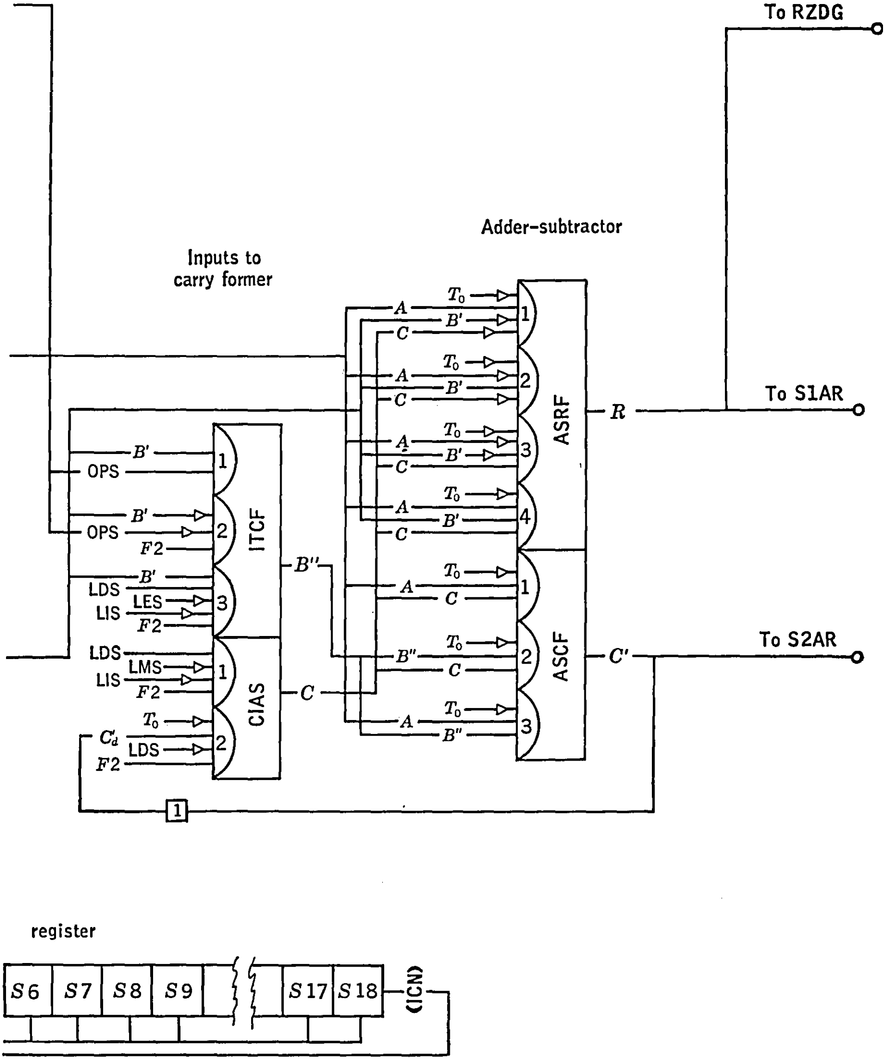


FIG. 18-11. Logical design of the arithmetic



unit: adder-subtractor and icand register.

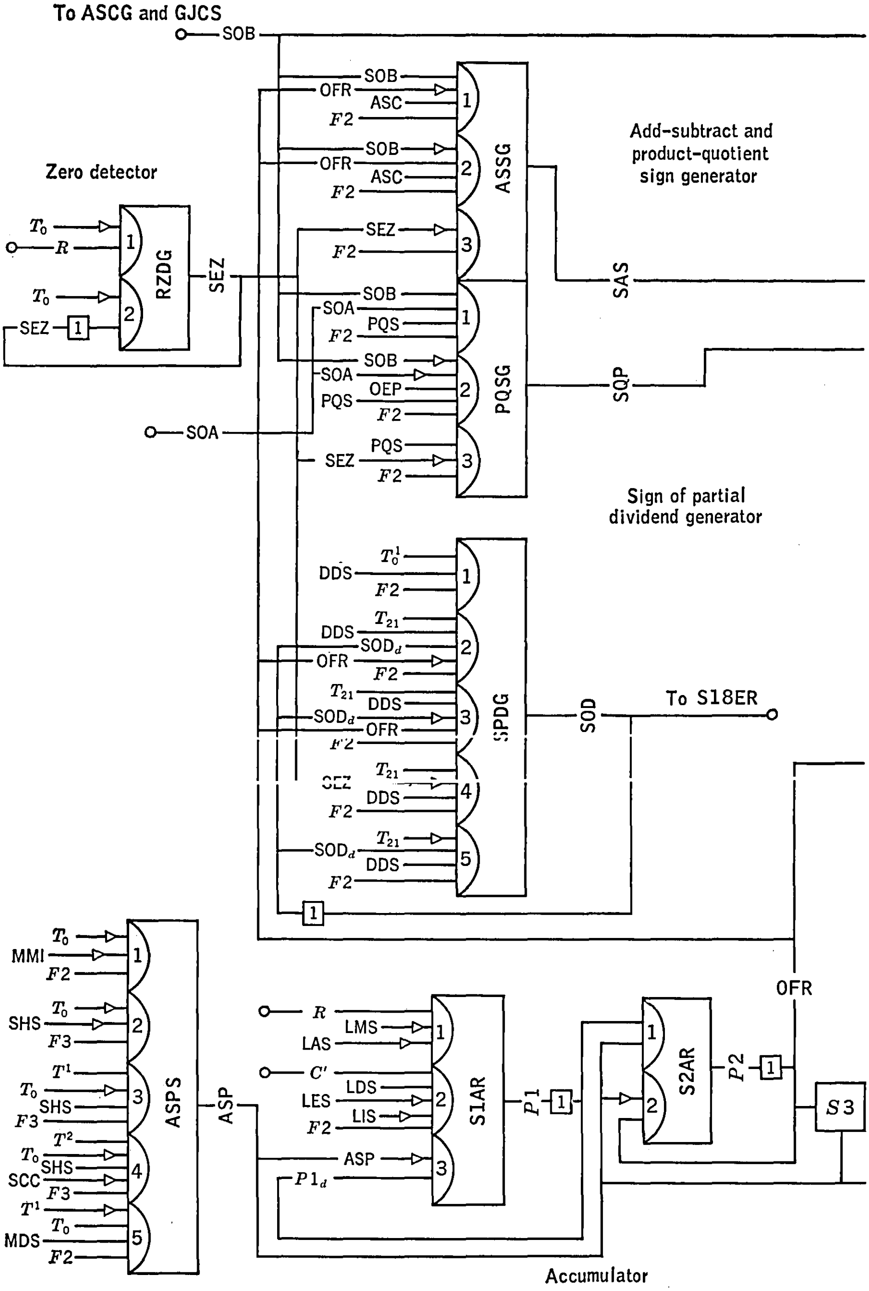
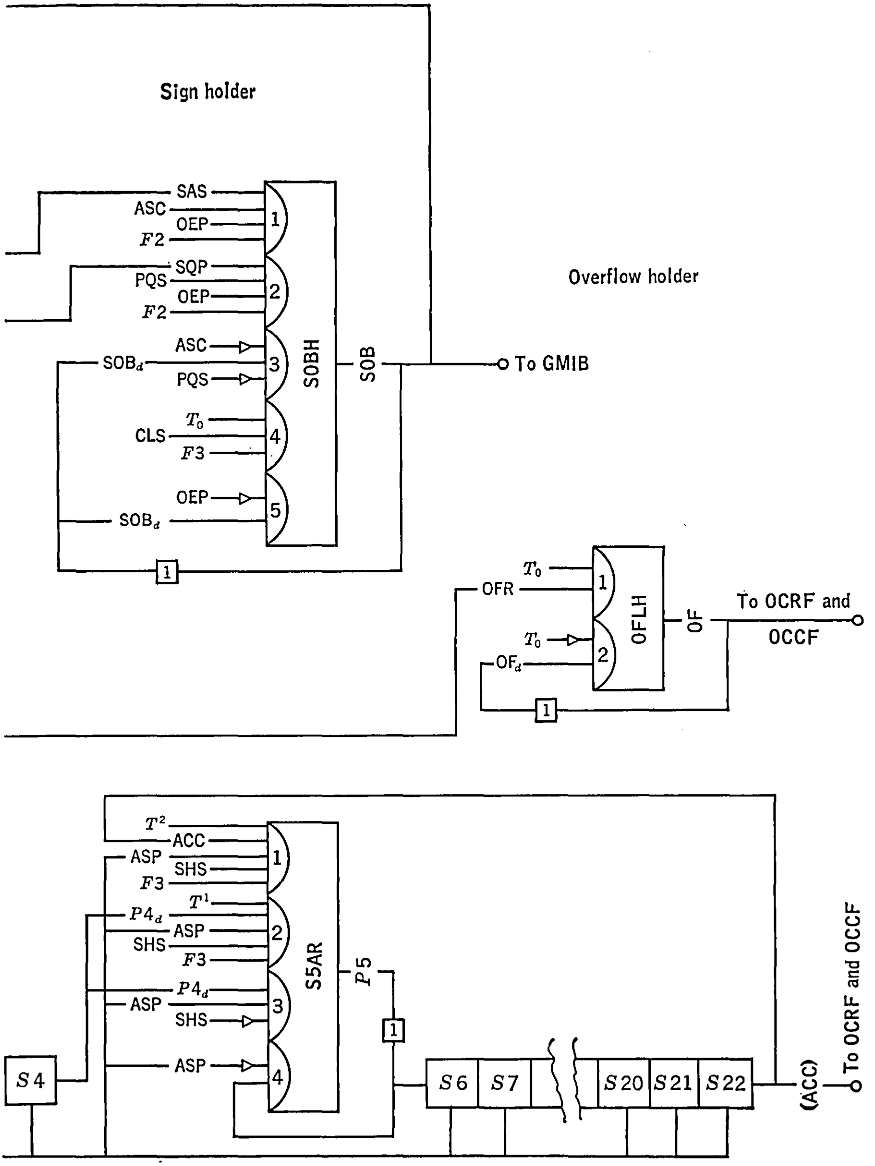


FIG. 18-12. Logical diagram of the arithmetic



register
unit: sign generator and accumulator.

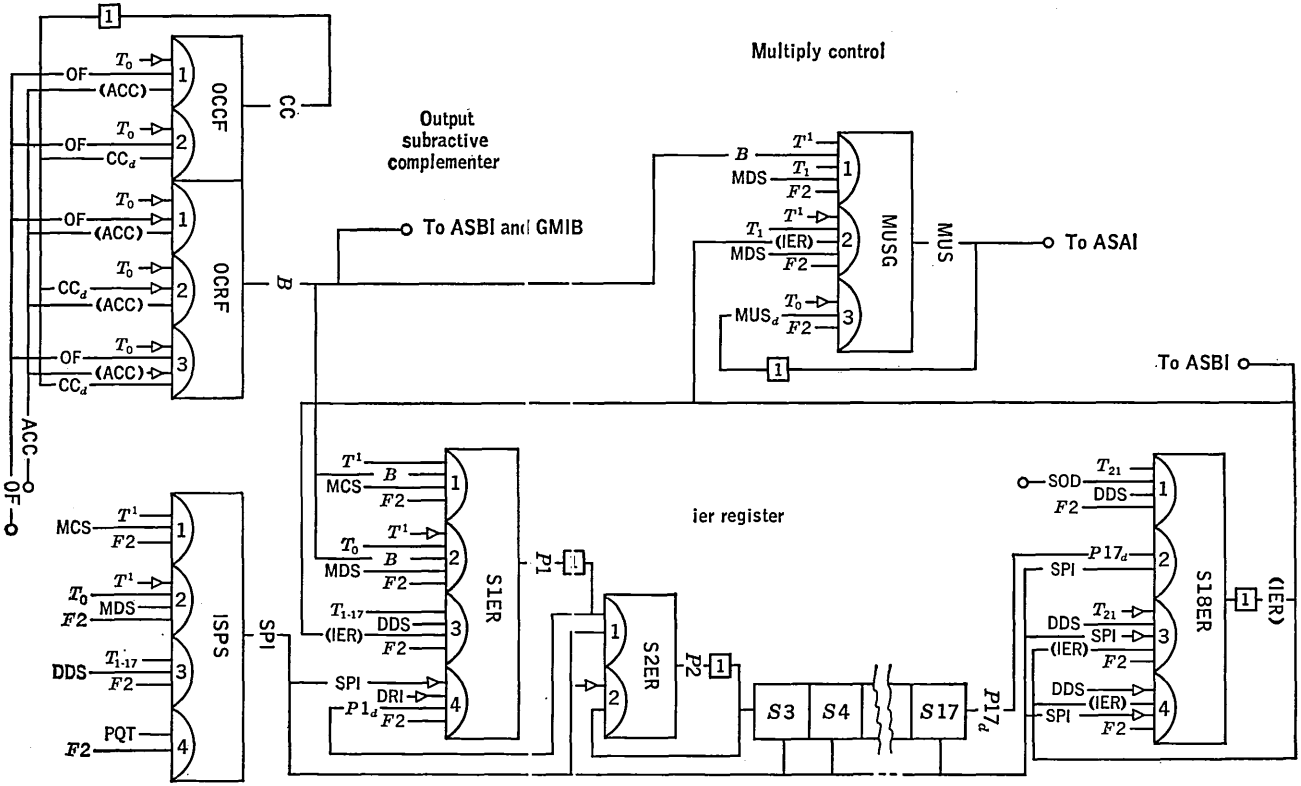


FIG. 18-13. Logical diagram of tl e arithmetic unit: output complementer and ier register.

Since there is no previous carry during T_1, the least significant bit of (α) is gated with the least significant bit of (acc) in package ASRF to form the result R. Signal R is fed to stage 1 of the accumulator (package S1AR) as well as to the zero detector (package RZDG).

During T_2 the next to least significant bits of (α) and (acc), with the carry formed during T_1, are fed to packages ASRF and ASCF, following the same path as the least significant bit. Again the output to package ASRF is fed to packages S1AR and RZDG, and the output of package ASCF is fed to package CIAS. Also during this unit time interval the (acc) is shifted one bit position to the right, thereby placing the least significant bit of the result in stage 2 of the accumulator.

This process continues for an additional 16 unit time intervals. At T_{18} the most significant bit of the result is formed, with a possible carry. A carry at T_{18} may be due either to an overflow in summing (i.e., a result greater than 1) or to an overlarge subtrahend in differencing. During T_{19-22} the A and B inputs are zero, and the result R is determined solely by C. Signal R from package ASRF continues to be fed to packages S1AR and RZDG until T_{22}. At T_{22} package GOEP forms signal OEP, which gates the sign of the result, formed in package ASSG, into package SOBH and concurrently kills the recirculation of signal SOA in package SOAH. Signal OEP also acts as an operation-ending pulse causing the machine to end the arithmetic operation and move on to phase IF1.

Multiplication Instructions. During $T_0{}^1$ of phase 2 the sign of (α) is fed from package GMOα to package SOAH, where it is stored. Operation signal MDS at gate 5 of package ASCG keeps signal OPS on until the end of the phase, and the adder-subtractor thus acts as an adder throughout this instruction. During T_1 the least significant bit of (acc) is sent through the output complementer (complemented if signal OF is present) to gate 1 of package MUSG and stage 1 of the ier register (gate 1 of package S1ER). Simultaneously signal B is gated with signal MDS in package MUSG to form the multiplier signal MUS. This signal is held throughout the minor cycle by recirculation through gate 2 of MUSG. Also during $T_1{}^1$ the least significant bit of (α) is passed into the icand register through gate 1 of package S1IR and fed to gate 3 of package ASAI. In that package it is gated with signal MUS such that (α) is allowed to enter the adder only if MUS is on. Then (α) (if MUS is present) and (acc) arefe d to the adder-subtractor through packages ASRF, ITCF, and ASCF. In package ASRF the result of addition is formed, and in package ASCF the carry into the bit position is formed. The output of package ASRF is fed to the first stage of the accumulator, through gate 1 of package S1AR, and to the zero detector, package RZDG. The output of package ASCF is delayed by one unit time interval and fed into package CIAS, which passes it on to packages ASRF and ASCF to be combined with the next bits of (α) and (acc). During T_2 the second least significant bit of (acc) is fed to stage 1 of the ier register, as the least significant bit of (acc) previously stored there is shifted one position. The second least significant bit of (α) is fed simultaneously, through the same paths as the least significant bit, to stage 1 of the icand register, the least significant bit being shifted once to make room for it, and to package ASAI, where it is gated with the MUS signal. During T_{3-18} the remainder (α) and (acc) follow the same paths, forming the first partial product in the accumulator and storing the multiplier [the original (acc)] in the ier register. At T_{19} the carry generated in T_{18}, if there was one, will be fed to the accumulator via the adder as the accumulator is shifted one position. During T_{20-22} the accumulator will be shifted three positions, so that the least significant bit of the partial product (which is in fact the least significant bit of the minor product) will be in the least significant position of that register.

During $T_0{}^2$ this least significant bit will be transferred out of the accumulator into gate 2 of package S1ER of the ier register, as the ier register is shifted one position to receive it. The least significant bit of the multiplier is thus "spilled over," and the second least significant bit assumes the end position of the register. During $T_1{}^2$ this bit is fed to gate 2 of package MUSG, where the multiplier signal for the second minor cycle is generated. During this time interval the least significant bits of the accumulator and the icand register will be fed to the adder-subtractor input packages. The least significant bit of the icand register will also be fed back into the icand register through gate 2 of package S1IR. The flow of the operands through the adder-subtractor during this minor cycle will proceed as described for the first minor cycle.

This procedure will continue for a total of 18 minor cycles, to complete the multiplication. At the end of the 18th minor cycle the major product, and the most significant bit of the minor product, will be in the accumulator; the rest of the minor product, with the most significant bit of the original (acc), will be in the ier register.

For either major or minor multiplication (instruction 42 or 32), the most significant minor-product bit will be transferred at $T_0{}^{19}$ into the ier register as that last bit from the multiplier is spilled over from ier stage 18. If the instruction is minor multiplication, then at $T_1{}^{19}$ the least significant bit of the minor product, stored in package S18ER, will be transferred to the most significant position of the accumulator, through gate 5 of package ASBI, gate 2 of package ASRF, and finally gate 1 of package S1AR, as the least significant bit of the major product is spilled over from the end of the accumulator. This process continues for an additional 17 unit time intervals, at the end of which the complete minor product will have been transferred to the accumulator. During T_{19-22}^{19} the accumulator is shifted four more positions, to place the least significant bit of the minor product in the least-significant-bit position of the accumulator. If the instruction is major multiplication, the major product is left in the accumulator and the minor product in the ier register, with the machine idling for T_{1-21}^{19}. At $T_{22}{}^{19}$ of either operation signal OEP comes on, gating the product's sign, which has been generated in package PQSG, into gate 2 of package SOBH, and killing the recirculation of signal SOA in package SOAH. Signal OEP also acts as an operations-ending pulse, causing the machine to move on to phase IF1.

Division Instruction. During $T_0{}^1$ of phase 2 the sign of (α) is fed from package GMOα to gate 1 of package SOAH. Operation signal DDS is fed to all gates of package ASCG to inhibit signal OPS. (The absence of signal OPS will cause the adder-subtractor to form the difference throughout this instruction; see Sec. 15-6.) Signal DDS is also fed to gate 1 of package SPDG, where it acts as a primer to set the sign of the partial difference (SOD) positive initially. During $T_1{}^1$ the least significant bit of the accumulator is sent through the output complementer (complemented if signal OF is present), then into gate 3 of package ASBI, delayed by one pulse period to form B' (see Sec. 15-6). Also during $T_1{}^1$ the least significant bit of (α) is passed into the icand register through gate 1 of package S1IR and through gate 1 of package ASAI into the adder-subtractor. Since package OCRF of the complementer was cut off at T_0, the B' input to the adder-subtractor will be zero at T_1. In package ASRF the first bit of $2(\text{acc}) - (\alpha) = B' - A$ is formed as package ASCF forms the carry into the next position, which is delayed and fed into package CIAS. The result of the subtraction, signal R, is fed to the zero detector, package RZDG, and to the first stage of the accumulator, package S1AR. At $T_2{}^1$ the second least significant bit of the original (acc) is routed through the same path as the least significant bit. As the second least significant bit of (α) is fed to the adder-subtractor and stage 1 of the icand register, the least significant bit of (α) is shifted to stage 2. In the adder-subtractor this bit is combined with the carry from $T_1{}^1$, *and with the delayed least significant bit of the original* (*acc*), to form the second bit of $2(\text{acc}) - (\alpha)$. This second bit is fed to the zero detector and to stage 1 of the accumulator, as the first bit is shifted to stage 2. The carry formed during T_2 is delayed and sent to the carry input package CIAS. This process continues through T_{18}, at which time (α) will have been transferred to the icand register and the first 18 bits of the first partial dividend will have been formed and stored in the 18 most significant bit positions of the accumulator. At T_{19} any carry generated in T_{18} will be fed to the (shifted) accumulator through the adder-subtractor. Such a carry, indicating that $2(\text{acc}) - (\alpha) < 0$, will be propagated until T_{22} (see Sec. 15-6). At $T_{21}{}^1$ the output of stage 2 of the accumulator (signal OFR) is gated with signal SOD in package SPDG to form the new SOD. This new SOD is fed to gate 1 of package S18ER, where it is stored temporarily as the most significant bit of the quotient.

During T^2_{1-17} the most significant bit of the quotient which is stored in S18ER is shifted 17 positions circularly, to package S17ER. The generation of the next most significant bit of the quotient is formed in the same way as the most significant bit, except that $(\alpha) = (\text{icand})$ is fed into the subtractor, through gate 2 of package ASAI, and is subtracted from the *absolute value* of (acc).

This process continues for an additional 16 minor cycles, at the end of which we shall have formed the 18 most significant bits of the quotient, which are stored in the ier register.

At $T_1{}^{19}$ the least significant bit of the quotient which is held in package S18ER will be transferred to the most-significant-bit position of the accumulator by way of gate 5 of package ASBI, then to gate 2 of package ASRF, and finally to gate 1 of package S1AR. This process continues for an additional 17 unit time intervals, at the end of which time the quotient will have been transferred to the accumulator. During T^{19}_{19-22} the accumulator is shifted four positions, so that the least significant bit of the quotient is in the least significant position of the accumulator. Also at $T_{22}{}^{19}$ signal OEP comes on, gating the sign of the quotient (generated in package PQSG) into gate 2 of package SOBH and killing the recirculation of signal SOA in package SOAH. Signal OEP also acts as an operation-ending pulse, causing the machine to move on to phase IF1.

Logical Instructions. During $T_0{}^1$ of phase 2 of any logical instruction the sign of (α) is fed from package GMOα to package SOAH. (This action will not have any effect on the logical instructions.)

In the *logical-addition* instruction, during T_1 the least significant bit of (α) is fed through gate 1 of package ASAI to package ASCF; the corresponding bit of (acc) (complemented if OF is present) is fed through gate 1 of package ASBI to package ITCF. Gate 3 of package ITCF allows (acc) to be fed to package ASCF. At the same time the carry input is held on by signal LDS at gate 1 of package CIAS. Then the outputs of packages ASAI, ITCF, and CIAS are gated together in package ASCF to form the result of the logical addition, signal C' (see Sec. 15-3) This signal is fed to the accumulator through gate 2 of package S1AR. This process continues until the most significant bit of the logical addition is formed at T_{18}. The sign of the result is SOB, the sign of the original (acc), recirculating through gate 3 of package SOBH. During T_{19-22} the accumulator is shifted four positions, so that the least significant bit of the result is in the least significant position of the accumulator. At T_{22} signal OEP comes on, ending the logical operation, inhibiting the recirculation of signal SOA in package SOAH, and moving the machine into phase IF1.

In the logical-multiplication instruction, during T_1 (α) is started through gate 1 of package ASAI to package ASCF, and (acc) (complemented if signal OF is present) is started through gate 1 of package ASBI to package ITCF. Gate 3 of package ITCF allows (acc) to enter package ASCF. Signals $\overline{\text{LMS}}$ and $\overline{\text{LDS}}$ in gates 1 and 2 of package CIAS inhibit C (see Sec. 15-3). In gate 3 of package ASCF, (α) and (acc) are gated together to form the result of logical multiplication, signal C'. This signal is fed to the accumulator through gate 2 of package SIAR. This procedure continues until the most significant bit of the logical multiplication has been formed at T_{18}. The operation during T_{19-22} is identical to that in logical addition.

In the *logical-equivalence* instruction, during T_{1-18} (α) is fed through gate 1 of package ASAI to package ASRF, and (acc) (complemented if signal OF is present) is fed through gate 1 of package ASBI to package ASRF. The carry input is held on during this minor cycle by signal LDS at gate 1 of package CIAS (see Sec. 15-3). Signals A, B', and C are combined in gates 3 and 4 of package ASRF to form the logical equivalence result, signal R. This result is fed to the accumulator through gate 1 of package S1AR. This process continues until the most significant bit of the logical equivalence is formed at T_{18}. The operation during T_{19-22} is identical to that in logical addition.

In the *logical-inequivalence* instruction, during T_{1-18} (α) is fed through gate 1 of package ASAI to package ASRF, and (acc) (complemented if signal OF is present) is fed through gate 1 of package ASBI to package ASRF. Signals $\overline{\text{LIS}}$ and $\overline{\text{LDS}}$ in gates 1 and 2 of package CIAS inhibit C (see Sec. 15-3). Then signals A and B' are gated together in gates 1 and 2 of package ASRF to form the logical inequivalence result, signal R. This result is fed to the accumulator through gate 1 of package S1AR. The procedure continues until the most significant bit of the logical inequivalence is formed at T_{18}. The operation during T_{19-22} is identical to that in logical addition.

Shift Instruction. During T^1_{1-22} of phase 3 (acc) is sent to package OCRF, where it is complemented if signal OF is present. The output B of package OCRF is fed through gate 2 of package ASBI, then through gate 2 of package ASRF, and back to the accumulator through gate 1 of package S1AR. Thus, at the end of the first minor cycle, the contents of the accumulator has been complemented if necessary and restored to its original position. The accumulator shift pulse signal ASP for the second minor cycle is generated in gate 4 of package ASPS. Starting again at $T_5{}^2$, the accumulator is shifted until signal SCC from package GSCC prevents further shifting by killing signal ASP. The shortened shift is accomplished by feeding the output of stage 22 of the accumulator, package S22AR, to gate 1 of package S5AR. The arithmetic unit will idle from the time that signal SCC comes on until signal OEP comes on at $T_{22}{}^2$. Signal OEP ends the operation and causes the machine to move on to phase IF1. The sign of the result is the original sign of (acc), recirculating through gate 3 of package SOBH.

Clear Instruction. During T_0 of phase 3 of the *clear* instruction the sign of B is forced positive by gate 4 of package SOBH. During T_{1-22} the accumulator shift pulse signal (ASP), generated in gate 2 of package ASPS, shifts the accumulator 22 positions. Since all inputs to the adder-subtractor are cut off during the operation, the accumulator is cleared. At T_{22} signal OEP comes on, ending the operation and causing the machine to move on to phase IF1.

Transfer Instruction. During $T_0{}^1$ of phase 3 signal SOB from package SOBH is transferred to the address of the memory through gate 3 of package GMIB (see Fig. 18-15). The sign of B is still retained in package SOBH by recirculation through gate 3. During T_{1-18} (acc) is fed into output complementer package OCRF, where it is complemented if signal OF is present. The output of the complementer, signal B, is transferred to address α through gate 2 of package GMIB. During T_{1-22} signal B is also fed through gate 2 of package ASBI, then through gate 2 of the adder-subtractor (package ASRF), and back into the accumulator through gate 1 of package S1AR. By T_{22} the original (acc) will have been restored to its original position in the accumulator (complemented if necessary). At T_{22} signal OEP comes on, ending the *transfer* operation and causing the machine to move on to IF1.

18-7. Instruction Register, Current-address Register, Instruction Decoder, Memory-selection Unit, and Counters

Instruction Register, Current-address Register, and Instruction Decoder (see Fig. 18-14). We have denoted the α bits of the instruction by αi, the reading of the current-address counter by $\alpha' i$, and the generated bits for memory selection, which may be $\alpha' i$ during phases IF1 and $F1$ or αi otherwise, by $\alpha^* i$.

The current-address counter is a straightforward cascading counter, which increases its count by 1 at each pulse from package 1CSG. This package will have an output at T_{22} of each phase 1, unless a jump is directed by signal JCS. When a jump is required, each αi is transferred to and loaded in the current-address counter, in a parallel fashion, by a pulse from package JCLG, as the previous αi are automatically erased.

The instruction register is a shift register stepped only during $F1$ in receiving (α) from the drum. The 6 most significant bits of the instruction, $\omega 6$ to $\omega 1$, enter the instruction decoder, which is simply a decoding matrix (see Exercises *a* through *f*). If the start button is pushed during the idling phase ISF, the current-address counter and the instruction register are both completely cleared by signal STC from package STCG.

The Channel and Sector Selectors (see Fig. 18-15). The signals α^*12, α^*11, α^*10, α^*9, and α^*8 open one of the 32 gates to the channels. Both the read-in and read-out gates are opened at the same time. A word will be read in or out depending on the gates of packages GMIB and GMOα.

The sector selector really chooses a time rather than a sector as directed by α^*7 through α^*1. It chooses that time when all the pairs of bits (Ci, $\alpha^* i$) are identical. This will occur first at T_0 of the proper minor cycle and will continue until the sector counter is again stepped at T_0 of the next minor cycle. The signal generated, called COI, changes the phase from an idling phase to an active phase. The active phase signal opens GMIB or GMOα so that a word may be read in or out.

Since we have decided against having a large *or*-gate package for the Pedagac, to obtain (α) we must use the principle that

$$X_1 + X_2 + X_3 = \overline{\bar{X}_1 \cdot \bar{X}_2 \cdot \bar{X}_3}$$

Some complications arise, however, for package GMOα. We need a function of the general form

$$(X_1 + X_2 + X_3) \cdot T \cdot (Y_1 + Y_2) = \overline{\bar{X}_1 \cdot \bar{X}_2 \cdot \bar{X}_3 + \bar{T} + \bar{Y}_1 \cdot \bar{Y}_2}$$

Hence we form $\bar{X}_1 \cdot \bar{X}_2 \cdot \bar{X}_3 + \bar{T} + \bar{Y}_1 \cdot \bar{Y}_2$ by means of our package format and invert the output to obtain the desired result (see Exercises *g* through *n*).

The Unit-time-interval, Sector, and Minor-cycle Counters (see Fig. 18-16). These are all parallel counters. The unit-time-interval counter is started by means of the drum synchronizing signal. The sector counter is not a single counter but is composed of two cascaded parallel counters, one of 4 bits and one of 3. This arrangement was necessary because each package has been allowed no more than five inputs to any *and* gate. The minor-cycle counter is stepped at T_0 by T_{22} delayed, except when it must be reset. The minor-cycle counter is reset to 1, not to 0 as are the other counters (see Exercises *o* through *w*).

EXERCISES

(*a*) Derive the expression for $\alpha'i$ from 1CS, JCL, αi, and its own recirculation. Remember that 1CS and JCL are mutually exclusive.

(*b*) Why are three gates needed to construct α^*i?

(*c*) Where is α^*10 used? Where is α^*3 used?

(*d*) Why is it necessary that STC inhibit only gate 1 of the αi in the instruction register, whereas it inhibits both gates 1 and 2 of the $\alpha'i$ in the current-address counter?

(*e*) As the instruction register is being loaded with (α), the instruction decoder generates a series of miscellaneous signals. Why is it that these generally erroneous miscellaneous signals do no harm? When do they become the correct signals?

(*f*) Redesign the current-address counter as a parallel counter (see Sec. 15-2).

(*g*) When should (α) be read into the arithmetic unit?

(*h*) When should (α) be read into the instruction register? Into the buffer?

(*i*) Why is no provision made in package GMOα for reading (α) into the arithmetic unit, the instruction register, or the buffer?

(*j*) Why is it necessary to invert only the output of GMOα?

(*k*) Derive the specific Boolean equation for (α) from the logical design drawing for GMOα.

(*l*) Why are three levels of gating necessary to form COI?

(*m*) Why should COI occur only at T_0?

(*n*) How is the sign of B read into the memory during the *transfer* instruction (i.e., through what gate of what package)?

(*o*) Does UTS come on at T_0? How do you explain this in the light of the fact that when DSS (the drum-synchronizing signal) comes on for the first time the count should start at T_0?

(*p*) Why does the package that forms T_{1-19} have T_{20} to inhibit the recirculation?

(*q*) Why need only signals T_0, T_1, T_{18}, T_{19}, T_{20}, T_{21}, and T_{22} be formed?

(*r*) Why does the sector counter not have a clearing signal?

(*s*) Why is the sector counter stepped by a delayed T_{22} rather than a T_0?

(*t*) How does the minor-cycle counter behave during phase IF2?

(*u*) How does the minor-cycle counter behave *before* UTS comes on?

(*v*) Describe the operation of package UTSG. (HINT: See Sec. 17-8.)

(*w*) Why is the minor-cycle counter reset to 1 at T_0 of every minor cycle except during phase 2 or 3?

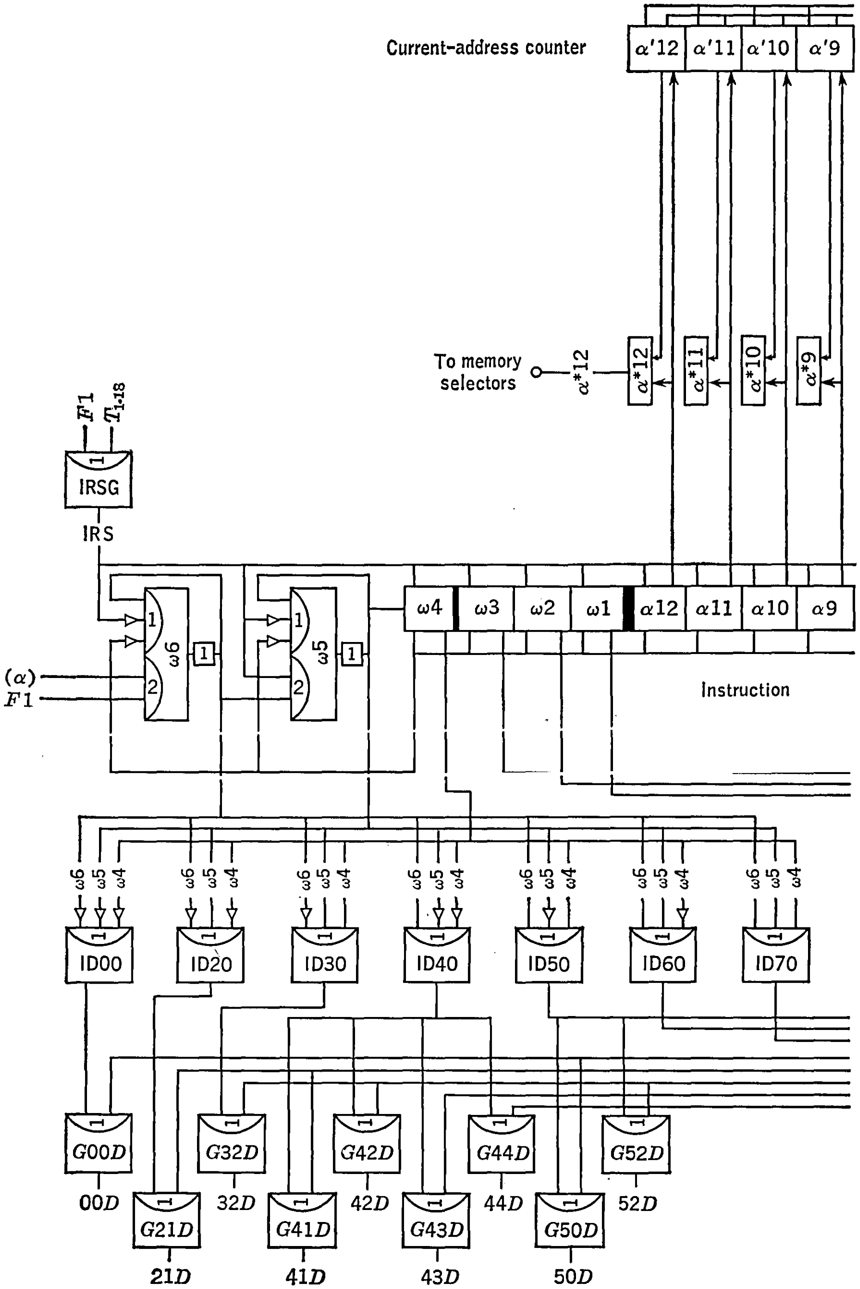
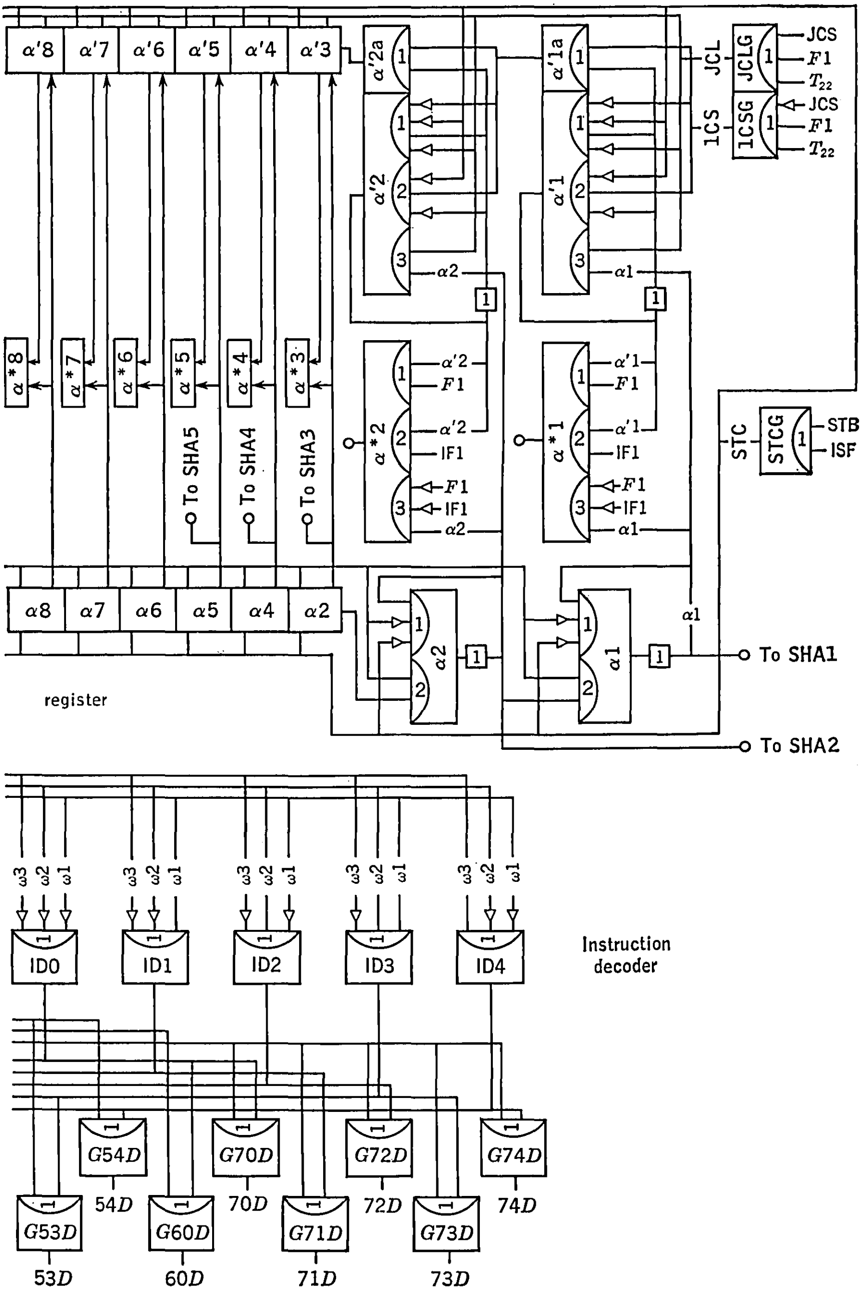


FIG. 18-14. Control unit: current-address counter,



instruction register, and instruction decoder.

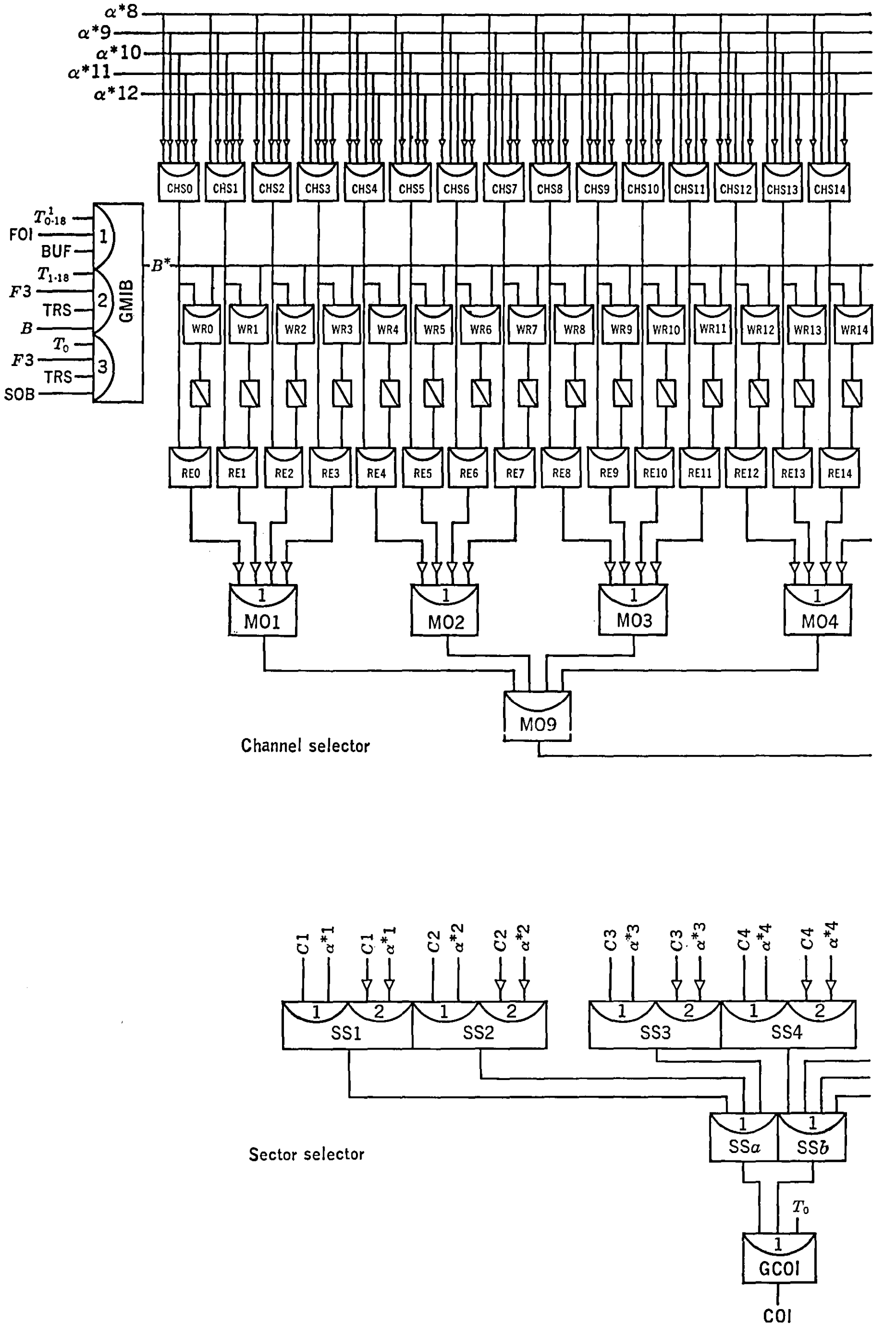
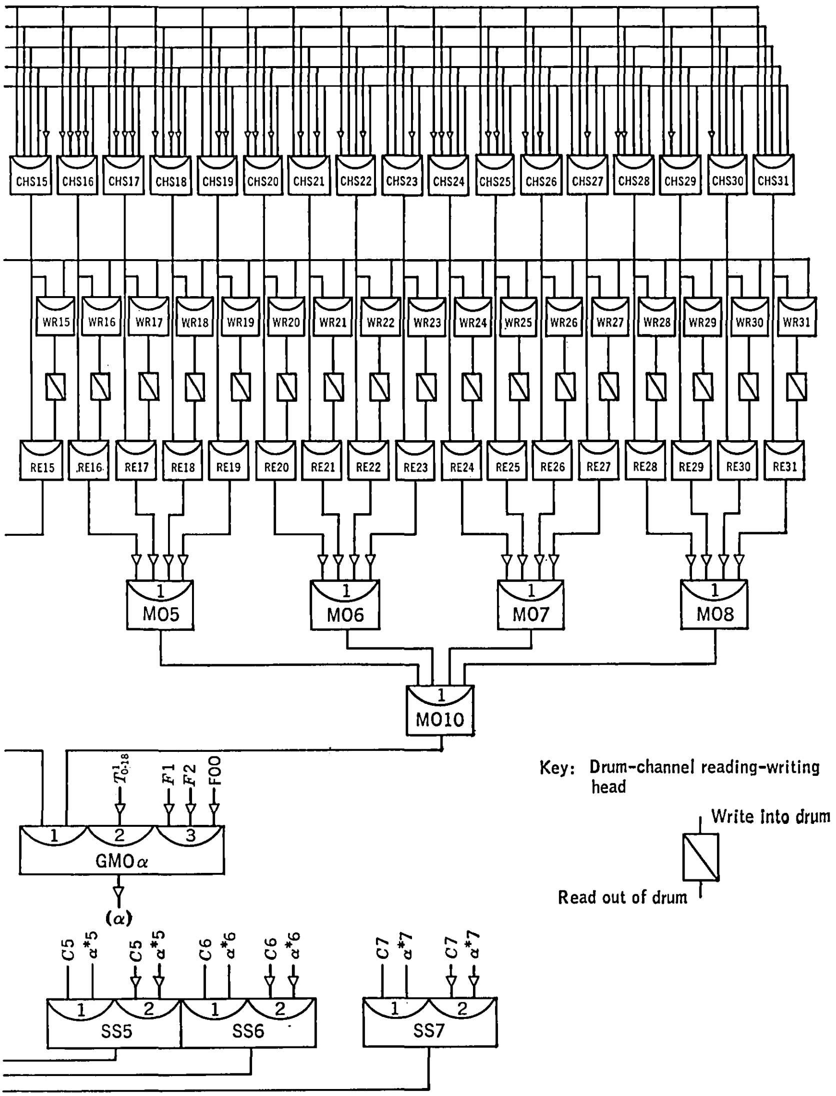
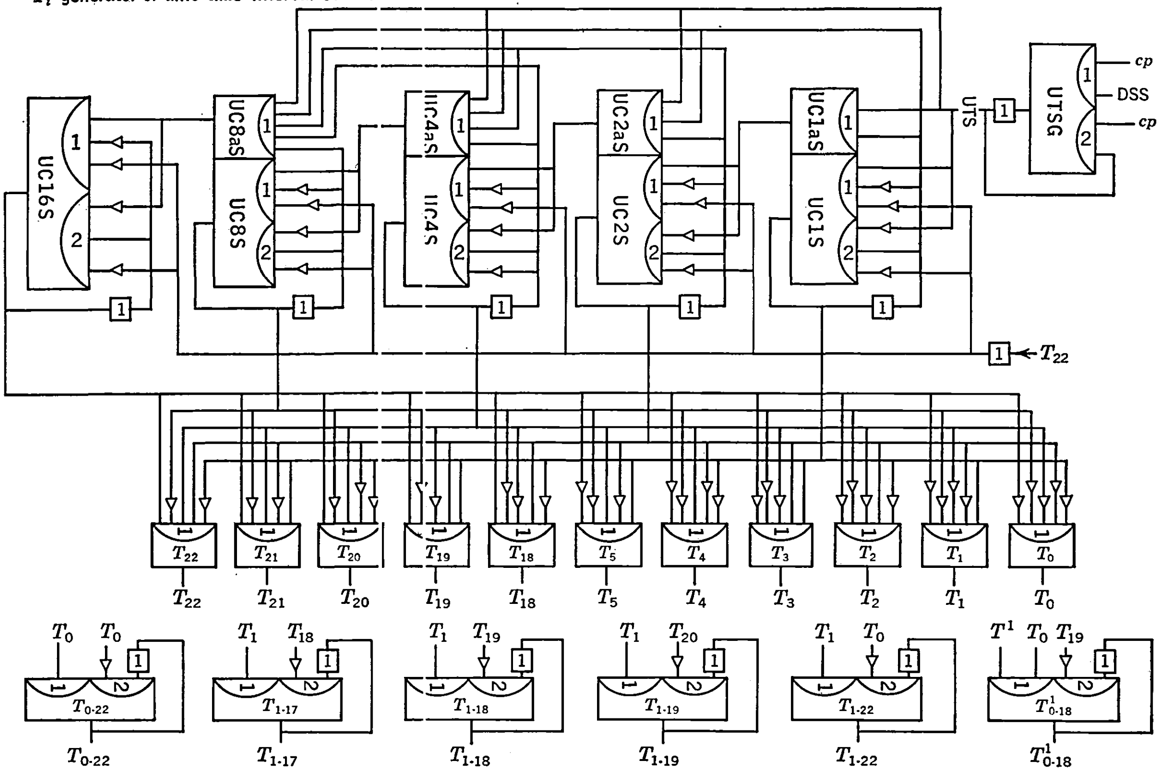


FIG. 18-15. Control unit: channel


CHS15 CHS16 CHS17 CHS18 CHS19 CHS20 CHS21 CHS22 CHS23 CHS24 CHS25 CHS26 CHS27 CHS28 CHS29 CHS30 CHS31
WR15 WR16 WR17 WR18 WR19 WR20 WR21 WR22 WR23 WR24 WR25 WR26 WR27 WR28 WR29 WR30 WR31
RE15 RE16 RE17 RE18 RE19 RE20 RE21 RE22 RE23 RE24 RE25 RE26 RE27 RE28 RE29 RE30 RE31
1
MO5
1
MO6
1
MO7
1
MO8
1
MO10
$T^1_{0:18}$
$F1$
$F2$
FOO
1
2
3
GMOα
(α)
C5
α*5
C5
α*5
C6
α*6
C6
α*6
C7
α*7
C7
α*7
1
2
SS5
1
2
SS6
1
2
SS7
Key: Drum-channel reading-writing head
Write into drum
Read out of drum

selector and sector selector.

T_i generator or unit-time-interval counte'



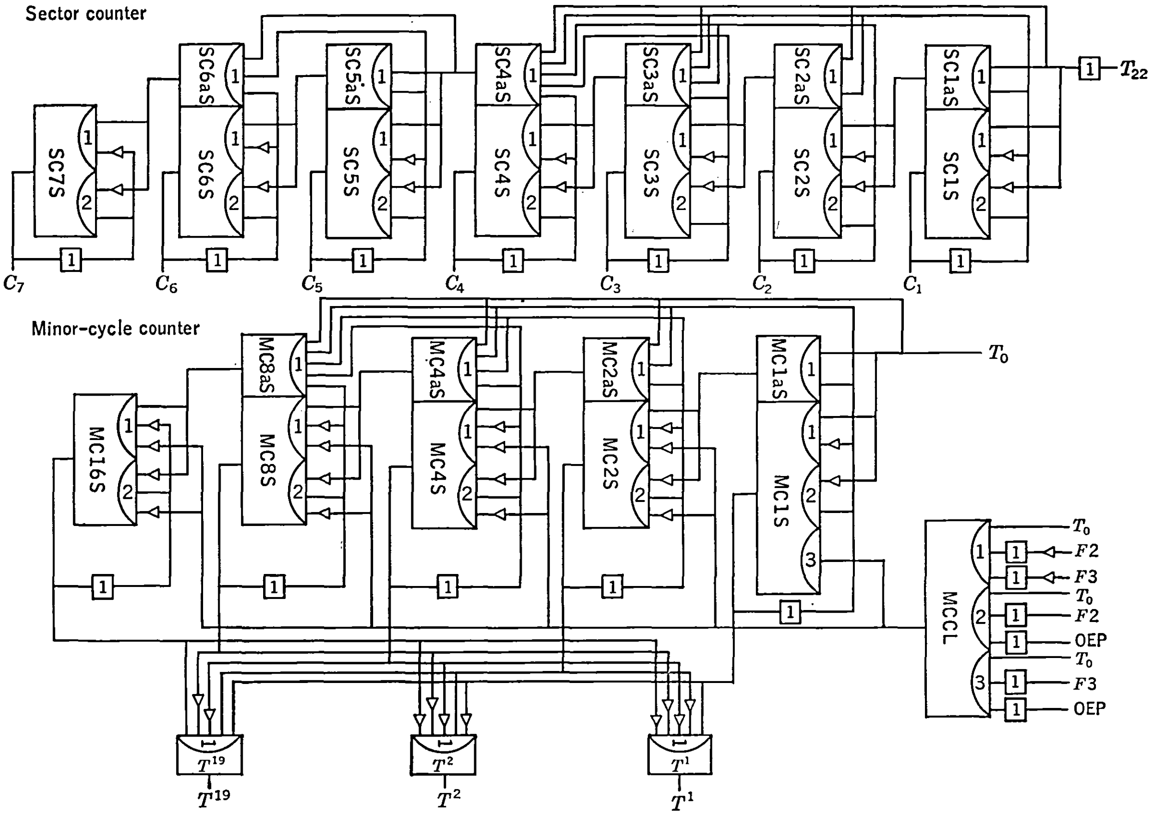


FIG. 18-16. Control unit: unit-time-interval counter, sector counter, and minor-cycle counter.

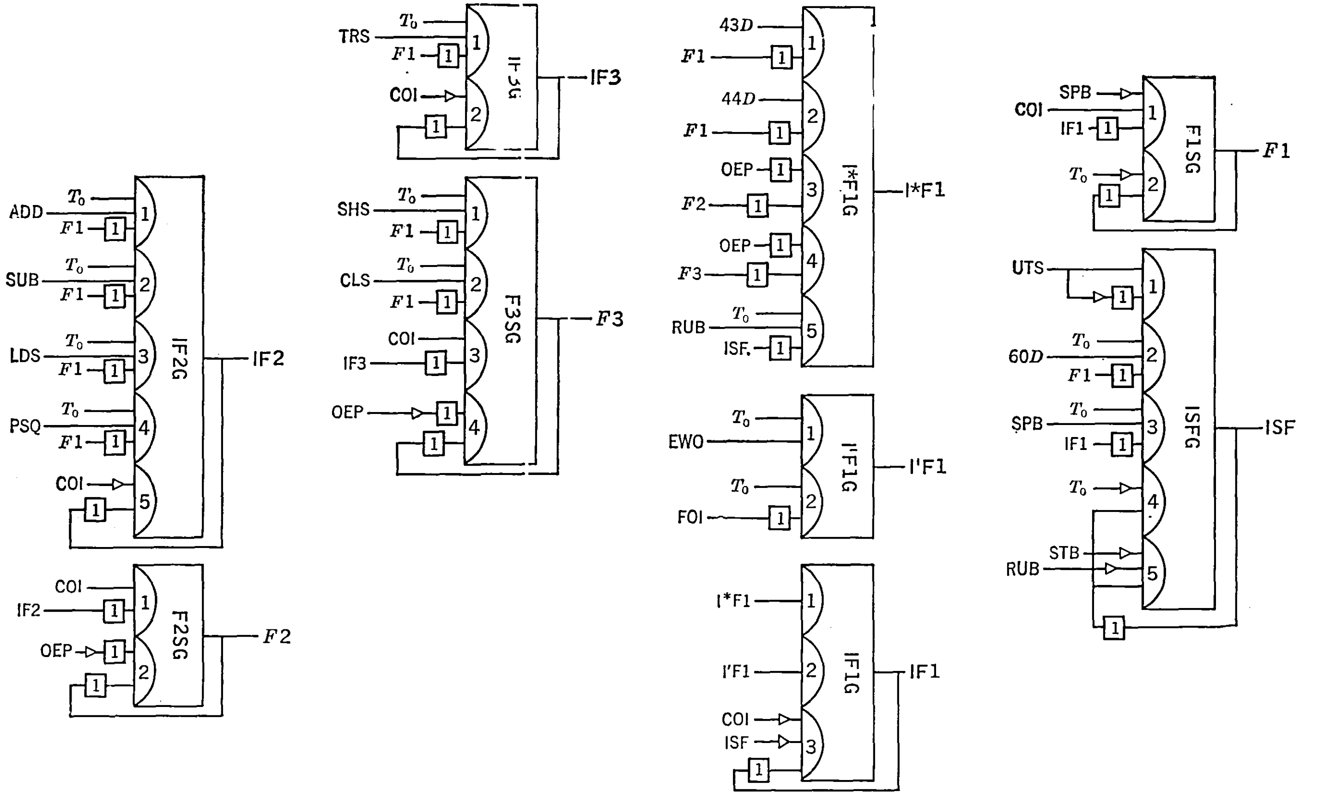


FIG. 18-17. Logica. design of the operations phase generator.

18-8. Phase Generater, Buffer, and Push Buttons

The *vertical description* of the *operations phase generator* (see Fig. 18-17, Exercises *a* through *p*) is as follows:

Phase 1 signal generator F1SG
Gate 1 initiates $F1$ after phase IF1 unless the stop button has been pushed.
Gate 2 holds $F1$ until the next minor cycle begins.

Idling phase IF2 signal generator IF2G
Gate 1 initiates IF2 for the *add* instruction.
Gate 2 initiates IF2 for the *subtract* instruction.
Gate 3 initiates IF2 for all logical instructions.
Gate 4 initiates IF2 for the *multiply* and *divide* instructions.
Gate 5 holds IF2 until the proper sector has been selected.

Phase 2 signal generator F2SG
Gate 1 initiates $F2$ after IF2.
Gate 2 holds $F2$ from $T_0{}^1$ until the end of the operation.

Idling phase IF3 signal generator IF3G
Gate 1 initiates 1F3 for the *transfer* instruction.
Gate 2 holds IF3 until proper sector is selected.

Phase 3 signal generator F3SG
Gate 1 initiates $F3$ for the *shift* instruction.
Gate 2 initiates $F3$ for the *clear* instruction.
Gate 3 initiates $F3$ after IF3, for the *transfer* instruction.
Gate 4 holds $F3$ from $T_0{}^1$ until the end of the operation.

Phase I*F1 auxiliary signal generator I*F1G
Gates 1 and 2 initiate IF1 after a *jump* instruction.
Gate 3 initiates IF1 after a phase F2 instruction.
Gate 4 initiates IF1 after a phase F3 instruction.
Gate 5 initiates IF1 for the run button.

Phase I′F1 auxiliary signal generator I′F1G
Gate 1 initiates IF1 after a word has been read into the output unit from the buffer.
Gate 2 initiates IF1 after a word has been read into the drum from the buffer.

Idling phase signal IF1 generator IF1G
Gates 1 and 2 initiate IF1 from I*F1 or I′F1.
Gate 3 holds IF1 until the proper sector has been selected or the stop button has been pushed.

Stopped idling phase signal ISF generator ISFG
Gate 1 initiates ISF at the first T_1 pulse formed after the power is turned on.

Gate 2 initiates ISF for a *stop* instruction.

Gate 3 initiates ISF when the stop button is pushed.

Gates 4 and 5 stop ISF at the next T_0 after the run button or start button is pushed.

The *vertical description* of the *in-out phase generator* (see Fig. 18-18, Exercises q through w) is as follows:

Output idling phase signal IFO generator IFOG

Gate 1 initiates IFO.

Gate 2 holds IFO until the proper sector has been selected.

Output operating phase signal FOO generator FOOG

Gate 1 initiates FOO after IFO.

Gate 2 holds IFO on for one minor cycle.

Output buffer phase signal FBO generator FBOG

Gate 1 initiates FBO after FOO.

Gates 2 and 3 hold FBO until T_0 of the first minor cycle after the complete word has been read out of the buffer.

Input buffer phase signal FBI generator FBIG

Gate 1 initiates FBI after phase 1 for a *read-in* (that is, 00) instruction.

Gate 2 initiates FBI at T_0 of the first minor cycle after the start button has been pushed after ISF.

Gates 3 and 4 hold FBI until T_0 of the first minor cycle after the word has been completely read into buffer.

Input idling phase signal IFI generator IFIG

Gate 1 initiates IFI after FBI.

Gate 2 keeps IFI on until proper sector is selected.

Input operating phase signal FOI generator FOIG

Gate 1 initiates FOI after IFI.

Gate 2 holds FOI for one minor cycle.

The Start, Run, and Stop Buttons (see Fig. 18-18, Exercises x through aa). The start button is used in initiating a new program on the computer. After the stop button has been pushed or a *stop* instruction has been executed, the start button breaks the sequence of instructions that were being computed. The start button will have effect only if pushed when the computer is in the stopped idling phase ISF. When it is pushed during ISF, the computer reads one word from the input unit into address 0000 and then takes (0000) as its next instruction. If the word read in is a *jump* instruction, the computer can be made to jump to any desired memory location and proceed with a desired program. The advantage of such a procedure, besides its being used for the initial read-in, is that it can be used for debugging purposes, as a manual method of jumping to different programs, and so forth.

Pushing the start button initiates signal STB′ in the start-button synchronizer of the control panel. If the button is pushed during phase ISF, STB′ initiates STB in package STBG immediately (except not at the beginning of a minor cycle). Signal STB in turn initiates STC (package STCG of Fig. 18-14), which clears the current-address counter and the instruction register. At the beginning of the next subsequent minor cycle STB initiates FBI (package FBIG) and allows ISF to be killed (package ISFG). Then at T_1, FBI (delayed) kills STB. Now, since the instruction is all zeros, the instruction operation will appear as 00 and the computer will proceed to read one word into 0000 during the successive phases IFI and FOI. Next follows phase IF1, and since the current-address register is all zeros, (0000) will be taken as the next instruction, to be executed as usual. If the start button is pushed when the computer is not in ISF, it will have no effect.

The stop button forces the computer into the stopped idling phase. That phase will be initiated only during or after phase IF1, bypassing phase $F1$. The reason for this will be made clear in the discussion of the run button.

The run button will have effect only if the computer is in the stopped idling phase. If the run button is pushed then, the computer will proceed with the computations from the point where it left off, taking the next instruction in proper sequence. If the stopped idling phase was initiated by a *stop instruction*, the current-address counter contains the address of the next instruction, for phase 1 had been completed. When the run button is pushed, this address will be taken as that of the next instruction. If the idling phase was entered via the stop button, the computer had gone from phase IF1 directly to the idling phase and had not brought the next instruction into the instruction register or increased the current-address counter, since phase 1 had been bypassed. Then, when the run button is pushed, the computer will read into the instruction register the instruction it would have read if the stop button had not been pushed.

The *vertical description* of the *push buttons and in-out buffer* (see Fig. 18-18, Exercises *bb* through *jj*) is as follows:

Start-button signal generator STBG

Gate 1 initiates STB with signal STB′ from the start-button synchronizer only when the computer is stopped.

Gate 2 holds STB on until FBI comes on.

Stop-button signal generator SPBG

Gate 1 initiates SPB with signal SPB′ from the stop-button synchronizer unless the computer is already stopped.

Gate 2 holds SPB until ISF comes on.

Run-button signal generator RUBG

Gate 1 initiates RUB with signal RUB′ from the run-button synchronizer only when the computer is stopped.

Gate 2 holds RUB until IF1 comes on.

End of write-out signal EWO generator EWOG
- Gate 1 initiates EWO when the buffer has been completely read into output unit.
- Gate 2 holds EWO until IF1 has been initiated at the next T_0.

End of write-in signal EWI generator EWIG
- Gate 1 initiates EWI when the buffer has been completely loaded from the input unit.
- Gate 2 holds EWI until IF1 has been initiated at the next T_0.

Read into buffer from input unit RIBG
- Gate 1 tells the input unit to start shifting (pulse RIB initiates SIU).

Read out of buffer to output unit ROBG
- Gate 1 tells the output unit to start shifting (pulse ROB initiates SOU).

First package of buffer BU1
- Gate 1 lets the contents of the input unit (signal IEQ) enter the buffer during phase FBI.
- Gate 2 lets (α) enter the buffer during phase FOO.
- Gate 3 holds the bit until shifted.

Buffer-shift pulse generator BSPG
- Gates 1 and 4 produce shifts synchronous with the input and output units.
- Gates 2 and 3 produce shifts synchronous with the computer clock.

Synchronization. The Pedagac is so designed that signals SPB′, RUB′, and STB′ can be of indefinite length. Hence the push-button synchronizers may be merely type *A* packages in which the outputs of the actual buttons are gated with clock pulses. In order that no error result from pushing two buttons simultaneously, we have SPB′ inhibit RUB′ and STB′, and we have RUB′ inhibit STB′. In addition we have STB inhibit RUB′, in case the start button were pushed immediately before the run button.

Synchronizing the input and output units of the Pedagac is equally simple. The input unit, for example, will put out a signal indicating when it is ready to deliver the next bit of a word. That signal initiates SIU in a type *A* package; SIU is then killed by the inversion of that signal, delayed by one unit time interval [cf. gate 1 of package ISFG (Fig. 18-17)]. Signal SOU is similarly generated by the output equipment. The end-of-shift signals for the buffer, EWI′ and EWO′, are also synchronized in this manner, but with *two* units of delay (why?). This method assumes that the signals of the input and output unit have been processed to eliminate intermittencies, such as might result from relay chatter.

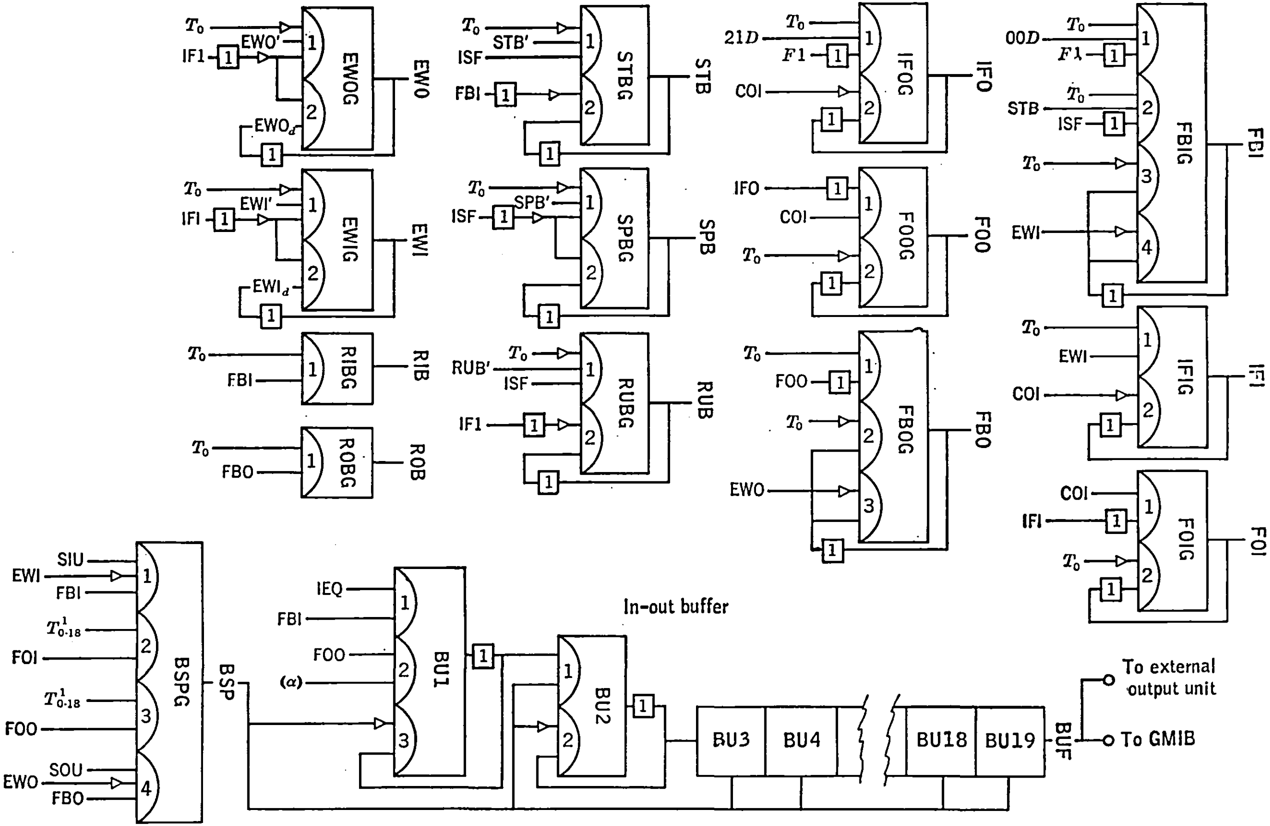


FIG. 18-18. Logical diagram of the push-button signal generators, in-out phase generator, and buffer.

EXERCISES

(*a*) How does gate 2 of package F1SG keep $F1$ on during Γ^1_{0-22}?

(*b*) What is the purpose of having the stop-button signal SPB inhibit the initiation of phase 1?

(*c*) In gates 1 to 4 of package IF2G why are T_0 and the unit delay on $F1$ both necessary?

(*d*) In package IF2G at what unit time interval will IF2 go off, and why?

(*e*) In gate 2 of package F2SG, why are the unit delays necessary?

(*f*) Both IF2 and IF3 perform the same function, namely, selecting the drum sector. Are they both necessary? (HINT: How would $F2$ and $F3$ be initiated if IF2 and IF3 were not available? Remember that only five gates are allowed to each package.)

(*g*) Why should the run button initiate phase IF1 in package I*F1G? Why must both ISF and RUB act on gate 5 of package I*F1G?

(*h*) Why is an auxiliary I*F1G package necessary?

(*i*) Show that I′F1 can come on only once after phase FBO.

(*j*) Why is $F1$ initiated in package F1SG from two sources?

(*k*) What are all the functions of the signal ISF generated by package ISFG (that is, where is this signal used, and why is it necessary)?

(*l*) Why are both button conditions for stopping ISF necessary (see gate 3 of package ISFG)?

(*m*) How does the minor-cycle counter behave when ISF is on?

(*n*) Show that gate 1 of ISFG turns ISF on *only* when UTS comes on for the first time. Why is it necessary to resort to an unconventional technique in this case?

(*o*) Why is it necessary to have the computer idling in phase ISF before the start button is pushed for the first time?

(*p*) Suppose that the start button is pushed when the computer is *not* in phase ISF; will the computer go into phase FBI and read one word into 0000? Why?

(*q*) What must be done on the control panel before the start button is pushed if it is desired to read in one word into 0000 during the course of debugging a program?

(*r*) Why is T_0 necessary in gate 1 of package IFOG?

(*s*) Why can FOO be inhibited simply by a T_0? (See gate 2, package FOOG.)

(*t*) Why can FBI be initiated after either phase 1 or phase ISF? (See gates 1 and 2 of package FBIG.)

(*u*) In gate 1 of package IFIG why did we *not* use a delayed FBI with T_0?

(*v*) How does the computer know that a word is to be read into the drum from the buffer during FOI, but *from* the drum into the buffer during FOO?

(*w*) Since both phases IFO and IFI perform the same function, namely, selecting a drum sector, why are they *both* necessary? (HINT: If both IFO and IFI were not available, how would FOO and FOI be initiated? Remember also the start button.)

(*x*) Describe all ways for entering idling phase ISF.

(*y*) Describe all ways for going out of idling phase ISF.

(*z*) When the stop button is pushed, why is it advantageous for the computer first to complete one minor cycle of phase IF1 and then to change *directly* to the idling phase, bypassing phase $F1$? (HINT: Remember the operation of the run button and that the computer must pass through at least one minor cycle of phase IF1 during every instruction-execution cycle.)

(*aa*) Trace on the diagrams the actions of the start, run, and stop buttons.

(*bb*) Why must STB (start-button signal) be turned off as FBI comes on? (HINT: Suppose that STB was left on; what would happen? See packages ISFG and FBIG.)

(*cc*) Why must gate 1 of STBG have a $\bar{T}_0$ input? (HINT: See the instruction shift register, and note that an error would result if STB′ were pushed at T_0. Why?)

(*dd*) Are the $\bar{T}_0$ inputs to gate 1 of packages SPBG, RUBG, EWOG, and EWIG necessary? Why?

(*ee*) Why must the ISF input to package SPBG be delayed?

(*ff*) What is the purpose of the EWI and EWO inhibitions on gates 1 and 4 of package BSPG?

(*gg*) Are the T^1_{0-18} inputs of gates 2 and 3 of package BSPG strictly necessary? Why?

(*hh*) Why must EWI and EWO wait, respectively, for IF1 and IFI to come on before they are turned off?

(*ii*) What is the purpose of package RIBG, and of package ROBG?

(*jj*) In package BU1 of the buffer why is it not necessary that the shift pulse be included as an input into gates 1 and 2, as it is into gate 1 of BU2?

18-9. Additional Topics

a. Redesign the Pedagac, using delay-line registers (including the current-address register and the instruction register).

b. Modify the design of the Pedagac to include other instructions such as the *replace* arithmetic operations and to admit relative addressing (see Chap. 4).

c. Design the Pedagac as a floating-point computer.

d. Redesign the Pedagac with a parallel arithmetic unit and parallel rapid multiplication and division.

e. Design the special-purpose business and logistics computer described in Sec. 8-5.

f. Incorporate the comparator unit, control computer unit (see Sec. 15-7, Additional Topics *c* and *d*), business unit, and general-purpose Pedagac into a unified system as described in Sec. 8-6.

PART 5

ELECTRONIC DESIGN OF DIGITAL CIRCUITS

CHAPTER 19

PROBLEMS AND LIMITATIONS IN ELECTRONIC REALIZATION

19-1. Introduction

In Parts 3 and 4 we considered digital circuits to be abstract "black boxes" with certain logical properties; we abstractly represented digital information by means of the symbols 0 and 1. Assuming the availability of such digital circuitry, we showed how a digital computer could be logically designed by appropriately connecting such circuits. In this part of the book we shall consider the problems involved inside our black boxes, problems of the electronic design of the digital circuits themselves.

Electronic signals are the physical realization of information processed in a digital electronic computer. In this part of the book we shall be concerned primarily with the electronic processing of these signals. The present chapter is concerned with the effect of the actual electronic circuits themselves on our building-block organizational concept. No longer can we neglect signal delay through amplifiers, or package-coupling problems, or absolute package reliability. The results of this present chapter are tacitly assumed in discussing the general considerations of circuits using semiconductors (diode and transistor) elements, in Chap. 20, and of circuits using magnetic elements, in Chap. 21. Memory methods and input-output techniques present many special problems, some of which will be considered in Chap. 22. Finally in Chap. 23 we shall return to the design of the Pedagac; and, specifying the electronic design of a particular type of package, we shall consider the nontrivial wiring diagrams, giving the actual pin connections between packages.

It should be emphasized, however, that almost every topic covered in this part of the book is a full-fledged and rapidly growing field in its own right. We can expect to do no more than present some of the basic principles involved. As with the previous parts of the book, the method of exposition of principles is by specific examples selected for that purpose. The extensive bibliographies given in the Additional Topics attest to the fact that only introductory material can be covered in such a short treatise as this.

In the present chapter various systems for the electronic realization of 0, 1 are first described. The next general problem in the physical realization of digital circuits is the effect of finite delay during amplification, which demands an introduction of the concept of clock phases. Finally, the actual use of circuits brings up the problem of reliability, which although not directly influencing general circuit considerations falls naturally into this chapter.

19-2. Types of Digital Gating Systems

Electronic Functions of Packages. In general the package must perform four functions: (1) gating, (2) amplification and reshaping, (3) retiming (i.e., resynchronization), and (4) storage. The gating must be accomplished for the logical processing of the signals. This logical processing requires the signals to pass from package to package, through various circuit components such as resistors, transistors, diodes, capacitors, and inductors, which cause the signals to become somewhat attenuated and distorted (see Exercise *a*). In order that the signals may not become so distorted as to be undetectable by a following package, they must be amplified and reshaped. For proper operation of the logical processing all signals entering the same gate must be synchronized, even though they may have passed through different sets of components (involving different numbers of gates) on the way to this gate. Finally, delaying or storing the signal is often an important part of the logical processing.

We would not leave the reader with the idea that all these four features must be included in every package, for frequently these functions are distributed over sets of specialized packages. As an illustration, it sometimes happens that signals pass through a special amplification and reshaping package after having passed through several packages that perform only the gating and synchronizing functions; there are of course many possible variations. On the other hand some provision must always be made for the accomplishment of each of the four functions.

We shall consider in detail three methods for performing these functions: the dynamic, or regeneration, method; the static, or flip-flop, method; and the phase, or parametric, method. The general principles involved will be presented by means of examples, but the practical implementation of these methods in computer technology is by no means limited to these illustrations.

Dynamic, or Regeneration, Method. In this method the unit signal representation is in the form of a (synchronized) square voltage pulse, the zero in the form of no pulse. Amplification of the pulse is accomplished by standard circuitry, but the pulses are reshaped and retimed by means of logical gating. As an illustration, consider Fig. 19-1*a*, where the pulse representations of 0 and 1 are shown. A typical package is shown in Fig. 19-1*b*; specific circuits for the gates and amplification will be considered in detail in the next chapters. Because of the clock pulse at every input *and* gate the transmission of a pulse from one package to the next occurs only as the clock pulse is on. The clock pulse itself, always correctly shaped and timed in a special clock oscillator, is sent directly to each input *and* gate. The leading edge of the clock pulse can therefore be used to reshape the leading edge of an attenuated input unit pulse, as illustrated in Fig. 19-1*c*. The input signal must enter the *and* gate ahead of the clock pulse, so that the resulting waveform at point *A* of the circuit will be as shown. The trailing edge of the input pulse still remains to be shaped. To do this, the amplified pulse is recirculated through a special two-input *and* gate with a clock pulse, but *without a delay* (other than that of the amplifier itself). The recirculation through this *and* gate will hold the output at the unit level until the clock pulse goes off. Then the trailing edge of the clock pulse will shape the trailing edge of the signal pulse. The output signal with and without such *regeneration* reshaping is shown in Fig. 19-1*c* (the offset of the

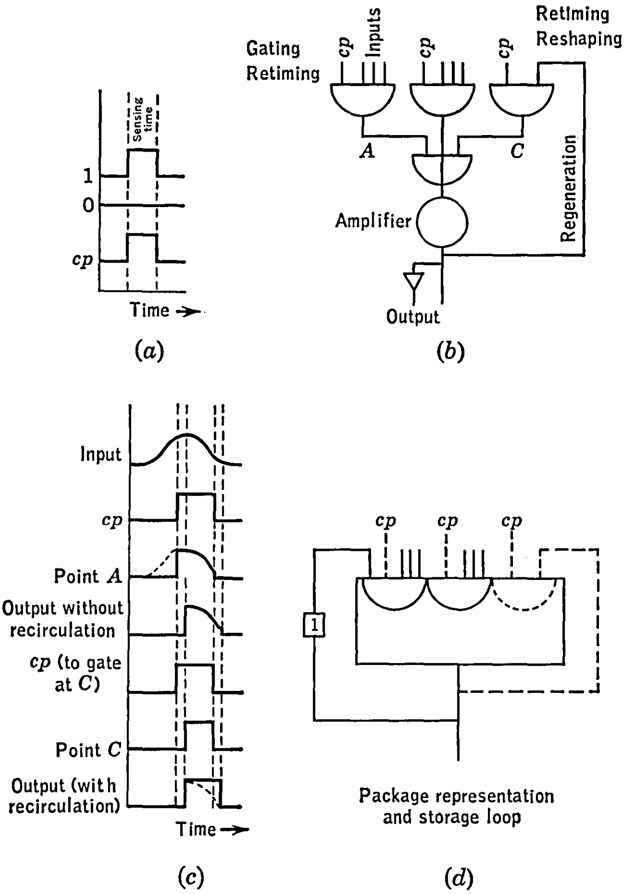


FIG. 19-1. Dynamic, or regenerative, method.

output signal to the right in the figure indicates the effect of delay inherent in the amplifier). Note that the circuit of Fig. 19-1*b* is of the type used in the package of the Pedagac; in our logical drawings we have always omitted the clock pulses and the recirculation *and* gate (cf. Fig. 19-1*d*).

This scheme has resulted in a package that gates and amplifies the signals, and reshapes and retimes them by using clock pulses to determine the leading and trailing edges of the signals. For storage, however, we must connect to the package a special delay line, as shown in Fig. 19-1*d*. Frequently special packages are made containing only such delay lines; these are then wired in as required by the logical processing diagrams. A variation of this technique is an *or-and-or* gating arrangement, rather than just the *and-or* arrangement. The inputs to the circuit would be to the first level of *or* gates; the clock pulses and leading-edge shaping would occur in the next level of *and* gates, with the recirculation accomplished as already described.

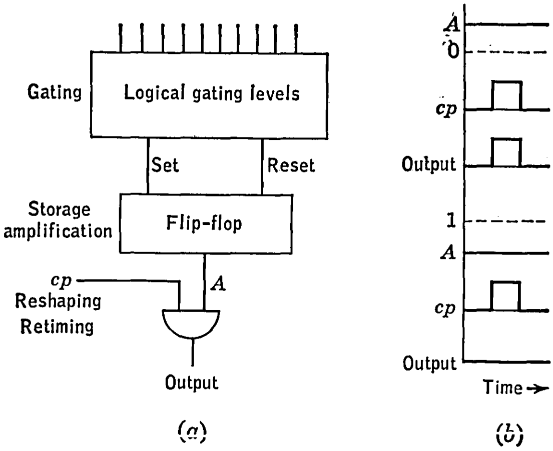


FIG. 19-2. Static, or flip-flop, method.

Static, or Flip-Flop, Method. In this system the electronic signals pass through a gating network into a flip-flop. The unit or zero is recorded as the state of the flip-flop. The flip-flops usually have two stable voltage levels, one representing a unit, the other a zero. The voltage level held in the flip-flop is gated out by a clock pulse to become the information pulse signal for the gates of the next package. The 0 and 1 signal representations are voltage pulses in the gates but are constant voltage levels in the flip-flops. Figure 19-2*a* represents the gating and flip-flop combination, with the *and* gate that combines flip-flop output and clock pulse. Amplification is virtually accomplished in the flip-flop; retiming and reshaping are accomplished by the clock pulse in the output *and* gate (see Fig. 19-2*b*). The flip-flop is of course capable of storage as well.

A standard gating arrangement and flip-flop combination can be incorporated into a single package. However, it is frequently more convenient to have separate flip-flop and gating packages. In either case there must be a limit on the number of gating levels or packages through which a signal can be sent before a flip-flop is used. Sometimes the gating packages themselves contain provision for amplification, or separate amplification packages are included, to increase the number of gating packages possible between flip-flop packages. However, the flip-flops would still accomplish the retiming, storage, and reshaping functions. We consider specific gating and flip-flop circuits in the next chapters.

Phase, or Parametric, Method. This system is entirely different from the two mentioned above. We include it here to indicate the diversity of conceivable methods; the principles involved also have important application to digital microwave circuitry (see Sec. 21-5). The 0, 1 information is represented in this case by the *phase* of a signal sine wave with respect to a standard "clocking" sine wave of double the frequency. (Compare the above methods, which represent information by the *amplitude* of a signal.) The technique is based on the phenomenon of *parametric amplification*, which we now describe.

We introduce the concept of a parametric amplifier by means of the rough mechanical analogy shown in Fig. 19-3*a*. As the bob of the pendulum swings past the lowest point of its path, with its greatest velocity, the cord is pulled, reducing the effective length l of the pendulum by Δl. Since angular momentum is conserved, the linear velocity of the weight increases by the ratio $l/(l - \Delta l)$ and the swing is amplified. At the top of the ascending arc the cord is released, allowing the arm to return to its original length l, and the procedure is repeated on the return arc. *Thus the frequency of the exciting pull on the cord is twice the frequency f of swing of the pendulum* (see Fig. 19-3*b*). The energy absorbed by the pendulum as its swing is amplified is the difference between the work required to pull the cord against the combination of the weight of the bob and its centrifugal force F at the low point of its swing and the work returned as the bob drops at the top of its swing *without centrifugal reaction.* The instantaneous input power is $P = F\,dl/dt$, and the work absorbed per cycle is $\int_0^{1/f} P\,dt$. Thus for a small sinusoidal change of length of frequency $2f = 2(\omega/2\pi)$, we have $l = l_0 - \Delta l \sin 2(\omega t + \alpha)$, with α giving the phase relation. Assuming a small swing of the pendulum (of angular frequency $\omega = 2\pi f$), and noting that the centrifugal force F will be proportional to the square of the velocity of the swing, we have approximately $F = K \sin^2 \omega t = \frac{1}{2}K(1 - \cos 2\omega t)$. Thus the work per cycle is

$$\Delta W = \int_{\omega t=0}^{\omega t=2\pi} P\,dt = \int_0^{2\pi/\omega} \tfrac{1}{2}K(1 - \cos 2\omega t)[-2\omega\ \Delta l \cos 2(\omega t + \alpha)]\,dt$$
$$= \pi k\ \Delta l \cos 2\alpha$$

which has its maximum positive value when $\alpha = 0$ or π, which is when the pull has its maximum velocity at the low point of the pendulum swing (i.e., at times $t_1, t_3, \ldots$ of Fig. 19-3*b*). Note that, if the pendulum was initially at rest, no horizontal motion would be produced by pulling the cord: there must be an initial swing to the pendulum in order that amplification take place. The initial phase α of this motion should be 0 or π for maximum amplification; *clearly the amplified motion will have the same phase as the initial motion.*

We can now understand the parametric amplifier, illustrated in Fig. 19-3*c*, in terms of the mechanical analog. The coils on cores A and B are wound in *opposite* directions, an exciting current-source wire threads the two cores, and a bias keeps the cores near the knee, or bend, of the hysteresis Φ-I curve (see Fig. 19-3*d*). The exciting wire is driven with a current source I_d of frequency $2f$, where f is the resonant frequency of the main LC circuit. As we have seen in the mechanical analog, there must be some small oscillation at the start. At time t_1 the small main-circuit current I_c will increase the flux Φ in one core, say B, and decrease the flux in the other, A (see Fig. 19-3*d*). Thus for the same ΔI_d the increase $\Delta\Phi_A$ will be greater than $\Delta\Phi_B$, the latter being practically zero. The induced voltage in A, namely, $V_A = d\Phi_A/dt$, is in the direction to increase I_c. This is just like the pull up on the bob at the bottom of the swing. At t_2, $I_c = 0$; hence both cores are at the same point on the hysteresis loop and the induced voltages, being opposed, cancel—i.e., like letting out the bob at the top of its swing. At t_3 the cores interchange positions on the hysteresis curve since the current I_c is reversed; since $\Delta\Phi_A$ will be negligible, $V_B = d\Phi_B/dt$, in the reverse direction, will help increase the current I_c. The gain in power (for coils of N turns each) is

$$P = NI_d \frac{d\Phi}{dt} = NI_d \frac{d\Phi}{dI_d}\frac{dI_d}{dt}$$

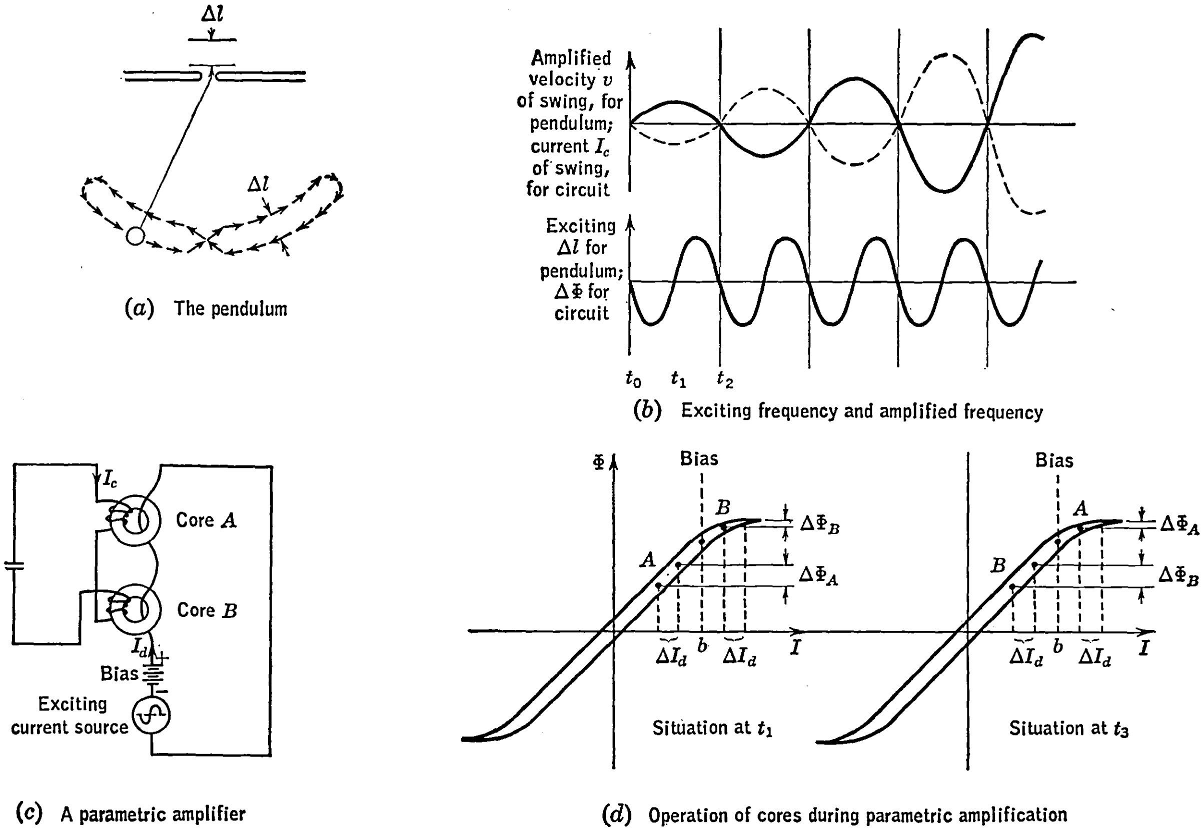
where $\Phi = \Phi_A + \Phi_B$. Hence, since the bias on I_d is large with respect to its total variation, we have for the work input the approximation

$$\Delta W = \int_{\omega t=0}^{\omega t=2\pi} P\,dt = NI_d \int_0^{2\pi/\omega} \frac{d\Phi}{dI_d}\frac{dI_d}{dt}\,dt$$

We have just seen that, with a sinusoidal current I_c of frequency f in the LC circuit, the slope $d\Phi/dI_d = d\Phi_A/dI_d + d\Phi_B/dI_d$ varies with a frequency

$$2f = 2\frac{\omega}{2\pi}$$

Thus, to a first approximation $d\Phi/dI_d = K(1 - \sigma \cos 2\omega t)$, with K and



(*a*) The pendulum

(*b*) Exciting frequency and amplified frequency

(*c*) A parametric amplifier

(*d*) Operation of cores during parametric amplification

FIG. 19-3. The parametric amplifier.

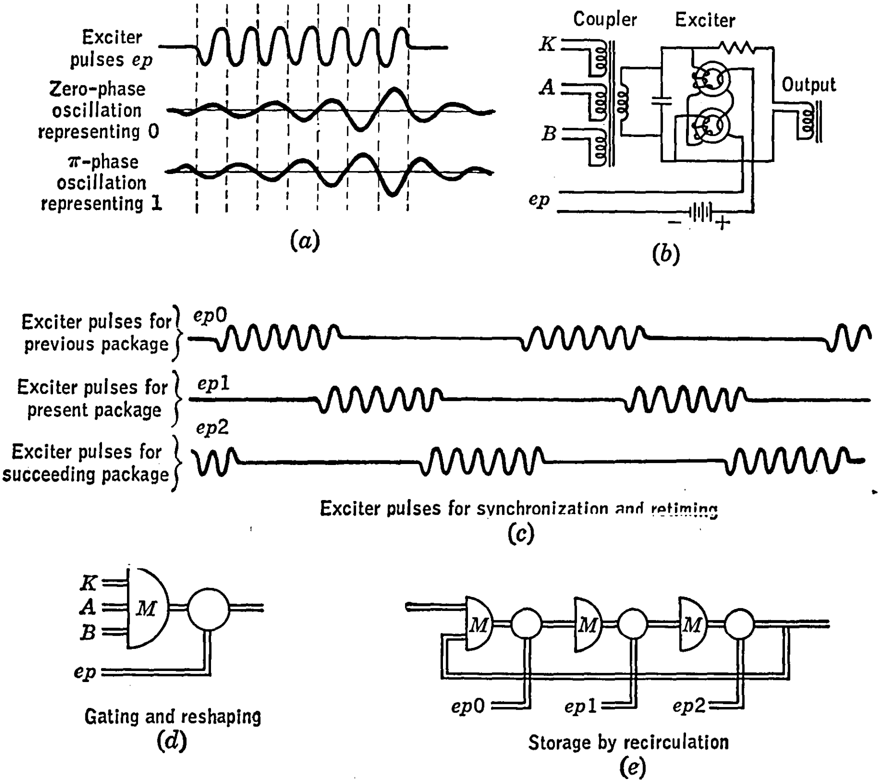
σ appropriate constants. For the current I_d we have

$$I_d = I_b + \Delta I_d \sin 2(\omega t + \alpha)$$

whence

$$\frac{dI_d}{dt} = 2\omega \, \Delta I_d \cos 2(\omega t + \alpha)$$

Thus the integral for ΔW is of the same form as for the pendulum above.


Exciter pulses ep
Zero-phase oscillation representing 0
π-phase oscillation representing 1
Coupler
Exciter
K
A
B
Output
ep
Exciter pulses for previous package
ep0
Exciter pulses for present package
ep1
ep2
Exciter pulses for succeeding package
K
A
B
M
ep
M
M
M
ep0
ep1
ep2

(a)

(b)

Exciter pulses for synchronization and retiming

(c)

Gating and reshaping

(d)

Storage by recirculation

(e)

FIG. 19-4. Phase, or parametric, method.

Summarizing, the parametric amplifier will use energy of a frequency $2f$ to amplify an initial small oscillation of frequency f, and the amplified result will have the same 0 or π phase as the initial small oscillation (see Fig. 19-4*a*). This latter observation is most important, for a 0 or 1 is represented by an oscillation in the 0 or π phase, respectively. The zero or unit information is transmitted into the amplifier by means of weak coupling, as illustrated in Fig. 19-4*b*, to initiate a small oscillation of appropriate phase; this is in turn amplified. The output is a coupling device to another amplifier. The exciting set of pulses of frequency $2f$ synchronizes the timing and is analogous to the clock pulse described above. Different sets of exciting pulses are used for the previous, present, and next successive packages (see Fig. 19-4*c*). Each set of pulses overlaps slightly as shown, and in this way the signal is transmitted from one package to another. Figure 19-4*d* represents a schematic for a parametric-amplifier package, where the M part is the input coupler, and the circle is the amplifier supplied with leads for the exciting pulses. Storage can evidently be obtained simply by direct recirculation, as shown schematically in Fig. 19-4*e*.

Gating can be accomplished as follows: Suppose that there are three coupling windings. The coupling will be in the same phase as the majority of the three windings, since if two of the windings are in different phases (π radians apart) they will cancel and the phase of the third winding will prevail. Hence, to make an *and* gate, let the winding K (see Fig. 19-4*b* and *d*) be in phase 0; then the phase of the exciter will be π only if both A and B are in phase π (that is, both units). To make an *or* gate, we let winding K have phase π. Then the exciter will be in phase π if the phase of A or B (or both) is π. Negation can be obtained by reversing the polarity of the coupling transformer input. This circuit, called a *Parametron*, is symbolically illustrated in Fig. 19-4*d*, where M means *majority*.

EXERCISES

(*a*) If a square waveform is introduced into a pure (1) resistor, (2) diode, (3) capacitor, (4) inductor, and (5) transistor, how will the attenuation affect the shape of the output in each case, assuming a series resistive load? Show mathematically how to account for each change of shape.

(*b*) Consider a flip-flop package with two inputs R and S such that the output Q' is given in terms of its state Q by $Q' = \bar{R}Q + S$. Design a serial adder, using the static, or flip-flop, method. (HINT: The carry and result should be the outputs of flip-flops.)

(*c*) Logically design a serial adder, using only four Parametrons each of which has three inputs as in Fig. 19-4. The circuit should form both the carry and the result. (HINT: Use the Parametrons directly as majority gates, and not as *and* or *or* gates.)

19-3. Clock Phases and Synchronization

Transfer, Repetition, and Overlap Times. In dealing with actual components, certain electronic-circuit responses become important. The first of these relates to the *transfer time* (t_T), the time required for the leading edge of a pulse to pass through a package. Since the leading edge of a pulse may not be vertical, we should more precisely define the transfer time as that interval of time from the moment when the leading edge of the incoming pulse is high enough to be detected, to that moment when the resulting output pulse of the package rises to this same level (see Exercise *a*). On the other hand, the *repetition time* t_R is the time interval that must elapse between two successive input pulses, i.e., the time required for a circuit to settle down so as to be able to distinguish the next successive input pulses. Even though the output information signal resulting from the logical processing of input signals by a gating network may be detected, it does not follow that the network is ready to acquire another input. In general it is the repetition time that limits the speed with which digital information can be processed by a circuit, although not infrequently the proponents of a particular circuit may quote the usually shorter transfer time instead. Finally the *overlap time* t_L must be considered; this is the time interval during which the input signal must be sustained in order to be detected.

Clock Phases and the Unit Time Interval. In Part 4 we deliberately avoided the problems that arise because of the finite response times of the package circuitry. However, as becomes clear from the considerations of Sec. 19-2, we can no longer postpone such a discussion. For this purpose let us first review our concept of the *unit time interval* t_U as the indivisible unit of time in so far as the logical design of the computer is concerned (see, for example, Sec. 15-4). The unit time interval is the shortest possible time interval that can exist between the leading edges of two successive pulses entering the same gating circuit. Clearly then

$$t_U \geq t_R$$

For purposes of logical design all times were taken as integral multiples of the unit time interval, and also for simplicity we chose t_U as the time required for a bit to go from the memory through a gating network to a register (see Sec. 15-1). Thus, for our logical considerations, the leading edges of the clock pulses were one unit time interval apart. However, from the electronic-design point of view this situation is not realistic. A pulse certainly passes through more than one package in a unit time interval; there is a finite delay through each package; and therefore for each successive package a distinct clock pulse must be available to synchronize input pulses. Thus, if the pulses had to pass through four packages in a unit time interval, there would be four successively delayed clock pulses, denoted by *cp*1, *cp*2, *cp*3, and *cp*4. These are called *clock-phase pulses,* and with just the four pulses we would speak of a *four-phase clock.* A package can then be denoted by the clock phase with which it is synchronized, as a *cp*1 package, a *cp*2 package, etc. Evidently it is more efficient to have the same clock phases for all parts of the computer. Hence the number of phases at minimum must be equal to the maximum number of packages that any single pulse in the computer will have to traverse in a pulse period. We shall analyze these timing considerations in more detail.

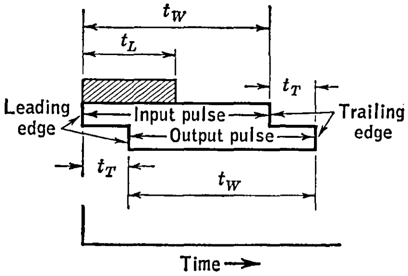


FIG. 19-5. Timing diagram for a package, where t_W is the pulse width.

Clock-phase Sequencing. In Fig. 19-5 we have diagramed the various times associated with a package. The shaded area represents the input overlap time t_L between packages; the input and output pulses are offset by the transfer time t_T; the *pulse width* is represented by t_W. Figure 19-6 illustrates these times for a four-clock-phase system; the diagram of Fig. 19-5 is there repeated for each package. Each unit time interval then contains the four clock-phase pulses *cp*1, *cp*2, *cp*3, and *cp*4. Observe that $t_U \geq 4(t_W + t_T - t_L)$, or for n clock phases $t_U \geq n(t_W + t_T - t_L)$.

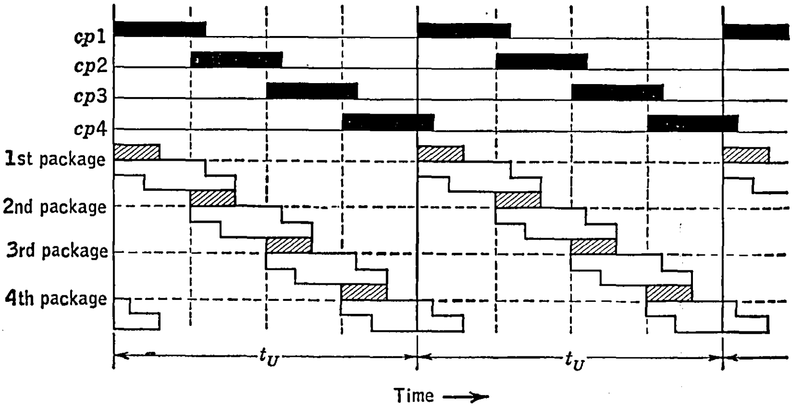


FIG. 19-6. Timing diagram for a four-clock-phase system.

The times t_W, t_T, and t_L are not necessarily independent, since they are determined by the characteristics of the circuit. Suppose that $t_W = t_L$; then $t_U \geq nt_T$, or $n \leq t_U/t_T$. Information can be processed most rapidly when t_U is a minimum, i.e., when $t_U = t_R$. In this case

$$n \leq \frac{t_R}{t_T}$$

which gives the electronic limitation on the maximum number of packages or gating levels that can be used in a unit time interval for most rapid operation. Observe, however, that it is not always possible to have $t_W = t_L$, as is the case, for example, in the dynamic package, where the leading edge of the clock pulse must be coincident with the maximum rise of the incoming pulse. Then all we can say is that the limitation on n is given by

$$n \leq \frac{t_R}{t_W + t_T - t_L}$$

As mentioned above, all parts of the computer will be synchronized by the same clock phases. However, sometimes some particular gating network requires a pulse to go through fewer than the maximum number of packages in a unit time interval. There are two methods for handling such a situation. The first is simply to delay the output pulse during the other clock phases. The second is to arrange the timing of the circuitry in such a way that some clock phases may be skipped. In a four-clock-phase system, if the timing were arranged as in Fig. 19-7 the output of a *cp*1 package could become the input of a *cp*4 package, as illustrated by the shading. Thus either one or two clock phases could be skipped, in the situation of the figure. (See Exercise *c*.)

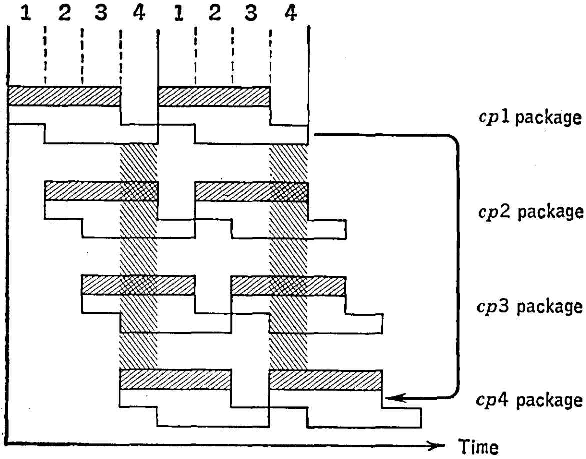


FIG. 19-7. Skipping clock phases.

EXERCISES

(*a*) Discuss the transfer time, repetition time, and overlap time for the following pure components: (1) resistor; (2) capacitor; (3) inductor; (4) transistor. Assume a square-wave input.

(*b*) Using Fig. 19-1, describe t_T, t_R, and t_L for a dynamic package.

(*c*) Suppose that a circuit were designed so that $t_T = \frac{3}{4}t_L$, $t_R = 4t_L$, and $t_U - t_W \geq t_L$. Find a value for t_W that will minimize t_U and still allow for skipping up to two clock phases.

(*d*) For Parametrons the clock pulse has the analogy of a group of sine waves of frequency $2f$. Suppose that 20 of these waves are required (i.e., "pulse" width) and that the overlap time must be 5 of these waves. Draw a diagram analogous to Fig. 19-6 for a three-clock-phase system of this type, indicating the waves as in Fig. 19-4*a*.

19-4. Methods for Increasing Circuit Reliability

General Considerations. Circuit reliability is of primary importance for digital-computer and -control systems, for clearly even a *single* inoperative gate can cause entirely erroneous results. Two possible methods of solutions to this problem are, first, increasing the reliability of the electronic-circuit design itself and, second, increasing the circuit reliability through redundant information-processing components.

A digital circuit has the general characteristic that it either operates successfully or does not; there is no gradation in its operation, such as would be characteristic of radio circuits. The object of digital-electronic-circuit design is to obtain a circuit that will operate successfully under a wide variation of component parameter conditions. For instance, consider the problem of choosing resistance values R for a transistor circuit that will utilize transistors of widely varying β [ratio of collector (output) current to base (control) current] and a not too well regulated voltage source V. (A full discussion of transistors will be found in the next chapter.) Suppose that only the clear area on the V-β plane of Fig. 19-8 represented those values of V and β for which resistance values could be chosen to make the circuit work and that for a given point (V,β) the value of R could vary within the volume illustrated. Then that resistance value R_0 should be chosen which will maximize the area of intersection of this volume and the plane of constant resistance. Of course in general many more parameters are involved, and much more complicated situations occur than that illustrated. Also such calculations may be sensitive to the precise definition of when the circuit is or is not working.

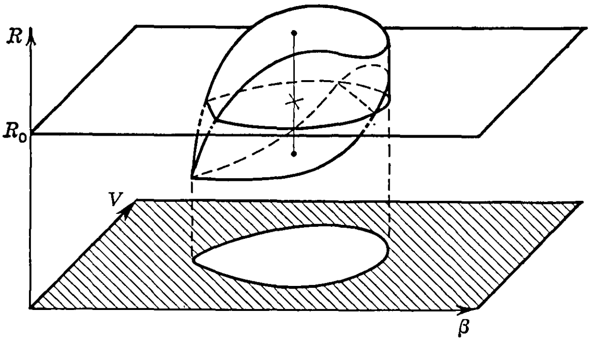


FIG. 19-8. Operating range of characteristics.

Redundant information can be introduced for increasing circuit reliability in a variety of ways. The simplest is perhaps the *parity-bit check*, often used in reading a word into the computer. Here either a zero or a unit, the *parity* bit, is used as an extra bit of the word to make the total number of units of the word always *odd* or always *even* as desired. For example if the word were 0110 and we desired an even number of bits, the parity bit would be 0 and the final word 00110; if the word were 0111, the parity bit would be 1 and the final word 10111. When the word is read, the number of bits is checked; in this way a single error that may have occurred during storage or calculations can be detected. Of course more elaborate systems can be used incorporating self-correcting codification schemes into increased reliability techniques (see Sec. 7-7). These methods, however, lend redundancy to the digital system rather than to the component digital circuits themselves.

We shall here consider two approaches for the introduction of redundancy directly into the gating circuits themselves; these are called the probabilistic-logic approach and the majority-logic approach.

Probabilistic Logic.† A digital circuit has a nonzero but usually small probability p of failure at any time. By means of electronic-circuit design p is made as small as possible. The question arises: Can a logical combination of such circuits have a probability of failure $q \ll p$? The answer is affirmative. For consider a collection of *and* gates each with probability of failure p, such that if A_1 and A_2 are inputs the output with failure will be A_1 (instead of $A_1 \cdot A_2$). Then, as we shall presently show, either output of the configuration of Fig. 19-9 will be the logical product of the inputs, $A_1 \cdot A_2$, with a failure probability of $p^3/4$ ($p^3/4 \ll p$ for small p). In fact, if we used n levels of gating, the total circuit would be equivalent to an *and* gate with failure probability $p^n/4$.

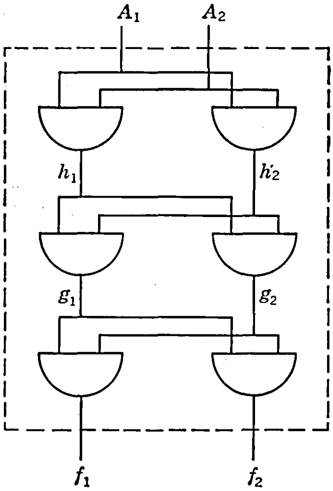


FIG. 19-9. If each *and* gate of the configuration has a failure probability p, then $f_1 = A_1 \cdot A_2$ (and $f_2 = A_1 \cdot A_2$) has a failure probability $p^3/4$.

First let us observe how to write the designation number of a function circuit with a failure probability. Consider, for example, the function h_1 of Fig. 19-9, where $h_1 = A_1 \cdot A_2$ when operating but $h_1 = A_1$ with failure probability p. Note that $\#(A_1 \cdot A_2) = \overset{0123}{0001}$ and $\#A_1 = \overset{0123}{0101}$ differ only in position 1; in fact position 1 will have a unit with probability p and a zero otherwise. Hence we write

Position	0	1		2	3
Probability	1	$\bar{p}$	p	1	1
$\#h_1$	0	0	1	0	1

where $\bar{p} = 1 - p$. Similarly, if $h_2 = A_2$ at failure, with probability p, and $h_2 = A_1 \cdot A_2$ otherwise, then we would write

Probability	1	1	$\bar{p}$	p	1
$\#h_2$	0	0	0	1	1

Our probabilistic logic involves a probabilistic matrix (P_{ji}), which corresponds to an (R_{ji}) matrix of the kind described in Chap. 13. The probabilistic matrix has probabilities as its elements, and a nonzero element p_{ji} of (P_{ji}) means that the jith element of the corresponding (R_{ji})

† This method is based on the author's adaptation for reliable circuit theory of a concept proposed by W. S. McCulloch in connection with neuron net theory (see *MIT Research Lab. Electronics, Quart. Progr. Rept.* 49, Apr. 15, 1958).

matrix is a unit *with probability p_{ji}*. Recall that (R_{ji}) is associated with an array of designation numbers and that a unit in the jth row and ith column of (R_{ji}) means that the ith position (column) of the array is composed of the binary number j. Hence a nonzero probability element in the jth row and ith column of (P_{ji}), namely, p_{ji}, means that the *probability that the array's ith position is composed of the binary number j is p_{ji}.* As an illustration, consider the array of designation numbers (where we have repeated the basis for convenience):

i	0	1		2		3
$\#A_1$	0	1		0		1
$\#A_2$	0	0		1		1
$\#h_1$	0	0	1	0	0	1
$\#h_2$	0	0	0	0	1	1
Probability	1	$\bar{p}$	p	$\bar{p}$	p	1

From this

$$(P_{ji}) = \begin{array}{c|c} & \begin{array}{cccc} i\ 0 & 1 & 2 & 3 \end{array} \\ \begin{array}{c} j\ 0 \\ 1 \\ 2 \\ 3 \end{array} & \begin{pmatrix} 1 & \bar{p} & \bar{p} & 0 \\ 0 & p & 0 & 0 \\ 0 & 0 & p & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \end{array}$$

where $\bar{p} = 1 - p$ as above. Observe that since the elements in a column of the (R_{ji}) matrix are associated with a mutually exclusive and exhaustive set of possibilities (i.e., possible columns of the designation-number array) then the arithmetic sum of the probabilities in a column of the corresponding (P_{ji}) matrix must be unity.

Let us recall that the (R_{ji}) matrix associated with two levels of gating is the logical matrix product of the (R_{ji}) matrices of each level. This is also true of the corresponding (P_{ji}) matrices, except that the ordinary (*not* the logical) matrix multiplication is performed. For example, suppose that failure of gate g_1 (see Fig. 19-9) will result in $g_1 = h_1$ and failure of gate g_2 in $g_2 = h_2$, each with failure probability p; then the same (P_{ji}) matrix will be associated with the second gating level as for the first. Thus we form

$$\begin{pmatrix} 1 & \bar{p} & \bar{p} & 0 \\ 0 & p & 0 & 0 \\ 0 & 0 & p & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \times \begin{pmatrix} 1 & \bar{p} & \bar{p} & 0 \\ 0 & p & 0 & 0 \\ 0 & 0 & p & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix} = \begin{pmatrix} 1 & \bar{p} + \bar{p}p & \bar{p} + \bar{p}p & 0 \\ 0 & p^2 & 0 & 0 \\ 0 & 0 & p^2 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

To see that this result is the desired probabilistic matrix, observe that *the value of each element represents the probability that the element is a unit* in the corresponding (R_{ji}) matrix. For the arithmetic sum of products involved in finding an element of the result matrix is simply the sum of the probabilities of the *exhaustive* and *mutually exclusive ways* in which the corresponding (R_{ji}) element can be a unit. Each product of this sum has two terms, which represent the probabilities of *independent* events (of being units) that must occur if the element is to be a unit in that way. From our resulting adapted matrix we can now form the result array [observing that $\bar{p} + \bar{p}p = (1 - p) + (1 - p)p = 1 - p^2 = \overline{p^2}$]:

Position i	0	1		2		3
Probability	1	$\overline{p^2}$	p^2	$\overline{p^2}$	p^2	1
$\#g_1$	0	0	1	0	0	1
$\#g_2$	0	0	0	0	1	1

A word of caution at this point: *We cannot in general take this result array as representing all possible designation numbers* by choosing all combinations of columns in each position, as we did above when solving type 3 problems with the (R_{ji}) matrix. For in general designation numbers can be formed from the probabilistic result array that *cannot* result from any simultaneous combination of gate failures. On the other hand all designation numbers that *can* result from gate failures can be formed from some column combinations appearing in the result array.

In our example suppose that we desire to find for the first two levels of gating the probability that an erroneous result for g_1 will be obtained. This probability is equal to 1 minus the probability that g_1 is not in error. For no failure $g_1 = A_1 \cdot A_2$; thus, from the probabilistic array, $\#g_1 = 0001$ with probability $1 \cdot \overline{p^2} \cdot (\overline{p^2} + p^2) \cdot 1 = \overline{p^2} = 1 - p^2$, and the probability of failure is $1 - (1 - p^2) = p^2$.

Continuing with our example of Fig. 19-9, assume that the failure of gate f_1 will result in $f_1 = g_1$ and failure of the f_2 gate in $f_2 = g_2$. Then the same (P_{ji}) matrix holds for the third level of gating, and we form

$$\begin{pmatrix} 1 & \bar{p} & \bar{p} & 0 \\ 0 & p & 0 & 0 \\ 0 & 0 & p & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \times \begin{pmatrix} 1 & \overline{p^2} & \overline{p^2} & 0 \\ 0 & p^2 & 0 & 0 \\ 0 & 0 & p^2 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} = \begin{pmatrix} 1 & \overline{p^2} + \bar{p}p^2 & \overline{p^2} + \bar{p}p^2 & 0 \\ 0 & p^3 & 0 & 0 \\ 0 & 0 & p^3 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Observing that $\bar{p}^2 + \bar{p}p^2 = 1 - p^3 = \overline{p^3}$, we have the array:

Position i	0	1		2		3
Probability	1	$\overline{p^3}$	p^3	$\overline{p^3}$	p^3	1
$\#f_1$	0	0	1	0	0	1
$\#f_2$	0	0	0	0	1	1

As is easily seen, for no failure $f_1 = A_1 \cdot A_2$; hence the probability that $\#f_1 = 0001$ is $1 \cdot \overline{p^3} \cdot (\overline{p^3} + p^3) \cdot 1 = \overline{p^3} = 1 - p^3$, and the probability of failure is $1 - (1 - p^3) = p^3$.

One further remark is necessary. We have just determined the probability of *functional failure* of the output of the circuit of the figure. Observe that, even if there is functional failure of the circuit, for three of the four possible values of A_1, A_2 the bit output will be *correct,* namely, when A_1, A_2 take on values

$$\begin{array}{cccc} A_1 & 0 & 0 & 1 \\ A_2 & 0 & 1 & 1 \end{array}$$

Let us suppose that they occur randomly, i.e., that the probability of a given combination of A_1 and A_2 is $\frac{1}{4}$. Then the probability of a *bit failure* is $p^3/4$. In Fig. 19-10 we have graphed the probability of bit failure (dashed lines) and functional failure (solid lines) for our example, as functions of p.

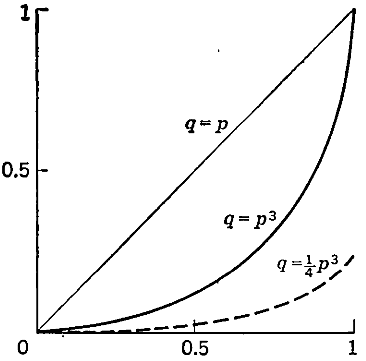


FIG. 19-10. p is the probability of failure of a gate; q is the probability of failure of the configuration in Fig. 19-9.

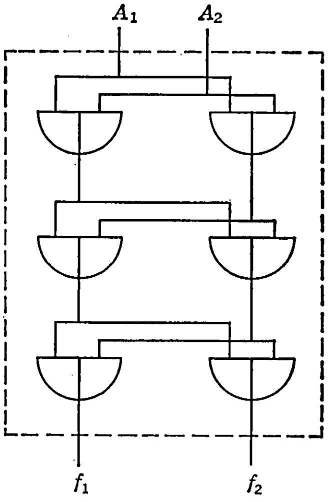


FIG. 19-11. Configuration for increasing the reliability of *or* gates.

It is easy to see that our probabilistic-logic technique is perfectly general and will apply to any sets of functions appropriately connected in levels (see, for example, Fig. 14-5). However, in order to cover all possible output functions with their corresponding probabilities, the nature of the malfunction and its probability must be explicitly stated. A little experimentation will show that only for certain special arrangements of some functions and malfunctions will the output be significantly more reliable than the component circuits. Our example illustrated one of these special cases. Consider now two further important examples, one concerned with increasing reliability of *or* gates and the other concerned with alternate levels of *or-not* and *and-not* circuits. The *or*-gate configuration appears in Fig. 19-11, in which for inputs, say W_1, W_2, the failure results in an output of W_1 (instead of $W_1 + W_2$). Each level has the same (P_{ji}), namely,

$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & p & 0 & 0 \\ 0 & 0 & p & 0 \\ 0 & \bar{p} & \bar{p} & 1 \end{pmatrix}$$

For n levels the functional-failure probability is p^n as in the case of the *and*-gate configuration. The alternate *or-not, and-not* circuit levels of Fig. 19-12, where for inputs W_1, W_2 the failure results in an output of $\bar{W}_1$ (instead of $\overline{W_1 + W_2}$ in the former or $\overline{W_1 \cdot W_2}$ in the latter), present respective probability matrices

$$\begin{pmatrix} 0 & \bar{p} & \bar{p} & 1 \\ 0 & 0 & p & 0 \\ 0 & p & 0 & 0 \\ 1 & 0 & 0 & 0 \end{pmatrix} \quad \text{and} \quad \begin{pmatrix} 0 & 0 & 0 & 1 \\ 0 & 0 & p & 0 \\ 0 & p & 0 & 0 \\ 1 & \bar{p} & \bar{p} & 0 \end{pmatrix}$$

If there are an *odd* number of levels n, then the functional-failure probability for the *or-not* circuit is p^n.

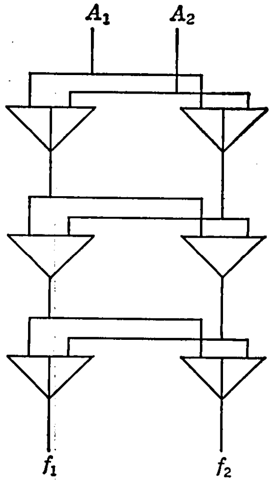


FIG. 19-12. Configuration for increasing the reliability of *or*-inverter gates.

Majority Logic. The probabilistic-logic approach has the disadvantage that the precise forms of possible functional failures must be known. The majority-logic approach does not have this disadvantage; however, the increased reliability that can be gained is probably not so great. An additional circuit element must be used, called a majority circuit, and as we shall see, to gain any reliability at all the failure probability of the majority circuit must be less than that of the gating circuits themselves. The majority-logic technique depends on the formation of three replicas of the gating functions, each being fed into the majority circuit (see Fig. 19-13*a*, where M represents the majority element). At any time the output bit of the majority circuit will be that of the majority of the inputs. Thus, if p_m represents the probability that an *operating* majority element will put out an erroneous result and p the failure probability of a gating circuit (where in Fig. 19-13*a* F_1, F_2, and F_3 are gating circuits), then

$$p_m = 3p^2(1 - p) + p^3 = 3p^2 - 2p^3 \tag{19-1}$$

since for failure either *two* of the three inputs or *all three* inputs must have failed. Of course it is possible to have majority circuits work on a majority of three out of five inputs, four out of seven, etc. (see Exercise *g*). In any case the majority element itself may fail, with failure probability m. Thus the *total* probability of failure of the output of the majority element becomes

$$p' = p_m(1 - m) + m = m(1 - p_m) + p_m \tag{19-2}$$

We certainly desire that $p' < p$ if the majority-logic scheme is to contribute at all to increased reliability. Solving for m in $p' < p$, find

$$m < \frac{p - p_m}{1 - p_m} = \frac{p - 3p^2 + 2p^3}{1 - 3p^2 + 2p^3} \tag{19-3}$$

($<p$, since p is less than unity). Note that, as $p \to \frac{1}{2}$ or $p \to 0$, $m \to 0$; thus m has a maximum for $0 < p < \frac{1}{2}$. To find this maximum, let

$$m = \frac{p - 3p^2 + 2p^3}{1 - 3p^2 + 2p^3}$$

and set $dm/dp = 0$, finding

$$4p^3 - 9p^2 + 6p - 1 = 0 \tag{19-4}$$

Factoring, we get $(4p - 1)(p - 1)(p - 1) = 0$, whence $p = \frac{1}{4}$ for maximum m. Substituting $p = \frac{1}{4}$ into Eq. (19-3), $m < \frac{1}{9}$ at best.

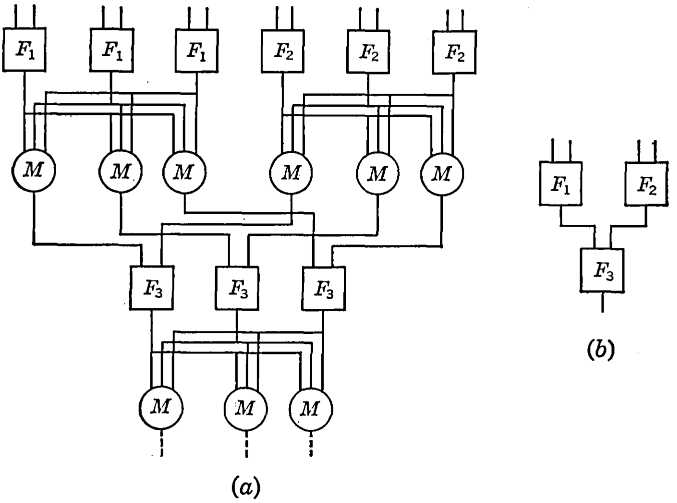


FIG. 19-13. (a) Use of majority circuits; (b) same circuitry without majority elements.

Since p' is limited by the probability of the majority circuit, let us use three such circuits, as shown in Fig. 19-13a. The probability p'_f that either or both inputs to any F_3 of the figure are erroneous becomes

$$p'_f = 2p'(1 - p') + p'^2 = 2p' - p'^2$$

The probability of an erroneous output for F_3, considering both failure of F_3 and erroneous inputs, is

$$p'' = p'_f(1 - p) + p$$

If we did not use the majority circuits (see Fig. 19-13b), this probability would be

$$s = 1 - (1 - p)^3 = 3p - 3p^2 + p^3$$

Observe that, if we let m equal the expression in Eq. (19-3), then $p'' = s$; that is, there would be no point in using the majority circuits at all unless m is less than that value.

Now let us consider the next level of majority circuits shown in Fig. 19-13*a*. Analogous to Eqs. (19-1) and (19-2), we have

$$p''_m = 3p''^2 - 2p''^3$$

and

$$p''' = p''_m(1 - m) + m$$

giving us, analogous to Eq. (19-3),

$$m < \frac{p'' - p''_m}{1 - p''_m}$$

From Eq. (19-4), by substituting p'' for p, we can find that value of p'' for which m is a maximum, namely, $p'' = \frac{1}{4}$. But the largest value m can take is $(p'' - p''_m)/(1 - p''_m)$, which gives

$$p'' = s = 3p - 3p^2 + p^3$$

Hence the maximum value of m must correspond to a p that satisfies

$$3p - 3p^2 + p^3 - \tfrac{1}{4} = 0$$

Solving this for p, $p = 1 - \sqrt[3]{\tfrac{3}{4}} = 0.092$. Substituting this back into Eq. (19-3), we find that

$$m < 0.070$$

for any advantage to be obtained from two levels of majority circuits. In fact m would have to be even smaller if we chose p other than 0.092 for our example. Thus we find that the effectiveness of majority circuits is highly dependent on the reliability of the majority circuits themselves.

EXERCISES

(*a*) Devise the logical design of a circuit that will accomplish a parity check.

(*b*) Devise the logical design of a circuit that will correct two errors in an error-correcting codification of the type described in Sec. 7-7.

(*c*) Using the methods of probabilistic logic, determine the failure probability of n levels of *and* gating of the type shown in Fig. 19-9.

(*d*) Consider the probabilistic analysis of an *and*-gating configuration of the type shown in Fig. 19-9, but allow *three* inputs to each gate, and have three such *and* gates on each level.

(*e*) Consider the extension of the configuration of Fig. 19-12 to n levels of *or*-inverter and *and*-inverter gating. What function is generated at the output of the even-level gates? What are the odd- and even-level output failure probabilities?

(*f*) Form a configuration, similar to that of Fig. 19-12, starting the first level with *and-not* circuits and then alternating. Analyze this configuration for failure probabilities.

(*g*) In majority logic suppose that $2n + 1$ functions F_1 are connected to a majority circuit whose output will be that of the majority of the inputs. What would be the general expression for p_m in terms of p? (HINT: Recall the binomial expansion.)

(*h*) Let $m = 0$, and graph p' as a function of p for $n = 1$, 2, and 3, where $2n + 1$ is the number of inputs to the majority circuit.

(*i*) Logically design a majority circuit for three inputs, using *and* gates and *or* gates.

(*j*) Plot p''' as a function of p for various values of m.

19-5. Additional Topics

a. Parametron References. The following articles contain additional information concerning the Parametron:

Goto, E.: New Parametron Circuit Element Using Non-linear Reactance, *KDD Hankyu Shiryo,* no. 52, November, 1954. (In Japanese.)

———: On the Application of Parametrically Excited Non-linear Resonators, *Denki Tsushin Gakkai-shi,* vol. 38, pp. 770–775, October, 1955. (In Japanese.)

———: The Parametron, a Digital Computing Element Which Utilizes Parametric Oscillation, *Proc. IRE,* vol. 47, pp. 1304–1316, August, 1959. (A correction, by E. Goto, R. G. Allen, and J. E. Mezei, appeared *ibid.,* p. 1840, November, 1959.)

Hushimi, D., and K. Kataoka: Memory for the Parametron, Employing Nondestructive Readout of Information Stored as Residual Polarization, *Proc. Japan. Inst. Elec. and Communs. Engrs.,* vol. 40, pp. 448–453, April, 1957. (In Japanese.)

Introduction to Parametron, *Kenshi Kogyo,* vol. 4, no. 11, p. 4, 1955. (In Japanese.)

Muroga, S.: Elementary Principle of the Parametron and Its Application to Digital Computers, *Datamation,* vol. 4, pp. 31–34, September–October, 1958.

——— and K. Takashima: The Parametron Digital Computer MUSAS(H)INO-1, *IRE Trans. on Electronic Computers,* vol. EC-8, pp. 308–316, September, 1959.

Takada, S.: "Hipac-1 (Hitachi Parametron Automatic Computer)," Hitachi, Ltd., Tokyo, December, 1958.

Takahashi, H.: The Parametron, *Taugakkai-shi,* vol. 39, no. 6, p. 56, 1956. (In Japanese.)

Terada, H.: The Parametron an Amplifying Logic Element, *Control Eng.,* April, 1959, pp. 110–115.

Wigington, R. L.: A New Concept in Computing, *Proc. IRE,* April, 1959, pp. 516–523.

Yamada, S., et al.: Magnetic Core Memory of the Parametron Digital Computer Musasino-1, *Japan. IEE J.,* vol. 33, pp. 1149–1155, November, 1958. (In Japanese, English summary.)

b. References on Reliability

Bellman, R., and S. Dreyfus: On the Computational Solution of Dynamic Programming Processes—XVI: Reliability of Multicomponent Devices, *Rand Corp. Rept.* RM-2245, Sept. 2, 1958.

Depian, L., and N. T. Grisamore: Reliability Using Redundancy Concepts, *George Washington Univ. Tech. Rept.,* February, 1959.

Lowenschuss, O.: Restoring Organs in Redundant Automata, *Inform. and Control,* vol. 2, no. 2, pp. 113–136, June, 1959.

McCulloch, W. S.: Stable, Reliable, and Flexible Nets of Unreliable Formal Neurons, *MIT Research Lab. Electronics, Quart. Progr. Rept.,* April, 1958, pp. 118–129.

———: Stability of Biological Systems, *Brookhaven Symposia in Biol.,* vol. 10, pp. 207–215, 1958.

——— and W. Pitts: A Logical Calculus of the Ideas Immanent in Nervous Activity, *Bull. Math. Biophys.,* vol. 5, 1943.

Moore, S. F., and C. E. Shannon: Reliable Circuits Using Less Reliable Relays, *J. Franklin Inst.,* vol. 262, pp. 191–208, 281–297, 1956.

Moskowitz, F., and J. McLean: Some Reliability Aspects of System Design, *Astia Document* AD 66802.

Pieruschka, E.: Mathematical Foundation of Reliability Theory, *Astia Document* AD 203824.

Proc. Statist. Tech. Missile Evaluation Symposium, Aug. 5–8, 1958.

Von Foerster, H.: Basic Concepts of Homeostasis, *Brookhaven Symposia in Biol.*, vol. 10, pp. 216–242, 1958.

Von Neumann, J.: Probabilistic Logics, in C. E. Shannon and J. McCarthy (eds.), "Automata Studies," pp. 43–98, Princeton University Press, Princeton, N.J., 1956.

Wing, Omar: Reliability Study of Communication Systems, *Astia Document* AD 204139.

CHAPTER 20

SEMICONDUCTOR ELEMENTS IN DIGITAL-CIRCUIT DESIGN

20-1. Introduction

Modern computers are almost all being constructed with transistors and other solid-state components, because of the generally greater reliability, longer life, smaller size, and lower power requirements of these components, compared with vacuum tubes. In this text we therefore consider only circuits using solid-state components, since in any case the tube circuits used in the older computers are almost directly analogous to their transistor counterparts. First, in Secs. 20-2 and 20-3 the physics of diodes and transistors is briefly discussed, so that the reader may gain an intuitive grasp of their properties and functional mechanisms. Next, in Sec. 20-4 is presented a simplified transistor equivalent circuit, which is most suitable for digital-electronic-circuit design. Since the subject of transistor equivalent circuits constitutes a large field in itself, many loose ends are left for the inquisitive reader to follow up elsewhere; but the main transistor properties useful for our purposes are stressed. Sections 20-5 and 20-6 discuss various examples of the application of diodes and transistors to digital circuits. Since computer-circuit engineering is advancing rapidly, our purpose in presenting these examples primarily is to illustrate the principles of the methods employed, and not the specific circuits themselves. Finally in Sec. 20-7 we present a brief introduction to the potentially highly important tunnel diode.

20-2. Solid-state Semiconductor Devices

Semiconductors. The solid-state devices under consideration all depend for their operation on the properties of *semiconductor* materials, which have an electrical conductivity less than the high conductivity of a metal but greater than the low conductivity of an insulator. Germanium (Ge) and silicon (Si) are the most commonly used semiconductors. As solids they form crystals in which the atoms are arranged in a lattice, each atom being surrounded by four other atoms. Each germanium atom has four valence electrons, and in the lattice structure each of these valence electrons is shared with one of the neighboring atoms through a *covalent bond.* The pairs of dots between adjacent atoms in the lattice pictured in Fig. 20-1*a* indicate the shared electrons of the covalent bond. The core of each germanium atom (i.e., the nucleus and the filled shells of electrons) has a net +4 charge (in units of electron charge), which is balanced by the four valence electron charges so that the lattice is electrically neutral. The valence bonds are relatively weak and can be broken by the thermal energy of the lattice vibrations. A broken bond will *free* an electron so that it may carry current, leaving in its place what is called a *hole* (see Fig. 20-1*b*).

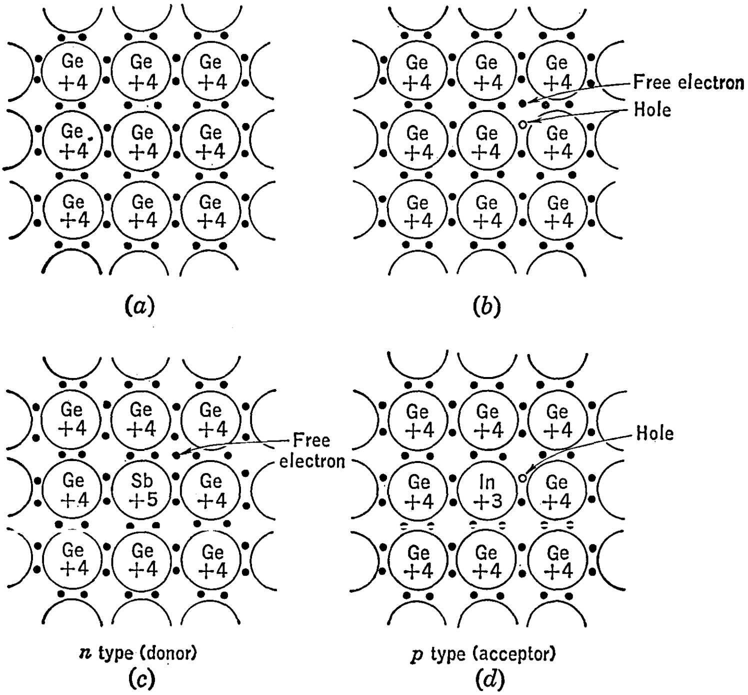


FIG. 20-1. Diagrammatic representation of a germanium lattice section. The pairs of dots represent shared electrons. (*a*) No free electrons, no holes; (*b*) thermal energy frees an electron leaving a hole; (*c*) pentavalent impurity presents a free electron; (*d*) trivalent impurity leaves a hole.

The energy required to break a bond and free an electron (i.e., to make a hole) is about 0.75 ev for Ge and 1.12 ev for Si. The neighborhood of the free electron will of course have a net negative charge, and similarly the neighborhood of the hole will have a net positive charge.

The conductivity of the material arises from two sources: the first is the *free* electrons; the second is concerned with the holes. When a hole appears, it is relatively easy for a neighboring covalent electron to break its bond and fill the hole, leaving a hole in its previous position. Thus it *appears* as though the hole had *moved,* as though there were movement of a positive charge. For this reason we speak of the conductivity due to the virtual movement of holes *whenever an electron fills one hole, leaving another.* Electron-hole pairs are continually being formed by thermal agitation, while other electron-hole pairs disappear by recombination. The conductivity is proportional to the number of electron-hole pairs available at any time: for pure germanium at room temperature there is about one electron-hole pair for each 5×10^{10} germanium atoms.

The electronic properties of interest to us, however, arise when minute amounts of impurities are introduced into the crystal lattice. First consider a five-valence-electron impurity such as antimony (Sb). As shown in Fig. 20-1*c*, the Sb atom will find a place in the Ge lattice and share four of its valence electrons with its Ge neighbors in covalent bonds. However, the fifth valence electron will not be bound and therefore will behave as a free electron, contributing to the conductivity of the material. (More accurately this fifth electron is lightly bound but can be freed by an energy of about 0.01 ev.) Such an impurity is called a *donor*, or *n-type* impurity, because it contributes (donates) electrons, or negative (*n*) charges. These electrons appear in addition to those being formed normally in the germanium by thermal energy.

On the other hand a three-valence-electron impurity, such as indium (In), can be introduced into the lattice. As shown in Fig. 20-1*d*, the three valence electrons of the In atom will form covalent bonds with its Ge neighbors; but the fourth bond will be incomplete, leaving a hole which can accept an electron. Such impurities are called *acceptors*, or *p-type* impurities (*p* for positive). These holes appear in addition to those normally appearing in the germanium.

In summary, in a pure semiconductor (called an *intrinsic* semiconductor) the numbers of free electrons and holes are equal and are determined by statistical equilibrium in pair formation and recombination. In a semiconductor with pentavalent or trivalent impurities (called a *doped* semiconductor) the conductivity is increased by electric-charge carriers that are either predominantly electrons (*n*-type) or predominantly holes (*p*-type).

Semiconductor Junctions and Diodes. The motion of the holes and electrons through a semiconductor is due *not* to mutual repulsion or attraction but rather to the process of *diffusion*, which is a "random-walk" phenomenon analogous to the motion of gas molecules. In general, if there is a high concentration of holes in one part of a semiconductor and a low concentration in another part, the holes will *diffuse* from the high-concentration area into the low-concentration area. The application of an electric field will cause the diffusing holes to *drift* generally in the direction of the field. The same of course holds for the free electrons (drifting against the field).

Suppose that we form a junction between a *p*-type and an *n*-type semiconductor [see Fig. 20-2*a*(1)]. Initially there will be relatively greater concentrations of holes in the *p*-type and of electrons in the *n*-type semiconductor. Hence the holes of the *p* type will diffuse across the junction into the *n* type, and the free electrons of the *n* type will diffuse into the *p* type. The result will be a recombination of holes and electrons near the junction to form negative ions of, for example, In in the *p*-type semiconductor and positive ions of Sb in the n-type semiconductor. A net charge density will be formed, as shown in Fig. 20-2b(1). This charge density will of course be associated with an electric potential, as shown in Fig. 20-2c(1). The effect of this potential is evidently to inhibit the diffusion of holes into the n-type and of electrons into the p-type material. Thus a state of equilibrium will be established.

This equilibrium is, however, dynamic—i.e., there are still random holes and free electrons moving back and forth across the boundary. The movement of holes from the p-type to the n-type material, or of free electrons from the n-type to the p-type material, is called *majority-carrier current* (I_{maj}), since the movement is of the *majority charge carriers* (due to the doping) in the materials in which the movements originate. Conversely the movement of holes from the n type to the p type, and of electrons from the p type to the n type, is called *minority-carrier current* (I_{min}). This latter current is due to hole-free electron pairs being formed in each material when Ge bonds are broken by thermal energy; then holes produced in the n type will "fall down the potential barrier" into the p-type material, while the electrons produced in the p type will be drawn into the n type by theelectr on potential. In equilibrium, then, with due regard to the directions of these currents, $I = I_{maj} - I_{min} = 0$, as illustrated in Fig. 20-2d(1).

Consider now the effect of applying across the p-n junction a potential which opposes the flow of I_{maj} (said to be in the *reverse* direction), as shown in Fig. 20-2(2). Holes in the p type, and electrons in the n type, will drift *away* from the junction, the charge density will extend farther from the junction on both sides, and the potential barrier to majority-carrier current will be increased, whence the majority-carrier current itself will be decreased. The minority-carrier current, on the other hand, is dependent only on the rate of formation of hole-electron pairs and hence will stay approximately the same as for the condition of no applied voltage. The net current across the junction in the reverse direction, i.e., from n-type to p-type material, will therefore be very small.

But consider the effect of applying a potential across the junction in the direction of I_{maj} (that is, in the *forward* direction), as shown in Fig. 20-2(3). In this case the charge density is confined closer to the p-n boundary, the potential barrier is reduced, and more majority carriers are able to diffuse across the boundary. The minority-carrier current still remains about the same, being relatively independent of the applied voltage.

It can be shown by means of statistical mechanics that

$$I_{maj} = I_{min}e^{V/\kappa} \qquad (20\text{-}1)$$

and hence

$$I = I_{maj} - I_{min} = I_{min}(e^{V/\kappa} - 1) \qquad (20\text{-}2)$$

where I = net current, amp (in direction of majority current)
V = applied voltage
$\kappa = T/C$

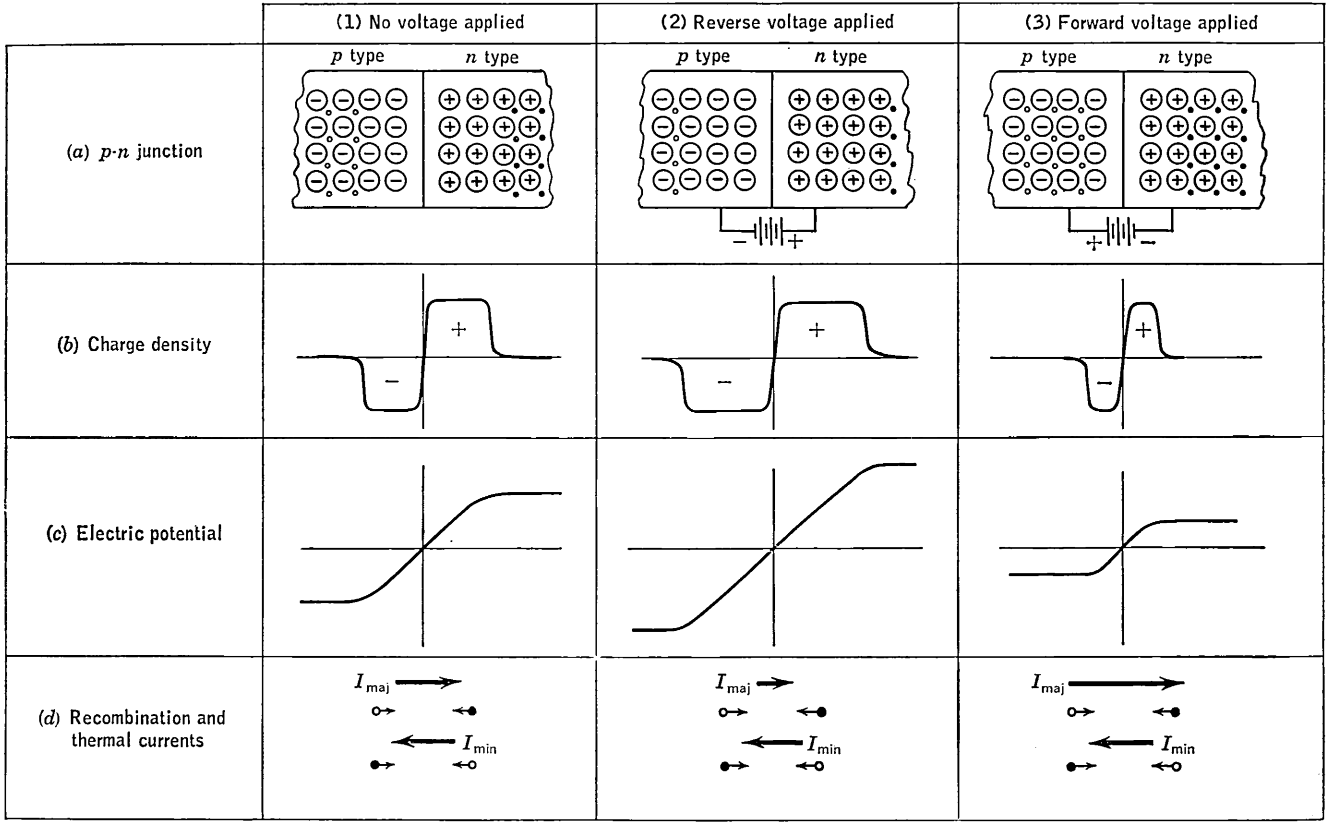


FIG. 20-2. The *p-n* junction (1) with no applied voltage, (2) with reverse applied voltage, and (3) with forward applied voltage. (*a*) The distributions of holes and electrons; (*b*) the charge-density distributions near the junction resulting from negative ions (−) and positive ions (+); (*c*) the resulting electric potentials; and (*d*) the directions of hole and electron movement (short arrows with circles and dots, respectively) and the relative magnitudes of the majority-carrier current I_{maj} and the minority-carrier current I_{min}.

with T the temperature of the semiconductor in degrees Kelvin and C the ratio of the electronic charge to Boltzmann's constant (11,600)—see the solid line in Fig. 20-3. A semiconductor device with one such p-n junction across which a voltage can be applied, called a *diode*, has the characteristic of high conductivity in the forward direction and low conductivity (very high back resistance) in the reverse direction. However, if the reverse voltage is made very large, an entirely different phenomenon occurs, the *Zener effect:* at some critical reverse voltage the back resistance goes practically to zero, and the current is limited only by the voltage-source resistance (see the dashed line in Fig. 20-3). The Zener effect is caused by the breakdown of valence bonds to produce new current carriers; these carriers acquire energy from the field and cause further carriers through ionization by collision, and so forth, as so-called *avalanche breakdown* occurs.

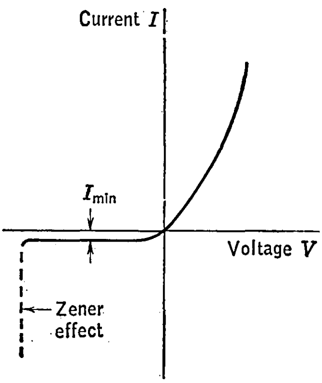


FIG. 20-3. Voltage-current characteristic of an n-p junction diode.

EXERCISES

(*a*) If I_{min} = 10 μa, compute the voltage-current characteristic of a diode at 20°C and at 30°C.

(*b*) What other elements besides Ge, Sb, and In can be used to make a semiconductor diode?

20-3. Transistors

Physical Description. The physical model of a transistor can most easily be understood as an extension of the diode model. In the diode, or single-junction case, the small minority-carrier current across the junction is approximately independent of the applied voltage because it depends primarily on the rate of production (by thermal agitation of the Ge covalent bands) of holes in the n-type and electrons in the p-type semiconductors. The only way to increase this reverse current is to supply the n-type material with holes or the p-type with electrons. Since a p-type semiconductor can be a supplier of holes, suppose that we form a second junction with a p-type material to the right of our original diode (see Fig. 20-4). With no applied voltage we still have not increased the current in the reverse direction, for the holes and electrons will come to equilibrium as shown in 1*a* of Fig. 20-4, to give an equilibrium electric potential as shown in 1*b*. However, if we decrease the potential difference of the middle n-type with respect to the added right-hand p-type material, then as we have already seen above, holes will diffuse into the n-type material. These holes can now diffuse *across* the n-type substance and

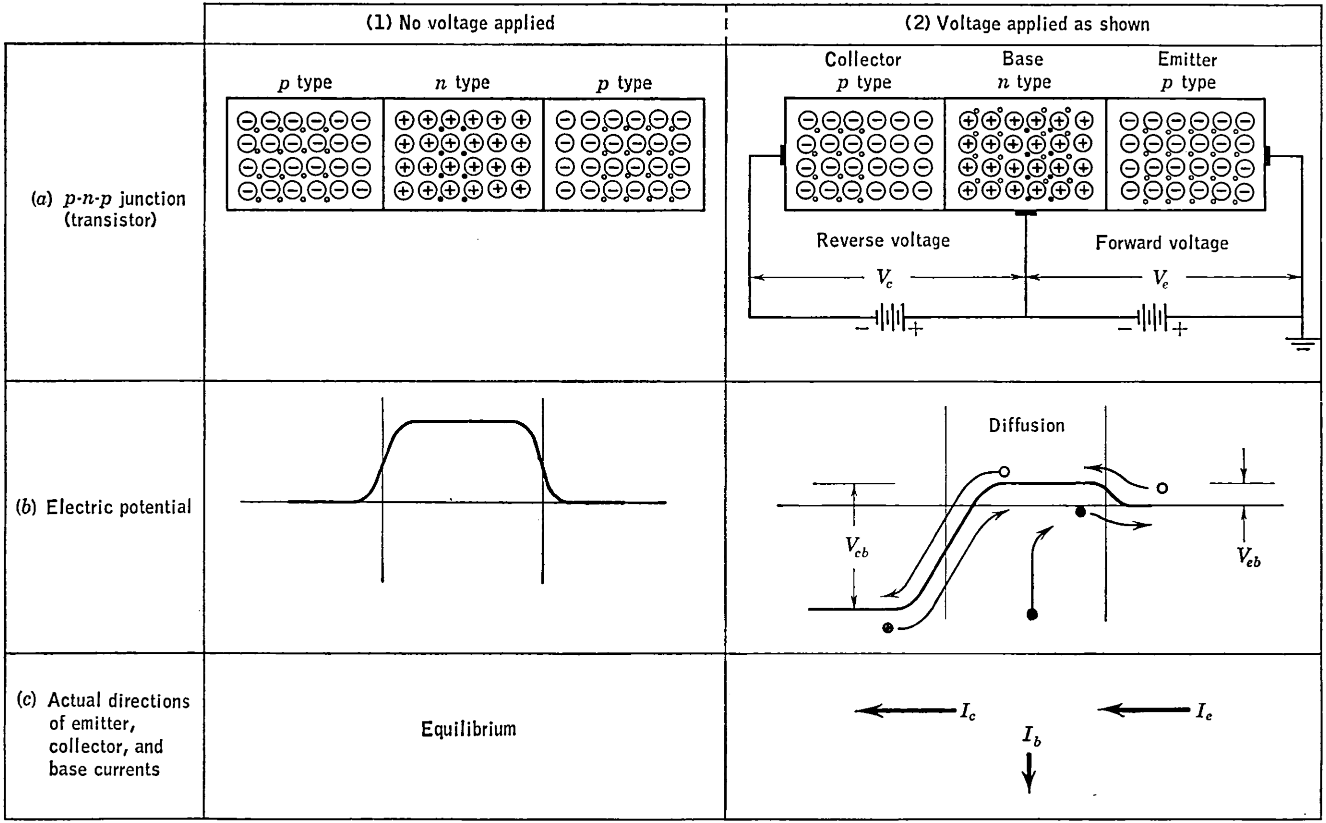


FIG. 20-4. The *p-n-p* junction (transistor) with (1) no applied voltage and (2) forward and reverse voltages applied across the respective junctions. (*a*) Distributions of holes and free electrons; (*b*) electric potentials; and (*c*) the resultant current directions.

into the left-hand p-type substance, causing an increase in the reverse current across the left junction, as desired. We can better our chances of getting a reverse current by *increasing* the potential difference across the junction in question (the left junction) so that the holes diffusing across the n-type substance will "fall down the potential hill." The resultant potential curve is illustrated in Fig. 20-4(2).

The two-junction device just described is called a *p-n-p transistor*, the center n-type substance is called the *base*, the right-hand p-type supplier of holes the *emitter*, and the left-hand p-type substance the *collector*. The use of transistors in digital circuits depends primarily on their *switching properties*. The emitter-to-base voltage controls the conduction of current through the device, from the emitter through the base to the collector. If the applied emitter-to-base voltage V_e is small, no diffusion of holes can occur across the emitter-base junction. Increasing V_e in the forward direction allows holes to enter the base from the emitter, to diffuse across the base, to fall down the collector-base potential hill V_c, and thus to contribute to an increased collector current. Figure 20-5 presents the conventional symbolism for a transistor and also gives the conventional current directions, in terms of which the actual collector and base currents, I_c and I_b as shown in Fig. 20-4c(2), are of course *negative* during conduction.

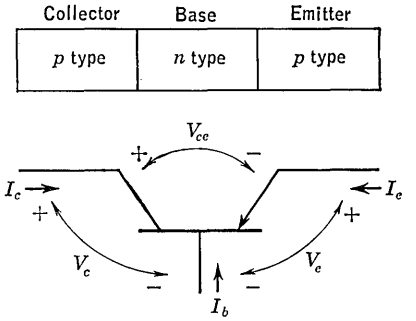


FIG. 20-5. *p-n-p* transistor symbolism, with conventional current and applied-voltage directions.

It is important that the base current I_b be relatively small during conduction so that voltage or current amplification can be obtained. This goal is achieved in the physical design of the transistor. To minimize the contribution to I_b of electrons crossing the base-emitter junction, the doping of the base is made much smaller than the doping of the emitter; thus the base-emitter current consists primarily of holes. Second, the contribution to I_b of electrons recombining with the holes diffusing across the base is minimized by making the base very thin; this reduces the time required for the holes to diffuse across the base from emitter to collector, preventing too many recombinations. Since the conduction through the base-emitter junction is mainly by holes, a measure of the number of recombinations that do occur in the base is the proportion actually reaching the collector, of the holes that crossed the emitter-base junction. This is given by the ratio of the change in magnitude of the collector current I_c to the change in magnitude of the emitter current I_e for a fixed voltage across the collector-base junction. Calling this ratio α, we have

$$\alpha = \left|\frac{\Delta I_c}{\Delta I_e}\right|_{V_c=\text{const}} \tag{20-3}$$

The value of α is close to unity, typical values being around 0.98.

Since from the diode equation (20-2) the contribution from the collector-base junction is $I_{min}(e^x - 1)$ for $x = V_c/\kappa$ and the contribution from the holes of the emitter current is αI_e, we have (observing our current conventions for transistors), replacing I_{min} by I_{co}, the collector minority carrier current, approximately

$$I_c = I_{co}(e^x - 1) - \alpha I_e \qquad \text{for } x = \frac{V_c}{\kappa} \tag{20-4}$$

When the transistor is conducting, the collector-base junction is reverse-biased; e^x is very small compared with unity and can be neglected. Hence Eq. (20-4) reduces to the approximation $I_c = -I_{co} - \alpha I_e$. From our current convention we have

$$I_b + I_c + I_e = 0 \tag{20-5}$$

whence $I_c = -I_{co} + \alpha(I_b + I_c)$. Thus *at constant* V_c, for the ratio, called β, of the changes in magnitude of the collector and base currents, we have

$$\beta = \left|\frac{\Delta I_c}{\Delta I_b}\right|_{V_c} = \frac{\alpha}{1 - \alpha} \tag{20-6}$$

Characteristic Curves of Transistors. Figure 20-6*a* shows typical V_c versus I_c characteristic curves, for constant I_e, of a transistor in the *grounded-base configuration.* Observe that the V_c and I_c axes are *negative* according to our current and voltage conventions (Fig. 20-5). Three regions of transistor operation can be distinguished (see Fig. 20-6*a*): the *cutoff* region 1, the *active* region 2, and the *saturation* region 3. In the cutoff region the base-emitter voltage difference is too large to allow for diffusion of holes into the base from the emitter, even though there is a reverse voltage applied to the collector-base junction. Thus $I_e = 0$, and since the base-emitter voltage is reversed, $e^x \ll 1$; hence from Eq. (20-4), $I_c = -I_{co}$. In the active region the transistor conducts as described above. Saturation occurs when the collector-base voltage difference is too small to keep the holes flowing through the junction. The collector *p*-type material then becomes saturated with holes, and the collector current drops to zero. In fact the collector may become forward-biased and supply holes back to the base: the base then receives holes from both the emitter and the collector, and the hole density in the base becomes large. We shall have more to say about this phenomenon when we discuss over-all time responsiveness of transistors.

Figure 20-6*b* shows a set of V_{ce} versus I_c characteristic curves for constant I_b, for the same transistor connected in a *grounded-emitter configuration.* The active and saturated regions are clearly seen; the cutoff

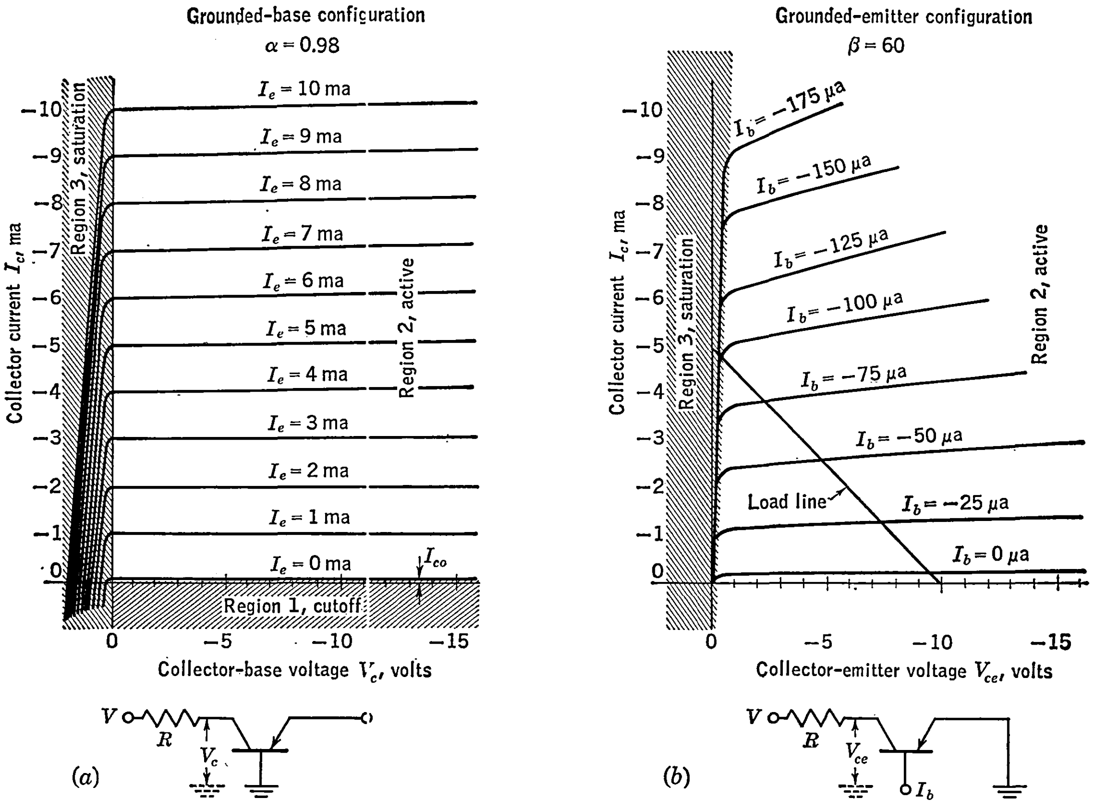


FIG. 20-6. (*a*) Grounded-base I_c versus V_c characteristic curves for constant values of I_e; (*b*) grounded-emitter I_c versus V_{ce} characteristic curves for constant values of I_b. Curves are approximately those for the Raytheon 2N414 transistor, which has $\alpha = 0.98$, $\beta = 60$ (at −12 volts), average $I_{co} = -0.002$ ma, and average $I_{eo} = -0.002$ ma.

region is not, because the base current at cutoff is equal to or greater than I_{co}. These curves are generally more useful than the V_c-I_c curves because the current gain β can be easily calculated from these. For example, if the collector is attached to a -10-volt supply, through a resistor $R = 2{,}000$ ohms, then from the active region in the curves we can observe that a change in I_b from 0 to -100 μa will cause changes in V_{ce} of from -9.8 volts to -0.2 volt and in I_c of from -0.005 ma to -4.82 ma (see the *load line* in Fig. 20-6*b*).

Response Times. There is a delay in transmitting the leading edge of a pulse through a transistor. This delay can be related to the density of holes in the base (i.e., of the principal *minority* current carriers with respect to the collector-base junction). Consider, for example, Fig. 20-7, which illustrates the density distribution of holes in the base. In the cutoff region both emitter and collector are reverse-biased with respect to the base, and hence whatever holes (minority carriers) due to thermal pair generation are present in the base will diffuse toward the emitter and collector, so that the hole-density distribution is as shown in the figure. When an input signal drives the emitter-base junction into the active (forward-biased) region, holes diffuse from the emitter into the base and the hole density builds up near this junction as shown in Fig. 20-7. The hole-density gradient stimulates the diffusion of holes to the collector-base junction; they are quickly gathered into the collector by the potential hill at that junction, to produce the collector signal response. The delay in the transmission of the leading edge of the signal, i.e., the *rise time*, is therefore the time required to build up the hole-density gradient in the base and initiate the final hole-diffusion rate. When the input signal is removed, collector current will continue to flow until the gradient is sufficiently reduced; the time required for this is the *fall time*, the lag in the trailing edge of the signal. If an input signal drives the transistor into the saturation region, the density of holes will rise at the collector-base junction as well. When the input signal is turned off, all these holes must be removed from the base by the collector before the output voltage signal can begin to fall. This delay, or *storage*, can be several times the fall time for the active region. Clearly for transistors used as switches, where speed is essential, care should be taken to keep the transistor from operating in its saturation region. However, in a large proportion of digital circuits the transistor is driven into saturation by the input pulse because of other considerations; these will be discussed in the next section.

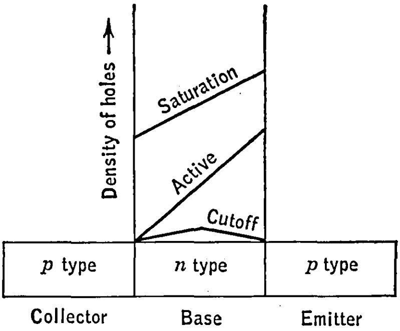


FIG. 20-7. Density distribution of holes in the base of a transistor for the cutoff, active, and saturation regions.

As an illustration of the storage time t_s, consider the grounded-emitter configuration of Fig. 20-8*a*. With an input base-voltage pulse V_{b1} of width t_w a base current $I_{b1} = V_{b1}/R_b$ will flow, provided that R_b is large compared with the internal impedance of the transistor. The rise time t_r of the collector current includes a very small initial delay (not indicated in the figure). During the storage time t_s the impedance of the transistor is still small, and $I_{b2} = V_{b2}/R_b$. After the transistor has returned to its active region, I_b falls to zero as the impedance of the transistor increases, during the normal fall time (see Fig. 20-8*b*, *c*, and *d*).

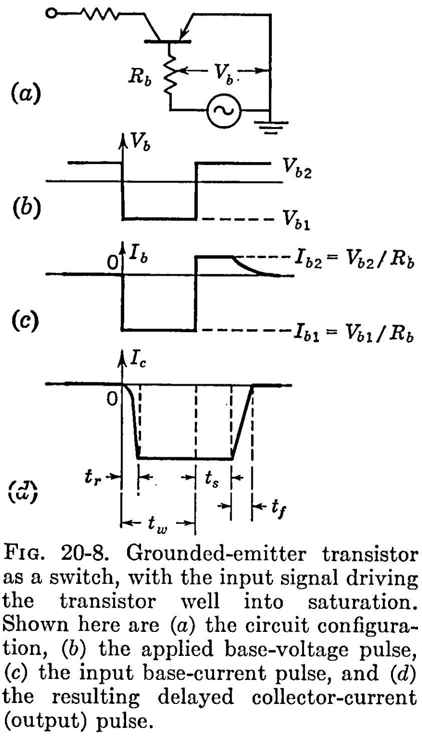


FIG. 20-8. Grounded-emitter transistor as a switch, with the input signal driving the transistor well into saturation. Shown here are (*a*) the circuit configuration, (*b*) the applied base-voltage pulse, (*c*) the input base-current pulse, and (*d*) the resulting delayed collector-current (output) pulse.

Elementary Analysis. An elementary analysis of a transistor, approximately valid in all three regions, can be instructive. The main difficulty of such an analysis is that constants such as α or I_{co} can vary appreciably in different transistors of the same type (although manufacturers are steadily improving reliability). The analysis is based on Eq. (20-4), which we repeat here, using α_n to signify the "normal" α:

$$I_c = I_{co}(e^x - 1) - \alpha_n I_e \qquad \left(\text{for } x = \frac{V_c}{\kappa}\right) \tag{20-7}$$

This is simply the equation for the characteristic curves giving I_c as a function of V_c for constant values of I_e (see Fig. 20-6*a*). Suppose that next we consider the transistor used in reverse, i.e., the collector used as the emitter and the emitter used as the collector. Then we can write an equation analogous to (20-7):

$$I_e = I_{eo}(e^y - 1) - \alpha_I I_c \qquad y = \frac{V_e}{\kappa} \tag{20-8}$$

where α_I is the α of our "reverse" transistor. In writing Eqs. (20-7) and (20-8) we must be careful of the sign conventions for I_c, I_e, V_e, etc., which were summarized in Fig. 20-5. By substituting Eq. (20-8) into (20-7) we can determine I_c as a function of the parameters I_{co}, I_{eo}, α_n, and α_I and the voltages V_c and V_e,

$$I_c = \frac{I_{co}}{1 - \alpha_n \alpha_I}(e^x - 1) - \frac{\alpha_n I_{eo}}{1 - \alpha_n \alpha_I}(e^y - 1) \tag{20-9}$$

Similarly by substituting Eq. (20-7) into (20-8) we find

$$I_e = \frac{I_{eo}}{1 - \alpha_n\alpha_I}(e^y - 1) - \frac{\alpha_I I_{co}}{1 - \alpha_n\alpha_I}(e^z - 1) \tag{20-10}$$

However, α_n, α_I, I_{eo}, and I_{co} are not independent; it can be shown (from the physics of the device) that they are related by the equation

$$\alpha_n I_{eo} = \alpha_I I_{co} \tag{20-11}$$

Equations (20-7) and (20-8) can be solved for the junction voltages V_c and V_e,

$$V_c = \kappa \ln\left(1 + \frac{I_c + \alpha_n I_e}{I_{co}}\right) \tag{20-12}$$

$$V_e = \kappa \ln\left(1 + \frac{I_e + \alpha_I I_c}{I_{eo}}\right) \tag{20-13}$$

The characteristic curves giving I_c as a function of V_{ce} for constant I_b (see Fig. 20-6*b*) can be derived by observing that $V_{ce} = V_c - V_e$. Then, letting $z = (V_c - V_e)/\kappa$, from Eqs. (20-1) and (20-2), using (20-5) and (20-11), we have

$$I_c = \frac{\alpha_I(1 - e^z)I_{co} - \alpha_n(\alpha_I + e^z)I_b}{\alpha_n(\alpha_I - e^z) - \alpha_I(1 - \alpha_n e^z)} \tag{20-14}$$

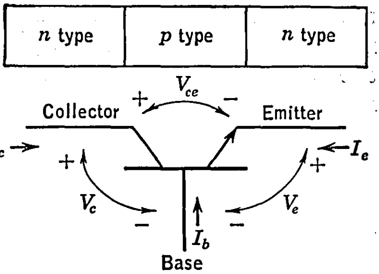


FIG. 20-9. *n-p-n* transistor symbolism with current and voltage conventions.

n-p-n Transistor. A transistor can also be constructed by placing a *p*-type semiconductor between two layers of *n*-type substance, as shown in Fig. 20-9. The arrow on the emitter lead specifies the direction in which current will flow when the emitter-base junction is biased in the *forward* direction, for active-region operation. However, in both the *n-p-n* and *p-n-p* transistors the conventional currents I_c, I_e, and I_b are directed into the transistor (see Exercises *h* through *m*).

EXERCISES

(*a*) How would you make an analogy between a transistor and a triode vacuum tube?

(*b*) Explain from our physical model of the transistor why α is close to but less than unity. (HINT: Where does the base current I_b arise?)

(*c*) What is the physical explanation of the flatness of the V_c versus I_c, constant I_e, curves?

(*d*) In Fig. 20-8 suppose that the input signal V_{b1} did not drive the transistor all the way to saturation; how would curves *c* and *d* appear for I_b and I_c, respectively?

(*e*) For the *p-n-p* transistor of Fig. 20-6 in the grounded-base configuration, if the collector is connected through a 2,000-ohm resistor to a −10-volt supply, and if $I_b = -75\ \mu$a, find the values of V_{ce}, V_c, V_e, I_c, and I_e.

(*f*) Find an expression for V_e at cutoff. (HINT: At cutoff $I_e = 0$, $I_c = I_{co}$.)

(*g*) Derive an expression for I_b when the transistor is in the cutoff region.

(*h*) Draw a figure analogous to Fig. 20-4 for an *n-p-n* transistor.

(*i*) For an *n-p-n* transistor, how would typical V_{ce} versus I_c, constant I_e, and V_c versus I_c, constant I_b, characteristic curves appear? (HINT: Be careful of conventional current directions.)

(*j*) How would an *n-p-n* transistor be used in the grounded-emitter configuration as a switch?

(*k*) What are the main current carriers in the base of an *n-p-n* transistor operated in its active region?

(*l*) Derive an equation for the V_c versus I_c, constant I_e, (grounded-base) characteristic curves of an *n-p-n* transistor.

(*m*) Derive an equation for the V_{ce} versus I_c, constant I_e, (grounded-emitter) characteristic curves of an *n-p-n* transistor.

20-4. Transistor Equivalent Circuit†

Equivalent Circuits. The preceding section has discussed the development of the junction transistor from the viewpoint of qualitative physical theory and the steady-state direct-current (d-c) voltage equations. In order to apply the transistor as an element in an electronic circuit, it is necessary to redescribe the physical device in a form more useful from the viewpoint of circuit-design techniques. This is accomplished by means of the transistor's equivalent circuit, describing the device in terms of familiar circuit elements which can be treated by the standard methods of network analysis. The emitter, base, and collector terminals of the transistor can be connected to the external circuitry in three possible configurations, the grounded-emitter, grounded-base, and grounded-collector modes. Because of its superior switching characteristics the grounded-emitter mode is most widely used in logical switching-circuit design and will be dealt with exclusively in this section. And since the majority of digital transistor circuits employ a voltage-step input to the base (rather than a current-step input), we shall limit ourselves to this type of input.

In this section we shall show a transistor equivalent circuit, in several forms, emphasizing how the circuit is used. The discussions given concerning the origin of the circuits are largely descriptive and heuristic in nature and should not be interpreted as "proofs" in any sense. The development of the small- and large-signal circuits themselves is beyond the scope of the text.

Our main purpose will be to derive the transient rise-time, saturation-state, storage-time, and transient fall-time characteristics of the transistor in the grounded-emitter mode for a large-signal input voltage pulse applied through the base. This involves a preliminary study of the

† This section is based on a private communication on an original development of Sidney B. Geller, Paul A. Mantek, and Don R. Boyle of the National Bureau of Standards.

large-signal d-c behavior as well as the small-signal behavior of the transistor. It is well first to digress for a moment to review the IRE notational standards for variable d-c and modulation parameters. The d-c parameters are indicated by capital letters, the transient, or modulation-level, parameters by lower-case letters (as illustrated in Fig. 20-10 for voltages). The subscripts indicate the reference points: A capital subscript means that a reference point is at true zero, that the total displacement is involved; a lower-case subscript indicates the d-c reference, involving only the local modulation displacement. (Constant parameters may be denoted by any convenient symbols, provided that no ambiguity arises.)

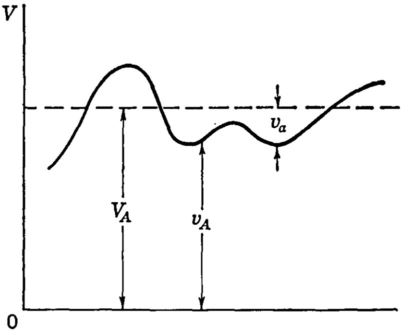


FIG. 20-10. Notational review.

Large-signal D-C Equivalent Circuit. From the equations of the previous section (with I_B substituted for I_b) it can be shown that

$$I_B = \frac{I_{eo}e^{V_e/\kappa}}{(1-\alpha_I\alpha_n)(\beta+1)} - \frac{I_{eo}}{(1-\alpha_I\alpha_n)(\beta+1)^2} - I_{co} \tag{20-15}$$

However, since $I_{co} \ll I_B$ in the active and saturated regions, and also since

$$\frac{I_{eo}e^{V_e/\kappa}}{(1-\alpha_I\alpha_n)(\beta+1)} \gg \frac{I_{eo}}{(1-\alpha_I\alpha_n)(\beta+1)^2}$$

we find approximately

$$I_B \cong \frac{I_{eo}e^{V_e/\kappa}}{(1-\alpha_I\alpha_n)(\beta+1)} \tag{20-16}$$

Also $\quad I_c = -\beta I_B + I_{co} \cong -\beta I_B \quad$ and $\quad I_e \cong (\beta+1)I_B \quad$ (20-17)

Solving for V_e/I_B in Eq. (20-16), with $\kappa \cong 1/39$ at room temperature, we find

$$\frac{V_e}{I_B} = R_{BE} \cong \frac{1}{39I_B}\ln\frac{(1-\alpha_I\alpha_n)(\beta+1)I_B}{I_{eo}} \tag{20-18}$$

The variable resistance R_{BE} can be interpreted as a *diffusion-process resistance* in the base. However, the effective base-to-emitter voltage is somewhat less than the external voltage; the reduction is the loss across a small equivalent internal resistance, denoted by r_{bb}, which must be separated from the diffusion resistance. We shall replace the symbol V_e by V_{BE}, to represent the effective internal base-to-emitter voltages. With V_{BE} replacing V_e, Eqs. (20-17) and (20-18) enable us to draw the main features of the circuit (Fig. 20-11); here R_X is the external resistance and V_X the externally applied voltage step. The value of r_{bb}, from 50 to 1,000 ohms, is usually obtained by disconnecting the emitter and then measuring the collector-to-base resistance, at high frequency so that the capacitance from base to collector effectively short-circuits the current generators.

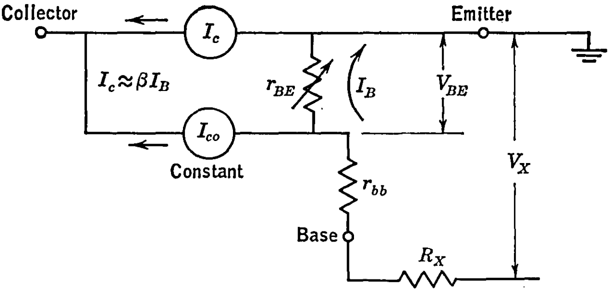


FIG. 20-11. Direct-current large-signal equivalent circuit.

From the figure, with no output current

$$I_B(R_X + r_{bb}) - V_X = V_{BE}$$

Since by Eq. (20-18) V_{BE} (that is, V_e) is a function of I_B, this equation can be solved by finding the intersection of the graphs of

$$y = I_B(R_X + r_{bb}) - V_X \qquad \text{and} \qquad y = V_{BE}$$

when they are plotted as functions of I_B (see Sec. 6-3). Figure 20-12 gives this latter curve of V_{BE} as a function of I_B. The V_{BE}-I_B curve is a key feature of this development and will be utilized extensively in the following discussions. It is easily obtained, either from Eq. (20-18) or by direct d-c measurement of a particular transistor.

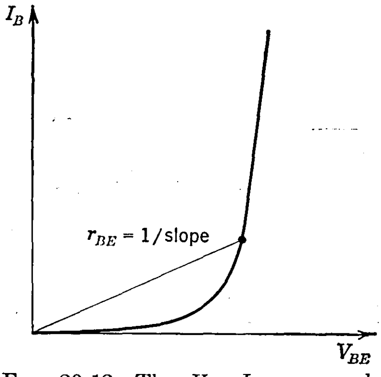


FIG. 20-12. The V_{BE}-I_B curve, obtained from Eq. (20-18) or by direct measurement.

Dynamic Behavior and the Small-signal Equivalent Circuit. The transition from the d-c to the dynamic behavior of the transistor can be made as follows: In the d-c state I_B is the current that flows through the diffusion resistance, contributing to V_{BE} as shown in Fig. 20-11. In the dynamic case, however, a capacitance appears to parallel the diffusion resistance. The axes of Fig. 20-12 should be relabeled with i_{BE} and v_{BE}, in accordance with the IRE notation. The curve, however, remains the same; in Fig. 20-13 it has been redrawn more accurately for a typical

2N414 transistor. A capacitance appears which is a combination of the diffusion-process capacitance and the space-charge capacitance at the base-collector junction; it is denoted by C_{df}. For the dynamic state

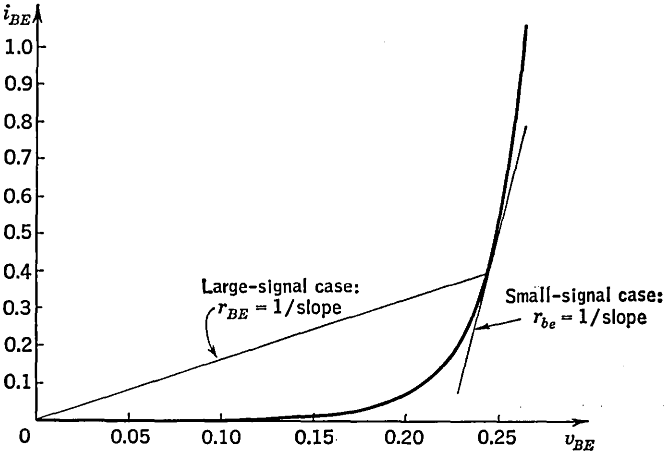


FIG. 20-13. The v_{BE}-i_{BE} curve of a typical 2N414 transistor.

we can rewrite Eqs. (20-16) and (20-17) appropriately:

$$i_{BE} \cong \frac{I_{eo}e^{v_{BE}/\kappa}}{(1-\alpha_I\alpha_n)(\beta+1)} \tag{20-19}$$

and

$$I_c \cong \beta i_{BE} \qquad I_e \cong (\beta+1)i_{BE} \tag{20-20}$$

The small-signal case occurs when in a steady d-c state, with applied voltage V_X, a small increase in voltage is applied, say $\Delta V_X = v_X$, producing small increases in i_{BE}, I_c, and v_{BE}, of $\Delta i_{BE} = i_{be}$, $\Delta I_c = i_c = \beta i_{be}$, and $\Delta v_{BE} = v_{be}$, respectively. However, for a small signal the diffusion resistance can be taken as a constant with the value $v_{be}/i_{be} = \Delta v_{BE}/\Delta i_{BE}$ (see Fig. 20-13). On taking the derivative of Eq. (20-19) we find this resistance to be

$$r_{be} = \frac{v_{be}}{i_{be}} = \frac{1}{(1/\kappa)i_{BE}} \tag{20-21}$$

Thus we can draw the small-signal (variational) equivalent circuit as shown in Fig. 20-14. A circuit analysis easily yields, *for small signals,*

$$i_{be}(t) = \frac{v_x}{R_X + r_{bb} + r_{be}}\left\{1 - \exp\left[-t\Big/\frac{r_{be}C_{df}(R_X + r_{bb})}{R_X + r_{bb} + r_{be}}\right]\right\} \tag{20-22}$$

for a small step v_x. (For the return step, see Exercise *b*.) From the physical analysis that describes the alternating-current (a-c) behavior of a transistor it can be shown that $C_{df} = (1/\kappa)i_{be}T_B^*$, where T_B^* is a constant for a given collector load resistance R_L and collector supply

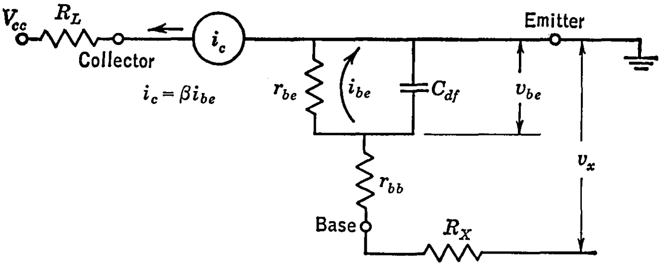


FIG. 20-14. Small-signal equivalent circuit.

voltage V_{cc}. Thus from Eq. (20-21) we have

$$r_{be}C_{df} = T_B^* \tag{20-23}$$

which can be substituted into Eq. (20-22). Therefore T_B^* is an invariant time constant, independent of r_{be} and C_{df}, which vary with r_{BE}. To obtain T_B^*, note that if R_X is very large, i.e., if the base is driven by a current source i_x, then $v_x = i_xR_x$ and Eq. (20-22) becomes approximately

$$i_{be} = i_x(1 - e^{-t/T_B^*})$$

and T_B^* becomes the time required for i_{be} to rise to $1 - e^{-1} = 63.2$ per cent of its final value i_x. (There is an analytic method for obtaining T_B^* for any value of R_L from a *single* rise-time measurement, due to Geller, Mantek, and Boyle of the National Bureau of Standards.)

Large-signal Equivalent Circuit: Rise Phase. The large-signal equivalent circuit is shown in Fig. 20-15. The main difference between it and the small-signal circuit is that r_{be} is replaced by the diffusion resistance r_{BE} and I_{co} no longer cancels out. Because of the nonlinearity of r_{BE} and C_{df}, the exact equation for $i_{BE}(t)$ is difficult, if not impossible, to obtain. A simple graphical method, however, is available, based on the nodal differential equation for the currents into the junction of C_{df} and r_{bb}:

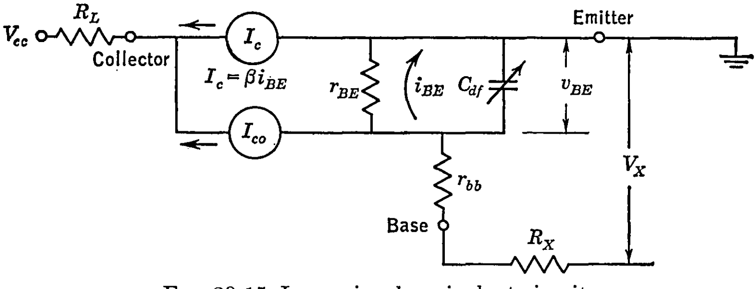


FIG. 20-15. Large-signal equivalent circuit.

$$\frac{V_x - v_{BE}}{R_X + r_{bb}} = i_{BE} + I_{co} + C_{df}\frac{dv_{BE}}{dt} \tag{20-24}$$

From Eq. (20-23), $C_{df} = T_B^*/r_{be}$; from Fig. 20-13 we see that

$$\frac{1}{r_{be}} = \frac{di_{BE}}{dv_{BE}}$$

(i.e., the slope), and hence $C_{df} = T_B^*(di_{BE}/dv_{BE})$. Substituting in Eq. (20-24) we have, since $(di_{BE}/dv_{BE})(dv_{BE}/dt) = di_{BE}/dt$,

$$\frac{V_x - v_{BE}}{R_X + r_{bb}} = i_{BE} + I_{co} + T_B^* \frac{di_{BE}}{dt}$$

Solving for di_{BE}/dt,

$$\frac{di_{BE}}{dt} = \frac{1}{T_B^*}\left[\left(\frac{V_x - v_{BE}}{R_X + r_{bb}} - I_{co}\right) - i_{BE}\right] \tag{20-25}$$

Hence at any point the slope of the $i_{BE}(t)$ curve is represented by the hypotenuse of a triangle constructed at that point having the constant T_B^* as base and the bracketed expression as altitude.

Figure 20-16*a* shows the construction, by linear approximation (cf. Sec. 6-10), of the $i_{BE}(t)$ curve for an input voltage pulse of constant voltage V_X and of relatively long duration starting at $t = 0$. To the left is drawn the v_{BE}-i_{BE} curve, with ordinates to the same scale as on the $i_{BE}(t)$ curve. Intersecting this curve is drawn the "load" line representing the equation

$$i = \frac{V_X - v_{BE}}{R_X + r_{bb}} - I_{co} \tag{20-26}$$

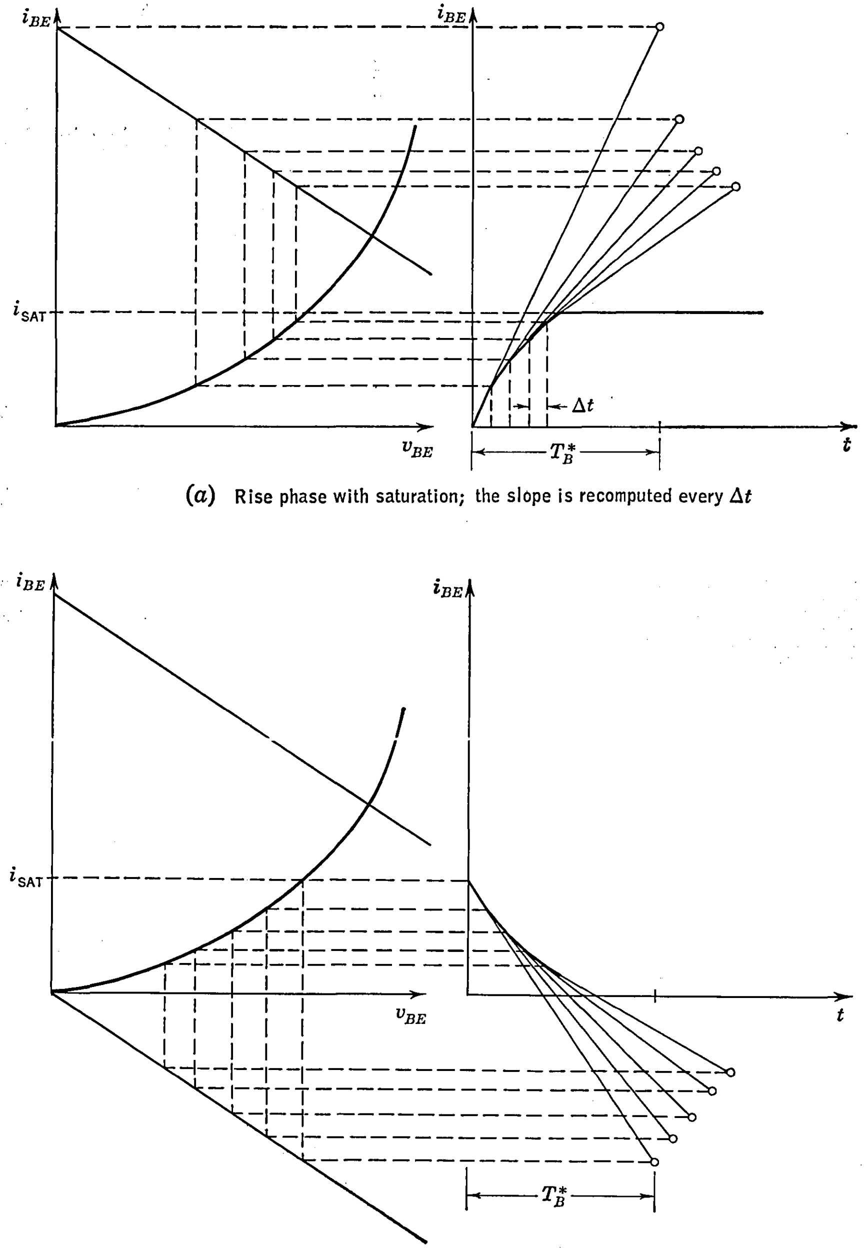
Observe that the distance between the line and the curve, along a line of constant v_{BE}, is the bracketed expression of Eq. (20-25) (displaced by the proper i_{BE}). Starting from zero, the $i_{BE}(t)$ curve is approximated by line segments at constant intervals of time. From a given point of the curve a horizontal line is drawn intersecting the v_{BE}-i_{BE} curve to find the v_{BE} corresponding to the current at this point. From this intersection a vertical line is drawn intersecting the "load" line, measuring off $(V_X - v_{BE})/(R_X + r_{bb}) - I_{co} - i_{BE}$. From this latter intersection a horizontal line is drawn back to a point T_B^* beyond the abscissa of the original point on the $i_{BE}(t)$ curve. The curve is then extended, along the line (the hypotenuse) joining these latter two points, to the next incremental value of t.

The process is continued until i_{BE} reaches saturation, and the curve levels off as shown in Fig. 20-16. The intervals of time chosen need not be uniform, but for greater accuracy may be made smaller in the region where the curvature of the graph is greatest.

Saturation and Storage. Saturation is initiated when

$$I_c = \frac{V_{cc}}{R_L} \tag{20-27}$$

When saturation occurs, then we have $i_{BE} = V_{cc}/\beta R_L = I_{BS}$ and



(*a*) Rise phase with saturation; the slope is recomputed every Δt

(*b*) Fall phase after saturation and storage

FIG. 20-16. Graphical method for determining $i_{BE}(t)$.

$v_{BE} = V_{cc}r_{BE}/\beta R_L = V_{BS}$. Thus in our graphical iteration we must stop when the saturation value of i_{BE} is obtained, and this value then continues until the end of the storage time. To account for this phenomenon in our equivalent circuit, we introduce a diode in series with a battery of strength $V_{BS} = V_{cc}r_{BE}/\beta R_L$ for r_{BE} and β at saturation (see Fig. 20-17*a*). During saturation the circuit behaves as in Fig. 20-17*b*,

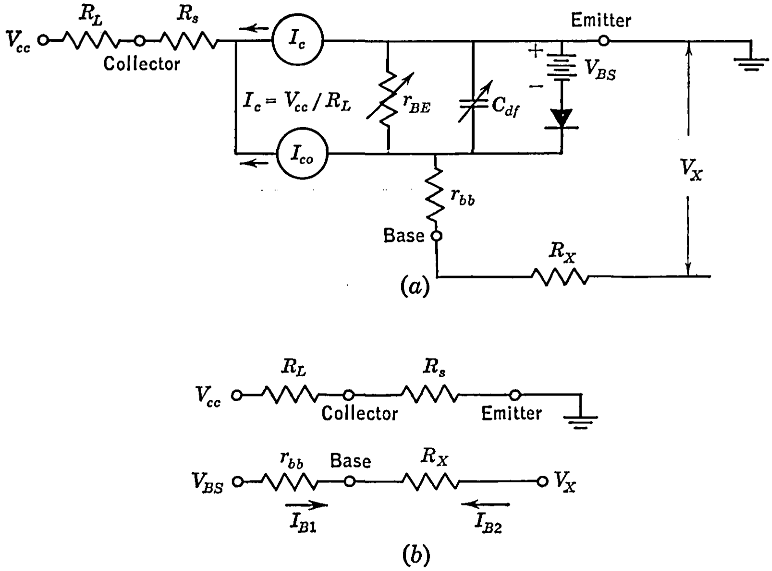


FIG. 20-17. Large-signal equivalent circuit including provision for saturation. (*a*) Equivalent circuit adapted for saturation; (*b*) behavior of equivalent circuit during saturation.

where a small-interval collector resistance R_s of about 4 ohms now appears. We have drawn R_s in Fig. 20-17*a* also, but because of I_c it has no effect before saturation is reached.

As the trailing edge of the V_X pulse cuts off, storage begins, saturation lasting until the storage is ended. If the base current during saturation, but before storage begins, is denoted by I_{B1}; if the reverse base current during saturation but after storage has begun (i.e., after $V_X = 0$) is denoted by I_{B2}; and if $I_{BS} = V_{cc}/\beta R_L$, then it can be shown that the storage duration is given by

$$t_S = K_S \ln \frac{I_{B1} - I_{B2}}{I_{BS} - I_{B2}} \tag{20-28}$$

where K_S is constant for relatively deep saturation. Here K_S is the excess base charge stored per unit excess base current. Equation (20-28) holds only for the case when I_{B2} is a constant reverse current; for the case when I_{B2} is not constant, e.g., if a capacitor is placed across R_X, other techniques, beyond our present scope, must be used to obtain the storage time.

Large-signal Equivalent Circuit: Fall Phase. When storage has ceased, the fall phase begins. Or if V_X returned to zero before saturation was reached, the fall phase would then be initiated immediately. For the fall phase the equivalent circuit of Fig. 20-15 is again valid. Equation (20-25) holds, with $V_X = 0$. Figure 20-16*b* repeats the graphical analysis of Fig. 20-16*a*, with the "load" line given by Eq. (20-26), with $V_X = 0$:

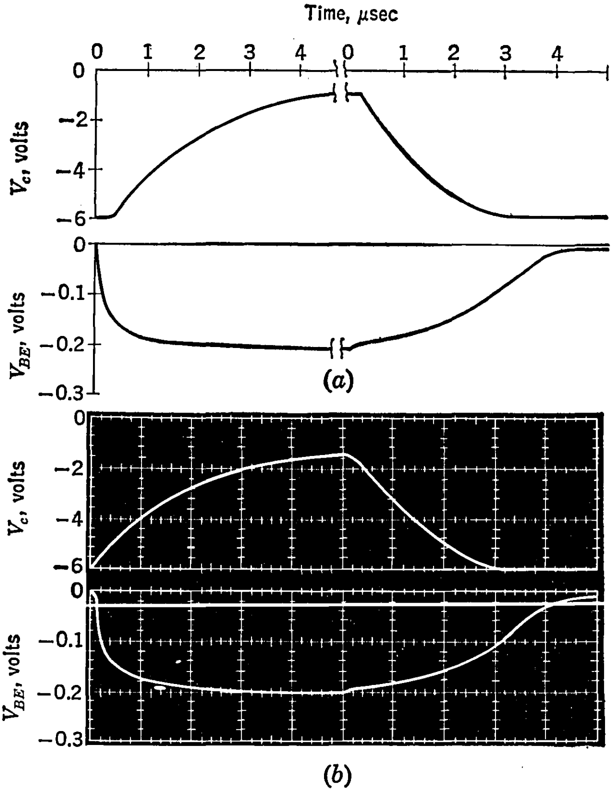


FIG. 20-18. Response of 2N414, with input pulse $V_x = 0.52$ volt (voltage drive to the brink of saturation). (*a*) Simulated; (*b*) experimental.

$$i = -\frac{v_{BE}}{R_X + r_{bb}} - I_{co}$$

Neglecting I_{co} (as is usually done), the line passes through the origin.

Summary. The analysis of Fig. 20-16 may be used for large-signal inputs other than the voltage pulse. Here the two "load" lines would be replaced by a family of lines, found by evaluating Eq. (20-26) for V_X at corresponding time increments.

Given the $i_{BE}(t)$ curves, we can deduce the $V_c(t)$ curve, since

$$V_c(t) = i_cR_L = R_L\beta i_{BE}(t)$$

Figure 20-18 compares an oscilloscope tracing with the computed rise and fall curves of $V_c(t)$ for a voltage pulse at the base of a 2N414 transistor. The increments used for the computed curve were taken fairly small, to illustrate the adequacy of the construction.

EXERCISES

(*a*) Derive Eq. (20-15) from the equations of the previous section.

(*b*) Derive Eq. (20-22) from the equivalent circuit of Fig. 20-14, and show that the return-step current is given by

$$\frac{v_{be}}{R_X + r_{bb} + r_{be}} \exp \frac{-t}{r_{be}C_{df}(R_X + r_{bb})/(R_X + r_{bb} + r_{be})}$$

Consider a 2N414 transistor for which $\beta = 55$, $r_{bb} = 100$ ohms, and $K_S = 2$ μsec. For the purposes of the following exercises take

$$T_B^* \cong 1.5 \times 10^{-6} + R_L(14 \times 19^{-12})56$$

At room temperature $1/\kappa = 39.6$.

(*c*) If $R_X = 20{,}000$ ohms, $R_L = 1{,}000$ ohms, and $V_{cc} = -12$ volts, and if a voltage pulse V_X of -12 volts and 5 μsec duration is applied, draw a graph describing the transient behavior of the voltage at the collector terminal during the rise phase, saturation (including storage time), and the fall phase. Use the curve of Fig. 20-13.

(*d*) For the situation of Exercise *c* find the time required for the collector current to rise to 90 per cent of its final value. What are the emitter current, base current, and base voltage at this time?

(*e*) Repeat Exercise *c*, but let $R_L = 500$ ohms. Compare the results and give a qualitative explanation.

(*f*) Repeat Exercise *c* for $R_L = 300$ ohms and V_X a voltage pulse of -4 volts and 1 μsec duration. Compare the results and give a qualitative explanation.

20-5. Diode-gated Circuits

Diode Gating. Figure 20-19 indicates the conventions and symbols used for a diode. Suppose that a *unit* of a computer word is represented by some signal-voltage level V_+ and a *zero* by a *lower* signal-voltage level V_- (this is sometimes referred to as positive logic). Then the diode current arrangement in Fig. 20-20 will yield an *or* gate. For the output voltage will be the zero signal level V_- only if no diode is conducting; i.e., if any one of the inputs is a unit voltage V_+, the current through resistance R will raise the output potential to the unit signal level V_+. For two diodes the condition-function table would be

$$\begin{array}{rl} A & 0101 \\ \underline{B} & \underline{0011} \\ \text{Output} & 0111 = A + B \end{array}$$

The dashed diode connections in Fig. 20-20 indicate that the gate can have any number of inputs.

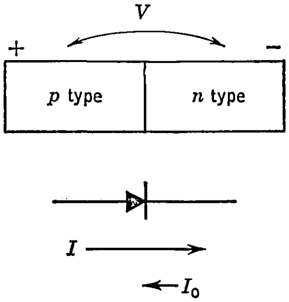


FIG. 20-19. Conventional diode symbolism for semiconduction diode. Polarity of applied voltage V for conduction, and directions of low-resistance current I and high-resistance current I_0 are shown.

Suppose on the other hand that we let V_+ be the zero voltage level and V_- be the unit level (negative logic). Then our condition-function table would be

$$\begin{array}{rl} A & 0101 \\ B & 0011 \\ \hline \text{Output} & 0001 = A \cdot B \end{array}$$

Thus we see that *the interpretation of the signal-voltage levels is important for the logical analysis of the gate.* In Fig. 20-21 we have turned the diodes around and attached the output through the resistor to a V_+ voltage supply. It is seen that, for the same interpretation of 1 and 0, reversing the diodes changes an *or* gate into an *and* gate and an *and* gate into an *or* gate.

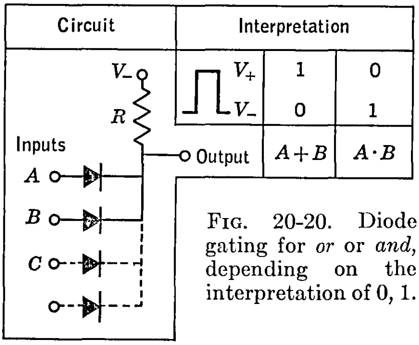


FIG. 20-20. Diode gating for *or* or *and*, depending on the interpretation of 0, 1.

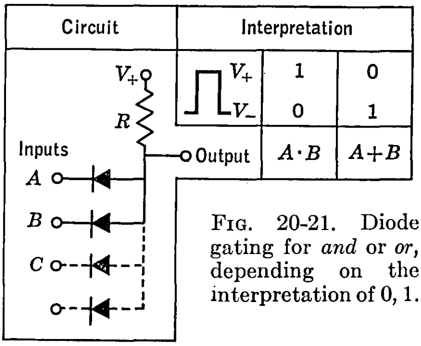


FIG. 20-21. Diode gating for *and* or *or*, depending on the interpretation of 0, 1.

In our discussion so far we have neglected the effect of the diode forward resistance r_f. When this is taken into account, it is seen that the voltage swing from V_- to V_+ on the input side of the diodes will be greater than that on the output side and will depend on the number of diodes conducting at any time. For the circuit of Fig. 20-20, with an input voltage swing from V_- to V'_+ and with n diodes conducting, the output voltage V_+ will be given by

$$V_+ = V_- + \frac{R}{R + r_f/n}(V'_+ - V_-) = \frac{r_fV + nRV'_+}{r_f + nR}$$

which increases toward V'_+ as n increases. When a constant value for V_+ is required, the output voltage can be "clamped" through a diode so that it will never exceed a predetermined V_+. This clamping voltage should clearly be not more than $(r_fV_- + RV'_+)/(r_f + R)$, the value of V_+ above for $n = 1$. Similarly for the circuit of Fig. 20-21, with an input voltage swing from V'_- to V_+, the output could be clamped to a predetermined V_-. However, one should note that the supply voltages connected to the resistors R must be the same as the maximum or minimum input voltages V'_+ or V'_- (why?). The outputs from several of the circuits of

Fig. 20-21 can be the inputs to a circuit of Fig. 20-20 (or vice versa); here both V_+ and V_- could be clamped. Figure 20-22 shows such a combination with the input to the amplifier clamped between V_+ and V_-, where $V_+ \leq V'_+$ and $V_- \geq V''_- > V'_-$. The supply resistors (R in Figs. 20-20 and 20-21) should be large enough to keep the supply current small but should not be large compared with the back resistance r_b of the diodes.

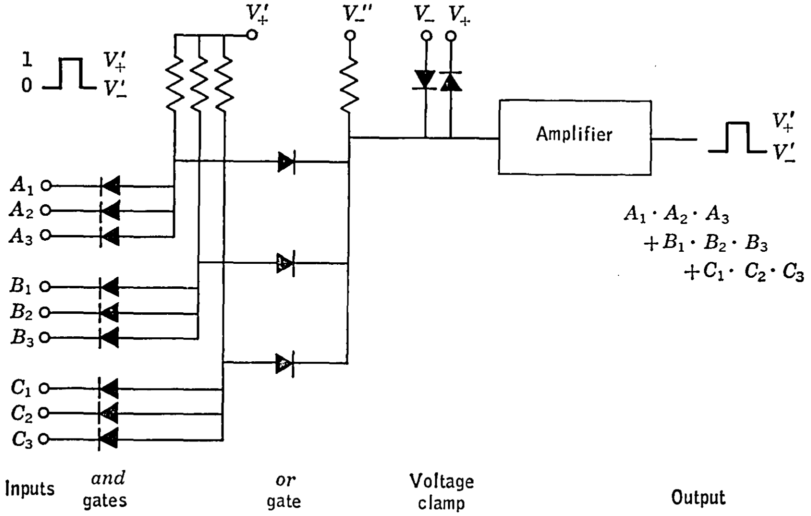


FIG. 20-22. A diode-gate *and-or* circuit.

Transistors as Amplifiers. An ordinary diode is a passive circuit element with no means for reinforcing signal strengths, so that after a few levels of such diode gates as described above the output signals become too much attenuated to be usable. Therefore in all diode-gated circuitry some form of active element is included to reamplify, and sometimes reshape, the signals. The earliest computers used vacuum tubes as amplifiers, but the postwar developments in solid-state devices have brought tubes to obsolescence in computer uses. We consider in this section transistors used purely as amplifiers of diode-gate output, as we shall consider magnetic elements purely as amplifiers in Sec. 21-3.

Figure 20-23*a* shows the simplest of gated-amplifier circuits, a single gate followed by the amplifier. For the 0, 1 levels as shown, the diode configuration acts as an *or* gate; the transistor amplifies the gate output, but in reverse polarity. Note that a clock pulse cannot be used here, in the ordinary way, since the output would be a zero whenever the clock was on. Hence this circuit is usually employed in conjunction with flip-flop synchronization and retiming. (Compare Exercise *b* of Sec. 20-6. Because of the lack of clocking, and because of considerations required by the logic used with these packages, they are generally called *or* inverters rather than *or-not* gates.) In the design of such a package the collector load resistor R_L should be as small as possible, since the base currents for succeeding stages are drawn through R_L (when this transistor is cut off), and the drop across it must not be too great. Similarly R must be as large as possible, to limit the currents through the preceding load resistors, but not so great that it cuts down the base current when the previous stages are fully loaded (i.e., have the maximum drop across their load resistors). The small base-to-emitter resistor r helps reduce saturation and consequent storage time. Figure 20-23*b* shows the optimum values for the circuit parameters, for a type 2N414 transistor, as computed from the equivalent circuit given in Sec. 20-4. (Observe that R_X for the equivalent circuit is computed from the parallel combination of R and r and that the V_X used must be reduced to its Thévenin equivalent.) The 100-$\mu\mu$f capacitor shunting R helps remove the excess holes in the base when the input goes positive.

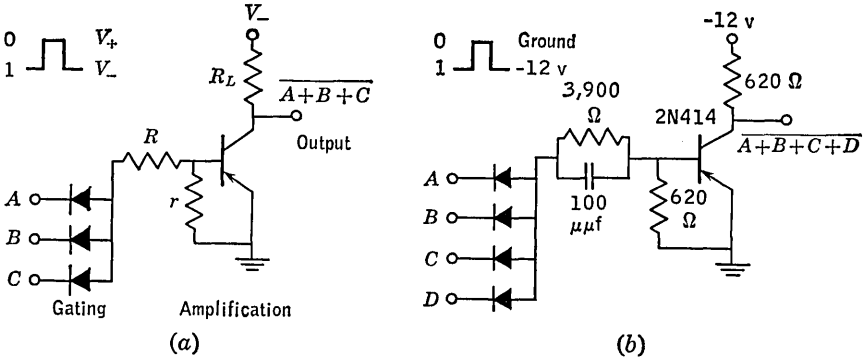


FIG. 20-23. *Or*-inverter packages.

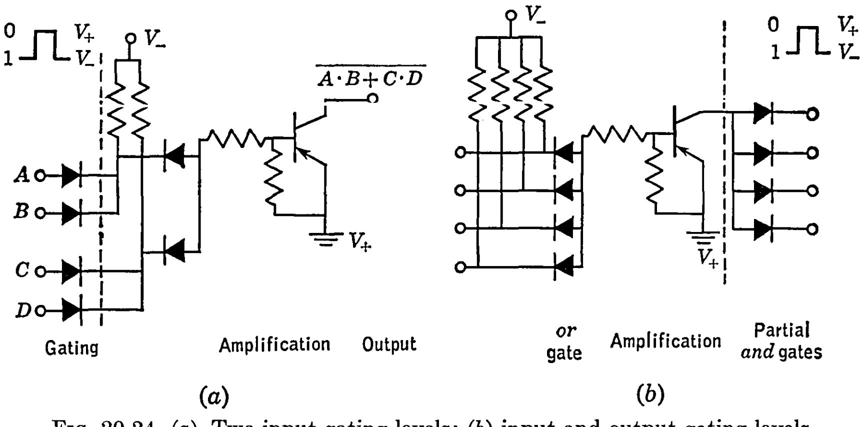


FIG. 20-24. (*a*) Two input gating levels; (*b*) input and output gating levels.

More than one level of diode gating can be used with transistors.

Figure 20-24*a* illustrates a level of *and* gates followed by an *or* gate. Of course each *and* gate can have more inputs than the two illustrated, but once a package is designed the number of inputs to each *and* gate is fixed. Another arrangement of levels of diode gating is illustrated in Fig. 20-24*b*. Here, instead of having both levels of gating precede the transistor, one level precedes and one level follows the transistor. In both circuits the parallel combination of *and*-gate resistors in the following stages acts as a collector resistor. The circuits of Fig. 20-24*a* and *b* differ only in the packaging: the dashed line in (*a*) shows where the separation was made in (*b*), and the dashed line in (*b*) shows where the separation was made in (*a*). Of course each of the output diodes of *b* is connected to a *different succeeding* package, and each input is connected to *several* preceding packages. An advantage of the circuit of (*b*) over that of (*a*) is that the packaging has now limited the number of *outputs* rather than the number of inputs; for packages whose output capacity is quite limited by electronic-design considerations, this arrangement is more natural.

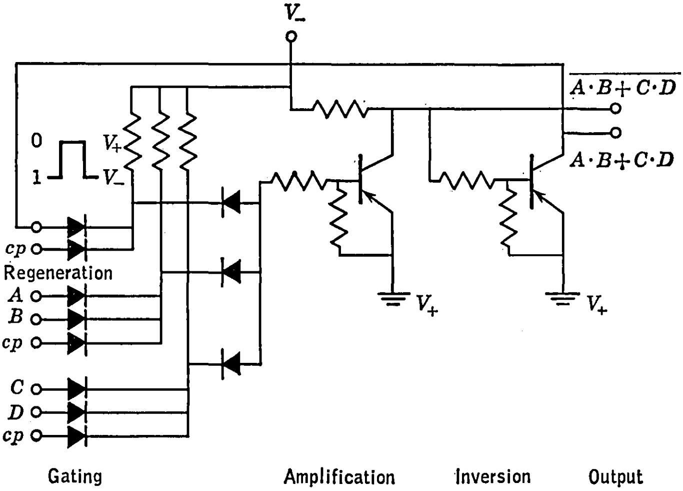


FIG. 20-25. Dynamic circuit with regeneration.

Retiming and reshaping, as well as amplification, can be incorporated into a transistor diode-gating package by means of regeneration as described above. Figure 20-25 illustrates such a package, designed as a simple extension of the circuits of Fig. 20-24*a* and *b*. Note that a second transistor must be used to invert the output before recirculation; thus we have available both the direct and complemented outputs. A collector resistor must be added for the first transistor, to allow for the case of no loading on the complemented output. Pulse transformers could also be used to provide the inversion for the regeneration; here again both direct and complemented outputs result.

EXERCISES

(*a*) For the diode voltage-current characteristic drawn in Exercise *a* of Sec. 20-2, calculate V''_-, V_-, and V_+ of Fig. 20-22 for $V'_+ = 6$ volts and $V'_- = -6$ volts and $R = 10{,}000$ ohms.

(*b*) For the circuit of Fig. 20-23*a* compute I_c, V_c, I_e, i_{BE}, and V_B for $V_- = -12$ volts, $R_L = 620$ ohms, $R = 3{,}900$ ohms, $R = 620$ ohms, and $V_X = -12$ volts, when the output is connected to 1, 5, and 10 other packages.

(*c*) Find the rise and fall times for an input voltage swing from 0 to -12 volts for the circuit of Fig. 20-23*a*, for the values given in Exercise *b*, when the output is connected to 1, 5, and 10 other packages. (Use the transistor values given under the Exercises in Sec. 20-4.)

(*d*) In order to increase the number of packages that can be driven from the circuit of Fig. 20-24*a*, how would you adjust the value of the *and*-gate resistor? The transistor base resistors? What are the limitations to such an adjustment?

(*e*) In order to increase the number of packages that can be driven from the circuit of Fig. 20-23*a*, how would you adjust the resistors R_L, R, and r? What are the limitations to such an adjustment?

(*f*) For the circuit of Fig. 20-25 predict the output voltage rise and fall as a function of time for each transistor, when the *and*-gate resistance is 5,000 ohms, the input resistance to the base of each transistor is 2,000 ohms, the base-emitter shunt resistance is 500 ohms, and the collector resistance of the first transistor is 5,000 ohms, for a voltage input swing of 0 to -10 volts, of duration 1 μsec, and for five additional packages being driven from each output; for one additional package being driven from each output. (Use only four segments in the graphical approximation, and the values of β, T_B, etc., as given under the Exercises in Sec. 20-4, as well as the curve of Fig. 20-13.)

20-6. Transistor Gating and Flip-Flops

Resistance-coupled Transistor Circuits. Figure 20-26*a* illustrates an *or*-inverter circuit gated with resistors only. The circuit is designed so that, if any one of the inputs A, B, or C receives a negative voltage signal, the current through the input resistance is sufficient to turn the transistor on; hence, with the negative signal representing a unit, the output is $\overline{A + B + C}$. The advantages of such a resistance-gated circuit are low cost, simplicity, and high reliability, since semiconductor components are minimized by the elimination of diodes. The disadvantage of the circuit is that if two or more inputs are on at the same time the operating speed is lowered. This occurs because the transistor is driven more deeply into saturation as more inputs are actuated; the effect of more inputs is to place more input resistors in parallel between V_- and the base, lowering the effective resistance of the input and increasing the base-emitter current. In order to maximize the number of transistors that any stage can drive, the collector resistor should be made as small as possible: For when the transistor is not conducting, connecting more outputs to the collector (i.e., more drives) requires a larger current through the collector resistor; the collector voltage may rise to such a value that the succeeding transistors can no longer be turned on. Note that only one level of gating can be accomplished per transistor stage.

In Sec. 18-1 the advantages of combinations of *or* and *and* inverters were described from a logical point of view. In Fig. 20-26*b* is illustrated a mutually compatible *or*-inverter–*and*-inverter circuit pair. The *and* inverter uses an *n-p-n* transistor instead of the *p-n-p* of the *or* inverter. Since the *or* inverter operates between 0 and −12 volts, the emitter of the *n-p-n* transistor (*and* inverter) is connected to −12 volts and the collector load resistor is grounded, so that this circuit too may operate over this range. However, matching problems still occur, for the *p-n-p* transistor operates near and below ground, while the *n-p-n* transistor operates near and above −12 volts. On the other hand, the *or* inverter must be able to drive either an *and* inverter or another *or* inverter, as also must the *and* inverter. One solution to this problem is to have the bias resistor or collector load resistor adjustable according to the succeeding circuit requirements.

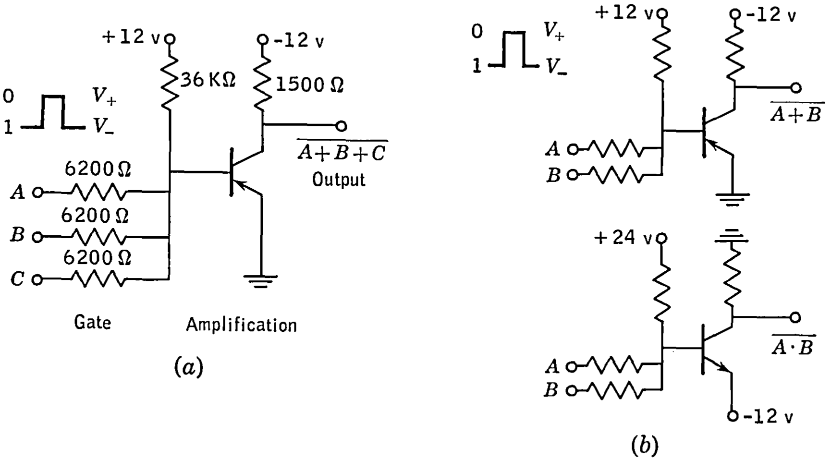


FIG. 20-26. (*a*) Resistor-coupled *or*-inverter package; (*b*) dual *p-n-p* and *n-p-n* circuits.

Direct-coupled Transistor Circuits. Figure 20-27 illustrates direct-coupled transistor logic. The advantage of this type of circuit is that it minimizes the number of different component types and the necessary power supply voltages and presents low power dissipation. In (*a*) is shown the *and* inverter, whose output (inverted) will appear only when *A* and *B* and *C* are units; in (*b*) is shown an *or* inverter, whose output (inverted) will appear if either *A* or *B* or *C* is a unit. Even though the voltage swing from 0 to 1 is small, the transistors are driven into saturation because of the direct coupling, limiting the switching speed. The transistors being driven must be evenly matched as to the base voltage necessary for saturation. Suppose in Fig. 20-27*c* that transistor T_1 is saturated at −0.3 volt, while T_2 requires −0.4 volt for saturation. As the driving transistor is cut off and the voltage falls to −0.3 volt, T_1 will conduct; its base current might be enough to keep the voltage from falling to −0.4 volt, and T_2 would never be driven to saturation.

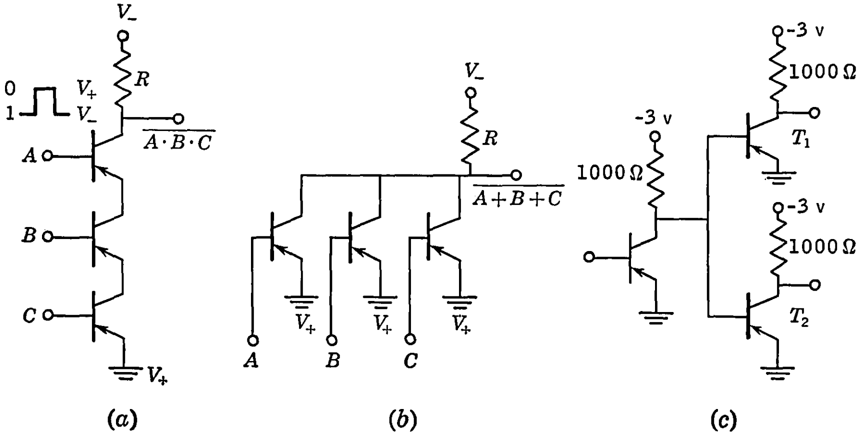


FIG. 20-27. Direct-coupled transistor circuits. (*a*) *and* inverter; (*b*) *or* inverter; (*c*) direct-coupled driving.

Current-switching Transistor Logic. As we have seen in Sec. 20-3, transistors driven into saturation can have long storage-time delays. For higher switching speeds, therefore, it is desirable to use only small voltage swings for 0 and 1 representation and to avoid using the transistors in the saturation region. The current-switching transistor method now to be described is designed for this purpose. The basic concept of

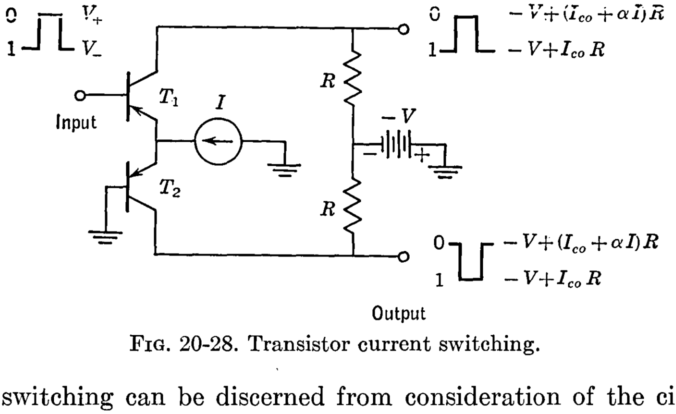


FIG. 20-28. Transistor current switching.

current switching can be discerned from consideration of the circuit of Fig. 20-28. Here the emitters of both transistors are connected to a *constant-current source.* The base of transistor T_2 is grounded, while the base of transistor T_1 receives the input to the circuit, a voltage swing just large enough to cause the transistors either to conduct or to be cut off. Suppose that T_1 is conducting. Then the current will go through the top resistor to $-V$ through T_1, raising the output voltage at the top to $-V + (I_{co} + \alpha I)R$. The emitter-base voltage on T_2 is too small for conduction; hence T_2 is in cutoff so that only I_{co} flows through the bottom resistor and the lower output voltage remains $-V + I_{co}R$. Now suppose that T_1 is cut off by the input signal. The current source will raise the voltage applied to the emitters until T_2 starts to conduct. The current will now flow through T_2, raising the bottom output voltage to $-V + (I_{co} + \alpha I)R$; the top output voltage level drops to $-V + I_{co}R$. In this way a small input voltage swing switches the current through one or the other of the transistors, thereby altering the two complementary output voltages.

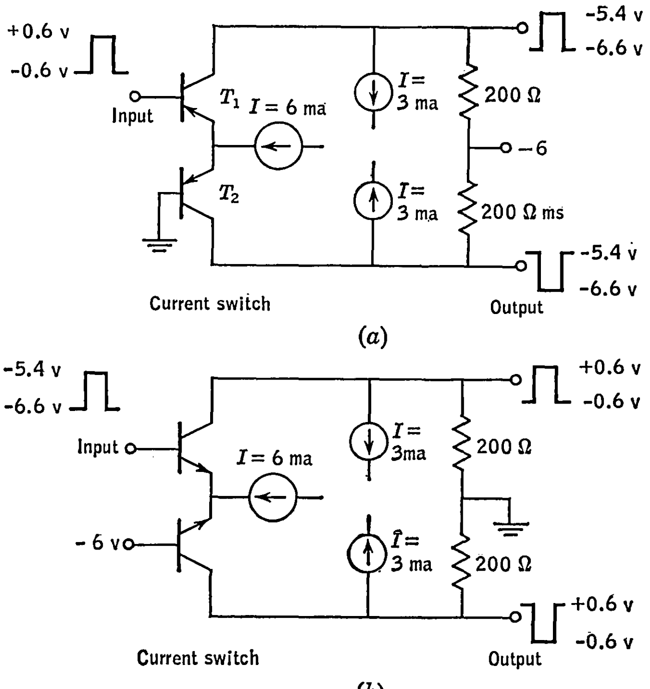


FIG. 20-29. Current-switching circuits for alternate matching.

In order that the voltage swing of the output be symmetric about $-V$, the configurations of Fig. 20-29 are used. Consider, for example, Fig. 20-29*a*: when T_1 is conducting (i.e., the input is at -0.6 volt), approximately 3 ma of the 6-ma source will go to the top current "sink"; the other 3 ma will pass through the 200-ohm resistors to the bottom current "sink," raising the top output voltage to approximately -5.4 volts, and lowering the bottom output voltage to about -6.6 volts. When the input is raised to $+0.6$ volt, T_1 is cut off and T_2 conducts, reversing the potentials at the outputs. The voltage levels given here are approximate because an exact calculation must take into account the effective internal resistances of the transistors; and, in addition, a current source is usually comprised of a large resistor in series with a voltage source, so that the current cannot be held precisely constant.

The outputs of the circuit of Fig. 20-29*a* swing between −5.4 and −6.6 volts, whereas its input changes from +0.6 to −0.6 volt. We must have another current-switching circuit, shown in Fig. 20-29*b*, whose input swing is −5.4 to −6.6 volts and whose output change is from +0.6 to −0.6 volt. Note that *n-p-n* transistors are used in circuit *b* and that the base of the bottom transistor is held at −6 volts instead of at ground level. Circuits of these two types are connected alternately.

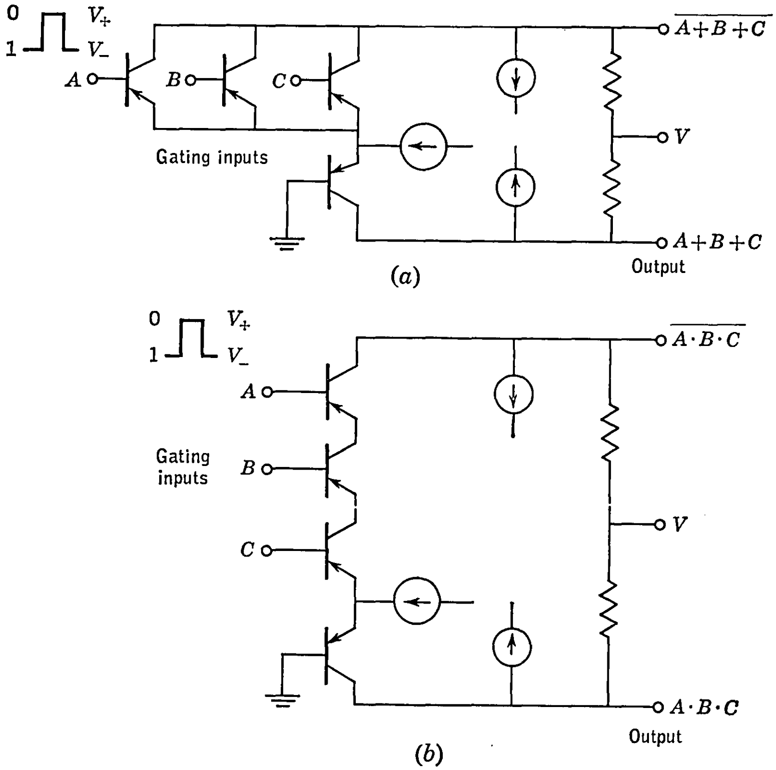


FIG. 20-30. Current-switching gating. (*a*) *or* gate; (*b*) *and* gate.

Current-switching logical operations are performed as shown in Fig. 20-30. In circuit *a*, if any of the transistors *A* or *B* or *C* is turned on, the current is switched, giving the complementary outputs $A + B + C$ and $\overline{A + B + C}$. In circuit *b* all three transistors *A* and *B* and *C* must be turned on to switch the current, giving the complementary outputs $A \cdot B \cdot C$ and $\overline{A \cdot B \cdot C}$.

Transistor Flip-Flop Circuits. The circuit of Fig. 20-31 is typical of transistor flip-flop circuitry. In one of the two stable states of the flip-flop, transistor T_2 is conducting, its collector voltage rising almost to 0 volts (ground). This collector voltage is applied through the resistor to the base of transistor T_1, cutting it off. Since T_1 is not conducting, its collector voltage will be about -12 volts; this voltage is applied to the base of transistor T_2, to keep it conducting (in saturation). The other stable state has T_1 conducting and T_2 not. Again the base-to-collector connections between T_1 and T_2 keep T_1 on (in saturation) and T_2 off (in cutoff). The outputs of the flip-flop, which are the two collector voltages, now reverse. Switching the flip-flop is accomplished by means of the capacitor and diode connected to the base of each transistor. The normal state of each input is at -12 volts; the diode prevents a reverse current from the base of the cutoff transistor. Suppose that T_2 is conducting and T_1 is cut off. When a positive signal is applied to the reset input, T_2 is cut off and its collector voltage falls to -12 volts. putting T_1 in saturation and reversing the state of the flip-flop. A positive pulse in the set input will now change the state back again. The switching of the flip-flop in this fashion is called *R-S* switching (from "Set-Reset"). This mode of operation corresponds to that described in Sec. 12-11, Additional Topic *b*. The small capacitor paralleling the resistor between the base of each transistor and the collector of the other acts simply to decrease the switching time by helping to clear out holes in the respective bases.

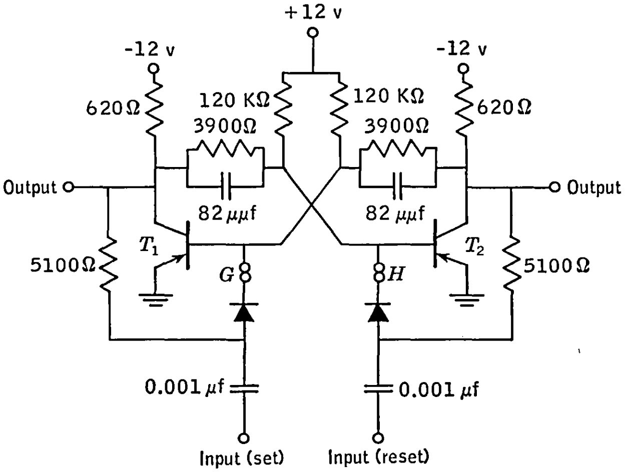


FIG. 20-31. Flip-flop package.

A single trigger pulse can be made to switch the flip-flop. Consider Fig. 20-32, which is the same circuit as that of the previous figure, except that the inputs have been connected together. Suppose that T_2 is conducting and T_1 is in saturation. The quiescent input voltage is -12 volts. The collector of T_2 will be at around 0 volts, while the collector T_1 will be at -12 volts; the base of T_2 will be at a small negative potential, while that of T_1 will be slightly positive. Observe that the voltage across the T_2 input capacitor will be 12 volts, while the voltage across that of the T_1 input will be about 0 volts, since the resistors R

are connected from each input capacitor to the corresponding collector. As a positive pulse (with 0 volts peak) appears at the input, the voltages across the capacitors remain momentarily unchanged. The voltage on the base of T_1 is unchanged since it was more positive than the input pulse peak; the base of T_2, however, is driven positive by the charge on its capacitor, far into the transistor's cutoff region. As the capacitors charge and discharge, T_2 is held at cutoff long enough for its collector voltage to bring T_1 on and the circuit settles into its other stable state. (The trailing edge of the pulse has no effect because of the diodes.)

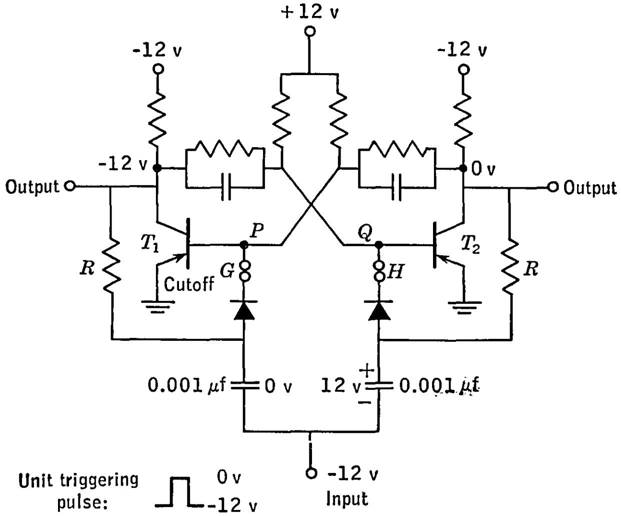


FIG. 20-32. Triggering the flip-flop with a single pulse.

Another positive pulse will of course again change the state, this time cutting off T_1, and so forth. This mode of operation corresponds to the T flip-flop described in Sec. 12-11, Additional Topic *b*.

Figure 20-33 shows several of these flip-flop circuits connected to form a shift register. Suppose that a unit in the flip-flop is represented by the right-hand transistor conducting, the left-hand transistor not, and a zero by the other flip-flop state. When the triggering shift pulse appears, we desire to transfer the state of the first flip-flop stage to the second flip-flop stage. This is accomplished by connecting the resistor part of our diode-capacitor inputs to the collector of each first-stage transistor as above, so that the state of the first stage can be "recorded" in the capacitors; but the diodes are now connected to the bases of the transistors of the *second* stage, so that the triggering excitation will have its effect there. However, we want the *second* stage to attain the same state that the first stage was in, and therefore we must connect the capacitor that senses the T_2 collector of the first stage to the base of the T_1 transistor of the second stage and the capacitor that senses the T_1 collector of the first stage to the base of the T_2 transistor of the second stage (why?), and so on, down the line of successive flip-flops in the register. Note that the first stage has two capacitors attached to its transistors' collectors. The other pair of capacitors load the shift register in the manner described for the circuit of Fig. 20-31.

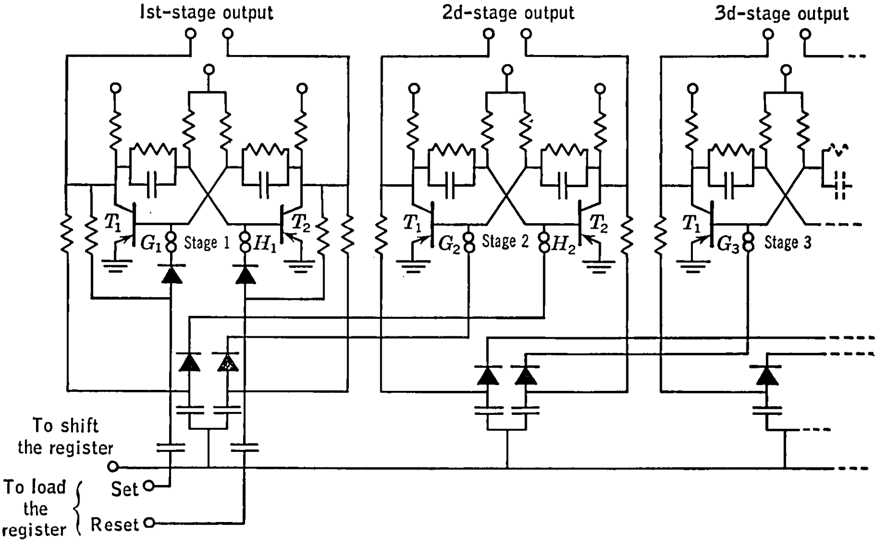


FIG. 20-33. Flip-flop shift register.

EXERCISES

(*a*) Discuss the circuit of Fig. 20-26, considering rise and fall times, number of inputs, saturation, and number of drives, etc. (HINT: Base the discussion on the transistor equivalent circuit.)

(*b*) Describe a method for graphically analyzing the rise and fall collector voltage-time curves for the circuit of Fig. 20-27*c*.

(*c*) How can the flip-flops of Fig. 20-31 be connected (through the *G-H* terminals) to make a serial binary counter?

(*d*) How can the flip-flops of Fig. 20-32 be connected to make a parallel binary counter?

(*e*) Discuss the use of clock pulses in a computer using the *or* inverter of Fig. 20-23, with and without the use of flip-flops. (HINT: In the latter case treat succeeding *or* inverters as *or* gates followed by *and* gates.) Does this discussion apply to the *or*-inverter circuit of Fig. 20-26?

20-7. The Tunnel Diode

Operating Principles of Tunnel Diodes.† The so-called *tunnel diode* is a new solid-state device, with a negative-resistance region in its char-

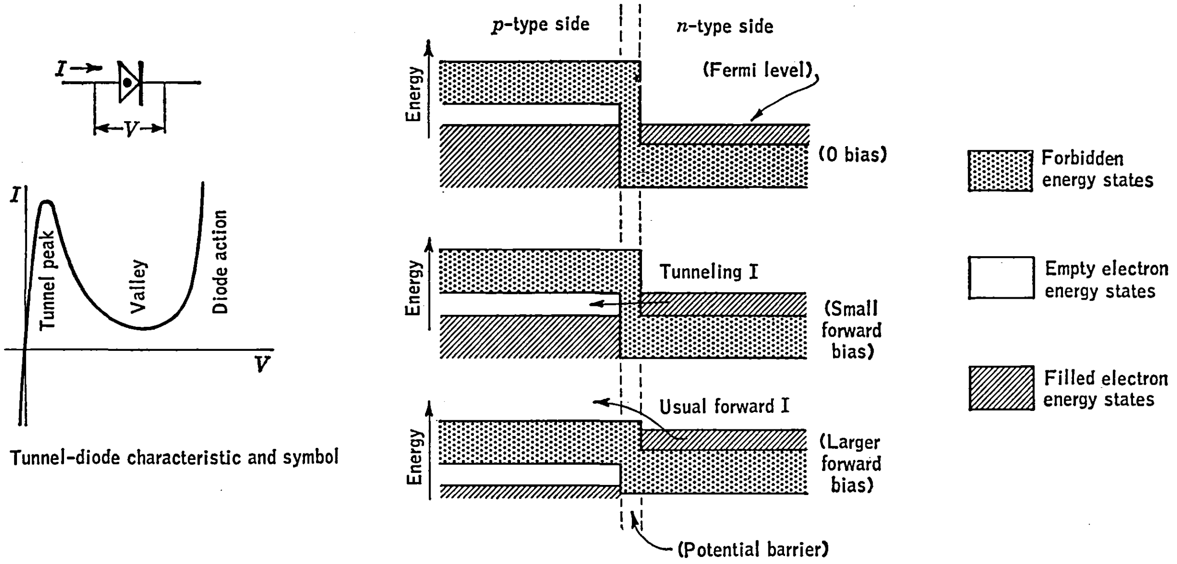
† The tunnel diode was discovered by Dr. Leo Esaki; see L. Esaki, New Phenomenon in Narrow Ge *p-n* Junctions, *Phys. Rev.*, vol. 109, pp. 603–604, Jan. 15, 1958.

acteristic curve, which can be used in microwave digital circuits for amplification and gating (see Fig. 20-34*a*). Utilizing this device, repetition rates (i.e., 1/unit time interval) of 1,000 to 3,000 megacycles, or more, may be realizable. Thus its potential advantage is very great speed in a comparatively reliable solid-state device. At present there are disadvantages, associated with problems of reproducibility and uniformity of characteristics as well as with the many circuit-design problems introduced by the high speed, which must be overcome.

The tunnel diode consists of a junction between extremely heavily doped (doped to "degeneracy") *n*-type and *p*-type semiconductors. It depends for its operation on the quantum-mechanical phenomenon known as *tunneling*, from which the diode derives its name.† Tunneling is an effect in which an electron can "tunnel" through a potential barrier, even though it does not have sufficient energy to "surmount" the barrier, provided the barrier is "thin" enough. The explanation of the tunnel diode's characteristics must therefore be given in terms of available energy states. The tunneling of electrons from the *n*-type side to the *p*-type side of the diode will occur provided that the junction potential barrier is thin, that electrons are available in the *n*-type side to do the tunneling, and that states are available in the *p*-type side into which the electrons can go. Figure 20-34*b* represents an energy-band diagram of the *p* and *n* sides of a tunnel diode, showing forbidden, empty, and filled energy states and their relation to the *Fermi level* in the diode. To simplify the illustration, the picture is shown for low-temperature conditions, under which the energy level between filled and empty states, as indicated by the Fermi level, is well defined; but this is not essential to our discussion. Any electron that tunnels through the barrier must fill an empty energy state at the same energy level from which the electron originally came. That is, in the diagram of Fig. 20-34*b*, electrons from the *n* side must tunnel into empty energy states "horizontally opposite" on the *p* side. Thus for zero bias no tunneling can occur, for there are no *p*-side empty energy states available at the same energy level as the *n*-side electron state. A *small forward bias*, however, brings the *n*-side electron states to a point horizontally opposite empty *p*-side states on the diagram, and tunneling occurs, accounting for the initial *tunneling peak* in the *V-I* characteristic. Further increase in the forward bias puts the *n*-side electron states opposite *p*-side forbidden energy states; tunneling ceases, accounting for the *valley* in the *V-I* characteristic. As the forward voltage becomes sufficiently large, the minority current dominates, as in the usual diode case, producing the rightmost *diode action* portion of the *V-I* characteristic.

Use of Tunnel Diodes in Digital Circuits. As an example of the use of the tunnel diode consider the circuit of Fig. 20-35*a*, due to E. Goto. The circuit is analogous to the Parametron considered above (Sec. 19-2)

† See, for example, L. I. Schiff, "Quantum Mechanics," chap. 5, pp. 92–95, McGraw-Hill Book Company, Inc., New York, 1949; or D. Bohm, "Quantum Theory," chap. 11, pp. 238–242, Prentice-Hall, Inc., Englewood Cliffs, N.J., 1951.

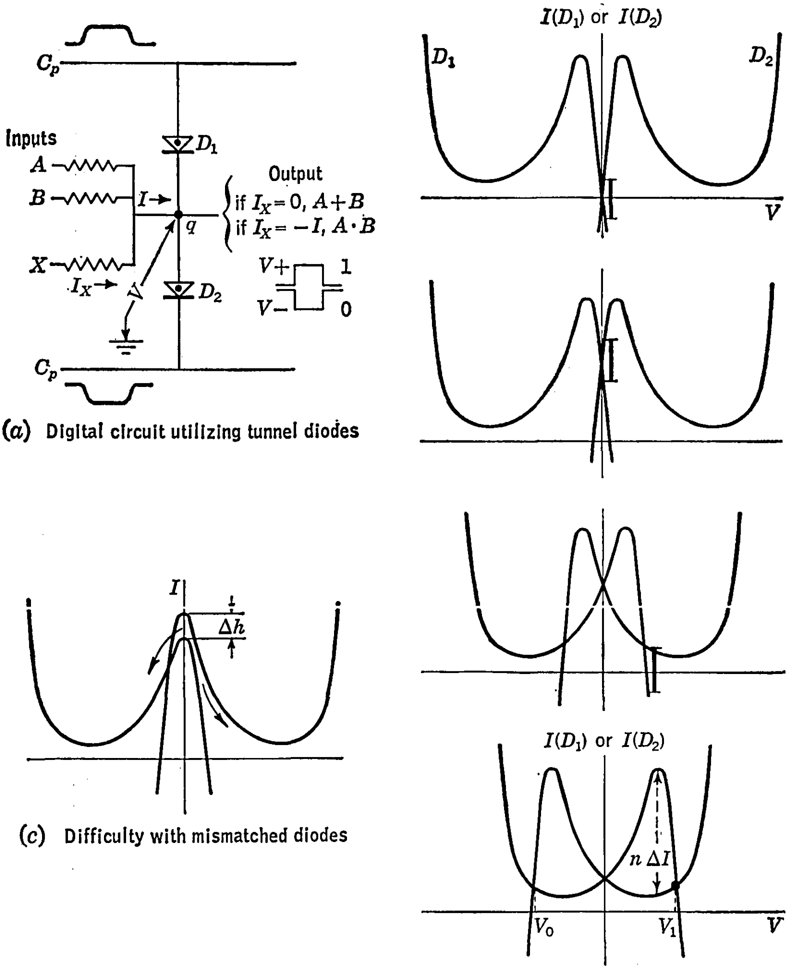


(*a*) Tunnel-diode characteristic and symbol

(*b*) Energy-band diagram of mechanism of tunneling

FIG. 20-34. Tunnel-diode characteristic and explanation of tunneling by available energy-band levels.

in that the clock pulse is utilized to build up a weak input signal; it differs from the Parametron in that its clock pulse and signal inputs and outputs are *direct-current* pulses and *not* wave pulses. The circuit uses two tunnel diodes in series and two clock pulses of opposite polarities;



(*a*) Digital circuit utilizing tunnel diodes

(*c*) Difficulty with mismatched diodes

(*b*) Successive stages in the operation of the circuit in (*a*)

FIG. 20-35. Diagram and operation of a digital circuit utilizing tunnel diodes.

the input to the circuit is a small *current* signal, which behaves as a small current source either into or out of node q depending on the sum of the currents through the large input resistors (see Fig. 20-35*a*). To see how this current signal is built up into a large voltage level by the diodes, consider the successive stages of the operation of the circuit illustrated in Fig. 20-35*b*. Here the vertical axis represents the current through either diode; the horizontal axis represents the voltage V of node q *with respect to ground.* Thus when the clock pulses are zero (i.e., no pulse), V must be negative to forward-bias diode D_1, positive to forward-bias diode D_2, as shown in the top diagram of Fig. 20-35*b*. As the positive clock pulse attached to D_1 comes on, its effect is to move the D_1 characteristic curve further to the right; as the negative clock pulse comes on, its effect is to move the D_2 curve further to the left, as shown successively in the lower diagrams. An input current I to node q will be divided between D_1 and D_2, moving the voltage of node q slightly positive until current I, as shown by the heavy vertical line in the top diagram, is achieved between the two curves. As the clock pulses come on, the movement of the curves forces the voltage of node q to move further positive, always keeping the difference between the currents through D_1 and D_2 equal to I, as shown by the successive positions of the heavy vertical line. Thus the signal is amplified, during a single clock-pulse period, and the voltage V_1 is approximately reached by node q, as shown in the bottom diagram. We say "approximately," for current will now be supplied to the output and after the clock pulse of the *previous* gate is turned off, current may also be drawn backward through the input. A similar argument shows that a voltage of approximately V_0 is reached if the original input signal was a negative current.

Observe that the two tunnel diodes of this circuit must be matched for approximately equal tunnel-peak heights of the characteristic. Consider the effect of a current mismatch of Δh, as shown in Fig. 20-35*c*. If the input current signal ΔI is less than Δh, then as the clock pulses are applied the direction of movement of the voltage V of node q cannot be predicted, for as the characteristic curves move across each other, an unstable point appears at which the voltage could either increase or decrease as indicated by the arrows on the figure. On the other hand, if $\Delta I > \Delta h$, the voltage of node q must remain on the same side of zero, as desired. Note, in the lowest diagram of Fig. 20-34*b*, that the limit on the total number of inputs plus outputs of the circuit is the difference in height between the tunnel peak and the valley, divided by ΔI; that is, $n_i + n_o \leq (h - y)/\Delta I$, for n_i inputs, n_o outputs, tunnel-peak height h, valley height y, and input current signals of magnitude ΔI.

Since the generated signal can propagate both forward and backward, we have a situation again analogous to the Parametron, as described in Sec. 19-2 (see Fig. 19-4). The same solution to that problem can be applied here: three clock phases are utilized, as shown in Fig. 20-36. The method of gating is also analogous to that of the Parametron in that

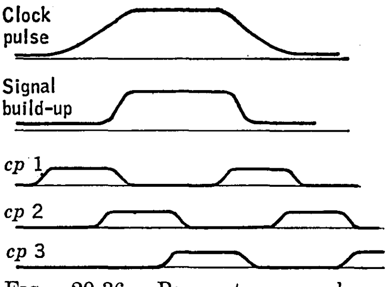


FIG. 20-36. Parametron analogy, showing signal build-up with clock pulse and a three-clock-phase system.

it is essentially of the majority-gate type. Thus, as indicated in Fig. 20-35*a*, if input X has no current, then an input on either A or B will result in a positive voltage output, producing an *or* gate if V_+ is a unit. If the input X has negative current (i.e., out of node q), then both A and B must be unit inputs in order that V_+ result, for an *and* gate. Observe that there is no negation. In the Parametron of Sec. 19-2 we simply reversed the two leads, since transformers coupled the circuits illustrated there. Here, however, two independent gates must be maintained for each gating operation, one for the direct, one for the inverse, output.

EXERCISES

(*a*) On transparent paper, draw the characteristic curves of Fig. 20-34*a* for D_1 and D_2; by sliding the curves across each other, as in Fig. 20-35*b*, observe carefully what happens as the two tunnel peaks cross.

(*b*) Design a serial adder using the circuit given in Fig. 20-35*a*. More than three inputs may be utilized. What is the minimum number of circuits required?

(*c*) Design a shift register utilizing the tunnel-diode circuit of Fig. 20-35*a*. How is the "delay" accomplished?

(*d*) Design a digital circuit using only a single tunnel diode. (HINT: Current-bias the tunnel diode just below the tunnel peak; then an additional current pulse will drive the voltage to the diode action slope. What is the limitation on the number of inputs and outputs of this circuit? Why is a three-phase clock necessary? How can gating be accomplished?)

20-8. Additional Topics

a. Selected Bibliography on Semiconductor Physics

Bridgers, H. E., J. H. Scaff, and J. N. Shive: "Transistor Technology," Bell Laboratories Series, 3 vols., D. Van Nostrand Company, Inc., Princeton, N.J., 1959.

Dunlap, C. W., Jr.: "An Introduction to Semiconductors," John Wiley & Sons, Inc., New York, 1957.

Hannay, N. B.: "Semiconductors," Reinhold Publication Corporation, New York, 1959.

Katz, H. W.: "Solid State Magnetic and Dielectric Devices," John Wiley & Sons, Inc., New York, 1959.

McAfee, K. B., E. J. Ryder, W. Shockley, and M. Sparks: Observations of Zener Currents in Germanium P-N Junctions, *Phys. Rev.*, vol. 83, pp. 650–651, August, 1951.

McKay, K. G.: Avalanche Breakdown in Silicon, *Phys. Rev.*, vol. 94, pp. 877–884, May, 1954.

Middlebrook, R. D.: "Junction Transistor Theory," John Wiley & Sons, Inc., New York, 1957.

Miller, S. L.: Avalanche Breakdown in Germanium, *Phys. Rev.*, vol. 99, pp. 1234–1241, August, 1955.

Scobey, J. E., W. A. White, and B. Salzberg: Fast Switching with Junction Diodes (Operating at Reverse Breakdown), *Proc. IRE*, vol. 44, pp. 1880–1881, December, 1956.

Shive, John N.: "Properties, Physics, and Design of Semiconductor Devices," Bell Laboratories Series, D. Van Nostrand Company, Inc., Princeton, N.J., 1959.

Shockley, W.: The Theory of P-N Junction in Semiconductors and P-N Junction Transistors, *Bell System Tech. J.*, vol. 28, pp. 435–489, July, 1949.

———: "Electrons and Holes in Semiconductors," D. Van Nostrand Company, Inc., Princeton, N.J., 1950.

———, M. Sparks, and G. K. Teal: P-N Junction Transistor, *Phys. Rev.*, vol. 83, pp. 151–162, July, 1951.

Warschauer, D. M.: "Semiconductors and Transistors," McGraw-Hill Book Company, Inc., New York, 1959.

b. Selected Bibliography on Transistors and Equivalent Circuits

Coblenz, A., and H. L. Owens: "Transistors," McGraw-Hill Book Company, Inc., New York, 1955.

Deuitch, D. E.: Junction Transistor Switching Characteristics, *Transistors, RCA Labs.*, March, 1956, pp. 609–639.

DeWitt, D., and A. L. Rossoff: "Transistor Electronics," McGraw-Hill Book Company, Inc., New York, 1957.

Easely, J. W.: The Effect of Collector Capacity on the Transient Response of Junction Transistors, *IRE Trans. on Electron Devices*, vol. ED-4, pp. 6–14, January, 1957.

Ebers, J. J., and J. L. Moll: Large-signal Behavior of Junction Transistors, *Proc. IRE*, vol. 42, pp. 1761–1772, December, 1954.

Enenstein, N. H.: A Transient Equivalent Circuit for Junction Transistors, *IRE Trans. on Electron Devices*, vol. PGED-4, pp. 37–54, December, 1953.

Farley, B. G.: Dynamics of Transistor Negative Resistance Circuits, *Proc. IRE*, vol. 40, pp. 1497–1508, November, 1952.

Giacoletti, L. J.: Study of P-N-P Alloy Junction Transistors from D-C through Medium Frequencies, *RCA Rev.*, vol. 15, pp. 506–562, December, 1954.

Hunter, L. P.: "Handbook of Semiconductor Electronics," McGraw-Hill Book Company, Inc., New York, 1956.

Lo, A. W., R. O. Endres, J. Zawels, F. D. Waldhauer, and Chung-Chih Cheng: "Transistor Electronics," Prentice-Hall, Inc., Englewood Cliffs, N.J., 1955.

Marcovitz, M. W., and E. Seif: Analytical Design of Resistor-coupled Transistor Logical Circuits, *IRE Trans. on Electronic Computers*, vol. EC-7, pp. 109–119, June, 1958.

Moll, J. L.: Large Signal Transient Response of Junction Transistors, *Proc. IRE*, vol. 42, pp. 1773–1784, December, 1954.

Peterson, L. C.: Equivalent Circuits of Linear Active 4 Terminal Networks, *Bell System Tech. J.*, vol. 27, pp. 593–622, October, 1948.

Pritchard, R. L.: Electric Network Representation of Transistors—A Survey, *IRE Trans. on Circuit Theory*, vol. CT-3, pp. 5–21, March, 1956.

Shea, R. F. (ed.): "Principles of Transistor Circuits," John Wiley & Sons, Inc., New York, 1953.

Zawels, J.: Physical Theory of New Circuit Representation for Junction Transistors, *RCA Rev.*, vol. 16, pp. 360–378, September, 1955.

c. Selected Bibliography on Diodes and Transistors in Digital Circuits

Anderson, A. E.: Transistors in Switching Circuits, *Proc. IRE*, vol. 40, no. 11, pp. 1541–1558, November, 1952.

Beter, R. H., W. E. Bradley, R. B. Brown, and M. Rubinoff: Directly Coupled Transistor Circuits, *Electronics*, vol. 28, no. 6, pp. 132–136, June, 1955.

———, ———, ———, and ———: Surface-barrier Transistor Switching Circuits, *IRE Conv. Record*, pt. 4, pp. 139–145, 1955.

Booth, G. W., and T. P. Bothwell: Logic Circuits after a Transistor Digital Computer, *IRE Trans. on Electronic Computers*, vol. EC-5, pp. 132–138, September, 1956.

Carroll, W. N., and R. A. Cooper: Ten Megapulse Transistorized Pulse Circuits for Computer Application, Digest of Technical Papers, Transistor and Solid-state Circuits Conference, Feb. 20–21, 1958.

Chaplin, G. B. B.: Transistor Regenerative Amplifier as a Computer Element, *Proc. IEE*, vol. 101-3, pp. 298–313, 1954.

Clark, E. G.: Direct Coupled Transistor Logic Complementing Flip-flop Circuits, *Electronic Design*, vol. 5, pt. 1, pp. 24–27, June 15, 1957; pt. 2, pp. 34–47, Aug. 1, 1957.

Deuitch, D. E.: Transistor Circuits for Digital Computers, *Electronics*, vol. 29, no. 5, pp. 160–161, May, 1956.

Easley, J. W.: Transistor Characteristics for Direct-coupled Transistor Logic Circuits, *IRE Trans. on Electronic Computers*, vol. EC-7, pp. 6–16, March, 1958.

Grisamore, N. T., L. S. Rotolo, and G. U. Uyehara: Logical Design Using the Stroke Function, *IRE Trans. on Electronic Computers*, vol. EC-7, pp. 181–184, June, 1958.

Harris, J. R.: Direct-coupled Transistor Logic Circuitry, *IRE Trans. on Electronic Computers*, vol. EC-7, pp. 2–6, March, 1958.

Henle, R. A.: High-speed Transistor Computer Circuit Design, IRE-AIEE Transistor and Solid-state Circuits Conference, February, 1957, Philadelphia.

——— and J. W. Walsh: The Application of Transistors to Computers, *Proc. IRE*, vol. 46, pp. 1240–1254, June, 1958.

McMahon, R. E.: Designing Transistor Flip-flops, *Electronic Design*, vol. 3, no. 10, pp. 24–26, October, 1955.

Millman, J., and H. Taub: "Pulse and Digital Circuits," McGraw-Hill Book Company, Inc., New York, 1956.

Olsen, K. H.: Transistor Circuitry in the Lincoln TX-2, *Proc. Western Joint Computer Conf.*, February, 1957, pp. 167–171.

Page, C. H.: Digital Switching Circuits, *Electronics*, vol. 21, no. 11, pp. 110–118, September, 1948.

Prom, G. J., and R. L. Crosby: Junction Transistor Switching Circuits for High-speed Digital Computer Applications, *IRE Trans. on Electronic Computers*, vol. EC-5, pp. 192–196, December, 1956.

Rowe, W. D.: Transistor NOR Circuit Design, *Electronic Design*, vol. 6, no. 3, pp. 26–29, Feb. 5, 1958.

——— and R. G. Royer: Transistor NOR Circuit Design, *Trans. AIEE, Communs. and Electronics*, no. 31, pp. 263–267, July, 1957.

Slutz, R. J., A. W. Holt, R. P. Witt, and D. C. Friedman: Diode Capacitor Memory, *NBS Circ.* 551, Computer Development at the NBS, January, 1955, pp. 102–107.

Smith, C. V. L.: "Electronic Digital Computers," McGraw-Hill Book Company, Inc., New York, 1959.

Wanlass, C. L.: Transistor Circuitry for Digital Computers, *IRE Trans. on Electronic Computers*, vol. EC-4, pp. 11–16, March, 1955.

Wier, J. W.: Some Direct-coupled Circuits Utilizing N-P-N and P-N-P Transistors in Combination, *Univ. Ill., Digital Computer Lab., Rept.* 65, Aug. 31, 1955.

Yourke, H. S., and E. J. Slobodzinski: Millimicrosecond Transistor Current Switching Techniques, *Proc. Western Joint Computer Conf.*, February, 1957, pp. 68–72.

———: Millimicrosecond Transistor Current Switching Circuits, IRE-AIEE Transistor and Solid-state Circuits Conference, February, 1957, Philadelphia.

d. Articles on Other Semiconductor Devices

Aldrich, R. W., and N. Holonyak, Jr.: Multiterminal P-N-P-N Switches, *Proc. IRE*, vol. 46, p. 1236, June, 1958.

Chang, K. K. N.: Low-noise Tunnel Diode Amplifier, *Proc. IRE*, vol. 47, pp. 1268–1269, July, 1959.

Early, J. M.: P-N-I-P and N-P-I-N Junction Transistor Triode, *Bell System Tech. J.*, vol. 33, p. 519, May, 1954.

Grinich, V. H., and I. Haas: Application of PNπN Triode Switches, Digest of Technical Papers, pp. 52–53, IRE-AIEE Solid-state Circuits Conference, 1959, University of Pennsylvania.

Hamilton, D. J., J. F. Gibbons, and W. Shockley: Physical Principles of Avalanche Transistor Pulse Circuits, Digest of Technical Papers, p. 92, IRE-AIEE Solid-state Circuits Conference, 1959, University of Pennsylvania.

Kingston, R. H.: Solid-state Maser Amplifiers and Their Applications: A Survey, Digest of Technical Papers, p. 74, IRE-AIEE Solid-state Circuits Conference, 1959, University of Pennsylvania.

Mackintosh, I. M.: The Electrical Characteristics of Silicon P-N-P-N Triodes, *Proc. IRE*, vol. 46, p. 1229, June, 1958.

Moll, J. L., et al.: P-N-P-N Transistor Switches, *Proc. IRE*, vol. 44, no. 9, p. 1174, September, 1956.

Rupprecht, E. G., H. J. Patterson, and P. Miller: Hyperfast Diffused-silicon Diode and Transistor for Logic Circuits, Digest of Technical Papers, p. 72, IRE-AIEE Solid-state Circuits Conference, 1959, University of Pennsylvania.

Schwartz, S. J., and J. S. Sallo: Electrodeposited Transistor and Bit Wire Components, Digest of Technical Papers, p. 94, IRE-AIEE Solid-state Circuits Conference, 1959, University of Pennsylvania.

Shockley, W.: Unique Properties of the Four-layer Diode, *Electronic Inds. Tele-Tech*, August, 1957.

——— and J. F. Gibbons: Introduction to the Four-layer Diode, *Semiconductor Products*, January–February, 1958.

Wallmark, J. T., and S. M. Marcus: Integrated Devices Using Direct-coupled Unipolar Transistor Logic, Digest of Technical Papers, p. 58, IRE-AIEE Solid-state Circuits Conference, 1959, University of Pennsylvania.

e. Selected Bibliography on the Tunnel Diode

Chang, K. K. N.: Low-noise Tunnel-diode Amplifier, *Proc. IRE*, vol. 47, pp. 1268–1269, July, 1959.

Chow, W. F.: Tunnel-diode Digital Circuitry, Digest of Technical Papers, 1960 International Solid-state Circuits Conf., February, 1960, pp. 32–33.

Dacey, G. C.: Properties of Esaki (Tunnel) Diodes: A Survey, Digest of Technical Papers, 1960 International Solid-state Circuits Conf., February, 1960, pp. 6–7.

Esaki, L.: New Phenomenon in Narrow Ge *p-n* Junctions, *Phys. Rev.*, vol. 109, pp. 603–604, Jan. 15, 1958.

Lesk, I. A., and N. Holonyak, Jr.: Germanium and Silicon Tunnel Diodes—Design, Operation and Application, *IRE Wescon Conv. Rec.*, vol. 3, pt. 3, pp. 9–31, 1959.

Lewin, M. H.: Negative Resistance Elements as Digital Computer Components, *Proc. Eastern Joint Computer Conf.*, pp. 15–27, Boston, Dec. 1–3, 1959.

———, A. G. Samusenko, and A. W. Lo: The Tunnel Diode as a Logic Element, Digest of Technical Papers, 1960 International Solid-state Circuits Conf., February, 1960, pp. 10–11.

Miller, J. C., K. Li, and A. W. Lo: The Tunnel Diode as a Storage Element, Digest of Technical Papers, 1960 International Solid-state Circuits Conf., February, 1960, pp. 52–53.

Neff, G. W., S. A. Butler, and D. L. Critchlow: Esaki (Tunnel)-Diode Logic Circuits, Digest of Technical Papers, 1960 International Solid-state Circuits Conf., February, 1960, pp. 16–17.

Price, P. J., and J. M. Radcliffe: Esaki Tunneling, *IBM J. Research Develop.*, vol. 3, pp. 364–371, October, 1959.

Sklar, B.: The Tunnel Diode—Its Action and Properties, *Electronics*, vol. 32, no. 45, pp. 54–57, Nov. 6, 1959.

Sommers, H. S., Jr.: Tunnel Diodes as High-frequency Devices, *Proc. IRE*, vol. 47, pp. 1201–1206, July, 1959.

———, K. K. N. Chang, H. Nelson, R. Steinhoff, and P. Schnitzler: Tunnel Diodes for Low Noise Amplification, *IRE Wescon Conv. Rec.*, vol. 3, pt. 3, pp. 3–8, 1959.

Tiemann, J. J.: Tunnel Diodes and Their Use as Multifunctional Circuit Elements, Digest of Technical Papers, 1960 International Solid-state Circuits Conf., February, 1960, pp. 8–9.

CHAPTER 21

MAGNETIC ELEMENTS IN DIGITAL-CIRCUIT DESIGN

21-1. Introduction

Magnetic Circuits. Magnetic elements can have high reliability, mechanical strength, small size, light weight, small power consumption, and relatively low cost. As such they are an important kind of component for digital circuits. After a brief review in this section of some of the laws of physics governing magnetic phenomena, we consider in the next section properties of those magnetic materials most frequently used in digital circuits. Application of magnetic components to amplification in diode-gated circuits is considered in Sec. 21-3. In Sec. 21-4 application of the magnetic components to the gating functions is described, including the multiaperture cores that give promise of digital circuits devoid even of diodes and transistors. Finally in Sec. 21-5 we consider modern developments still in experimental stages, the Cryotron operating on magnetic principles, and microwave gating operating on electromagnetic principles. The topics of this chapter by no means cover all the types of magnetic components or circuits that have been reported in the literature. Our purpose primarily is to introduce the reader to as large a variety of concepts as possible without embarking on too extensive a discourse. Each of the concepts studied has many possible variations; and it is hoped that the basic principles of the use of magnetic components in digital circuitry, illustrated by the examples chosen for this chapter, will enable the student to understand other techniques appearing in the literature that must be omitted here.

Review of Ampère's Law. A magnetic vector field is associated with every current, every moving charge. According to Ampère's law, the infinitesimal contribution dH to the *magnetic field* at a point P made by the infinitesimal length $\mathbf{dl}_I$ of wire carrying current I is given by the vector product

$$\mathbf{dH} = \frac{I\,\mathbf{dl}_I \times \mathbf{r}}{4\pi r^3} \tag{21-1}$$

where $\mathbf{r}$ is the position vector from $\mathbf{dl}_I$ to point P (see Fig. 21-1*a*). Then according to the right-hand rule (see Fig. 21-1*b*) the direction of dH is perpendicular to the plane of $\mathbf{dl}_I$ and $\mathbf{r}$. Consider, for example, the field at the center of a circular loop, carrying a current I (see Fig. 21-1c):

$$\mathbf{H} = \int_0^l \frac{I\,\mathbf{dl}_I \times \mathbf{r}}{4\pi r^3} = I \int_0^{2\pi} \frac{r\,d\alpha}{4\pi r^2}\mathbf{k} = \frac{I}{2r}\mathbf{k}$$

where $\mathbf{k}$ is a unit vector, here directed toward the reader perpendicular to the plane of the figure. For N turns, each carrying the current I,

$$\mathbf{H} = \frac{NI}{2r}\mathbf{k} \tag{21-2}$$

The current-related dimensions for $\mathbf{H}$ are thus *ampere-turns per meter.*

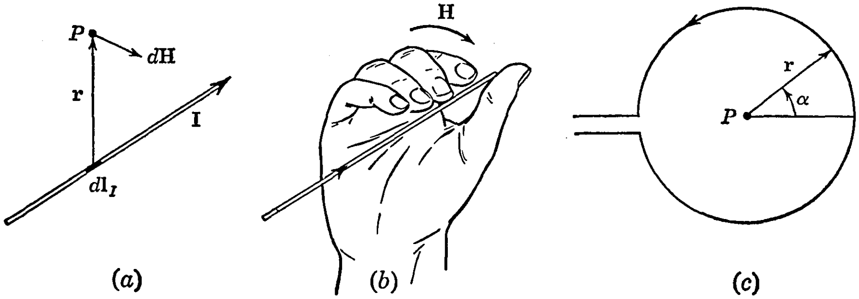


FIG. 21-1. (a) Ampère's law; (b) the right-hand rule for $\mathbf{H}$; (c) field in the center of a wire loop.

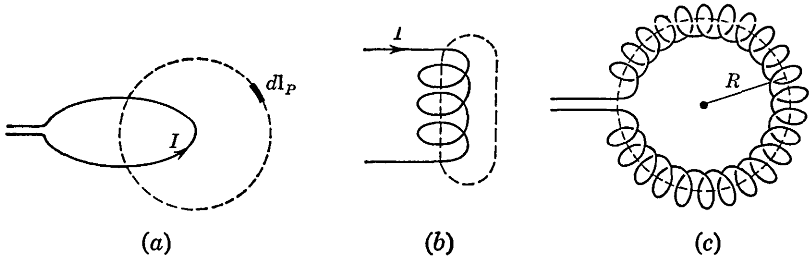


FIG. 21-2. (a) Path enclosing loop of wire; (b) path enclosing N loops; (c) path inside toroid.

A concept closely related to the magnetic field is that of *magnetomotive force,* mmf (which is *not* a force at all but rather is associated with work or energy), defined as the line integral of the scalar product

$$\mathfrak{F}_{AB} = \int_A^B \mathbf{H} \cdot \mathbf{dl}_P \tag{21-3}$$

It can be shown that, if the line of integration is taken as a closed path around a current I, then

$$\mathfrak{F} = \oint \mathbf{H} \cdot \mathbf{dl}_P = I \tag{21-4}$$

and is independent of the particular path chosen, so long as it encloses I (see Fig. 21-2a). If for a coil of N turns carrying a current I, the closed path is taken through the center and then round the outside of the coil (see Fig. 21-2*b*), then from Eq. (21-4)

$$\mathcal{F} = NI \tag{21-5}$$

Thus $\mathcal{F}$ can have the current-related dimensions of *ampere-turns.* Consider, for example, the path taken inside a toroidal coil (see Fig. 21-2*c*) of radius R such that H (the magnitude of **H**) is constant along the path. Then from Eq. (21-3), $\mathcal{F} = Hl$; but, from (21-5), $\mathcal{F} = NI$; hence $H = NI/l$, where $l = 2\pi R$.

Review of Faraday's Law. A magnetic field can be represented by lines whose directions at all points are those of the magnetic field vectors at those points. By convention the number of these lines crossing a unit area taken normal to their direction is made equal to the magnitude of the field vector. Such lines are called *lines of induction,* or *lines of flux.* The total number of lines of induction passing through a surface is called the *magnetic flux* through the surface. Consider a coil of N turns enclosing magnetic flux Φ (see Fig. 21-3*a*); if the flux is varied with time at a rate $d\Phi/dt$, then according to Faraday's law a voltage V is induced across the windings, of magnitude

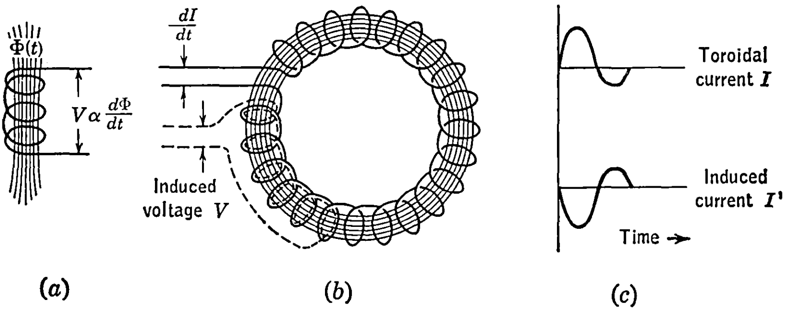


FIG. 21-3. (*a*) Flux change through coil; (*b*) toroid with secondary coil; (*c*) induced current direction.

$$V = -N\frac{d\Phi}{dt} \tag{21-6}$$

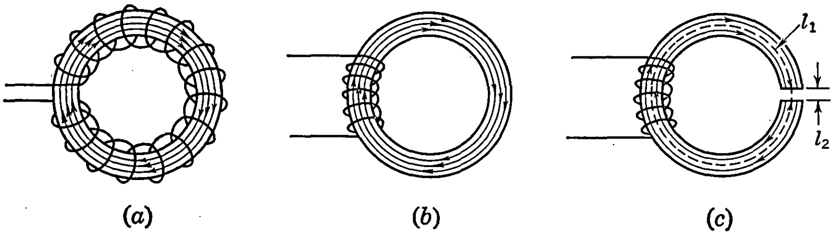
Note that, since H is given in ampere-turns per meter, we could give Φ in ampere-turn-meters; but then V would become something different from a volt. Because of this, Φ is more often given in dimensions of *volt-seconds per turn,* called the *weber.* Thus 1 volt is induced across one turn enclosing a flux change of 1 weber/sec. Since **H** is essentially flux density (flux per normal unit area), we can now express it in terms of *webers per square meter.* When expressed in these units, it is called the *magnetic induction,* or *flux density,* and is denoted by **B**. Thus $\mathbf{B} = \mu_0\mathbf{H}$, where μ_0 is a constant of proportionality between the different kinds of units (i.e., webers per square meter and ampere-turns per meter);

and in fact $\mu_0 = 4\pi \times 10^{-7} = 1.257 \times 10^{-6}$ weber/amp-turn-m (for a vacuum; see below).

Note that the flux density inside the toroid of Fig. 21-2*c* is $B = \mu_0 NI/l$. If the radius of the loops is r, then the flux in the loop is $\Phi = BA$, or

$$\Phi = \frac{\mu_0 ANI}{l} = \frac{NI}{l/\mu_0 A}$$

where $A = \pi r^2$. If we place a second coil of M turns inside the toroid (see Fig. 21-3*b*), then the voltage induced across this second coil, due to a rate of change of current equal to dI/dt in the toroid coil, becomes $V = -M\, d\Phi/dt = -MN\, (dI/dt)/(l/u_0 A)$.



(*a*) (*b*) (*c*)

FIG. 21-4. Coils with cores showing flux lines.

The direction of the induced voltage is given by Lenz's law, which states that the direction of an induced voltage is such as to oppose the cause producing it. Thus, as the flux inside a coil increases, the direction of the voltage induced must be such that a resulting current will produce a flux tending to *decrease* the imposed flux change; and as the imposed flux decreases, the resulting current will produce a flux tending to *increase* the imposed flux. For example, in our toroid when dI/dt is positive the induced dI'/dt is negative, and vice versa (see Fig. 21-3*c*).

Ferromagnetism. When certain materials, such as iron, nickel, cobalt, and their alloys, are placed in a magnetic field **H**, the flux density inside the material becomes greater than the applied field. This phenomenon is due to subatomic currents associated with electron spin, nuclear spin, or planetary motion in atoms. When an external field is applied, the subatomic currents are lined up so that their fields are in the same direction and the vector sum of these fields produces the observed phenomenon. The amount of this additional flux density is (roughly) proportional to the applied field. Thus the total flux density inside the material becomes

$$\mathbf{B} = \mu_0\mathbf{H} + \chi\mathbf{H} = (\mu_0 + \chi)\mathbf{H} = \mu\mathbf{H} \tag{21-7}$$

where χ is called the *magnetic susceptibility* of the material and μ is called the *permeability* of the material. The constant μ_0 introduced above is called the permeability of space. For example, if a wire of N windings carries a current I around a ferromagnetic core of susceptibility χ (see Fig. 21-4*a*), then $B = \mu_0 NI/l + \chi NI/l = \mu NI/l$. For the flux we have $\Phi = BA = NI/(l/\mu A)$. Even if the windings are located only on one part of the core, since the permeability of the core is so much greater than that of the surrounding space, most of the flux is still confined inside the core (see Fig. 21-4*b*). If a *small* air gap were made in the core, most of the flux lines would cross the gap and enter the material on the other side (see Fig. 21-4*c*) and the flux would be almost the same all around the ring.

Reluctance. Let us assume that the flux in Fig. 21-4*c* is confined within the ring, which has a constant cross-sectional area A_1, and within a constant area A_2 across the air gap. Choosing within the ring as a path of integration a line of constant magnetic field intensity H_1, Eq. (21-3) gives

$$\mathfrak{F}_1 = H_1 l_1$$

where l_1 is the length of the line within the metal. Integrating along the continuation of this line (having constant field intensity H_2) in the air gap, we find

$$\mathfrak{F}_2 = H_2 l_2$$

where l_2 is the length of the line within the gap. By Eq. (21-4)

$$\mathfrak{F} = \mathfrak{F}_1 + \mathfrak{F}_2 = H_1 l_1 + H_2 l_2$$

Since the total flux Φ is constant across A_1 or A_2, we have

$$\Phi = B_1 A_1 = B_2 A_2$$

or $B_1 = \Phi/A_1$ and $B_2 = \Phi/A_2$. Applying Eq. (21-7), we find $H_1 = \Phi/\mu_1 A_1$ and $H_2 = \Phi/\mu_2 A_2$, with μ_1 and μ_2 the permeabilities of the ring and air gap, respectively. Substituting these values in the above equation and solving for Φ, we have

$$\Phi = \frac{\mathfrak{F}}{l_1/\mu_1 A_1 + l_2/\mu_2 A_2} = \frac{\mathfrak{F}}{\mathfrak{R}_1 + \mathfrak{R}_2}$$

where $\mathfrak{R}_i = l_i/\mu_i A_i$ is called the *reluctance* of the path. As with resistances, the reluctances of series paths add, and the reciprocals of the reluctances of parallel paths add.

21-2. Magnetic Properties and Materials for Digital Circuits

Square-hysteresis-loop Materials. Consider a plot of **B** versus **H** for a magnetic core made of "square-loop" material (see Fig. 21-5). As the field **H** is applied, the flux through the core reaches a maximum, beyond which additional **H** will not produce appreciably more **B**. In this state practically all the subatomic currents are lined up, and the core is said to be *saturated* (see point *a* in Fig. 21-5). If the field is decreased to zero, the core will retain most of its flux, i.e., will behave as if it were a permanent magnet. The flux density B at this point (*b* in Fig. 21-5) is called the *retentivity*, or *remanence*. As H is decreased still more (i.e., as **H** is reversed in direction), the flux will suddenly start to drop rapidly (point *c*). The magnitude of **H** for which **B** = 0 (point *d*) is called the *coercive force,* or *coercivity.* The direction of the flux density will now change, and the core will again become saturated, but oppositely to the previous saturation state (point *e*). Further decrease of **H** will not appreciably change **B** (point *f*). If **H** is now increased, the plot will follow the lower line of Fig. 21-5 (points *f, g, h, i, j, a*), passing through the retentivity, or remanence, for this state (point *g*) and the coercivity (point *i*) for positive **H**. Thus the core can have two remanent states, which in binary circuits represent a unit (e.g., point *b*) and a zero (e.g., point *g*). The processes of changing **H** so that the core goes from one remanent state to the other is called *flipping,* or *switching, the core.* (If the coercive force is not exceeded, partial flipping of the core may occur—see the dashed lines of Fig. 21-5.) The reader should remember that not all magnetic materials exhibit such a square hysteresis loop.

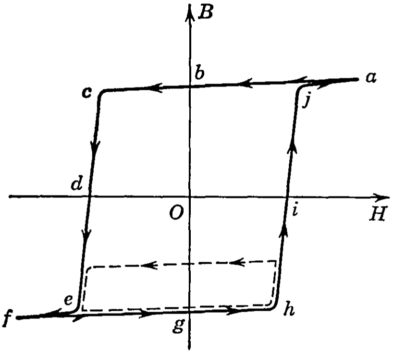


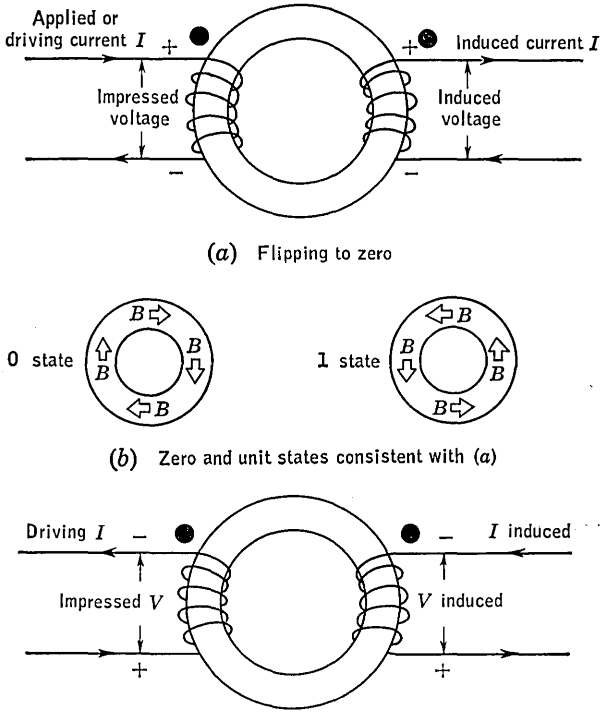
FIG. 21-5. Square hysteresis loop.

The most commonly used cores are made of *ferrite* materials, molded ceramics composed of iron oxide and other metallic oxides. The powdered ceramic is pressed into toroidal molds and sintered at around 1300°F until proper crystallization has occurred. Such cores are manufactured in a variety of sizes, for memories some as small as 0.05 in. in diameter. Cores have also been made with ultrathin (1/8 mil) metallic tapes wound on ceramic or steel bobbins, the layers reducing eddy currents. The ferrites are nonconductors, and therefore their eddy currents are automatically kept small. Crystal structure and uniformity of composition are important factors in obtaining a square hysteresis loop.

It has been shown that there exist in ferromagnetic substances small regions called *domains.* Within each domain all the electronic spins are parallel to each other—i.e., the domain is saturated. Adjacent domains have different directions of spins, i.e., of intrinsic **B**. The region between two domains is called the *domain wall.* Here the spin directions gradually change as the wall is crossed from one domain to another. In unmagnetized material the domain may have a linear dimension of 0.1 mm; the wall thickness is about 10^{-4} mm. When a field is applied, the domains originally oriented in the direction of the field begin to grow larger at the expense of the other domains; this is accomplished by the movements of the domain walls. Another process, called *domain rotation,* may also occur, where all the spins of a domain simultaneously rotate to the direction of the applied field.

Flipping the Cores. The dot notation for core windings is a very important point for an understanding of core circuits. The factors involved are (1) the zero or unit state of the core, (2) the primary (energized) coil and the secondary (induced) coil, and (3) the direction of the current in (i.e., polarity of the voltage across) each coil (see Fig. 21-6*a* and *b*). The dot notation organizes these factors according to the following conventions:

1. *Flipping to zero* (Fig. 21-6*a*). The driving current I or impressed voltage V will tend to flip the core to the zero state when the current enters the lead adjacent to the dot. When a core is flipped to the zero state, the induced current in the secondary coil will leave the coil at the lead adjacent to the dot as the induced voltage makes that lead more positive than the other.



(*a*) Flipping to zero

(*b*) Zero and unit states consistent with (*a*)

(*c*) Flipping to unit

FIG. 21-6. Dot convention for flipping cores.

2. *Flipping to unit* (Fig. 21-6*c*). The driving current I or impressed voltage V will tend to flip the core to the unit state when the current leaves the dot lead. When a core is flipped to the unit state, the induced current of the secondary coil will enter the coil at the dot lead, as the induced voltage makes the dot lead more negative than the other.

If the current is such as to drive a core into the state which it already occupies, evidently no flipping of the core will result, and hence no voltage will be induced in the secondary coil.

Consider a two-winding core (such as in Fig. 21-6*a*) which has a constant-current pulse source applied to its primary coil to switch it. Figure 21-7 illustrates the shapes of the resulting flux and corresponding induced-voltage curves, as functions of time, during the switching process. Of course the induced voltage appears in each of the coils, as a counter emf in the primary and as an induced emf in the secondary. Observe that there is an initial rise in the flux, Φ_i, and a peak in the voltage associated with the initiation of the current from the source. As the domains become aligned, the flux increases, with its rate of change being reflected in the magnitude of the induced voltage. When the saturation flux Φ_s is obtained, there is no further increase in the flux and the voltage drops to zero. When the current is turned off, the flux is slightly reduced to the remanent value Φ_r, and a corresponding spike appears in the voltage. If the current source now reverses direction, the core flips back to its original state.

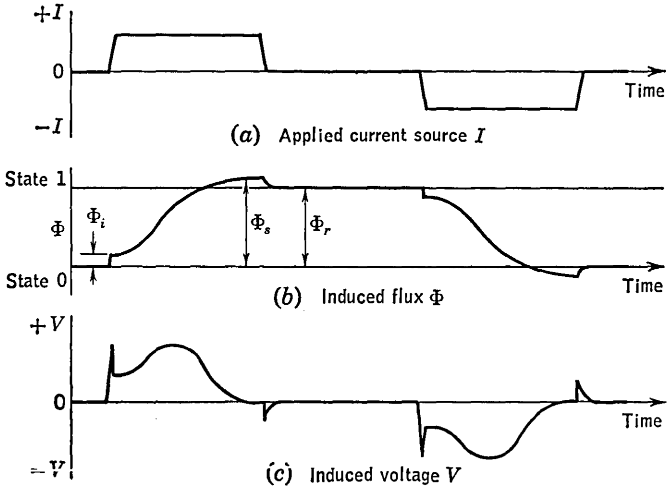


FIG. 21-7. Flipping the core with a constant-current source.

When a voltage source V is applied to the driving coil (see Fig. 21-8), there is an initial current rise of I_0 through the coil, until point *c* or *h* of Fig. 21-5 is reached. Then the switching of the core begins, and the counter emf in the driving coil limits the current flow to I_s (see Fig. 21-8*b*). During this time the coil appears as an approximately constant resistance R_c. Of course the voltage source must be connected with a resistor R in series with the coil, so that after the core is flipped (and R_c becomes zero) the current will rise only to a value $I = V/R$. Letting V_s be the voltage across the coil during switching, we have

$$\begin{aligned} V_s &= (I_s - I_0)R_c \\ \text{and} \qquad V &= IR = (I_s - I_0)R_c + I_sR \end{aligned} \tag{21-8}$$

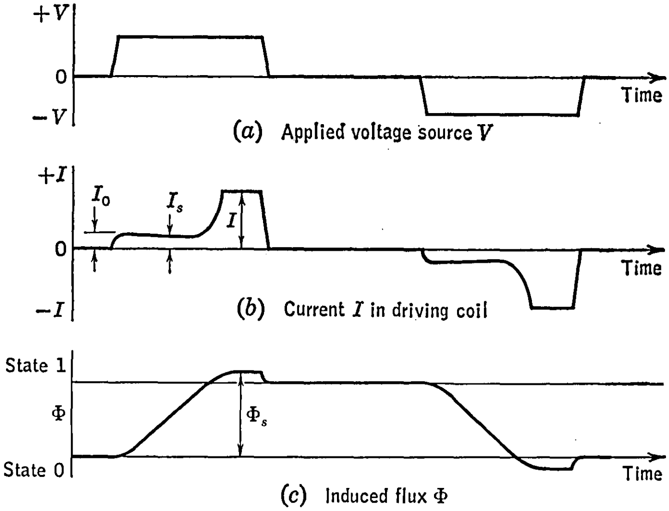
From these equations we find

$$V_s = \frac{RR_c(I - I_0)}{R + R_c}$$

and

$$I_s = \frac{V + I_0R_c}{R + R_c} \tag{21-9}$$

The core switching resistance R_c appears only during the time the core is switching, i.e., when $d\Phi/dt \gg 0$. Note that R_c is approximately constant in the constant-voltage case (i.e., the I_s part of the I curve of Fig. 21-8 is approximately flat), but the approximation is not so good for the constant-current case. (Why? See Fig. 21-7.)



(a) Applied voltage source V

(b) Current I in driving coil

(c) Induced flux Φ

FIG. 21-8. Flipping the core with a constant-voltage source.

The value of R_c can be estimated by means of considerations concerning the *switching time* of the core, the time τ required to switch or flip a core. This is frequently defined as the time required to switch the core through 80 per cent of Φ_s. It has been found experimentally that if a constant mmf $\mathfrak{F}_s$ (that is, constant NI_s) is used to switch a core, then the switching time is inversely proportional to the mmf $\mathfrak{F}_s$. Since the coercivity must be applied before the core switches, the plot of $1/\tau$ versus $\mathfrak{F}_s$ appears as in Fig. 21-9. The equation of this straight line is

$$\frac{1}{\tau} = \frac{\mathfrak{F}_s - \mathfrak{F}_0}{S} = \frac{N(I_s - I_0)}{S} \tag{21-10}$$

where S is the slope of the line. If $d\Phi/dt$ is considered constant during the switching (that is, R_c is constant), then $d\Phi/dt = 0.8\Phi_s/\tau$, whence from Eq. (21-6) $V_s = 0.8N\Phi_s/\tau$. Then by Eqs. (21-8) and (21-10) we find

$$R_c = \frac{0.8N^2\Phi_s}{S} \tag{21-11}$$

If R_c cannot be considered constant, then we can find an average R_c defined such that $\bar{R}_c = \bar{V}_s/(I_s - I_0)$. However, the average voltage during switching, by Eq. (21-6), is

$$\bar{V}_s = \frac{1}{\tau}\int_0^\tau V\,dt = \frac{0.8N\Phi_s}{\tau}$$

Then, from Eq. (21-10) we have once more

$$\bar{R}_c = \frac{0.8N^2\Phi_s}{S} \tag{21-12}$$

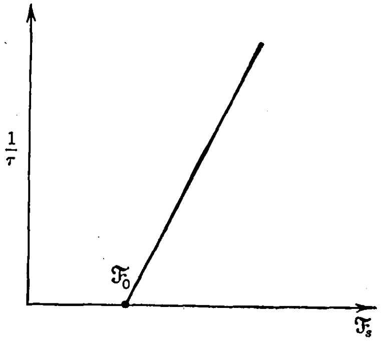


FIG. 21-9. Experimental plot of the reciprocal of the switching time τ versus the mmf $\mathfrak{F}_s$ ($\mathfrak{F}_0 = NI_0$ is the coercive mmf).

Getting back to the switching time, let us find the effect of the impressed voltage V in the case of constant-voltage drive, and of the driving current I in the case of constant-current drive. From Eqs. (21-10) and (21-9) we find

$$\frac{1}{\tau} = \frac{N}{S}\frac{I - I_0}{R + R_c}R \qquad \text{and} \qquad \frac{1}{\tau} = \frac{N}{S}\frac{V - RI_c}{R + R_c}$$

The constant-current case requires $R \gg R_c$, whence

$$\tau = \frac{S}{N(I - I_0)}$$

whereas for the constant-voltage case $R_c \gg R$, whence

$$\tau = \frac{S}{N}\frac{R_c}{N(V - RI_0)} = \frac{0.8N\Phi_s}{V - RI_0}$$

EXERCISES

(*a*) Prove Eqs. (21-9) from Eqs. (21-8).

(*b*) If, for a core, $S = 0.66 \times 10^{-4}$ (amp-turn)(sec)/m, $N = 50$ turns, $I_0 = 3$ ma, and $\Phi_s = 4 \times 10^{-8}$ weber, find R_c in ohms.

(*c*) Find the switching time τ for the core of Exercise *b* when a current source $I = 10$ ma is applied; when a voltage source of 10 volts through a resistance of 2,000 ohms is applied.

21-3. Diode-gated Magnetic-amplifier Circuits

Problems of Digital Magnetic Circuits. A discussion of an elementary core shift register will display the three important problems that arise in connection with magnetic circuits, namely, (1) separation of the *input and output phases* of each core stage, (2) *signal amplification,* and (3) *signal isolation* of a core from its adjacent stages. In the operation of a core the initial state is a zero; the core is flipped for a unit input and remains unchanged for a zero input. To sense the bit stored in a core, an attempt is made to flip it to the zero state: if an output signal is induced in a sensing coil, a unit was stored; no signal indicates that a zero was stored. Figure 21-10 illustrates the elementary register to be described, which has *two* cores per register bit. Each core of our shift register has three windings: an input coil that will flip the core if a unit signal arrives from the previous stage; a clock coil that is driven by a constant-current source to try to flip the core to zero; and an output coil that senses if the core was actually flipped by the clock. Note that the A and B clocks operate in alternate cycles. The simultaneous current pulses from the A clocks will produce output signals from stages 0 and 2, as these stages are "cleared" to the zero state. These output signals become the respective input signals to stages 1 and 3. When clocks B are activated, stages 0 and 2 are cleared and are in their *input phases,* while stages 1 and 3 are in their *output phases.* Because the cores are alternately cleared as the information is shifted from stage to stage, the register requires two cores per bit.

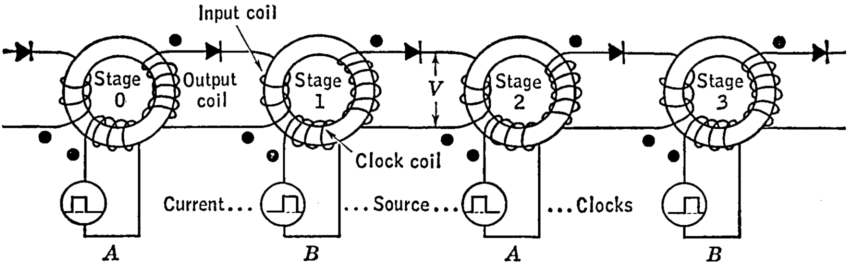


FIG. 21-10. Elementary core shift register, two cores per bit.

Assume that the B clocks are energized and that the stage 1 core is thereby flipped from the unit state to the zero state. Let the switching flux *change* be Φ_1; then a voltage $V(t) = N\,d\Phi_1/dt$ will be induced in the N turns of the stage 1 output coil. This voltage will be impressed across the M turns of the stage 2 input coil, flipping it to the unit state. But as core 2 flips, a voltage is induced across its clock and output coils. However, the A clocks are off, and no current can flow in the clock coil; and the induced voltage in the output coil reverse-biases the diode so that no current will flow. Thus during the flipping of the stage 2 core by the stage 1 core, the rate of induced flux is $d\Phi_2/dt = V(t)/M$. To find the total induced flux, observe that

$$\int_0^\tau V\,dt = N\Phi_1 = M\Phi_2$$

If $M < N$, then $\Phi_2 > \Phi_1$. This amplification of the signal is necessary

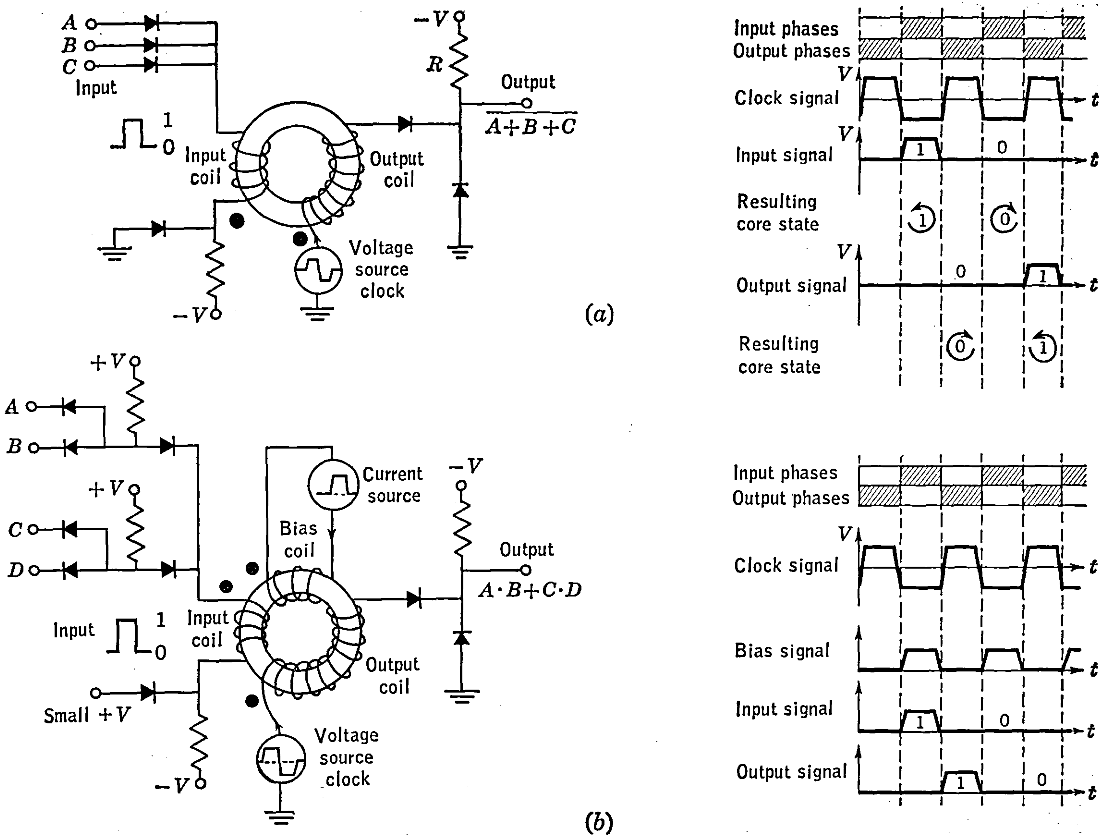


FIG. 21-11. Series clock-pulse core logic. (*a*) *or*-inverter gate; (*b*) direct instead of inverted output.

to ensure that the stage 2 core will flip even if there is some resistive loss and even if the clock and output coils of stage 2 do have some effect.

We have seen how stage 3 is isolated from stage 2 by the diode when the latter stage is flipped by stage 1. The problem still remains as to the effect of stage 1 on stage 0, for as the B clock flips stage 1, a voltage will be induced in the *input* coil of stage 1 and the resulting current will be in the forward direction of the diode. Since current will flow, clearly flux will be induced in the core of stage 0, but the induced flux Φ_0 will be only a fraction of Φ_1, since $\Phi_0 = (M/N)\Phi_1$. Actually Φ_0 will be even smaller because of resistive loss and because of some small effect of the input and clock windings on the core of stage 0, and only partial reversal of the flux will occur. But stage 0 is also receiving an input at this time; if the input is a unit, no problem exists, for the core will be flipped the rest of the way; if a zero input occurs, the partial flux reversal will remain. Now consider the effect of this partial reversal when clock A is energized. Some small voltage will be induced in the output coil of stage 0. There is danger of this being amplified by the ratio N/M; in practice this does not happen, for the coercive force must first be met, and also the diode will still have appreciable resistance at low forward biases. Thus a small voltage will not be amplified, whereas, as we have seen above, a large voltage will be amplified.

Series Clock-pulse Core Logic. In the shift register, signal amplification was obtained by having a smaller number of turns on the input coil than the output coil. Although this proves adequate for driving one core, other techniques must be employed to drive more than one. Suppose, for example, that the core of Fig. 21-11a is in the zero state. A positive voltage swing of the clock will tend merely to drive the core further into the zero state and there will be little change of flux in the core. Hence the output coil will appear as little or no resistance, and the clock will essentially be connected directly to the output. The output voltage will very nearly follow the positive clock voltage. On the other hand, if the core is in the unit state a positive voltage swing of the clock will flip the core back to the zero state. As we have seen in the previous section, the output coil will then appear as a resistance through which the clock is connected to the output. Now if $R_c \gg R_1$, most of the voltage drop from the clock voltage to $-V$ will appear across the output coil and the output voltage will not rise appreciably. In this way the core acts as a resistance that can be switched in or out, and the output is driven directly by the voltage-source clock (and not by the core). This type of amplifier is generally called a *Ramey* magnetic amplifier. Note that in this circuit, as in the shift register above (and the circuits below), the input phase of one stage is the output phase of the preceding stage, and thus there is necessarily a half-unit-time-interval delay between successive gates.

Let us now consider the entire circuit of Fig. 21-11a. Here gating is accomplished in the usual fashion by the input diodes. At the start of the input phase the core is always in its zero state. During the input phase a positive pulse arriving from A or B or C will flip the core into its unit state. Note that we must distinguish between zero and unit *signal levels* and zero and unit *core states.* We have chosen a positive signal pulse to represent a unit signal level: thus the core is flipped into its unit state by a unit pulse. During the input phase, however, the clock voltage will be negative, reverse-biasing the output-coil diode, so that no current flows in the output coil. During the output phase the output coil flips the core back to the zero state; the output voltage remains close to ground, and a zero output signal results. Thus the circuit acts as an *or* inverter. As the core is flipped, the input coil reverse-biases the input diodes and no current flows. The grounded diode near $-V$ on the input coil assures that the input diodes will be truly reverse-biased (why?).

The circuit can be used to produce an uninverted output by reversing the input coil and using a bias coil connected to a current source (see Fig. 21-11*b*). During the input phase the bias coil tends to flip the core to its unit state. If there is a unit input signal (i.e., positive signal), then the input coil will drive against the bias coil, tending to keep the core in its zero state. The result will be no net change in the flux of the core, and the core will remain in the zero state. During the output phase the output-coil resistance will be negligible and a positive (unit) signal pulse will result. However, for all-zero input signals (near ground voltage level), the bias coil will succeed in flipping the core to its unit state; then during the output phase the output coil will present its switching resistance, and a zero output signal will result.

Parallel Clock-pulse Core Logic. In the above circuit the voltage source is connected in series with the output coil. Magnetic amplification can also be obtained by connecting the voltage source in parallel with the input coil (see Fig. 21-12). In the input phase, if all the previous cores connected to an *and* gate are in the unit state, then they will present a (divided) switching resistance between the +40- and +1-volt supplies, so that a unit signal will pass through the *or* gate. But if at least one of the cores of each *and* gate is in the zero state, each *and* gate then presents a small resistance, the inputs to the gates will all fall to near 1 volt, and the output of the *or* gate will be a zero signal. Note that the previous cores of the active *and* gates are flipped back to the zero state by the current from the 40-volt supply (in part—see below).

During the input phase the voltage-source clock output will be positive. If the output from the *or* gate is a unit (a positive voltage at point S of the figure), a current will flow through the input coil of the core to ground, flipping the core to the unit state. If the output from the *or* gate is a zero, point S is near ground potential, the positive clock voltage reverse-biases its diode so that no current flows through the core input coil, and hence the core remains in the zero state.

During the output phase the voltage-source clock output swings negative, pulling down the voltage at S and with it the voltages at P and Q. The *and*-gate diodes isolate these negative voltages from the previous gates, which are now in their input phases. At the same time the clock voltage is applied to the input coil of this gate in a direction to flip the core to the zero state. If the core is left in that state, the flow of current will not be impeded by the core nor will the core state be altered. If the core is left in the unit state, the clock voltage on the input coil, together with the current in the output coil from the 40-volt supply if the output is connected to an active *and* gate, will flip the core back to the zero state, ready for the next input phase. The circuit of Fig. 21-12 is known as the *Hogue* magnetic amplifier.

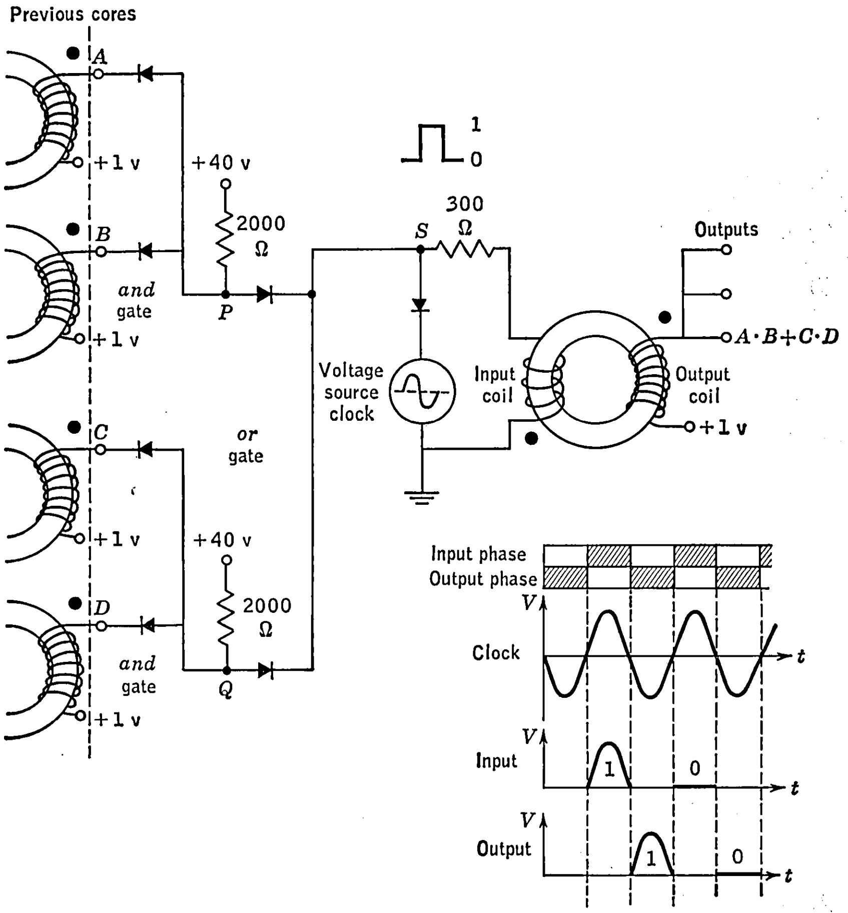


FIG. 21-12. Parallel clock-pulse core logic.

EXERCISES

(*a*) Why must a voltage source be used in the Ramey amplifier circuit (see Fig. 21-11)?

(*b*) Draw the states of the core of the Ramey amplifier after the input phase and after the output phase, for both zero and unit pulse inputs.

(*c*) Why is the small $+V$ needed in Fig. 21-11*b*? (HINT: Where will current flow in the *and*-gate–*or*-gate network during the output phase?)

(*d*) For the Ramey amplifier describe the state of the preceding core stage and the input coil of the present stage during the output phase of the present core circuit.

(*e*) For the Hogue amplifier (Fig. 21-12) describe the states of the preceding core stage and the input coil of the present stage during the output phase of the present core circuit.

(*f*) For the Hogue amplifier describe the effect on the next successive stage of flipping the core to a unit during the input phase.

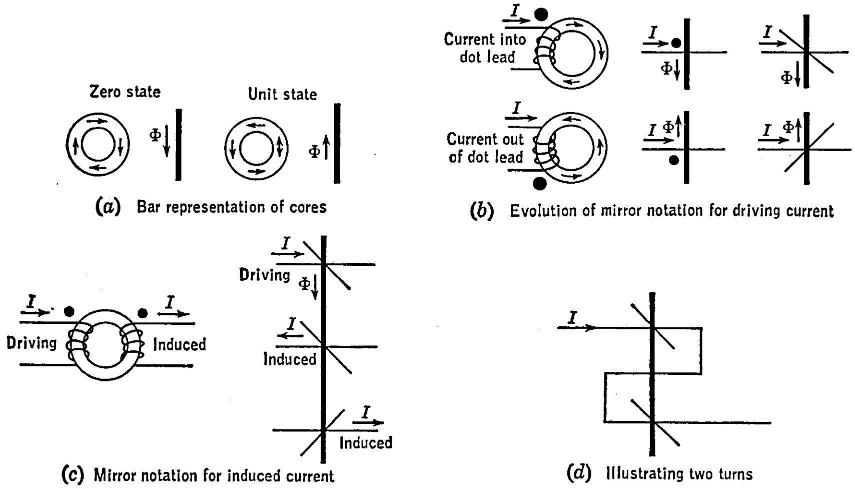
(*g*) Discuss the source or sources of amplification (power) in the Ramey and Hogue circuits.

(*h*) Devise a system for negation using the Hogue circuit. (HINT: Use an extra core.)

(*i*) Suppose in the Ramey amplifier of Fig. 21-11*a* that $V = -10$ volts, $R = 2{,}000$ ohms, the clock voltage source is a ±10-volt square wave at 200 kc, the core is as described in Exercise *b* of Sec. 21-2 with 50 turns on the input, 100 turns on the output windings, and inductances, respectively, of 2 and 4 μh. Calculate the maximum number of cores that can be driven. [HINT: Recall that $\int_0^t v\,dt = Li(t)$ when current flows through a coil and the core is not being flipped.]

21-4. Magnetic Gating

Gating with Cores. In the previous section we were concerned with the use of cores for magnetic amplification. We now consider the use of cores for the logical gating functions as well. But first we consider a new, shorter notation for core windings, the *mirror* notation. In this notation a core is represented by a vertical bar (see Fig. 21-13*a*); the unit and zero (counterclockwise and clockwise) flux directions of the core are imagined



(*a*) Bar representation of cores

(*b*) Evolution of mirror notation for driving current

(*c*) Mirror notation for induced current

(*d*) Illustrating two turns

FIG. 21-13. Mirror notation.

as upward and downward flux directions in the bar, respectively. A coil on the core is represented simply by an intersecting horizontal line (see Fig. 21-13*b*). A dot at the side of the bar could be used to represent the dot lead of the coil, as shown in the second column of Fig. 21-13*b*; the mirror notation replaces the dot by a diagonal line, slanted so that it acts like a mirror, "reflecting" the input-current direction into the resultant flux direction. For induced currents, see Fig. 21-13*c*, which shows the directions of the induced currents for both types of diagonal line. Frequently each intersection is considered a single turn, in which case more than one intersection must be used for several turns (see Fig. 21-13*d*).

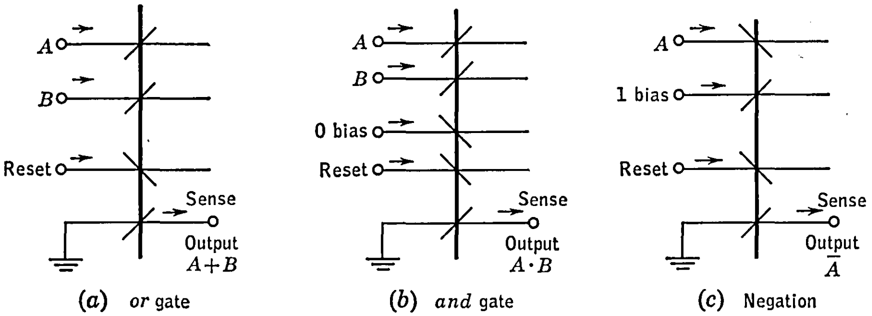
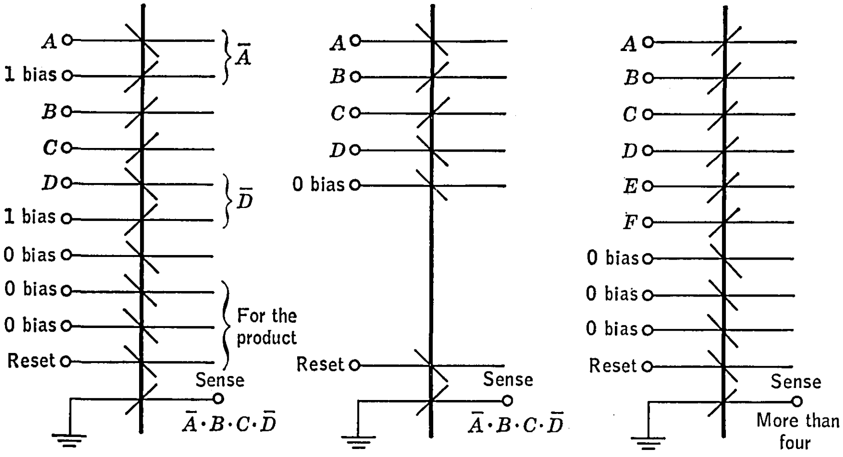


FIG. 21-14. Gating with cores.

Gating is accomplished in two phases, as was amplification. In the input phase the logical function is formed in the core, represented by the resulting zero or unit state of the core. During the output phase a reset winding is pulsed to reset the core to zero. A sense winding then determines whether the core flips or not, i.e., whether its state was a zero or unit; the current induced by flipping is a unit output. For example, consider the gating configurations of Fig. 21-14. In (*a*) is illustrated an *or* gate with inputs A and B. Either input A or input B alone provides sufficient current to switch the core to the unit state during the input phase. If both A and B occur, there will of course be twice as much current as necessary to switch the core, but the flux change will still be only to saturation, that is, Φ_s. During the output phase the reset pulse causes the sense wire to be energized if the core was left in a unit state, for an output $A + B$. In the circuit of (*b*) we introduce a new element, the bias wire whose current alone is sufficient to flip the core. During the input phase the bias works against inputs A and B, so that if the core is to be flipped to the unit state, both A and B must be energized. In the output phase the output will represent $A \cdot B$. Since this bias tends to change the core to the zero state, it is called a 0 bias. A 1 bias can also be used to produce negation, as shown in Fig. 21-14*c*. By the appropriate use of one or more bias wires any elementary Boolean product can be obtained. For example, Fig. 21-15*a* is a circuit generating

$\bar{A} \cdot B \cdot C \cdot \bar{D}$. Note that *for each negation* there is a corresponding 1-bias and, *for an n-term product*, $n - 1$ of the 0 biases must appear. However, each pairing of a 1 bias and a 0 bias cancels and hence can be eliminated, as shown in Fig. 21-15*b*. With less than $n - 1$ (gross) 0 biases the core logic becomes a logic of *majority* gating. For example, in Fig. 21-15*c* an output will appear if more than four of the inputs A, B, C, D, E, or F occur. Negations can be handled in the same manner as for the elementary products.



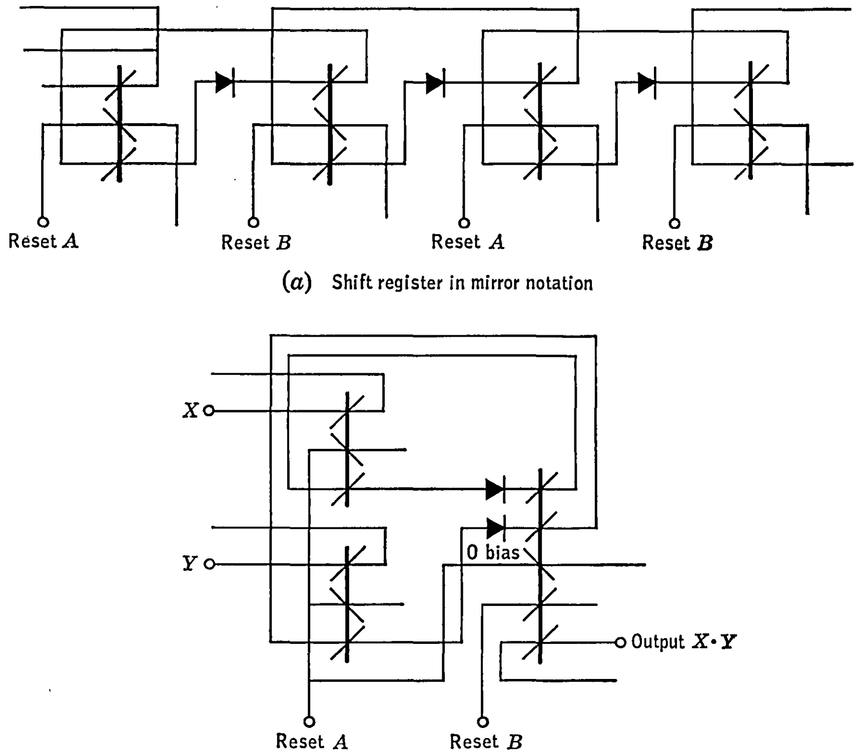
(*a*) Product (*b*) Simplified product (*c*) Majority

FIG. 21-15. Generating elementary products.

We have not yet considered the problems of isolation and amplification for our core-logic gating method. Adaptations of the Ramey or Hogue magnetic-amplifier circuits can be used, with a gating stage followed by an amplifying stage, and so forth. Or if the output need drive no more than a single subsequent core, the method used in the shift register of Fig. 21-10 can be employed. Figure 21-16*a* shows that shift register in the mirror notation, and Fig. 21-16*b* illustrates shifting into an *and* gate.

Gating with Flux. Note that when gating with cores we still had to use diodes for the transmission of information between cores. Gating by flux switching eliminates the necessity for using other than magnetic elements in the circuit: isolation, as well as a certain amount of amplification, can be obtained. Gating by flux switching is accomplished by means of multiaperture devices (abbreviated *MAD*), which are disks of square-loop ferrite material in which variously shaped holes have been made. The basic principle of a MAD can be described in terms of the two-hole device (called a *Transfluxor*) of Fig. 21-17. There are two windings, one through each hole as shown. If sufficient current I is passed through the larger hole, the direction of the flux (lines with arrowheads) will be the same all the way round. This is called the *clear* state.

In particular the flux in leg L_1 and leg L_2 will be saturated (Fig. 21-17*a*). A current in winding N_R can*not* change the flux locally round the small hole (see dashed lines), for this would imply an increase in flux in one of the already saturated legs L_1 or L_2. However, if the current in N_R is large enough, the flux can be reversed in leg L_1, but such a reversal *must* extend to the flux all the way round the large hole (see Fig. 21-17*b*), because the lines of flux must close on themselves.



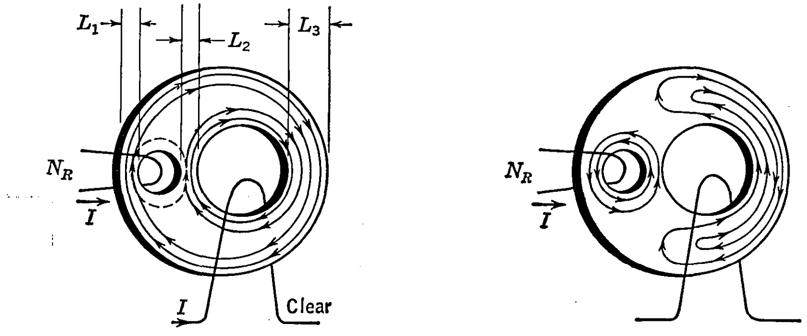
(*a*) Shift register in mirror notation

(*b*) Shifting into an *and* gate

FIG. 21-16. Use of shift-register method with core gating.

With the flux lines in this *set* state, a smaller current in N_R can now vary the amount of flux round the small hole (within the limits of saturation). For, in changing from the clear to the set state, the flux in the *long path* round the large hole has to be reversed; this requires a larger $\mathfrak{F} = N_R I = Hl$ than does just changing the flux in the *shorter path* round the small hole in the set state. To see how this occurs, consider the flux in leg L_1 of the MAD in the clear state (see point *a* of Fig. 21-18). As the current in N_R is increased, the flux goes from *a* to *b* to *c*, at which point the flux reversal begins, to the saturated state *d*, and on further driving to *e*. At and beyond *d* the MAD is in its set state.

Reversing the current in N_R, the flux of leg L_1 now follows the dashed line of Fig. 21-18, going to f, to g, from which point the flux in the smaller loop is switched, to h. The flux in leg L_1 can now be varied round the path h, (i), j, f, g by varying the current in N_R. Note that, in the set state, a small mmf $\mathfrak{F}_s$ will switch some flux in L_1, whereas in the clear state this will not occur (see point b' in Fig. 21-18), since then $\mathfrak{F}_c$ is needed.



(a) Clear (zero) state

(b) Set (unit) state

FIG. 21-17. Two-hole MAD (Transfluxor) showing *clear* and *set* states.

The operation of a three-hole MAD can be described in terms of the principles developed for the two-hole MAD. In the clear state, induced by a strong current through the "clear" wire, a small current in N_R or N_T cannot cause any flux change (see Fig. 21-19a). A large current in N_R will put the MAD in the set state (see Fig. 21-19b). Note that the reversal of flux takes place primarily in the *shortest* possible path round the large hole, through L_3 rather than L_4. This is to be expected: $\mathfrak{F}$ will roughly be inversely proportional to the distance from the wire, and as $\mathfrak{F}$ is increased, the coercive force for flux reversal is reached sooner round a shorter path than round a larger path. The exact geometry of the MAD thus becomes an important factor in its operation. In the set state a small current in N_T or N_R will be able to change the flux locally round its respective small hole.

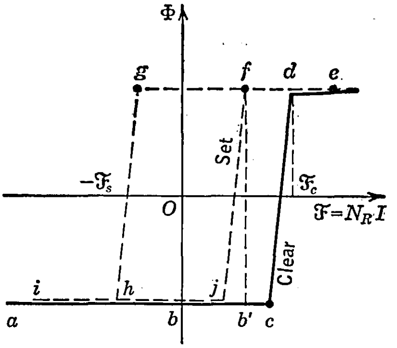
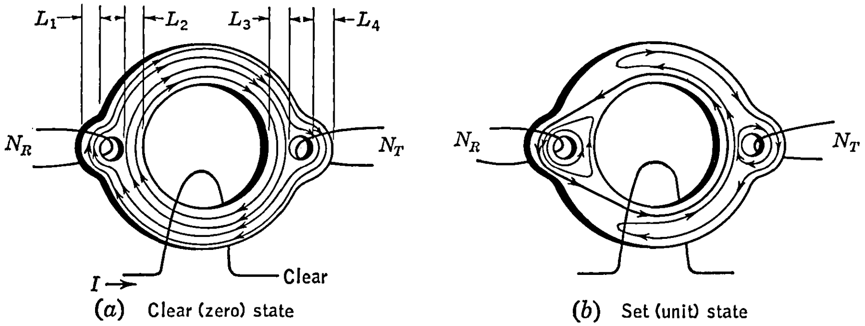


FIG. 21-18. Switching flux in L_1, clear and set states (solid line, clear state; dashed line, set state).

Note that a change in the flux round the small N_T hole will *not affect the state of the MAD*. This principle is used for *isolation* in MAD logic. Amplification is obtained as follows: Consider the coil N_T of one MAD connected to the coil N_R of another, as shown in Fig. 21-20. Let the set state represent the unit, and let the MAD receiving the unit initially be in the clear state. A relatively small current pulse I in the split winding shown in Fig. 21-20a will encounter a switching resistance in the



(*a*) Clear (zero) state (*b*) Set (unit) state

FIG. 21-19. Diagrammatic illustration of the flux lines in the clear and set states of the three-hole MAD.

set MAD; but low values of I will not affect the flux in the clear MAD, and no resistance will be developed. Hence the current will almost all pass through the clear MAD, finally rising to a level high enough to change the clear MAD into the set, or unit, state. This effect can be increased by making turns N_T greater in number than turns N_R. If a zero is to be transmitted (Fig. 21-20b), then both MAD's are in the clear state initially. The current pulse will be equally divided through N_T and N_R but will not be great enough in either to change states. Amplification results from this current-switching phenomenon. A shift register can be made on this principle (see Fig. 21-21), again using two cores (stages A and B) for each bit. There must now be four phases (the pulse wires of the figure) to shift each bit. First the B stages must be cleared, and then the A-to-B transfer pulse sets the B stages. Third, the A stages are to be cleared, and last the B-to-A transfer current pulse sets the A stages.

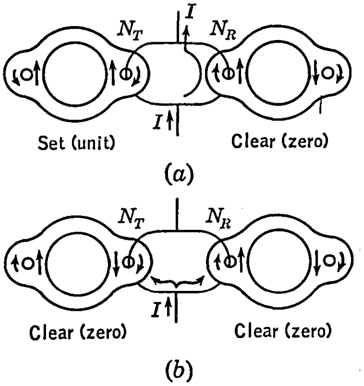


FIG. 21-20. (*a*) Transferring a unit; (*b*) transferring a zero.

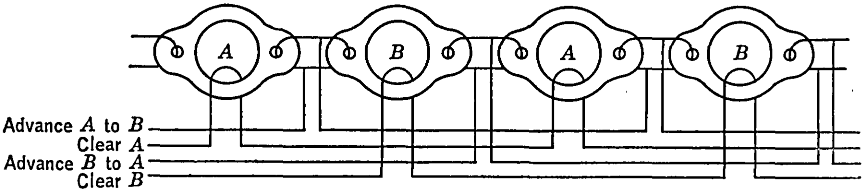


FIG. 21-21. MAD shift register.

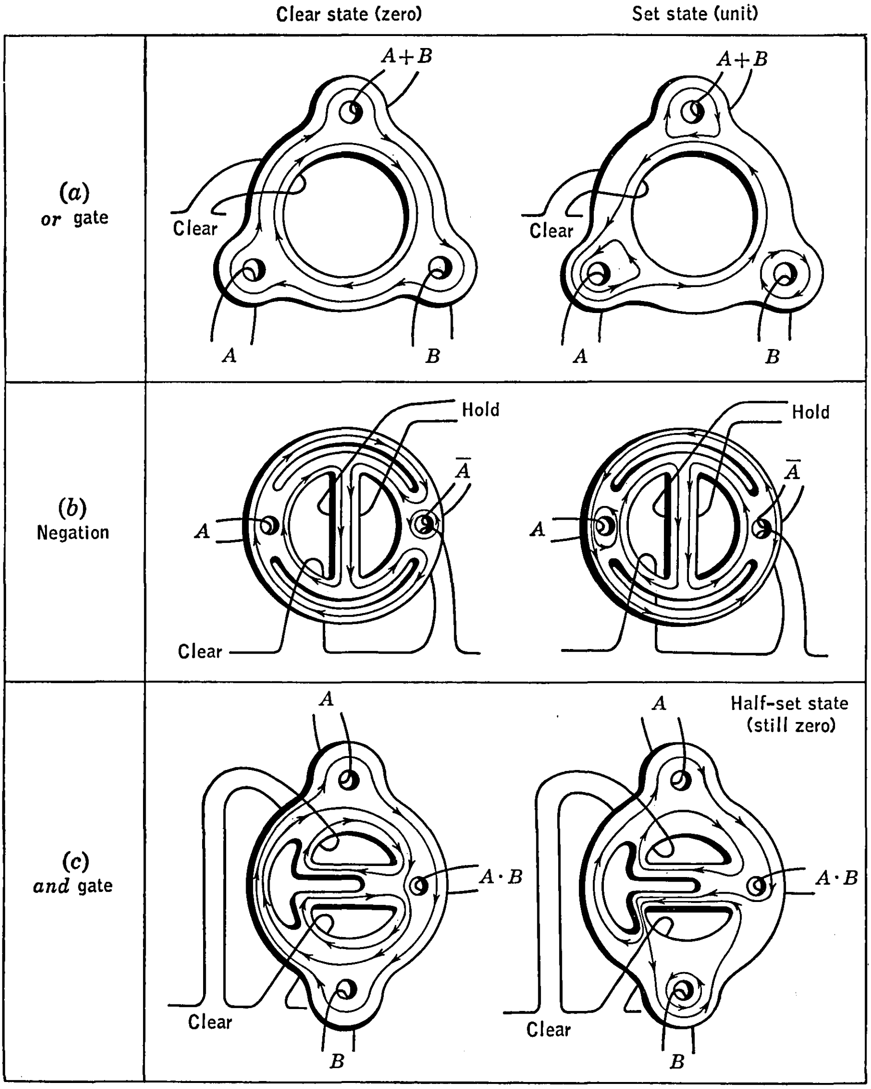


FIG. 21-22. Diagrammatic illustration of flux linkages in gating with flux.

Logical gating can be achieved in a single MAD element. Figure 21-22*a* represents an *or* gate. The zero state of an output aperture occurs when the flux is in the same direction in both legs about the aperture, i.e., when the flux does not link *locally* round the aperture. The unit state of an output aperture occurs when the flux does link locally round the aperture. If no unit (i.e., current pulse) is received in either A or B, the *or* MAD remains in the clear state; i.e., the output is a zero. But if an input is received at A, for example, the *shortest path* that will allow a flux change in the outside leg of aperture A is round the large hole as shown. This flux linkage leaves the output hole locally linked, and hence a unit output occurs. By symmetry it is seen that a unit output will result from $A + B$. Negation is a little more complicated—see Fig. 21-22*b*. Here the clear wire sets the output aperture to a locally closed state, i.e., a unit (note carefully the path of the clear wire). An input zero of course leaves this state unchanged, and the output is a unit. A unit input pulse, however, reverses the flux round the shortest possible path, the periphery of the core as shown. (Note that a *hold* wire keeps the flux in the cross leg in the direction shown, in both the clear and set states.) Thus the resulting flux round the output aperture does *not* link it, to form a zero output. An *and* gate (Fig. 21-22*c*) operates as follows: Consider a single unit input at B. A flux reversal in the outside leg of aperture B can occur by linking the nearest half of the center hole as shown. But the output aperture is still linked round the other half of the center hole and hence still remains a zero. It is clear that, if an input occurs at A *and* B at the same time, the output aperture will be linked locally.

EXERCISES

(*a*) Using the mirror notation, design a core circuit that will have an output if more than four of $\bar{A}$, B, C, $\bar{D}$, E, and $\bar{F}$ are inputs. Minimize the number of necessary wires.

(*b*) Making use of the majority feature of core gating, design a serial adder using only four cores (cf. Exercise *c* of Sec. 19-2).

(*c*) Draw a flux-linkage diagram for the case of A and B both being inputs to the MAD *or* gate.

(*d*) Draw a flux-linkage diagram for the case of A and B both being inputs to the MAD *and* gate, showing the unit output.

(*e*) Design a serial adder using MAD logical elements.

(*f*) How can single-aperture cores be used, in a way analogous to the use of multi-aperture devices, to form *and*, *or*, and *not* gates without the use of diodes or any other elements? (HINT: Use combinations of single-hole cores with *different hole diameters* wired appropriately together.)

21-5. Modern Computer Techniques

The Cryotron.† Cryogenics is the study of extremely low-temperature phenomena. *Cryotron* is the name given to a switching device whose operation is based on superconductivity, which occurs at extremely low temperatures. When certain elements are cooled to near absolute zero temperature, their electrical resistivity drops practically to zero. This transition into the superconductive state occurs at a sharply defined temperature, which is characteristic of the element but can be altered by an applied magnetic field. In general, the greater the magnetic field, the lower becomes the transition temperature This magnetic phenomenon is the basis for the Cryotron. Figure 21-23 shows the transition

† The Cryotron as a computer element was invented in 1955 by Dudley A. Buck.

temperature as a function of the applied magnetic field for tin (Sn) and lead (Pb), two metals capable of superconductivity. (Such low temperatures can be maintained in liquid helium.)

Figure 21-24 pictures the simplest of Cryotrons, consisting of crossed "wires" of tin and lead, operated at a temperature of about 2 to 3°K. When no current is flowing in the lead wire, both wires are superconducting. When current flows in the lead wire, the magnetic field created by this current flow forces the tin wire out of the superconducting state (see the dashed line in Fig. 21-23). The tin "wire" is made about ten times wider than the lead wire, to reduce its current density so that its magnetic field will not bring the lead wire out of the superconductive state. The lead wire, which is thus always in a superconducting state, is the controlling element; the tin wire, whose superconducting state is dependent on the current flow in the lead wire, is the controlled element. Because of the negligible resistance encountered, it is convenient to have a current source and to use the Cryotron as a current switching device. Figure 21-25*a* shows a Cryotron *and* gate; Fig. 21-25*b* shows an *or*-gate. In each circuit the current is switched to one or the other of the two outputs.

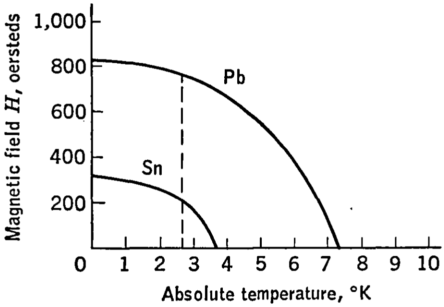


FIG. 21-23. Superconductor transition temperature as a function of applied magnetic field for Sn and Pb.

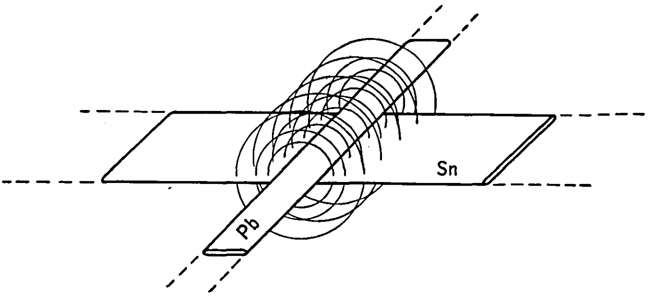
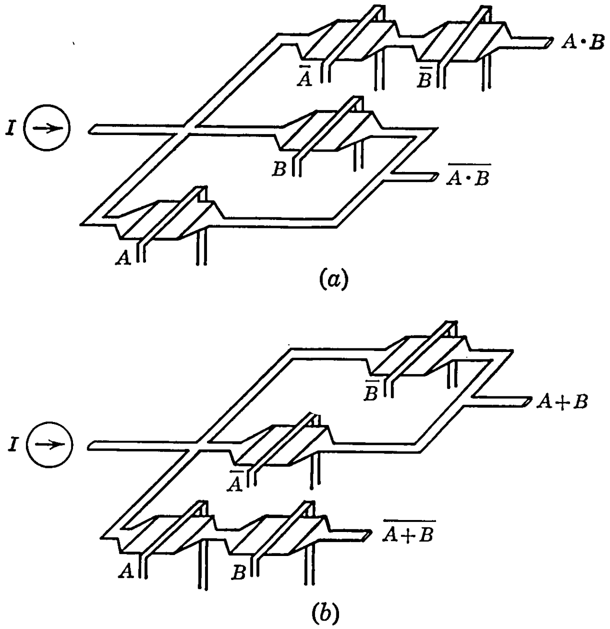


FIG. 21-24. A Cryotron.

In connection with Cryotrons it has been suggested that an electron-microscope beam be used to aid the construction of the circuit patterns from thin films. The use of film wires as illustrated may have the advantage of decreasing the required switching time to the millimicrosecond range. According to this suggested technique a thin film of Sn (or Pb) is placed as the object of the electron microscope and the electron beam moved to trace the desired path of a wire. It happens that, wherever the beam hits, a hard substance, namely, SiO_2, is formed from the tetraethoxysilane present in the beam chamber because of the oil-diffusion vacuum system. A small quantity of chlorine gas introduced into the chamber will etch the film, removing all metal not covered by SiO_2. Hence the trace of the electron beam is left as a flat wire. Finally, if desired, the SiO_2 can be removed by introducing a small amount of hydrogen fluoride gas into the chamber. It is expected that by this procedure "wires" of a diameter approximating 0.1 μ (10^{-5} cm) can be formed, several layers of such wires being formed over each other to complete the Cryotron gates. If this technique can be fully developed, then we can have over 100 million gates within 1 *square* centimeter. Such a process could significantly change the logical design concepts and introduce an entirely new capability into digital-computer engineering. However, such techniques are still in their early experimental stages.



(*a*)

(*b*)

FIG. 21-25. Cryotron *and* and *or* gates.

A Cryotron computer would operate inside a Dewar flask, or the like, filled with liquid helium, into which must lead the wires connecting the computer with the outside world. In the case of component failure there would be no way to repair the computer. This, coupled with the fact that a practically unlimited number of gates can be incorporated into the computer, indicates the possibly great effectiveness of the probabilistic-logic approach to increasing the reliability of gating circuit outputs (see Sec. 19-4).

Microwave Computer Components. Microwaves can be conveniently transmitted and gated by means of so-called *strip-line* components (see Fig. 21-26*a*). A strip line consists of a copper ground plane, covered by a layer of dielectric material, on top of which are placed strips of copper. Microwaves will be guided along the strip line, confined approximately between the ground plane and the top copper strip itself. It may be desirable to place another layer of dielectric on top of this and finally another copper ground plane; however, we do not illustrate this in our diagrams for the sake of clarity. Energy from a klystron or magnetron can be inserted into the strip line by means of a coaxial connection (see Fig. 21-26*b*). A line can be terminated by a wedge of carbon placed at its end to convert the electromagnetic energy into heat (see Fig. 21-26*c*). Diodes can also be placed in the line (see Fig. 21-26*d*).

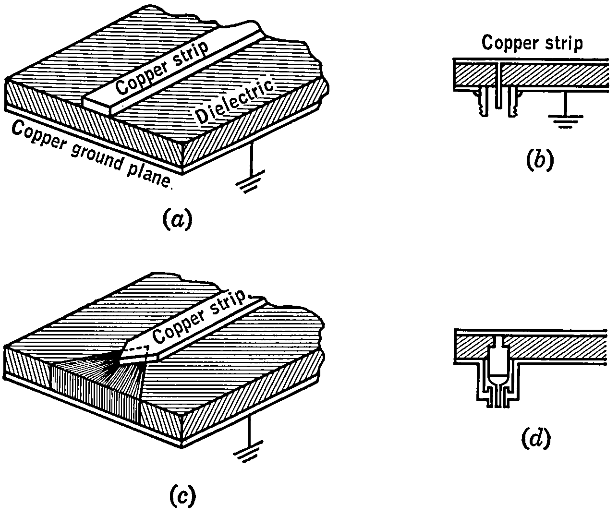


FIG. 21-26. Microwave components. (*a*) The strip line; (*b*) coaxial connection; (*c*) line termination with carbon wedge; (*d*) diode in the strip line.

Attempts are being made to utilize strip-line microwave techniques to obtain gating components with unit time intervals in the millimicrosecond range. One technique is to use a carrier wave of approximately 2,000 megacycles, modulated at about 500 megacycles, for unit-zero information transmission. Using these figures for illustrative purposes, a unit (or zero) signal would be represented by a burst of four 2,000-megacycle oscillations; actually, however, groups of 10 to 15 oscillations are presently being employed. There are two ways of representing unit and zero bits with these 2,000-megacycle pulse bursts, called the *phase-script* and *pulse-script* methods. The concept of phase script is just that described previously (Sec. 19-2) for a Parametron; the pulse script is analogous to the usual dynamic operation (see Fig. 21-27). Since it is easy to convert from pulse to phase script, and vice versa, either method can be utilized in different parts of the same computer, as convenience or electronic considerations may require.

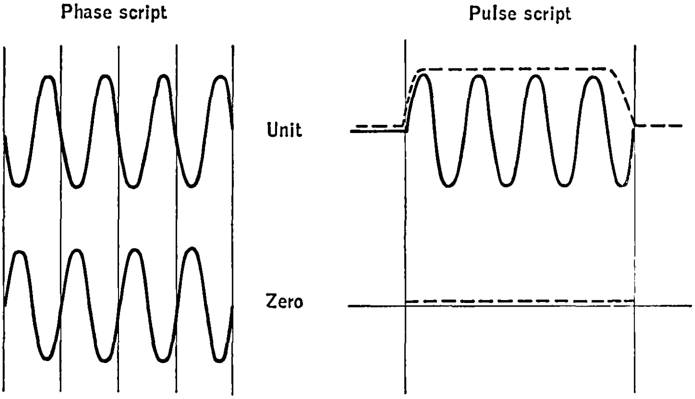


FIG. 21-27. Unit and zero representations in phase and pulse scripts.

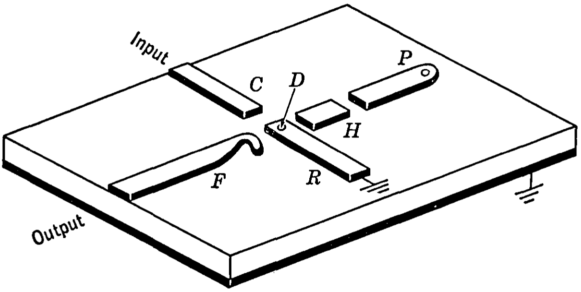


FIG. 21-28. Microwave parametric amplifier.

Consider the phase script first. A parametric amplifier (see Fig. 21-28) can be used, with a 4,000-megacycle primary source of energy entering the strip line at *P*. Any 2,000-megacycle input signal, coupled into the amplifier at *C*, will be amplified with its same phase in the 2,000-megacycle half-wave resonator *R*. A diode *D* in this resonator serves as a variable reactor; a 4,000-megacycle half-wave resonant strip *H* allows the 4,000-megacycle drive to enter *R* but prevents the 2,000-megacycle energy from escaping. A loosely coupled output arm *F* guides the amplified 2,000-megacycle signal to the rest of the circuit.

An *and* gate can be constructed by means of a so-called *hybrid ring* (see Fig. 21-29*a*). The ring is a circular section of strip line 1½ wave-lengths in effective circumference. A constant train of microwaves entering the ring through arm K as shown in the figure can split and go round the ring in both directions. When the two halves meet at C, they will be in phase; but when they meet at either arm A or arm B,

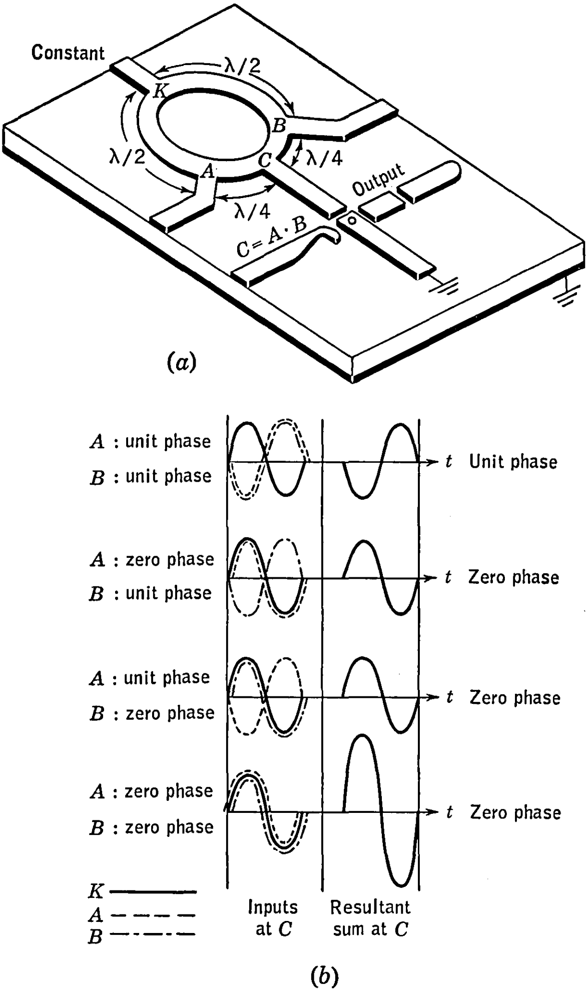


FIG. 21-29. (*a*) Hybrid ring; (*b*) time-phase relationships at arm C.

they will be ½ wavelength out of phase and will therefore cancel, feeding no energy into A or B. Neither will this energy be reflected back into arm K, because it will be ½ wavelength out of phase with the incoming wave at K. Similarly the energy entering the ring at A will cancel at arms K and B and will not be reflected back at A but will be in phase at C; energy entering at B will cancel at K and A, will not be reflected back at B, but will be in phase at C. If the constant wave is in the unit phase at K, then both A and B inputs will have to be in the unit phase for the resulting wave in arm C to be in the unit phase. For the constant wave entering at K will reinforce a zero-phase wave entering A or B or both (see Fig. 21-29*b*). Thus $C = A \cdot B$, and we have an *and* gate. An *or* gate can be obtained by having the constant input in the zero phase when it enters at K, instead of the unit phase. Negation is of course obtained by sending the wave through an extra ½ wavelength of strip line.

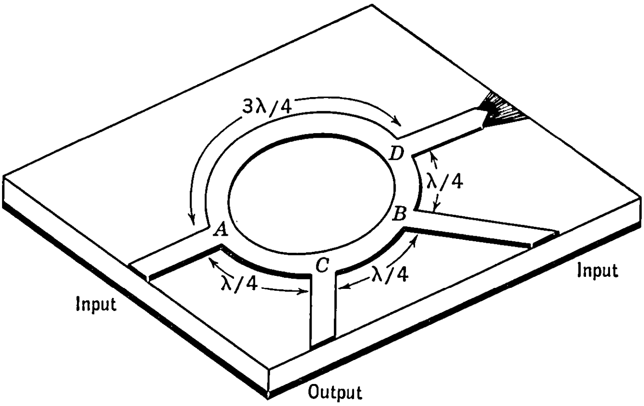


FIG. 21-30. Pulse-script *and* gate.

The pulse-script gating techniques use the hybrid ring also. For example, consider the ring used as an *and*-gate (see Fig. 21-30). Note again that the clockwise and counterclockwise waves due to an input at A cancel at B, and conversely waves entering at B cancel at A. If there is only one input, e.g., a train of pulses entering at A but not B, then half the power is absorbed at D and the other half enters C. On the other hand, if a train of wave pulses enters A and B at the same time (and in the same phase), then they will cancel at D (i.e., no power will be lost in D) and will reinforce at C. Therefore four times the power will appear at C when there is an input at both A and B than will appear at C when there is only one input. The operation of the *not* gate of Fig. 21-31 is as follows: If no pulse enters A, then the constant power entering at K will be divided between D and the output $\bar{A}$, presenting a pulse of waves at arm $\bar{A}$. If a group of waves enter at A, then there is canceling at $\bar{A}$ due to the constant input K and there is no output; all the power is absorbed at D.

Observe, however, that the parametric amplifier cannot be used in the pulse script, for even if there is no pulse (zero) input to the 2,000-megacycle resonator, 2,000-megacycle energy will still be accumulated (in an essentially random phase). Hence the pulse script must be converted into phase script; the phase script is amplified and then converted back to pulse script. For conversion from pulse to phase script consider the hybrid ring of Fig. 21-32, where the $\bar{A}$ input is delayed by a half wave-

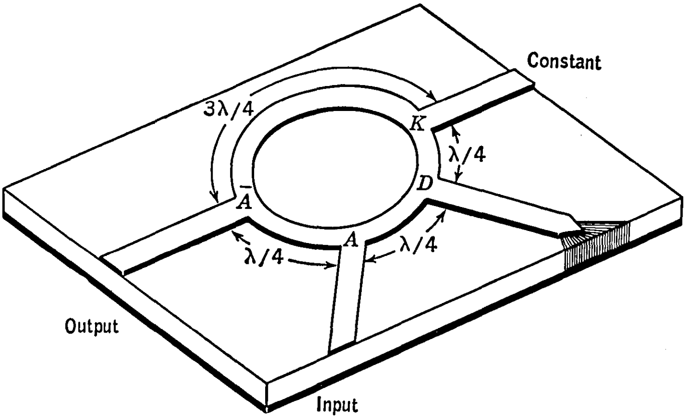


FIG. 21-31. Pulse-script *not* gate.

length so that it will be π radians out of phase with A. Only one at a time of these inputs will contain energy, and hence the wave output pulse will be in the unit or in the zero phase script depending on which input contains the energy. For conversion from phase to pulse script consider Fig. 21-33. A unit phase input at A will be reinforced at C by the constant. A zero phase input will interfere with and cancel the constant at C, and no output will result; the energy will be dissipated at D (why?).

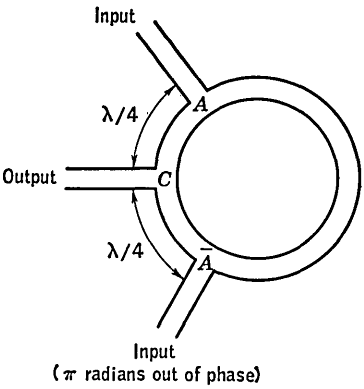


FIG. 21-32. Converting from pulse to phase script.

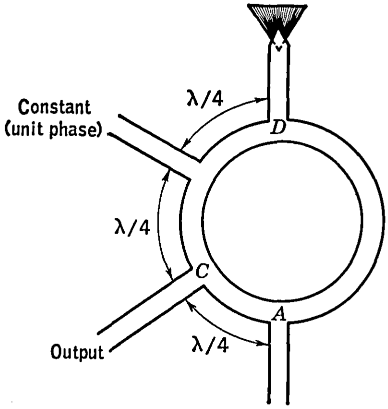


FIG. 21-33. Converting from phase to pulse script.

EXERCISES

(*a*) Design a serial adder, using the Cryotron gates illustrated in Fig. 21-25.

(*b*) Design a microwave serial adder, using hybrid rings in the phase script. Take advantage of the majority principle, and use only four rings (cf. Exercise *c* of Sec. 19-2 and Exercise *b* of Sec. 21-4).

21-6. Additional Topics

a. Selected Bibliography on the Physics of Magnetic Elements

Bates, L. F.: "Modern Magnetism," 3d ed., Cambridge University Press, New York, 1951.

Bauer, E. W., and L. P. Hunter: High Speed Coincident-flux Magnetic Storage Principles, *J. Appl. Phys.*, vol. 27, no. 11, pp. 1257–1261, November, 1956.

Bozorth, R. M.: "Ferromagnetism," chaps. 5 and 7, D. Van Nostrand Company, Inc., Princeton, N.J., 1951.

Brown, D. R.: Squareness Ratio for Coincident-current Memory Cores, *MIT Digital Computer Lab., Eng. Note* E-464, July, 1952.

Coleman, R. P.: Relation between Magnetic-core Switching Time and the Width of Ferromagnetic Resonance Peaks, *Proc. AIEE Conf. Magnetism and Magnetic Materials*, 1956, pp. 620–624.

DeBlois, R. W.: Domain Wall Motion in Metals, *J. Appl. Phys.*, vol. 29, pp. 459–467, 1958.

Digital Magnetic Circuits—Theory, Electrical Design, and Logical Design, *Burroughs Corp. Tech. Rept.* TR 59-27, vols. 1–3, April, 1959.

Gianola, U. F.: Switching in Rectangular Loop Ferrites Containing Air Gaps, *J. Appl. Phys.*, July, 1958, p. 1122.

Gyorgy, E. M.: Rotational Model of Flux Reversal in Square Loop Soft Ferromagnets, *J. Appl. Phys.*, vol. 29, p. 283, 1958.

Harrington, J. V.: On the Statics and Dynamics of Magnetic Domain Boundaries, *MIT Lincoln Lab. Tech. Rept.* 166, Jan. 13, 1958.

Hunter, L. P., and E. W. Bauer: High-speed Coincident-flux Magnetic Storage Principles, *J. Appl. Phys.*, vol. 27, pp. 1257–1261, November, 1956.

Katz, H. W. (ed.): "Solid State Magnetic and Dielectric Devices," John Wiley & Sons, Inc., New York, 1959.

Kilburn, T., G. R. Hoffman, and P. Wolstenholme: Reading of Magnetic Records by Reluctance Variation, *Proc. IEE*, vol. 103, pt. B, suppl. 2, Convention on Digital Computer Techniques, pp. 333–336, April, 1956.

Krell, A. I., C. Meyer, and R. D. Torrey: Switching Characteristics of Magnetic Cores as Circuit Elements, Transistor and Solid-state Circuits Conference, 1957.

McNamara, F.: Noise Problem in a Coincident-current Core Memory, *IRE Trans. on Instrumentation*, vol. I-6, pp. 153–156, June, 1957.

Menyuk, N., and J. B. Goodenough: Magnetic Materials for Digital Computer Components: Part I, A Theory of Flux Reversal in Polycrystalline Ferromagnetics, *J. Appl. Phys.*, vol. 26, pp. 8–18, January, 1955; Part II, Magnetic Characteristics of Ultrathin Molybdenum Permalloy Cores, *ibid.*, vol. 26, pp. 692–697, June, 1955.

Minnick, R. C., and R. L. Ashenhurst: Multiple-coincidence Magnetic Storage Systems, *J. Appl. Phys.*, vol. 26, pp. 575–579, May, 1955.

Rajchman, J. A.: Survey of Magnetic and Other Solid-state Devices for the Manipulation of Information, *IRE Trans. on Circuit Theory*, vol. CT-4, pp. 210–225, September, 1957.

Rossing, T. D., and S. M. Rubens: Effect of a Transverse Field on Switching Rates of Magnetic Memory Cores, *J. Appl. Phys.*, vol. 29, pp. 1245–1247, August, 1958.

Snoek, J. L.: "New Developments in Ferromagnetic Materials," Elsevier Press, Inc., Houston, Tex., 1947.

Van Sant, O. J.: Considerations for the Selection of Magnetic Core Materials for Digital Computer Elements, *IRE Conv. Record*, pt. 4, pp. 109–113, 1954.

b. Selected Bibliography on Magnetic-core Circuits, Both Diode-gate Amplifiers and Core Logic

Anderson, J. R.: A New Type of Ferroelectric Shift Register, *IRE Trans. on Electronic Computers*, vol. EC-5, pp. 184–191, December, 1956.

Andrews, L. J.: A Technique for Using Memory Cores as Logical Elements, *Proc. Eastern Joint Computer Conf.*, December, 1956, pp. 1139–1147.

Auerbach, I. L.: Ferromagnetic Cores with Microsecond Access, *Proc. Symposium on Large Scale Digital Computing Machines*, August, 1953, pp. 118–129.

Bambrough, B.: Digital Computer Based on Magnetic Circuits, *Proc. IEE*, vol. 104, pt. B, suppls. 5–7, pp. 485–490, 1957.

Bonn, T. H.: Analysis of Magnetic Switching Circuits, *Proc. Intern. Symposium on Switching Theory*, April, 1957.

Devenny, C. F., and L. G. Thompson: Ferromagnetic Computer Cores, *Tele-Tech & Electronic Inds.*, vol. 14, no. 9, pp. 58–59, 84–94, September, 1955.

Digital Magnetic Circuits—Theory, Electrical Design, and Logical Design, *Burroughs Corp. Tech. Rept.* TR 59-27, vols. 1–3, April, 1959.

Dorey, P. F., and O. Eldridge: Core-transistor Logical Element, *Proc. IEE*, vol. 104, pt. B, suppls. 5–7, pp. 512–522, 1957.

Evans, W. G., W. G. Hall, and R. I. Van Nice: Magnetic Logic Circuits for Control Systems, *Trans. AIEE*, pt. II, *Applications and Ind.*, vol. 75, pp. 166–171, July, 1956.

Freedman, A. L.: Magnetic Core Matrices for Logical Functions, *Electronic Eng.*, June, 1959, pp. 358–361.

Geyger, W. A.: "Magnetic Amplifier Circuits," 2d ed., McGraw-Hill Book Company, Inc., New York, 1957.

Guterman, S., and W. M. Carey: A Transistor-magnetic Core Circuit; a New Device Applied to Digital Computing Techniques, *IRE Conv. Record*, pt. 4, pp. 84–94, 1955.

———, R. D. Kodis, and S. Ruhman: Circuits to Perform Logic and Control Functions with Magnetic Cores, *IRE Conv. Record*, pt. 4, pp. 124–132, 1954.

Hogue, E. W.: Magnetic Amplifiers for Digital Computers, *Elec. Mfg.*, vol. 61, pp. 150–151, May, 1958.

———: A Saturable Transformer Digital Amplifier with Diode Switching, *Proc. Eastern Joint Computer Conf.*, December, 1956, pp. 58–64.

Isborn, C.: Application of Ferristor Type Saturable Reactors to Digital Instrumentation, *Paper* 56-33-2, *Proc. Instr. Soc. Am. Ann. Conf.*, September, 1956.

Karnaugh, M.: Pulse Switching Circuits Using Magnetic Cores, *Proc. IRE*, vol. 43, pp. 570–584, May, 1955.

Kodis, R. D., S. Ruhman, and W. D. Woo: Magnetic Shift Register Using One Core per Bit, *IRE Conv. Record*, pt. 7, pp. 38–42, 1953.

———, S. S. Guterman, and S. Ruhman: Logical and Control Functions Performed with Magnetic Cores, *Proc. IRE*, vol. 43, pp. 291–298, March, 1955.

Loev, D., W. Miehle, J. Paivinen, and J. Wylen: Magnetic Core Circuits for Digital Data Processing Systems, *Proc. IRE*, vol. 44, no. 2, pp. 154–162, February, 1956.

Meyerhoff, A. J.: Magnetic Core Components for Digital Control Applications, *Automatic Control*, April, 1956, pp. 15–19; May, 1956, pp. 13–15; July, 1956, pp. 34–40.

Newhouse, V. L., and N. S. Prywes: High-speed Shift Registers Using One Core per Bit, *IRE Trans. on Electronic Computers*, vol. EC-5, pp. 114–120, September, 1956.

Papoulis, A.: The Nondestructive Read-out of Magnetic Cores, *Proc. IRE*, vol. 42, pp. 1283–1288, August, 1954.

Ramey, R. A.: The Single-core Magnetic Amplifier as a Computer Element, *Trans. AIEE, Communs. and Electronics*, vol. 71, pp. 442–446, January, 1953.

———: On the Control of Magnetic Amplifiers, *U.S. Naval Research Lab. NRL Rept.* 3869, October, 1951. (Also available from Office of Technical Services, U.S. Department of Commerce, as PB 105747.)

———: On the Mechanics of Magnetic Amplifier Operation, *Trans. AIEE*, vol. 70, pt. II, pp. 1214–1223, 1951. (Also available from Office of Technical Services, U.S. Department of Commerce, as PB 104106 and as *U.S. Naval Research Lab. NRL Rept.* 3799.)

Russell, L. A.: Diodeless Magnetic Core Logical Circuits, *IRE Natl. Conv. Record*, pt. 4, pp. 106–114, March, 1957.

Sack, H. S., et al.: Special Magnetic Amplifiers and Their Use in Computing Circuits, *Proc. IRE*, vol. 35, pp. 1375–1382, November, 1947.

Sands, E. A.: An Analysis of Magnetic Shift Register Operation, *Proc. IRE*, vol. 41, pp. 993–999, August, 1953.

Scarrott, G. G., W. J. Harwood, and K. C. Johnson: The Design and Use of Logical Devices Using Saturable Magnetic Cores, *Proc. IEE*, vol. 103, pt. B, suppl. 2, Convention on Digital Computer Techniques, pp. 302–312, April, 1956.

Storm, H. F.: "Magnetic Amplifiers," John Wiley & Sons, Inc., New York, 1955.

Van Nice, R. I.: Magnetic Logic Circuit Control System Design Considerations, *Trans. AIEE*, vol. 75, pt. I, pp. 595–600, 1956.

c. Selected Bibliography on Multiaperture Magnetic Devices

Abbas, S. A., and D. L. Critchlow: Calculation of Flux Patterns in Ferrite Multipath Structures, *IRE Conv. Record*, pt. 4, pp. 263–267, 1958.

Abbott, H. W., and J. J. Suran: Multihole Ferrite Core Configurations and Applications, *Proc. IRE*, vol. 45, pp. 1081–1093, August, 1957.

Baldwin, J. A., Jr., and J. L. Rogers: Inhibited Flux—A New Mode of Operation of the Three-hole Memory Core, *J. Appl. Phys.*, vol. 30, pp. 58S–59S, April, 1959.

Bennion, D. R., and H. D. Crane: Design and Analysis of MAD Transfer Circuitry, *Proc. Western Joint Computer Conf.*, March, 1959, pp. 21–36.

Bobeck, A. H.: A New Storage Element Suitable for Large-sized Memory Arrays—The Twistor, *Bell System Tech. J.*, vol. 36, no. 6, pp. 1319–1340, November, 1957.

———: New Concept in Large-size Memory Arrays—The Twistor, *J. Appl. Phys.*, vol. 29, pp. 485–486, March, 1958.

——— and R. S. Title: The Twistor—Its Use in a Memory Array, *Proc. Nonlinear Magnetics and Magnetic Amplifiers Special Tech. Conf.*, August, 1958, p. 234.

Crane, H. D.: A High-speed Logic System Using Magnetic Elements and Connecting Wire Only, *Proc. IRE*, vol. 47, pp. 63–73, January, 1959.

Engelbart, D. C.: A New All-magnetic Logic System Using Single Cores, Digest of Technical Papers, p. 66, IRE-AIEE Solid-state Circuits Conference, 1959, University of Pennsylvania.

Gianola, U. F., and T. H. Crowley: The Laddic—A Magnetic Device for Performing Logic, *Bell System Tech. J.*, vol. 38, no. 1, pp. 45–72, January, 1959.

Lockhart, N. F.: Logic by Ordered Flux Changes in Multipath Ferrite Cores, *IRE Conv. Record*, pt. 4, pp. 268–278, 1958.

Newhouse, V. L., N. R. Kornfield, and M. M. Kaufman: A Transistorized Ferrite Plate Memory, *Proc. Natl. Electronics Conf.*, 1958, pp. 641–652.

Prywes, N. S.: Diodeless Magnetic Shift Registers Utilizing Transfluxors, *IRE Trans. on Electronic Computers*, vol. EC-7, pp. 316–324, December, 1958.

Rajchman, J. A.: Ferrite Apertured Plate for Random Access Memory, *Proc. IRE*, vol. 45, pp. 325–334, March, 1957.

———: Principles of Transfluxor and Core Circuits, *Proc. Intern. Symposium on Switching Theory*, April, 1956.

——— and A. W. Lo: The Transfluxor, *Proc. IRE*, vol. 44, pp. 321–332, March, 1956.

——— and ———: The Transfluxor—A Magnetic Gate with Stored Variable Setting, *RCA Rev.*, vol. 16, pp. 303–311, June, 1955.

Rumble, W. G., and C. S. Warren: Coincident Current Applications of Ferrite Apertured Plates, *IRE Wescon Record*, pt. 4, pp. 62–66, 1958.

Warren, C. S.: Ferrite Apertured Plate Memories, Digest of Technical Papers, p. 18, IRE-AIEE Solid-state Circuits Conference, 1959, University of Pennsylvania.

d. Selected Bibliography on the Cryotron

Bonn, T. H.: Magnetic Computer Has High Speed, *Electronics*, Aug. 1, 1957, pp. 156–160.

Bremer, J. W.: Cryogenic Devices in Logical Circuitry and Storage, *Elec. Mfg.*, vol. 61, no. 2, pp. 78–83, February, 1958.

Buck, D. A.: A Magnetically Controlled Gating Element, *Proc. Eastern Joint Computer Conf.*, 1956, pp. 47–50.

———: Superconductive Circuit Response Time: A Survey, Digest of Technical Papers, p. 28, IRE-AIEE Solid-state Circuits Conference, 1959, University of Pennsylvania.

———: The Cryotron, a Superconductive Computer Component, *Proc. IRE*, vol. 44, pp. 482–493, April, 1956.

Kraus, C. J.: Pro's and Con's on a Superconducting Memory, Digest of Technical Papers, p. 20, IRE-AIEE Solid-state Circuits Conference, 1959, University of Pennsylvania.

———: Operation of a Low-temperature Memory Element, Digest of Technical Papers, Transistor and Solid-state Circuits Conference, February, 1958.

McMahon, H. O.: Superconductivity and Its Application to Electric Circuits, *Proc. Symposium on the Role of Solid State Phenomena in Elec. Circuits*, 1957, pp. 187–195.

Richards, R. K.: Proposed New Cryotron Geometry and Circuits, Digest of Technical Papers, p. 30, IRE-AIEE Solid-state Circuits Conference, 1959, University of Pennsylvania.

Slade, A. E., and H. O. McMahon: A Cryotron Catalog Memory System, *Proc. Eastern Joint Computer Conf.*, 1956, pp. 115–120.

e. Kilomegagate Systems. Cryogenic techniques as well as other methods presently being developed give promise of feasibly obtaining billions of gates in a single computing system. The only known system of this type presently in existence is the mammalian brain. The following articles are concerned with problems of the analysis of brain mechanisms.

Farley, B. G., and W. A. Clark: Generalization of Pattern Recognition in a Self-organizing System, *Proc. Western Joint Computer Conf.*, 1955, pp. 86–91.

——— and ———: Simulation of Self-organizing Systems by Digital Computer, *IRE Trans. on Inform. Theory*, vol. PGIT-4, pp. 16–84, September, 1954.

Hebb, D. O.: "The Organization of Behavior," John Wiley & Sons, Inc., New York, 1959.

Rochester, N., J. H. Holland, L. H. Haibt, and W. L. Duda: Tests on a Cell Assembly Theory of the Action of the Brain, Using a Large Digital Computer, *IRE Trans. on Inform. Theory*, vol. IT-2, pp. 80–93, November, 1956.

Rosenblatt, F.: The Perceptron: A Theory of Statistical Separability in Cognitive Systems, *Cornell Aeronaut. Lab. Rept.* VG-1196-G-1, January, 1958.

———: The Perceptron, a Probabilistic Model for Information Storage and Organization in the Brain, *Psychological Rev.*, vol. 65, pp. 386–408, 1958.

———: Two Theorems of Statistical Separability in the Perceptron, *Proc. Symposium on the Mechanization of Thought Processes*, 1958. (Awaiting publication by H. M. Stationery Office, London; also printed without appendix as *Cornell Aeronaut. Lab. Rept.* VG-1196-G-2, September, 1958.)

———: Perceptual Generalization over Transformation Groups, *Proc. Interdisciplinary Conf. on Self-organizing Systems*, 1959. (Awaiting publication by Office of Technical Services, U.S. Department of Commerce.)

f. Selected Bibliography on Microwave Components

Albers-Schoenberg, E.: Ferrites for Microwave Circuits and Digital Computers, *J. Appl. Phys.*, vol. 25, pp. 152–154, February, 1954.

Baker, R. H., T. H. Meisling, and G. Nielson: "Application of Frequency-domain Techniques to Computers," Stanford Research Institute, Stanford, Calif., September, 1956.

Blattner, D. J., and F. Steizer: Fast Microwave Logic Circuits, *IRE Natl. Conv. Record*, pt. 4, pp. 252–258, March, 1959.

——— and ———: Microwave Carrier Technique for High Speed Digital Computing, Symposium on Microwave Techniques, Mar. 12, 1959, Washington, D.C.

Edson, W. A.: Frequency Memory in Multi-mode Oscillators, *Stanford Univ. Tech. Rept.* 16, July 19, 1954.

Forrer, M. P.: Investigation of Application of Frequency Memory Techniques, *Gen. Elec. Microwave Lab. Rept.* R55 ELS 12.3, Dec. 12, 1955.

———, L. Fein, and V. Met: Microwave Techniques for Computer Application, Quarterly and Final Reports on Office of Naval Research Contract Nonr-2127(00), December, 1956–December, 1958.

Giguere, W. J., J. H. Jamison, and J. C. Noll: Transistor Pulse Circuits for 160-MC Clock Rates, Digest of Technical Papers, p. 68, IRE-AIEE, Solid-state Circuits Conference, 1959, University of Pennsylvania.

Ortel, W. C. G.: Nanosecond Logic by Amplitude Modulation at X-band, Symposium on Microwave Techniques for Computing Systems, Mar. 12, 1959, Washington, D.C.

Steizer, F., and W. R. Beam: Parametric Sub-harmonic Oscillators, Digest of Technical Papers, IRE-AIEE Solid-state Circuits Conference, 1959, University of Pennsylvania.

Wigington, R. L.: A New Concept in Computing, *Proc. IRE*, vol. 47, pp. 516–523, April, 1959.

CHAPTER 22

MEMORY AND INPUT-OUTPUT METHODS

22-1. Introduction

Memory systems were touched upon briefly in Chap. 17, where our major concern was their integration into the logical design of the control circuitry. In the present chapter some of the electronic features of memory systems are considered. Only magnetic-core and -film memories and magnetic tapes and drums are considered in the brief treatment here presented, since these are by far the most important. In Sec. 22-2 problems concerning the selection ratio and the read-write cycle are discussed. Section 22-3 considers thin-film memories. We present the principle of nondestructive and simultaneous read-out in connection with thin-film memories, for it is felt that these concepts can most easily be given in this context. However, the principles involved can also be applied with single- or multiaperture cores, as is brought out in the Exercises. Section 22-4 emphasizes the problems involved in the interpretation of results obtained from magnetic tape and drum heads, in various forms of digital representation on magnetic surfaces. Magnetic tapes and drums are properly considered input-output equipment, and hence further consideration of input-output problems, in Secs. 22-5 and 22-6, follows naturally. Conversion of information from analog to digital representation and the reverse, considered in Sec. 22-5, is of central importance to many input-output devices. An introduction to the important sampling theorem is presented, and the principles of analog-to-digital conversion, and the reverse, are illustrated by examples. We refrain from detailed discussion of such circuits as voltage comparators, electronic switches, and ramp or saw-tooth wave generators, since these are not digital circuits and since adequate treatments of them can be found elsewhere. Finally in Sec. 22-6 we present several examples of the various types of input-output devices, to illustrate the variety of applicable techniques and available methods.

22-2. Magnetic-core Memories

Increasing the Selection Ratio. In Sec. 17-6 the elements of magnetic-core-memory coincident current selection were described (see Fig. 17-16 on page 567). In the simple system described in that section, to select a word two currents, each of $-I_c/2$ (half the coercive-force current), were passed through the cores, one current through the X wire, the other through the Y wire, corresponding, respectively, to the X and Y parts of the desired address. Only the cores of the proper address will then have the full current $-I_c$; of these only the cores recording units will flip to zero and be sensed by the Z wire. Some cores, not of the proper address, will be "half-selected," i.e., will have a current of $-I_c/2$ threaded through them. The ratio of the current in the selected core to the current in a half-selected core is called the *selection ratio*, in this case 2:1.

We shall discuss two methods for increasing the selection ratio to 3:1. In Fig. 22-1*a* we show just a single *x-y* plane. Here currents of $-I_c/2$ are passed through the x and y wires, as before. However, through all other x and y wires a current of $+I_c/6$ is passed. The total current through the selected cores will be the desired $-I_c$, but that through the nonselected cores will be either $-I_c/3$ or $+I_c/3$; thus the selection ratio is 3:1. The selection just described is for reading a word; for writing, the currents would be reversed. The second method for increasing the selection ratio involves the use of extra wires (see Fig. 22-1*b*) each of which threads all the cores in a single *x-y* plane. A current of $-2I_c/3$ is passed through the x and y wires corresponding to the desired address; a current of $+I_c/3$ is passed through the extra wire. The result is a current of $-I_c$ in the selected core, and either $-I_c/3$ or $+I_c/3$ in the other cores, giving the 3:1 selection ratio. Again, for writing, the currents are reversed.

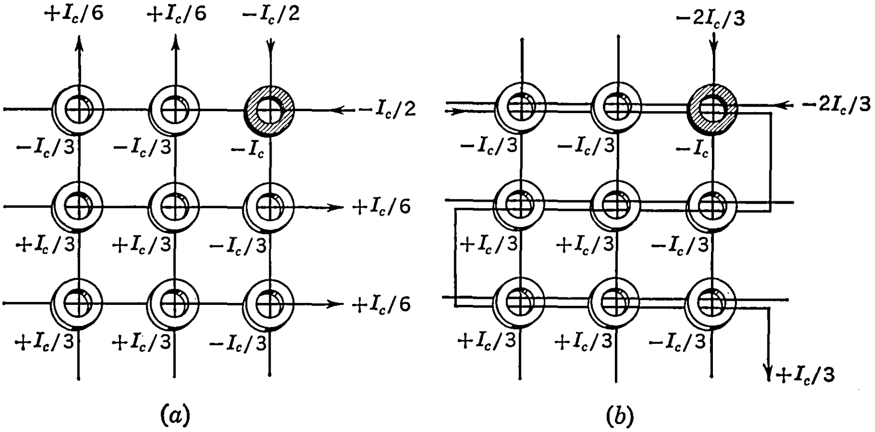


FIG. 22-1. Two methods for increasing the selection ratio to 3:1.

Core Selection Matrix and Read-Write Cycle for Core Memories. The theory of decoding matrices was discussed in Sec. 17-2, and in Chap. 18 selection matrices for the drum of the Pedagac were designed, using the diode-gated Pedagac packages. We now consider the design of selection matrices for core memories using cores as the gating elements. The technique is based on the methods of Sec. 21-4. Consider, for example, a 16-word core memory, with α^*1, α^*2 the x part of an address and α^*3,

α^*4 the y part. The x selection matrices appear as in Fig. 22-2. One of the cores, the *selecting core,* will flip to the unit state, and hence only one of the sense wires will carry a current, as desired. Of course for large memories several levels of gating may be necessary, as described in the theory of decoding matrices in Sec. 17-2.

Observe that, in writing into a core memory, all the cores of the selected memory location must be initially in the zero state. Then only the cores corresponding to the unit bits of the word will be flipped. In reading out of a core memory the attempt is made to flip all the cores of the selected memory location to the zero state. The unit cores of the word will then flip, and hence the unit bits will be sensed—but the memory location will be left all zeros. Therefore the word that has been read out must be stored and written back into this memory location. In order to facilitate the initial clearing before writing and the final write-back after reading, a single read-write cycle is usually established, which is repeated for every memory address selection, either read or write. First the selection matrices are set, and the selected word is read out of the memory into a memory buffer register, clearing the memory location. Note that at this time the selecting cores are the only cores of the x and y selection matrices that are in the unit state. Now the x and y reset signals are initiated, and the selecting cores flip back to the zero state, reversing the current in their sensing leads. As we have just seen, *this reversed current performs the selection* for writing into the memory. If the instruction being performed is "Read," then the word in the buffer is written back into the selected memory location as it is also transferred to some other part of the computer. If the instruction being performed is "write," the buffer is first loaded with the new word, which is then written into the selected memory address.

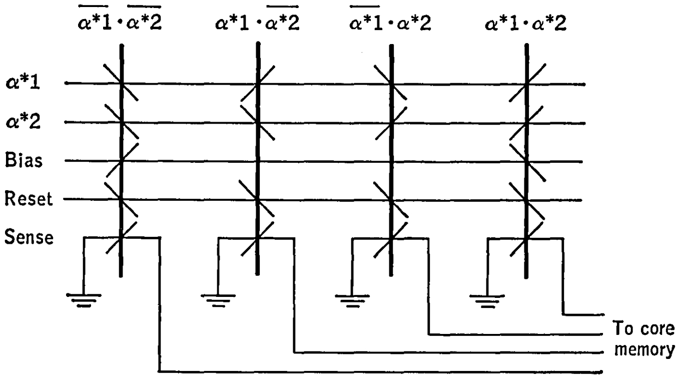
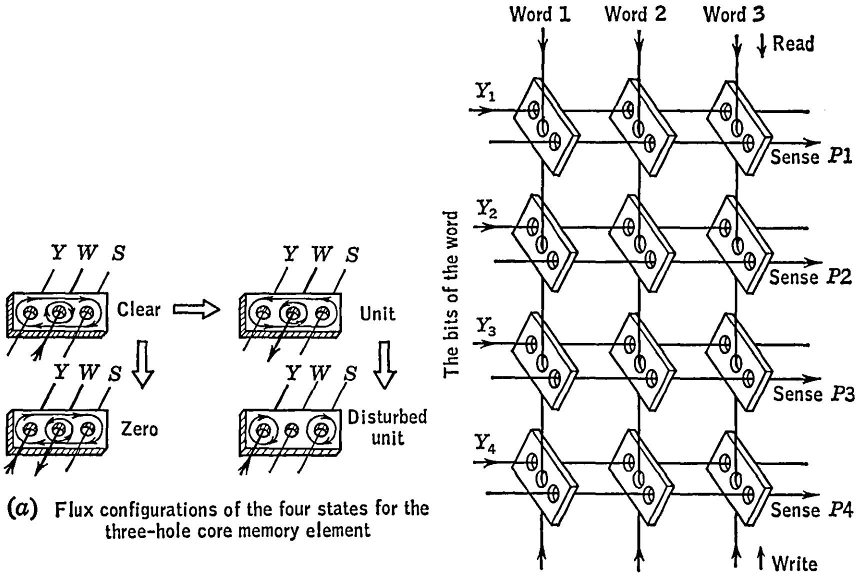

$\overline{\alpha^*1}\cdot\overline{\alpha^*2}$
$\alpha^*1\cdot\overline{\alpha^*2}$
$\overline{\alpha^*1}\cdot\alpha^*2$
$\alpha^*1\cdot\alpha^*2$
α^*1
α^*2
Bias
Reset
Sense
To core memory

FIG. 22-2. Core selection matrix.

Memory Applications of Multiaperture Devices. A disadvantage of the core memory systems described above is that the current through a single wire cannot be greater than the coercive force of the core for selection by coincident currents. This limitation in drive current places an upper limit on the memory-system speed with which the memory cores can be flipped. However, from the discussion of Sec. 21-2 it is clear that, if the selection, read, and write currents could be made large, the switching time could be made very small. We shall now describe a non-coincident-current memory system with this advantage, utilizing three-hole cores.

The three-hole ferrite device has a single wire through each hole (see Fig. 22-3*a*). The *Y* wire (through left hole) distinguishes the bits of the words; the *W* wire (through middle hole) distinguishes the words of the memory; the *S* wire (through right hole) senses the bits for read-out. Starting from the *clear* state (see Fig. 22-3*a*, upper left) produced by a current in the *W* wire, the unit state (upper right) is effected by a reversed current through the *W* wire and the zero state (lower left) by currents through *both* the *Y* and *W* wires. A typical memory configuration is shown in Fig. 22-3*b*. To write a word, the *W* wire for the desired memory location is selected, and through it a drive current is applied, upward as shown in the figure. At the same time currents are applied only in those *Y* wires corresponding to the zero bits of the word being read in. But consider the effect that a current in a *Y* wire has on a nonselected bit element. If the bit is in the zero state, the *Y* current has no effect; if the bit is in the unit state, the *Y* current will change the unit state to a disturbed unit state (lower right of Fig. 22-3*a*). Thus unit bits may be in either the normal unit or the disturbed unit stage. To read a word



(*a*) Flux configurations of the four states for the three-hole core memory element

(*b*) Three-hole core memory array of three words, each of four bits

Fig. 22-3. Three-hole multi-aperture-device memory.

out of the memory, a large current is applied again to the *W* wire, in a direction opposite to the write selection current. The core will be set to the clear state (upper left). In so switching, the flux direction round the rightmost hole is the same as that of the zero state but is reversed from that of the normal unit and disturbed unit states. Hence the sense wire *S* will have a current induced corresponding to a unit, but not to a zero (see Fig. 22-3*b*).

EXERCISES

(*a*) Describe a technique for producing a 4:1 selection ratio.

(*b*) Two-dimensional coincident-current selection methods have been discussed in the text. How would a three-dimensional coincident-current selection method work? (HINT: Break the address into three parts.) What selection ratios will be obtained?

(*c*) Design a core x-selection matrix for 4 bits in the x part of the address.

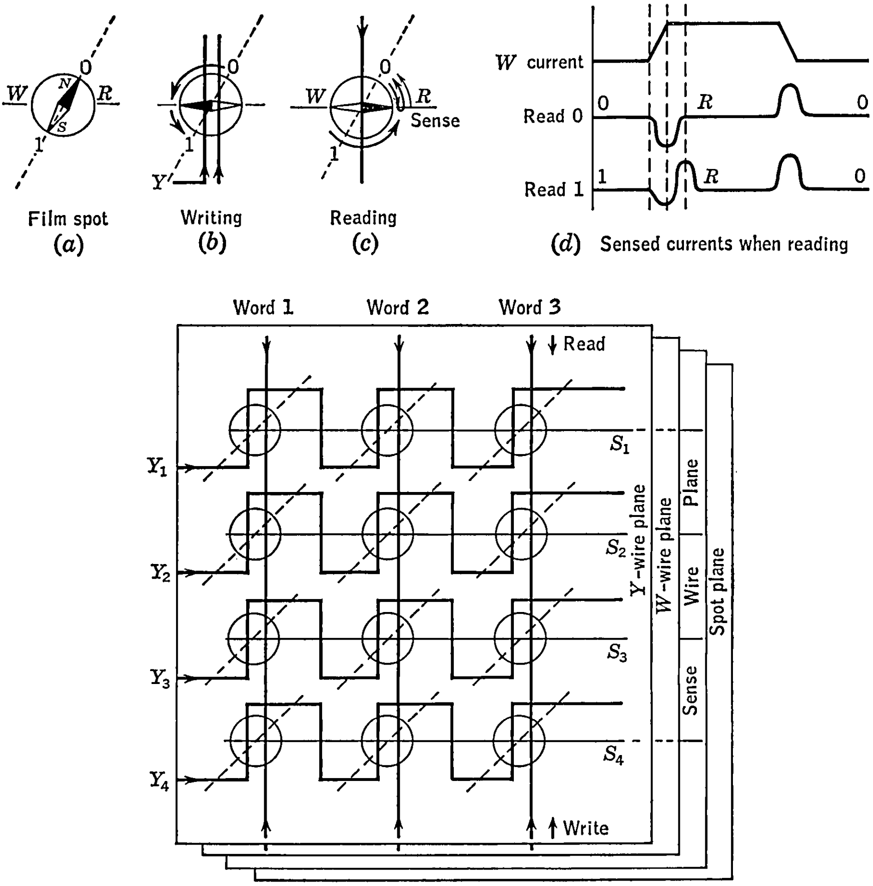
(*d*) Using the decoding techniques described in Sec. 17-2, design a two-level 8-bit x-selection matrix using cores.

(*e*) How would selection matrices operate for the three-hole core memory described in the text? How does this differ from the selection techniques described earlier?

22-3. Magnetic-film Memories

Thin Films. Thin-film memories require no threading of wires through holes and therefore can more easily be constructed by printed circuit techniques. The storage elements are vacuum-deposited nickel-iron-alloy films with square hysteresis loops, shaped into $\frac{3}{16}$-in.-diameter round spots about 2,000 A thick. The films are deposited in the presence of a magnetic field and as a consequence show a preferred direction of magnetization. The direction of magnetization can be represented by a magnetic dipole (like a compass needle) which has two stable states parallel to the preferred direction, the unit and zero states (see Fig. 22-4*a*). An appropriately applied magnetic field can cause the film dipole to *rotate* out of its unit or zero state; when the field is removed, the dipole will rotate back to the *nearest* stable state (that state requiring the least angle of rotation). Three (printed) wires are required, the *W* wire characteristic of a word, a *Y* wire characteristic of a bit of the word, and an *S* wire that senses the read-out signal for a bit, oriented as shown in Fig. 22-4*e* with respect to the preferred direction of the film spot.

The films of a storage location are always reset to zero before a word is written into it. The directions of the *W* and *Y* wires are parallel at the film spot, making an angle of 30° with the spot's preferred direction. When it is desired to write into a memory location, a current is applied to the corresponding *W* wire. For the unit bits of the word a current in the same direction as that of the *W* wires is applied to the *Y* wire; for the zero bits of the word a current in the opposite direction is applied to the *Y* wire. For the unit bits the combined applied fields of both *W* and *Y* wires rotate the magnetization to the *W* direction (see Fig. 22-4*b*); when the applied field is removed, the magnetization rotates to the unit preferred state since this is closest. For the zero bits the effects of the fields applied by the W and Y wires cancel, and the magnetization stays in the zero state. The field applied to the nonselected word spots by the current in the Y wire is not of itself sufficient to rotate the magnetization



(d) Sensed currents when reading

(e) Film memory

FIG. 22-4. (a) Film spot with compass representation; dashed lines show preferred direction, with 0 and 1 states. (b) Writing a unit, showing rotation to W and then to the unit state. (c) Reading, showing rotation to R and then to zero state. (d) Sensed currents when reading (both returning to the zero state). (e) Assembly of a film memory showing planes of printed wires and spot plane.

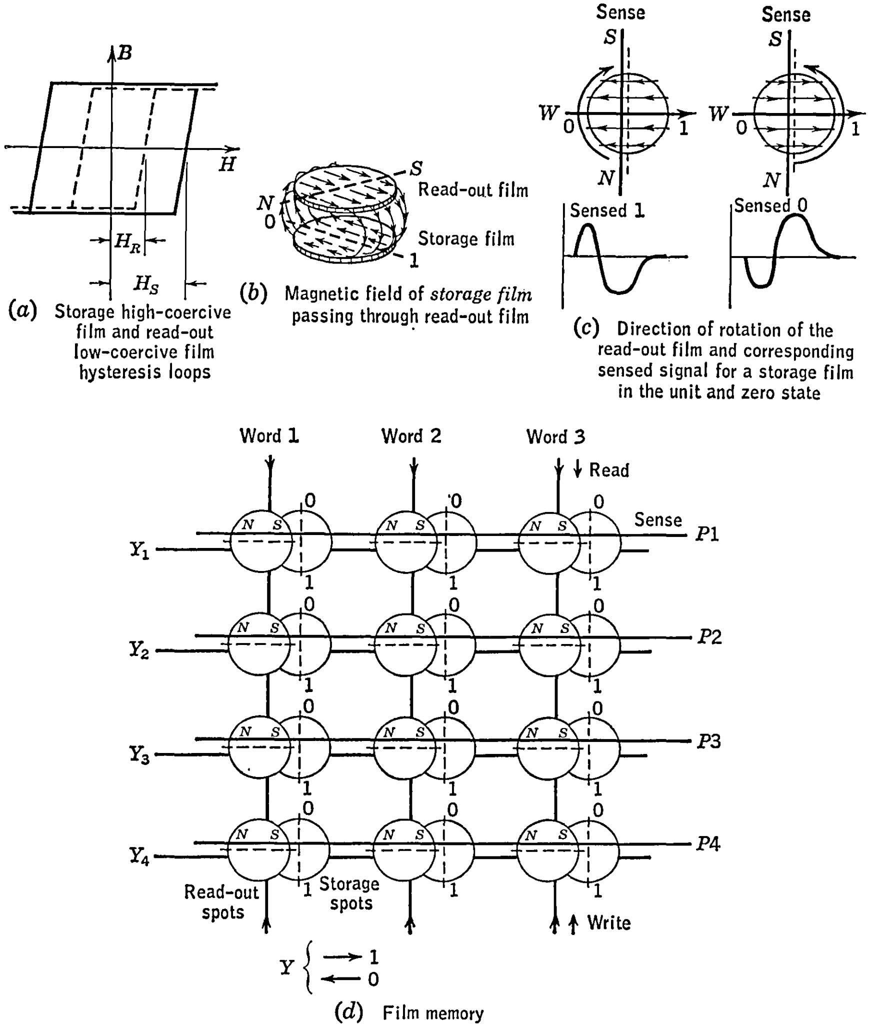
over 90° from the preferred direction, and hence those spots are not affected.

To read a word from the memory, a current is applied to the W wire in the direction opposite to that for writing. The zero-state spots rotate from the zero direction to R (see Fig. 22-4c); the unit-state spots rotate from the unit direction also to R, but the rotation occurs in the opposite sense to that of the zero-state spots. The sense wire S, placed perpendicular to the W wire to minimize stray pickup, gets a negative induced current from the former rotation and a positive induced current from the latter rotation; these currents correspond, respectively, to sensed zeros and units (see Fig. 22-4*d*).

However, both unit and zero spots will return to the zero state, and, as in the case of the memories described above, the word must be read back by means of the usual read-write cycle. Figure 22-4*e* shows a memory of three 4-bit words; the three wire planes are printed separately and placed over the spots as shown. Actually for greater sensing signals two spots per bit are used in a slightly different arrangement.

Nondestructive and Simultaneous Read-out from Thin-film Memories. In all memory systems considered so far, a read-write cycle and a memory buffer register were necessary, because the read-out destroyed the stored information. A nondestructive read-out memory has the advantage of eliminating the buffer register and the circuitry needed for a complete read-write cycle. In addition nondestructive read-out memory systems frequently enable reading from one word of the memory while simultaneously writing into another. This is because in such memories the mechanism of the write process is distinct from the mechanism of the read process.

As an illustration, consider the nondestructive read-out thin-film memory of Fig. 22-5. For each bit there are a pair of film spots, one for storage and one to facilitate read-out. The remanent state of the storage film spot is sensed by switching the read-out film spot in such a way as not to alter the magnetic state of the storage spot. The storage film spot is made of high-coercivity material and the read-out spot of low-coercivity material (see Fig. 22-5*a*); the two spots for a bit are placed close together (see Fig. 22-5*b*), with their preferred directions at right angles. The flux of the storage spot passes through the read-out spot as shown in that figure. To sense the state of the storage spot, a current is applied to a W wire running parallel to the preferred direction of the storage film. Suppose the magnetization of the read-out spot is initially parallel to its preferred direction with the north pole as indicated in Fig. 22-5*c*. Then two magnetic fields will be applied to the read-out film: the field of the storage film (shown as transverse arrows) and the field of the current in W. The latter field will reverse the direction of magnetization of the read-out film, but the direction of rotation for reversal will depend on the direction of the field due to the storage film, as shown in Fig. 22-5*c*. Depending on its direction, this rotation will induce an initial positive or negative current in a sense wire. Since the coercivity of the storage spot is greater than that of the read-out spot, the direction of magnetization of the former spot will not be affected provided that the reading current is only sufficient to flip the latter. A reverse current of the same limited magnitude must be applied through the W wire to reset the read-out spot so that its north pole is in the proper initial direction.



(*a*) Storage high-coercive film and read-out low-coercive film hysteresis loops

(*b*) Magnetic field of *storage film* passing through read-out film

(*c*) Direction of rotation of the read-out film and corresponding sensed signal for a storage film in the unit and zero state

(*d*) Film memory

FIG. 22-5. Nondestructive read-out thin-film memory. If the magnetization of the storage spot points to the 0, then a zero is stored; if it points to the 1, a unit. The *N* and *S* stand for the poles of the magnetization of the read-out film in the reset or normal state.

Figure 22-5*d* represents a memory of three 4-bit words. To read, the *W* wire for the desired word is selected and pulsed, the spots being sensed nondestructively as just described. To write into the memory, a current in the reversed (writing) direction is passed through the proper word wire, as at the same time currents corresponding to the desired zeros and units are passed through the *Y* wires. If a storage spot is to flip, both its *W* and *Y* wires must have currents; thus storage cores in nonselected words will not respond. Simultaneous read and write can occur (involving different words) provided that the coercivity of the read-out film is sufficiently less than that of the storage film. If word 3 were being read, for example, as word 1 is written, the currents in the sensing wires *will not* be great enough to influence the writing of word 1. The problem is rather to prevent the writing operation from influencing the reading of word 3. The writing current in word 1 holds its read-out spots in the reset state, so that none will flip to give a false read-out as the word 1 storage spots are flipped. The currents in the Y wires might contribute, with the resetting current in the W wire of word 3, to flipping a word-3 storage spot. This will not occur, however, if the reading and resetting current is much less than the required writing current; the difference between the currents depends on the difference in the coercivity of the spots of the pair.

EXERCISES

(*a*) In practice the rotation of a single spot may not induce sufficient current in the sense wire. How can the arrangement of Fig. 22-4 be revised so that two spots per bit are sensed?

(*b*) The two-aperture devices described earlier (Sec. 21-4) can be used for nondestructive and simultaneous read-out forms of memories. Using the principles developed in this section, design such a memory system. (HINT: Use the coincident-current method to select the large hole for writing into, and the small hole for reading out of, the memory.) Be careful when explaining why simultaneous read-out can occur.

22-4. Magnetic Tapes and Drums

Return-to-zero Writing Method. Both magnetic tapes and drums operate on the principle of writing onto and reading from magnetic material deposited on a moving surface. Figure 22-6*a* shows a reading and writing head. For writing, a current pulse I induces flux, some of which magnetizes the material on the moving tape or drum surface (see Fig. 22-6*b*). In the sensing of such a magnetized spot, as the spot moves past the head the lines of flux will pass through the read coil, inducing a small voltage difference. The writing current and induced reading voltage are small (a few milliamperes or millivolts, respectively). Figure 22-6*c* illustrates a current pulse and the intensity of the resulting spot of flux along the length of the tape; the shape of the induced voltage pulse across the read coil is shown, as is the final result of amplifying and limiting this voltage pulse. It is desirable that the head be designed to write with a small current, to produce a small spot on the tape, and to produce as large a reading voltage as possible. The gap in the head should be small to localize the spot and reduce magnetic-circuit reluctance and should maximize fringe effects so that the lines of flux pass into the tape. A low-reluctance path will increase the flux through the head on reading, and hence the head should have a short path and a large relative cross-sectional area as well as a small gap, and should be made of high-permeability material.

One of the first methods used to record binary information on a magnetic surface is illustrated in Fig. 22-7. The sequence of bits is shown in (*a*). For a unit there is a positive current pulse, for a zero a negative current pulse, but after each pulse there is a return to zero as shown in (*b*). If the spots are sufficiently spaced, at, say, *m* bits/in. on the tape, then the magnetization or flux per unit length is as shown in (*c*). If the spots are placed closer together, they may overlap, particularly for repeated units or repeated zeros, as shown in (*d*). When the spots are well spaced, as in (*c*), the voltage induced in reading this flux is as shown in (*e*). These pulses can be amplified and limited; then when the output is sampled at appropriate times, a voltage curve similar to (*b*) as desired will have been formed. For example, the sampling times may correspond to the pulses above the line in (*f*). Since each positive voltage swing is followed by a negative swing and each negative swing by a positive swing [see (*e*)], the output can be checked by sampling a second time for each bit, as shown in (*f*) by the pulses below the line (to indicate inversion). If a pair of samples do not agree, an error is indicated.

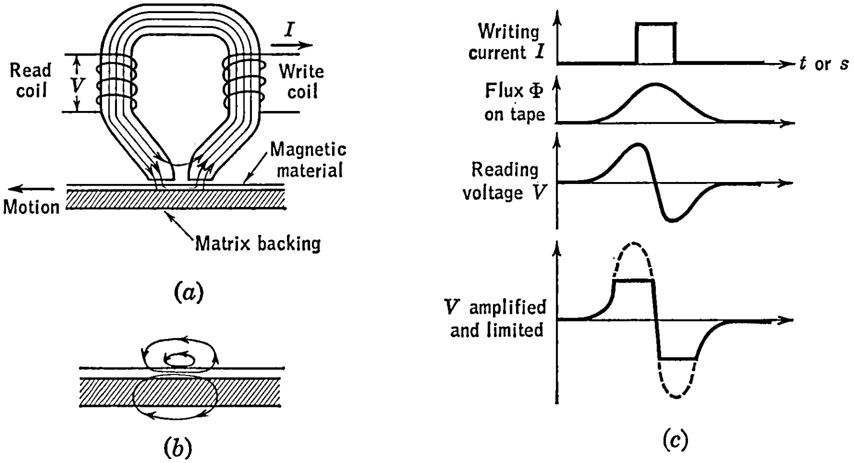


FIG. 22-6. (*a*) Magnetic reading and writing head; (*b*) flux spot in the magnetic material; (*c*) writing current flux, reading voltage, and processed reading voltage as a function of time *t* or distance *s* on the moving surface.

When the bits are packed densely, repeated units and repeated zeros require further consideration. For such repetitions can cause the accumulation of flux in the magnetic head, as shown in Fig. 22-8*g*, and this accumulation tends to saturate the head. The resulting voltage waveform is as shown in (*h*). Amplifying and limiting this voltage can cause erroneous results in the sampling. The wavy pulse lines in (*i*) represent positions at which errors can result, since as the head approaches saturation some of the positive and negative induced voltage swings are close to zero, as indicated by the three arrows in (*h*). In order to interpret a waveform such as (*h*) correctly, we might form its derivative, counting on the slopes to preserve the information. However, the derivative of a curve is very sensitive to noise; hence it is better to take a difference instead between the voltage waveform and itself delayed by one-half the distance between bit centers. The two waves are depicted in Fig. 22-9*j*, their difference in (*k*). This difference, while related to the derivative, is not so sensitive to errors due to noise (why?). When the waveform of (*k*) is amplified and limited, the pulse train of (*l*) results. These pulses are to be sampled within the *first half period following* the bit center (why?), as shown in (*m*). When the sampled pulses are inverted, the desired result is obtained, as shown in (*n*).

Non-return-to-zero Methods. Some of the problems that occur in the return-to-zero method arise from the fact that, when neighboring spots are magnetized in the same direction, their flux lines tend to link and thus between-bit sensing is weakened. In the first non-return-to-zero method that we shall discuss there is at least one reversal of magnetization direction per bit. Consider, for example, Fig. 22-10. Here a unit [of the word in (*a*)] is represented by a pulse crossing the zero axis from below, a zero by a pulse crossing from above [see arrows in (*b*)]. Between bit positions there may be an appropriate reversal, to get into position. The pulses of (*b*) represent the write current pulses. The voltage waveform of (*c*) is the result of reading a tape in this mode when close bit packing is used. The differencing method can then be applied [see the dashed lines in (*c*)], and the resulting difference will appear as in (*d*). After amplifying and limiting, the waveform of (*e*) is obtained, which when sampled in the first half of the period results in the desired pulses, shown in (*f*). It has been reported that with this method pulses packed at 880 bits/in. can be read. Normal packing densities (for all methods) range from 200 to 400 bits/in. for tapes.

A second non-return-to-zero method is illustrated in Fig. 22-11. In this method the flux on the tape is made to change direction *only when a unit appears.* Thus the write current appears as in (*b*) for the bit pattern of (*a*). The voltage waveform on reading will have a spike whenever a unit occurs [see (*c*)]. These spikes may be rectified to preserve symmetry [see dashed lines in (*c*)]. The amplified and limited result will then have a pulse for a unit, no pulse for a zero. The output from the head should be compared with a constant voltage of about half the normal peak, and any pulses below that level should be ignored as noise.

Clocking. In the previous paragraphs we have tacitly assumed some kind of sampling pulse generator synchronized with the reading of the tape so that the processed information is sampled at appropriate times. One technique for accomplishing this is to have on the tape an extra channel containing clocking pulses only. For example, in Fig. 22-12*a* we illustrate a four-channel tape with the fourth channel used for clock pulses. If this were the second non-return-to-zero scheme, the writing current pulses for each channel would appear as in (*b*); the voltage spikes that result from reading are shown in (*c*). Then to read channel 1, 2, or 3, the output of that channel is gated with the output of channel 4 as

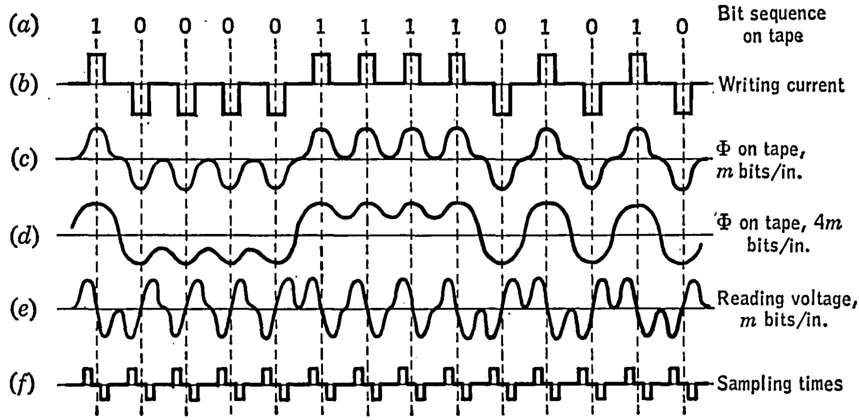


FIG. 22-7. Return-to-zero recording.

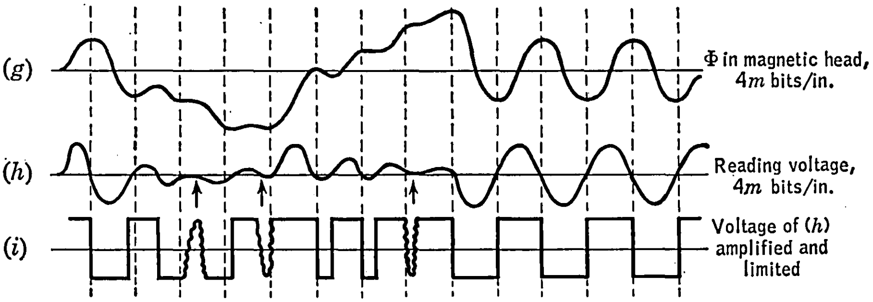

(g)
Φ in magnetic head, 4m bits/in.
(h)
Reading voltage, 4m bits/in.
(i)
Voltage of (h) amplified and limited

FIG. 22-8. Return-to-zero: problem of repeated bits.

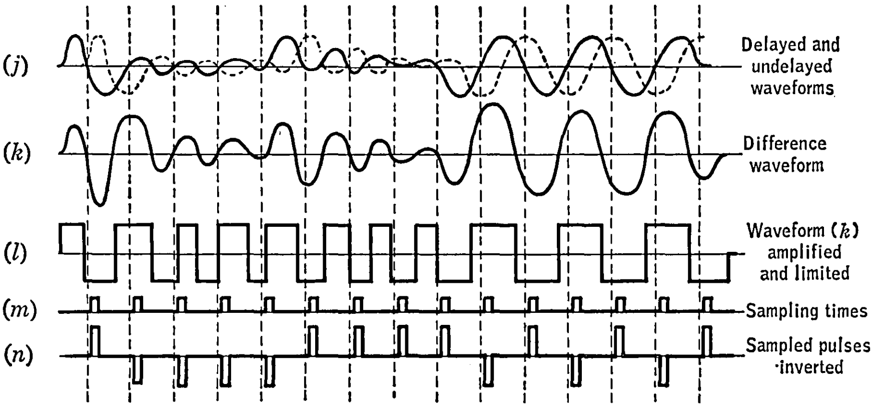


FIG. 22-9. Return-to-zero: method of differencing.

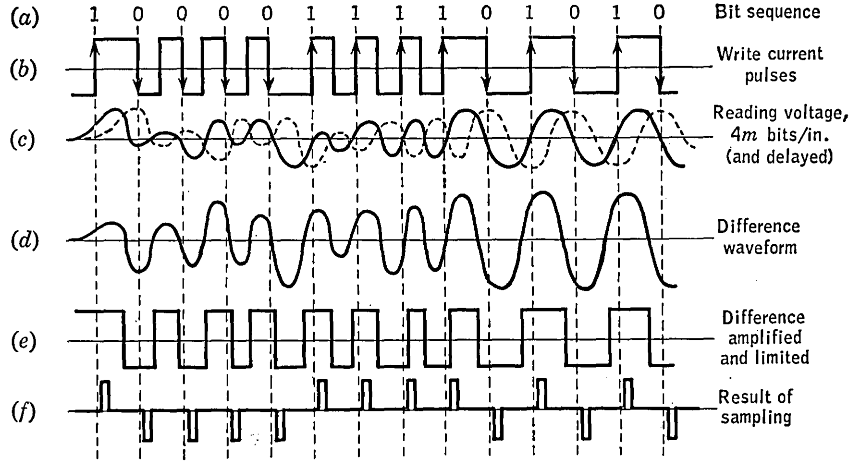


FIG. 22-10. Nonreturn to zero with differencing.

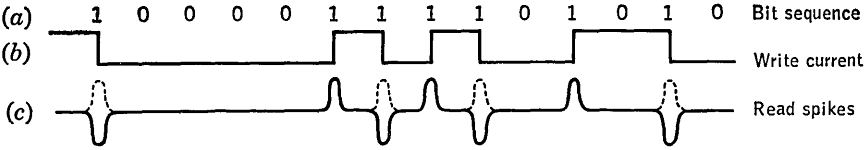
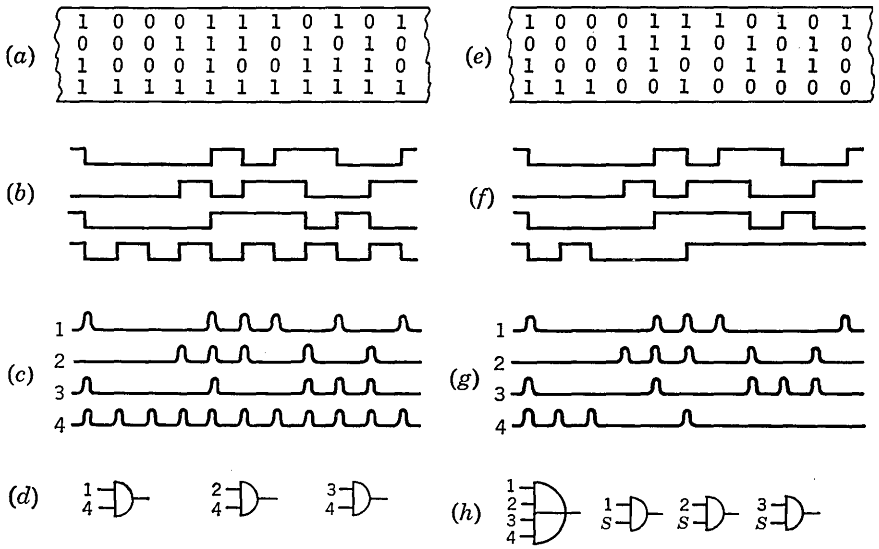


FIG. 22-11. Nonreturn to zero: change-over method.



(a)

1	0	0	0	1	1	1	0	1	0	1
0	0	0	1	1	1	0	1	0	1	0
1	0	0	0	1	0	0	1	1	1	0
1	1	1	1	1	1	1	1	1	1	1

(e)

1	0	0	0	1	1	1	0	1	0	1
0	0	0	1	1	1	0	1	0	1	0
1	0	0	0	1	0	0	1	1	1	0
1	1	1	0	0	1	0	0	0	0	0

FIG. 22-12. Clocking methods.

shown in (*d*). A slightly different scheme uses the extra channel to record an *odd parity bit* for pulses being read at the same time; this is illustrated in (*e*). As the 4 bits on each line are read out [(*f*) and (*g*)] by the four heads (one for each channel), the odd parity bit can be checked. The four outputs (*g*) are combined in an *or* gate, which has a unit output S for every line of pulses, because of the odd parity. Then S can be gated with channels 1, 2, and 3 as a clocking pulse [as in (*h*)].

The first non-return-to-zero method lends itself to a *self*-clocking scheme. It is easily observed that there are either one or two axis crossovers per bit of the processed information (see Fig. 22-10*d*). Where there are two crossovers, only one of them contains the desired significant information; the problem is to determine which this is. Starting with a significant crossover, an "information gate" can be formed that is one-half period in length and starts three-fourths period delayed from the significant crossover. This gate will then "bracket" another significant crossover. The sign of the differential crossover spike, indicating which way the crossover was made, will indicate whether the bit is a unit or zero. This scheme permits some variability in the periods, since they can vary by one-fourth period and still allow the information to be sensed. The clock starts from the new crossover each time and so is adjusted continually. In order to start the process, an information sample of alternating zeros and units is used, since these contain only significant crossovers.

Drums and Tapes. We have previously described various arrangements of words on the drum in channels and sectors, as well as methods for synchronizing the drum with the computer and for memory selection (see Secs. 17-6 and 17-8). The rotating speeds vary from 440 to 12,500 rpm, with 50 to 100 bits/in. along a channel and 15 to 30 channels/in. The reason for the low bit density is that to eliminate frictional wear the spacing between the head gap and the drum surface is as great as 0.001 to 0.002 in. This spacing results in considerable loss of spot resolution. In addition any eccentricity of the drum will vary the drum-to-head distance, introducing further aberrations. The drum surface is coated with a magnetic material, such as Fe_2O_3, or plated with the desired magnetic metal. There are a wide variety of drum diameters and lengths, diameters ranging from 2 in. to 4 ft, and lengths from ¼ in. to 3 ft.

Disks coated with magnetic material can also be used for storage. Figure 22-13 shows one such mechanism, whose read-write head is moved mechanically along the rapidly spinning disks to the proper disk level and is then inserted to the proper radius. By using an air cushion the separation between the head and disk can be made very small and kept relatively constant even if the disk were somewhat warped. Both sides of the disk are used, making the total available surface area very large. Each disk contains 7×10^5 bits; for a system utilizing 50 disks, the average access time is about 0.6 sec.

Magnetic tapes are coated strips of plastic (usually Mylar) or other nonmagnetic material, varying in width from ½ in. to several inches, the wider tapes containing more channels. Usually seven channels are used; each 7-bit rank represents an alphabetic or numeric character, but the seventh bit is a parity check or clocking channel bit. As we have discussed above, the ranks are organized in blocks; there is space containing no information left between blocks (equivalent to about three ranks) as an allowance for starting and stopping the tape between blocks. The mechanism for moving the tape must be capable of producing high acceleration on starting and stopping, for the tape speed while reading or writing must be relatively constant. One method is to have the tape pass over rapidly spinning wheels (see Fig. 22-14*a*). When the tape is to be moved, it is squeezed between one of these spinning wheels and an idling pressure wheel that is driven against it; the increased friction between the tape and the spinning wheel quickly brings the tape to full speed. To reduce inertia, the tape itself may be stored loosely in bins, as shown in Fig. 22-14*a*. Frequently reels of tape are used, as shown in Fig. 22-14*b*. Here, in order that the tape-driving wheels need not accelerate the reels, the proper

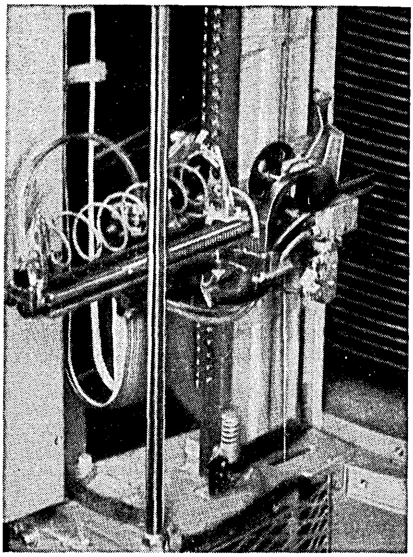
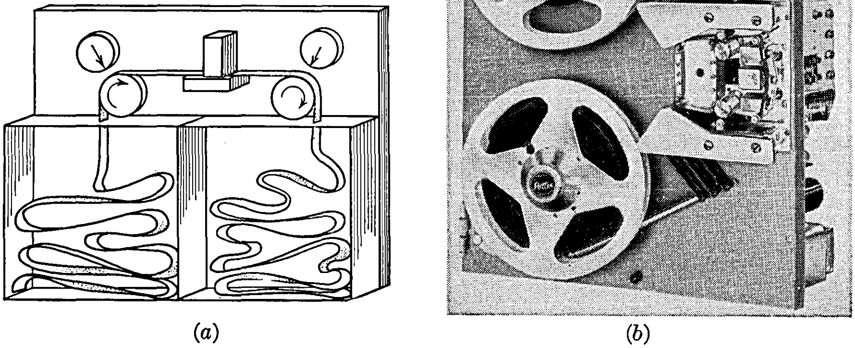


FIG. 22-13. Disk memory, showing magnetic head positioned between disks. (*Photograph courtesy International Business Machines Corp.*)



(*a*) (*b*)

FIG. 22-14. Magnetic-tape movement mechanisms. (*a*) Tape bins; (*b*) tape reels. (*Photograph courtesy Potter Instrument Co., Inc.*)

amount of slack tape is maintained by servomechanisms that drive the reels. Complete loops of tape, or tape belts, have the advantage of not needing to be rewound to obtain information at the beginning, if the tape is located at the end. Short tape belts can also be used for temporary storage, as illustrated by the small retrieval computer of Fig. 8-9 on page 269.

EXERCISES

(*a*) When the spots are well spaced, as in Fig. 22-7*c*, draw the amplified and limited waveform and the result of sampling.

(*b*) Draw the waveform of the derivative of Fig. 22-8*h* and then the result of amplifying and limiting this derivative.

(*c*) Draw a possible flux pattern for the write current shown in Fig. 22-10*b*, together with the read-head voltage waveform and the differential spikes.

22-5. Conversion from Analog to Digital and from Digital to Analog

Analog and Digital Conversion. Mountains of raw data continue to pile up at such a rate that research laboratories and test facilities are burdened with unreduced information. The recordings produced by strip chart pens, galvanometer deflections from oscillographs, frequency-modulated carriers on magnetic tape, oscilloscope waveform photographs, as well as outputs of thermocouples, strain gauges, manometers, theodolites, etc., can have no meaning until processed and analyzed. Digital computers have the capability of processing such data at rates comparable to those with which it was produced. However, the data must first be converted into a form usable by the digital computer. Rapid automatic analog-to-digital converters can fulfill this need. In addition, when a computer is used to control experimental procedures or manufacturing processes, the digital control signals must be interpreted in terms of mechanical displacements or electrical responses; hence the need also arises for automatic and rapid digital-to-analog conversion.

Digital information is represented in the form of numbers; analog information is represented in forms other than numbers, such as a mechanically generated shaft-rotation angle or an electronically generated voltage value. In general the converters can be placed in two categories, depending on whether the analog information is in the form of mechanical or electrical measurements. Of course there can be conversion between mechanical and electrical analog information prior to (or after) the digital conversion. Converting an analog measurement into a number containing more significant digits (or bits) than required by the relative error (see Sec. 6-11) of the analog information would not be desirable. Conversely converting a digital number into an analog measurement with a system capable of smaller relative analog error than would be required by the number of significant digits in the original number is unnecessary. Thus the analog-to-digital or digital-to-analog equipment should be well matched with regard to relative error or significance. A large variety of analog-to-digital and digital-to-analog converters are available, using many different techniques adapted to different special characteristics of the analog information. In this section we shall present by example some of the basic concepts involved.

Effects of Discrete Sampling. Usually analog information is presented to the converter in what appears as a continuously varying function (often of time), such as a varying angular velocity or a voltage waveform. The result of a single digital conversion, a number, can correspond to only one point of the analog function; the digital conversions must be made for a sequence of points—*discrete samples*—of the analog function. The problem always arises of how close together the samples of analog information should be chosen, i.e., at what increments of the independent variable (e.g., time) the digital conversion of the analog representation of the dependent variable should be made. Consider, for example, a localized burst of oscillations of frequency f of the dependent variable (see Fig. 22-15). *Clearly, to ensure that at least the rms values of the variation of the oscillations will be "seen" in the sequence of digitalized values, the sampling rate must be no less than* $4f$. Figure 22-15 shows two cases of a $4f$ sampling rate, one where the maximum and the other where the rms variation value can be seen in the sampled sequence. (Note that the value seen will be at least the rms value.)

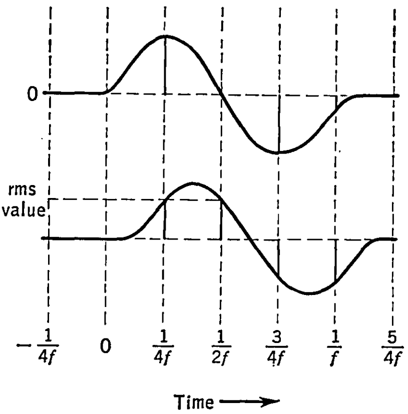
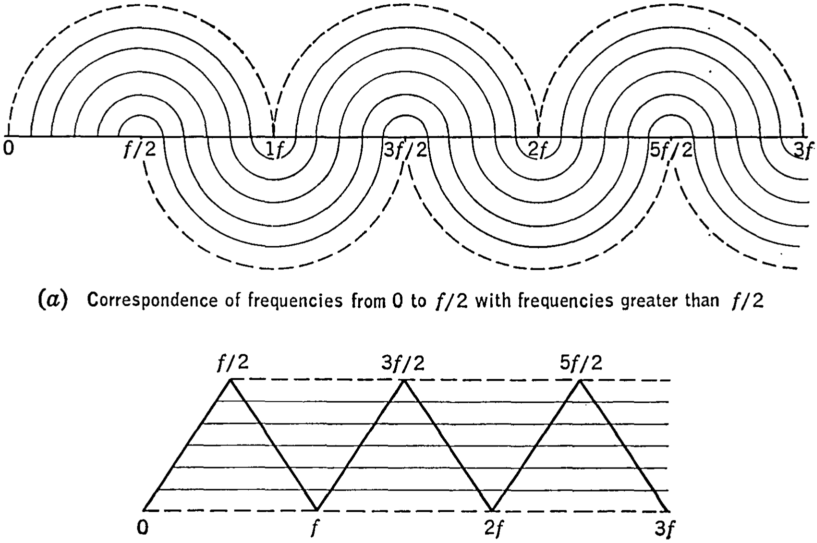


FIG. 22-15. Sampling at frequency $4f$ to obtain at least the rms value.

Consider next the problem of determining the necessary frequency of sampling that will enable the original signal to be recovered from the samples. By *recovering a signal* is meant the determination of all the coefficients of the Fourier series required for the exact reconstruction of that signal. We shall show that *it is necessary to take more than two points per cycle of the highest significant frequency component in a signal in order to recover that signal.* Consider a signal that is sampled at frequency f, that is, at times . . . , $-2/f$, $-1/f$, 0, $1/f$, $2/f$, . . . , . . . , k/f, . . . for integral k. The frequency components of the signal greater than $f/2$ *cannot be distinguished* from frequencies in the range 0 to $f/2$. To see this (noting that $f/2$ cps is the same as πf radians/sec), we need simply observe that for $0 \leq \epsilon \leq f/2$

$$\begin{aligned} &\cos\left[(n\pi f + \epsilon)t + \varphi\right] = \cos\left[(\pi f - \epsilon)t - \varphi\right] \\ \text{and}\quad &\cos\left[(n\pi f - \epsilon)t + \varphi\right] = \cos\left[(\pi f - \epsilon)t + \varphi\right] \end{aligned} \tag{22-1}$$

for odd integers $n > 0$, whenever $t = k/f$, that is, *whenever t is a sampling time.* Equations (22-1) mean that at times k/f each frequency greater than πf radians/sec, that is, $f/2$ cps, is indistinguishable from some frequency in the range 0 to $f/2$. In other words, we are dividing the frequency scale into intervals of length $f/2$ and observing that any frequency will fall in an interval $nf/2 + \epsilon$ or $nf/2 - \epsilon$ (for $0 \leq \epsilon \leq f/2$ and odd $n > 0$) and hence, at $t = k/f$, will correspond to some frequency



(*a*) Correspondence of frequencies from 0 to $f/2$ with frequencies greater than $f/2$

(*b*) The correspondence visualized as folding

FIG. 22-16. The frequency scale folded to illustrate the sampling theorem.

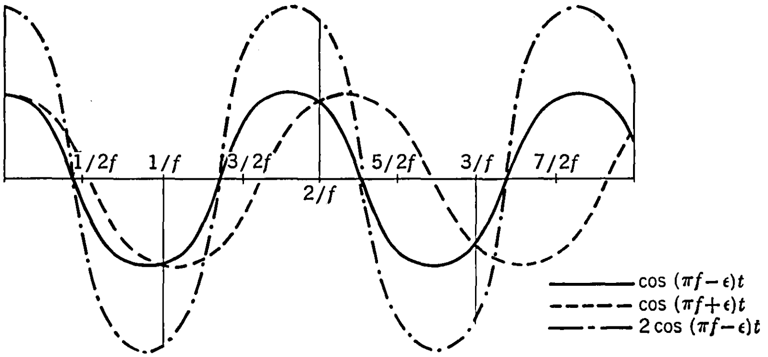


FIG. 22-17. Distortion of a frequency below half the sampling rate f by a frequency above half the sampling rate.

$\pi f - \epsilon$ in the interval 0 to $f/2$. This correspondence is illustrated in Fig. 22-16*a*, where the lines indicate the correspondences on the frequency scale. Figure 22-16*b* shows how the correspondence can be visualized as a "folding" of the frequency scale, all frequencies on the same horizontal line being indistinguishable when $t = k/f$. Figure 22-17 illustrates the effect of this correspondence. The $\cos(\pi f + \epsilon)t$ wave appears identical to the $\cos(\pi f - \epsilon)t$ wave when $t = k/f$. The result is an effective doubling of the amplitude of the $\cos(\pi f - \epsilon)t$ wave, in so far as the information contained in the samples is concerned (see the vertical lines in the figure), since, by Eqs. (22-1), $\cos[(\pi f + \epsilon)k/f] = \cos[(\pi f - \epsilon)k/f]$.

Now suppose that a signal has negligible frequencies higher than f^*. Then, in order to sample the signal unambiguously, it must be sampled at a frequency f such that $f/2 \geq f^*$, whence the *sampling frequency*

$$f \geq 2f^* \tag{22-2}$$

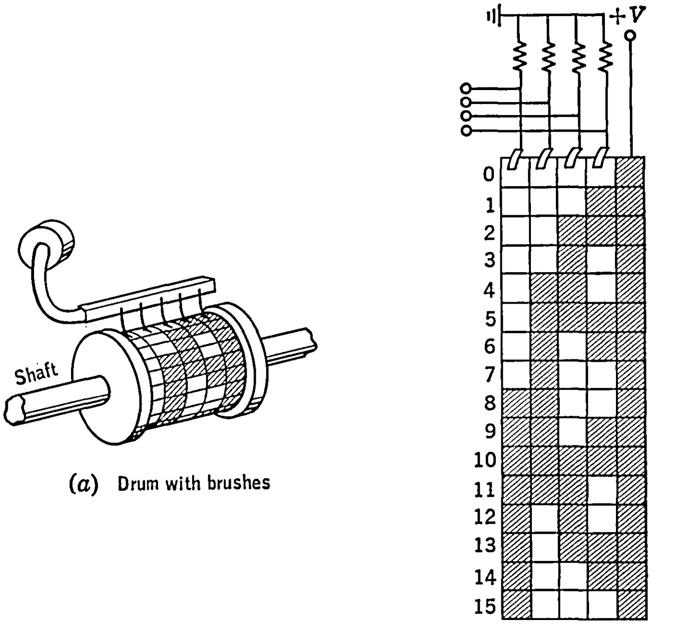
This result is called the *sampling theorem*. It is important to point out a pitfall in the use of the sampling theorem. It does *not* mean that it is necessary only to take more than two points per cycle of the highest frequency *which is of interest* in a particular application. In other words, it is not true that high frequencies cannot be passed by low-frequency sampling, since the high frequencies do affect the sample by falsely appearing as frequencies in the 0 to $f/2$ range. Another way of putting this is: "Sampling is folding; sampling is not filtering." If high frequencies are present in a function for which a low sampling rate is to be used, i.e., for which only the low frequencies are of interest, then the high frequencies *must first be filtered out* before the analog-to-digital conversion.

Mechanical Conversion. Consider first the digital conversion of the angular position of an analog shaft. This may be accomplished by means of the drum of Fig. 22-18. In this simplified illustration 16 different positions can be recognized as the four brushes sense the conducting pattern (shaded areas) on the drum surface. Notice that a cyclic code is used so that in the transition from one angular position to the next no serious error can result, since only one bit position at a time changes between successive angular positions.

The digital-to-analog conversion for an angular shaft position could make use of the analog-to-digital converter. A digital comparator determines when the angular position of the rotating shaft has the same magnitude as the digital input; at the instant of successful comparison the motor is stopped (see Fig. 22-19).

Mechanical analog-to-digital conversion and the reverse are capable of high accuracy, to over 12 significant bits. For example, the instantaneous angular position of the Pedagac drum can be obtained to within the position of a single bit, and there are $2^7 \times 23 = 2{,}944$, or more than 2^{11}, bits around a channel. This incidentally can be adapted as another method for analog-to-digital conversion.

Electronic Digital-to-analog Conversion. Let us consider two illustrations of digital-to-analog conversion. Our digital input can be represented as $N = a_{-1}2^{-1} + a_{-2}2^{-2} + a_{-3}2^{-3} + \cdots + a_{-m}2^{-m}$, where of course a_{-k} can have only the values 0 or 1 (see Sec. 3-2). Figure 22-20*a* illustrates a converter using constant-current sources, with the digital input taking the form of electronic switches opened or closed as shown,



(*a*) Drum with brushes

(*b*) Corresponding cyclic-code pattern wrapped around drum

FIG. 22-18. Direct-drive angular-shaft-position analog-to-digital converter.

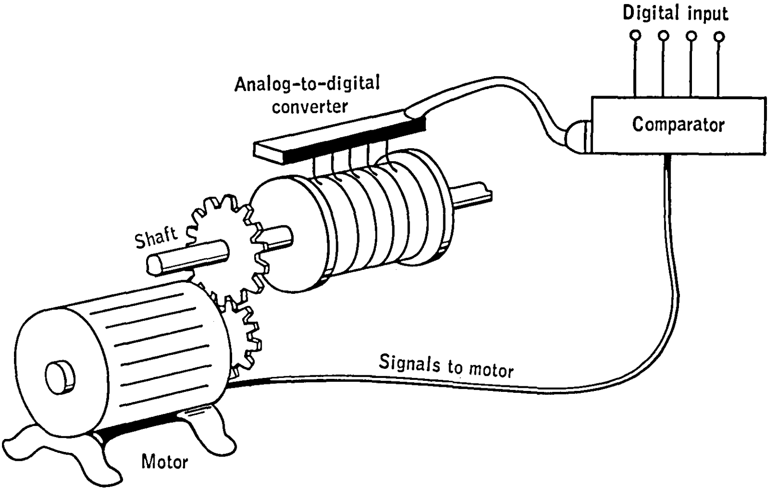


FIG. 22-19. Angular-shaft-position digital-to-analog converter.

depending on the value of a_{-k}. In the figure we have illustrated only 5 bits, though a circuit to convert any number of bits, within the limits of accuracy, may be constructed by using an extended ladder. Note that the value of the end resistances is half that of the interior resistances. The output here is the voltage V_o across the leftmost resistor. The precise analysis of the circuit will be left to the reader. But we shall note here that the three right-hand resistors can be combined into a single resistor of value R connected between point D and ground, so that, looking to the right from point C (or B, or A), we see the same resistance, $2R$, as from D. Similarly the resistance looking left from branches B, C, D, or E is $2R$, so that the current from B, C, or D divides into three equal parts, while the currents from branches A and E divide with $\frac{1}{3}I$ horizontal and $\frac{2}{3}I$ vertical. To reach the output resistor the current from branch C passes the node at B, where it is halved; also the current from branch D passes the nodes at C and B and is thus halved twice; and so forth. Therefore the current through the output resistor will be $\frac{2}{3}I$, if any, from branch A, plus $\frac{1}{3}I$, if any, from branch B, plus $\frac{1}{6}I$, if any, from branch C, plus $\cdot\cdot\cdot$; and the output will be

$$V_o = \tfrac{4}{3}IR(a_{-1}2^{-1} + a_{-2}2^{-2} + a_{-3}2^{-3} + a_{-4}2^{-4} + a_{-5}2^{-5})$$

(since $\frac{4}{3} \times 2^{-1} = \frac{2}{3}$, etc.)

A digital-to-analog converter that employs a single constant voltage source is illustrated in Fig. 22-20*b*. Note that when $a_{-1} = 1$ the a_{-1} switch presents $V - V_o$ across the $2^{-1}R$ resistance in arm A; but if $a_{-1} = 0$, the a_{-1} switch leads to ground, presenting V_o across the $2^{-1}R$ resistor, with the current in the opposite direction. The current in arm A can therefore be written as $I_A = (a_{-1}V - V_o)/2^{-1}R$, where a_{-1} is either a unit or a zero. This general form is true for the currents I_B, I_C, and I_D in arms B, C, and D, respectively. But, with no load on V_o, $I_A + I_B + I_C + I_D = 0$; hence

$$\frac{a_{-1}V - V_o}{2^{-1}R} + \frac{a_{-2}V - V_o}{2^{-2}R} + \frac{a_{-3}V - V_o}{2^{-3}R} + \frac{a_{-4}V - V_o}{2^{-4}R} = 0$$

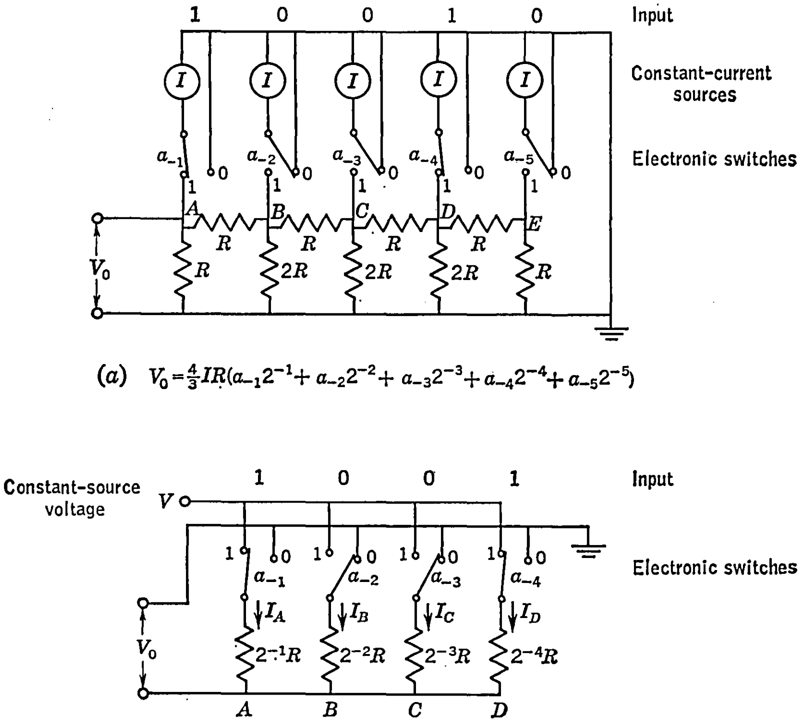
Solving for V_o,

$$\begin{aligned} V_o &= \frac{V(a_{-1}2^{-1} + a_{-2}2^{-2} + a_{-3}2^{-3} + a_{-4}2^{-4})}{2^{-1} + 2^{-2} + 2^{-3} + 2^{-4}} \\ &= \frac{V}{1 - 2^{-4}}(a_{-1}2^{-1} + a_{-2}2^{-2} + a_{-3}2^{-3} + a_{-4}2^{-4}) \end{aligned}$$

as desired.

Electronic Analog-to-digital Conversion. Three methods of analog-to-digital conversion will be described briefly. Many of the circuits involved, such as voltage comparators, voltage ramp generators, electronic switches, etc., are not primarily of a digital nature, and since their detailed design is well covered in other texts, we shall not consider it here. However, it is important to mention that many of these functions can be performed by means of operational amplifiers, used in conjunction with standard analog-computer techniques.

Our first conversion method, the feedback method, makes use of a digital-to-analog converter, which might be of the types described above. For illustration, let us assume that the range of input voltage is 0 to $2^4 - 1$ volts. The decision unit first generates the binary number 2^3,



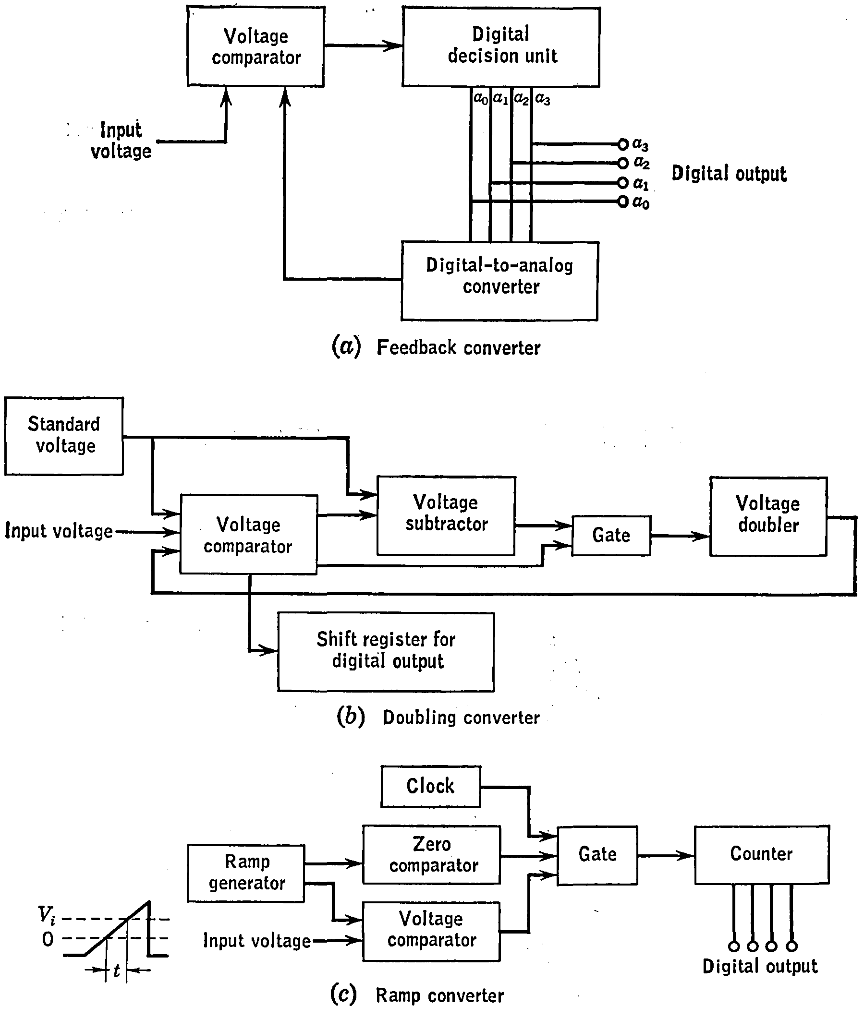
(a) $V_0 = \frac{4}{3}IR(a_{-1}2^{-1} + a_{-2}2^{-2} + a_{-3}2^{-3} + a_{-4}2^{-4} + a_{-5}2^{-5})$

(b) $V_0 = \frac{V}{1-2^{-4}}(a_{-1}2^{-1} + a_{-2}2^{-2} + a_{-3}2^{-3} + a_{-4}2^{-4})$

FIG. 22-20. Two forms of digital-to-analog converter, using (a) constant-current sources and (b) a constant-voltage source.

or 1000, by making $a_3 = 1$; this is converted to an analog voltage V_d and compared with the input voltage V_i. If $V_d \geq V_i$, then a_3 is left a unit; otherwise a_3 is returned to zero. Next a_2 is made a unit, and the analog representation of this number, namely, V'_d, is generated and compared with the input voltage. If $V'_d \geq V_i$, then a_2 is left a unit; otherwise a_2 is returned to zero. The process continues until all the bits of the converted number have been determined (see Fig. 22-21a).

The second, doubling method does not require digital-to-analog conversion. The bits of the digital result are sequentially generated, again in order from most significant to least significant bits. A standard voltage V_h, of one-half the full voltage range, is required. This is compared with the input voltage in the voltage comparator. If $V_i \geq V_h$, the most significant bit is a unit; if $V_i < V_h$, it is a zero. If it is a unit, then $V_i - V_h$ is formed in the voltage subtractor and sent to the voltage doubler. If it is a zero, then V_i is sent directly to the voltage doubler.



(a) Feedback converter

(b) Doubling converter

(c) Ramp converter

FIG. 22-21. Three types of electronic analog-to-digital converters.

Here either $2(V_i - V_h)$ or $2V_i$ is formed and sent back as an input to the voltage comparator to be compared once more with V_h. The result of this comparison determines the next most significant bit, and the process continues until all the bits of the number are formed (see Fig. 22-21b). The two methods just described take one unit time interval for each bit of the number generated.

The third, or ramp, method is essentially a counting method, but it does not require extensive control circuitry. A linearly increasing voltage generator is initiated and its output sent to a zero comparator. When the voltage reaches the zero level, the counter is activated and begins to count clock pulses. Meanwhile the ramp, or increasing, voltage is being compared with the input voltage. When the ramp voltage equals the input voltage, the counter is stopped; the final digital count so obtained is proportional to the input voltage. In this method the time required depends on the magnitude of the analog voltage (see Fig. 22-21*c*).

EXERCISES

(*a*) Describe both of the electronic digital-to-analog methods for n bits (i.e., develop the appropriate formulas for a general integer n).

(*b*) Logically design the decision unit for the analog-to-digital converter of Fig. 22-21*a*.

22-6. Input-Output Methods

Electronic Digital Transducers. Almost all information put into a computer is generated in other than electronic binary form. Such input information may be generated by a computer programmer in the form of handwritten numbers on a coding sheet, by a strain gauge in the form of a voltage waveform, by missiles in the form of a radar signal, by bank checks in the form of printed serial numbers, by an experimental process in the form of the sequential frames of motion-picture film, and so forth. Similarly output information generated by the computer in electronic binary form usually must be converted into a more easily assimilated form: for example, the conversion of generated functions into graphs or tables, of military data into map or pictorial displays, of business information into bills and invoices, of processed statistical data into printed results, and so forth. A transducer may be considered a device to convert information from one form to another; if one of these is an electronic binary or digital form, the device is called an electronic digital transducer. The study of such computer input-output transducers is in itself a large field, which cannot be adequately covered in this short section. The methods and techniques involved are not, for the most part, closely related to digital circuits. However, brief descriptions of some specific examples will give the reader an understanding of many of the general problems that occur. The use of conventional electric-typewriter, punched-paper-tape, punched-card, and magnetic-tape inputs and outputs for computers has already been mentioned. We shall therefore in this section consider other forms of input-output devices, stressing the digital aspects of these mechanisms.

Let us first note that input digital transducers must perform three functions: The input information must first be put into digital form by an analog-to-digital converter or by some switching mechanism such as brushes sensing holes in cards, or intermittent light sensed by a phototube. Next the digital information must be converted into the proper electrical form, using conventional electronic techniques. Finally the electronic bits must be transformed into a word format and appropriately synchronized with the clock of the computer. Output digital transducers must perform the reverse process. Some aspects of these problems have already been considered when the push buttons and input-output buffer of the Pedagac were described (Sec. 18-8).

Direct Switched Input. Other than push buttons, the so-called *joy stick* is the simplest mechanism for computer input. It consists of a stick that can be moved in two dimensions; the displacement of the stick is mechanically digitalized and recorded directly in a special word of the computer's memory, analogously to the signal sensing word of Sec. 8-3. One typical application of a joy stick is as an aid in tracking aircraft, inserting into the computer the X, Y coordinates of a radar-signal pip on a cathode-ray tube. A code is written so that the displacement of the joy stick directs, through the computer, the position of a spot on the cathode-ray tube. Then by means of the joy stick the operator maneuvers the spot until it appears directly over the radar-signal pip. Then a button is pushed, and the computer records the position of the joy stick, which now gives the position of the radar-signal pip (see Fig. 22-22). A great many variations of this technique exist. For example, the position of the spot may be directly proportional to the displacement of the stick; a foot pedal might be added so that when it is pushed the spot will acquire a velocity proportional in magnitude and direction to the displacement of the stick.

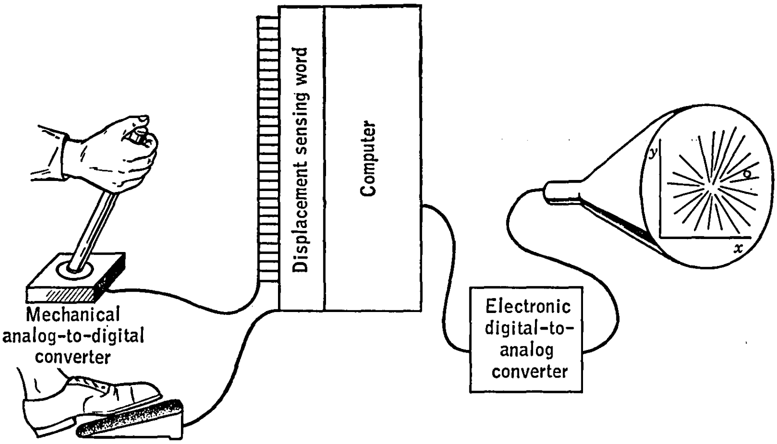
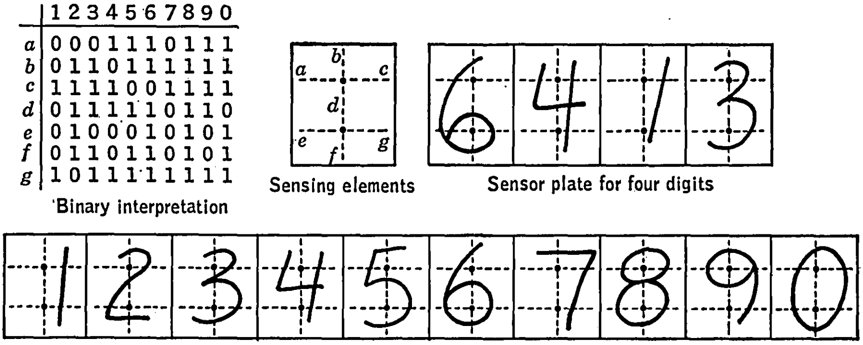

Displacement sensing word
Computer
y
x
Mechanical analog-to-digital converter
Electronic digital-to-analog converter

FIG. 22-22. Joy stick and display system.

Manual Write-in. Another simple input to a computer is a writing sensor plate, whose purpose is to enable numbers to be written directly into the computer. The scheme is based on a system of writing numbers around a pair of dots previously printed on a paper. The paper is placed on a board that senses whether the pencil crosses the lines *a*, *b*, *c*, *d*, *e*, *f*, and *g* as a digit is written in each position (see Fig. 22-23). Each digit crosses the lines or not in a different pattern, as shown in the array of the figure: a unit means that the pencil crosses the corresponding line when writing that number, and a zero that it does not, as recorded by the sensing plate. After the number has been completely written, the pencil is touched to the square in the lower right-hand corner, and the 7 bits of each number are recorded by the computer. The computer then interprets this code appropriately. A similar innovation can be made for the entire alphabet, printed or script. An application of such a scheme is to enable many long-distance telephone operators to record telephone numbers into an automatic bill-forming computer.



	1	2	3	4	5	6	7	8	9	0
a	0	0	0	1	1	1	0	1	1	1
b	0	1	1	0	1	1	1	1	1	1
c	1	1	1	1	0	0	1	1	1	1
d	0	1	1	1	1	1	0	1	1	0
e	0	1	0	0	0	1	0	1	0	1
f	0	1	1	0	1	1	0	1	0	1
g	1	0	1	1	1	1	1	1	1	1

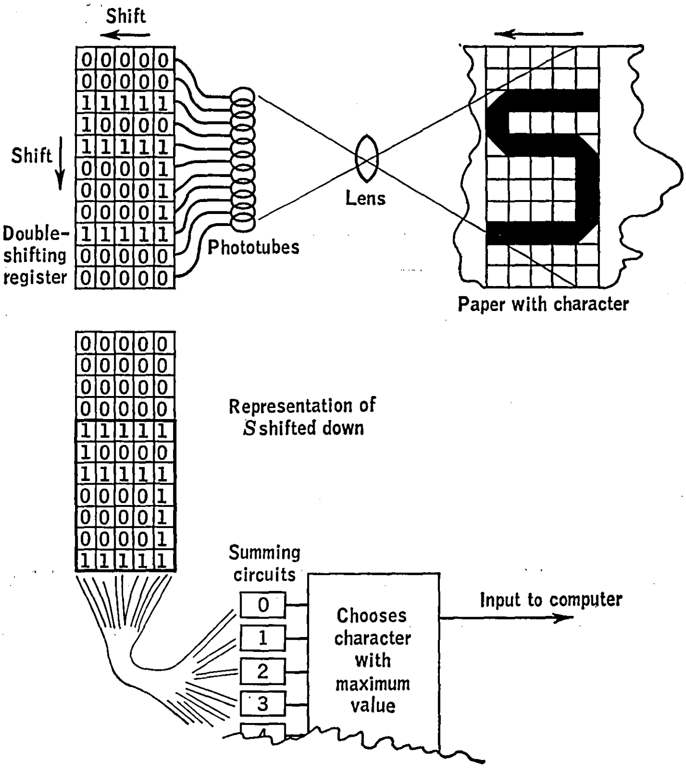
Binary interpretation

FIG. 22-23. Writing sensor plate and binary interpretation of digits.

Character-reading Machines.† There are many methods for automatically reading printed matter into a computer. We have chosen to describe a method that utilizes digital circuitry extensively and is capable of reading 10,000 characters/sec. As shown in Fig. 22-24, a lens focuses a vertical-strip area of the character onto sensing phototubes. The phototubes divide the strip image into 11 squares, considering each as black (a unit) or white (a zero). These 11 bits are then recorded in 11 rows of a shift register. The character is sensed in this way five times as the paper on which it is printed moves past the lens. Each time, each of the 11 shift-register rows is shifted to accommodate the new information, which enters only on the right. When the fifth strip is sensed, the full character has been recorded as shown, as zeros or units in a 5 × 11 matrix. The machine is able to separate characters as they come past because the shifting is initialed only when the first black area is sensed; then the machine shifts only five times and after a short delay waits for the next black area. To avoid errors due to varying vertical displacement of the characters on the paper, after the 11 shift-register

† Based on private communication with J. Rabinow, Rabinow Engineering Co., Takoma Park, Md.

rows have been loaded the columns become shift registers, and the recorded information is shifted down until a unit reaches the bottom row. The value (0 or 1) of each bit in the registers can now be sensed by two wires from each of the bit locations, the direct and inverted outputs. Seven wires (for example) are chosen from critical areas for each character, so that if that character is recorded the voltages of the seven wires will add up to seven times the voltage on each wire. In order to determine which character is recorded in the registers, we need merely look for the one corresponding to the *maximum* voltage. This maximum type of decision is used to avoid errors due to broken printed characters, dirt spots, etc.

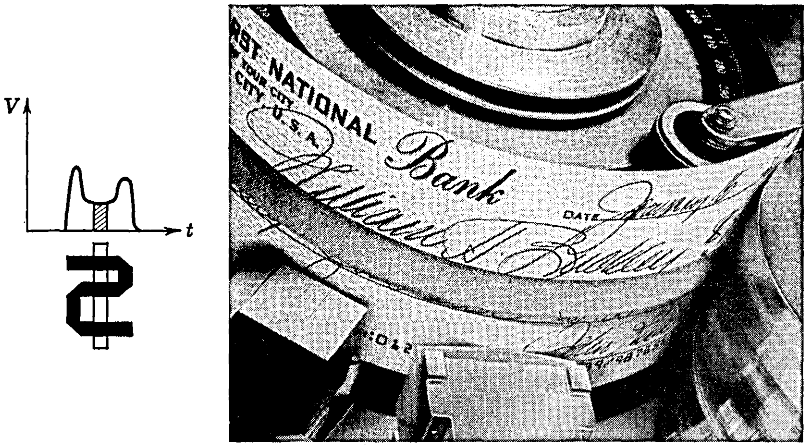


0	0	0	0	0
0	0	0	0	0
1	1	1	1	1
1	0	0	0	0
1	1	1	1	1
0	0	0	0	1
0	0	0	0	1
0	0	0	0	1
1	1	1	1	1
0	0	0	0	0
0	0	0	0	0

0	0	0	0	0
0	0	0	0	0
0	0	0	0	0
0	0	0	0	0
1	1	1	1	1
1	0	0	0	0
1	1	1	1	1
0	0	0	0	1
0	0	0	0	1
0	0	0	0	1
1	1	1	1	1

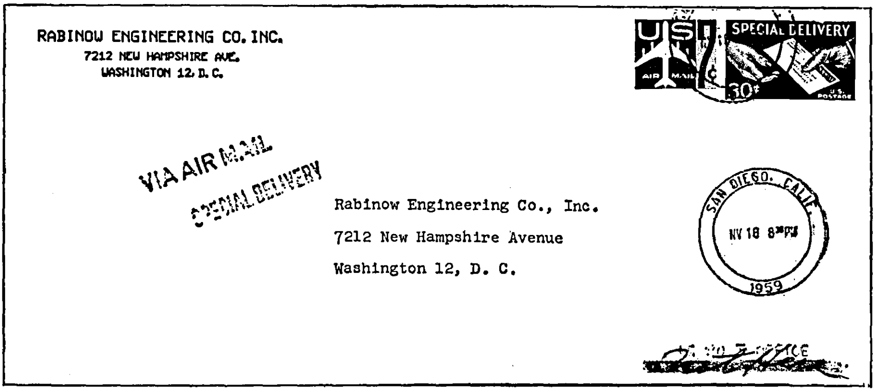
FIG. 22-24. Parallel-scan character-reading machine.

Another technique has the characters printed in magnetic ink. Again a vertical strip is sensed, this time by a magnetic recording head; an induced voltage waveform is generated, characteristic of the figure passing by (see Fig. 22-25*a*). The characters are so designed that the waveforms can easily be distinguished. This method has been standardized for reading serial numbers on bank checks (see Fig. 22-25*b*). For sorting mail automatically still another scheme has been proposed. On the lower right-hand corner of each letter is sprayed a strip of magnetic material, which acts like magnetic tape. An operator reads the hand- or typewritten address and with a keyboard puts a coded address onto the



(*a*) (*b*)

FIG. 22-25. Magnetic printing and character recognition. (*a*) Induced voltage waveform when magnetic head covers area shown as the number moves past; (*b*) check-reading machine showing the rotating drum that holds the check and the magnetic head that reads the stylized numbers. (*Photograph courtesy International Business Machines Corp.*)



RABINOW ENGINEERING CO. INC.
7212 NEW HAMPSHIRE AVE.
WASHINGTON 12, D. C.

US AIR MAIL SPECIAL DELIVERY 30¢ U.S. POSTAGE

VIA AIR MAIL
SPECIAL DELIVERY

Rabinow Engineering Co., Inc.
7212 New Hampshire Avenue
Washington 12, D. C.

SAN DIEGO, CALIF.
NV 18 8 PM
1959

FIG. 22-26. A letter that was sent across the country, showing the magnetic-ink strip in the lower right-hand corner. Even though "Not Here" was written over the strip, the code was not disturbed. (*Photograph courtesy Rabinow Engineering Co., Inc.*)

strip as the letter passes by the magnetic writing head. This magnetic address can be read for automatic sorting at various postal distribution points. Writing or stamping over the strip does not affect the code (see Fig. 22-26).

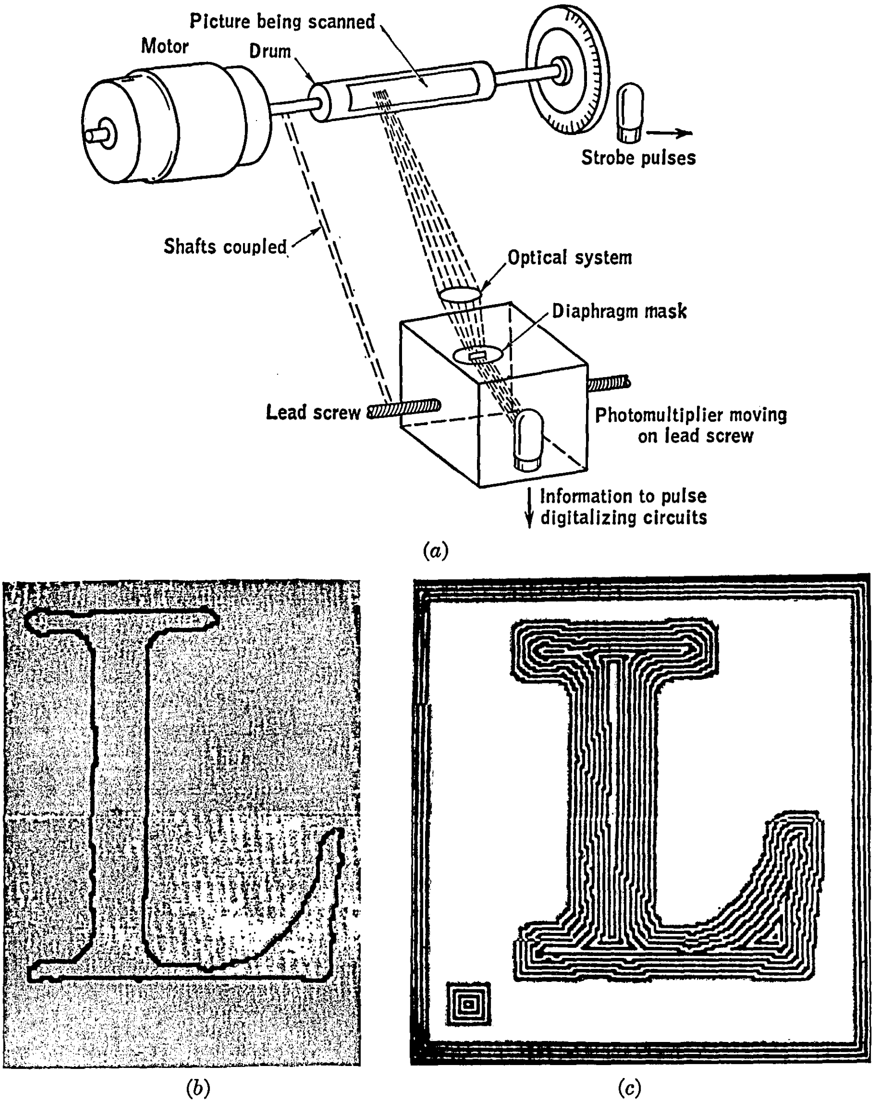


FIG. 22-27. (a) Mechanism of picture reader; (b) finding the boundary of the letter L; (c) the result of repeating the process; note what happened to a spot of dirt in the lower left-hand corner.

Picture-reading Machine. An input device of particular interest is a picture reader. Here a photomultiplier tube scans the picture as if it were marked off in squares like graph paper. If a particular square is darker than some relative index, then this square is recorded in the computer as a unit, otherwise as a zero (see Fig. 22-27a); in this way the picture is recorded in the memory. The contents of the memory can be displayed on a cathode-ray tube so that a unit bit presents a bright spot and a zero a dark spot. This display indicates essentially how the zeros and units actually appear in the memory. Such a picture-reading machine has great potentialities. Maps may be rapidly recorded in a computer's memory for weather investigation, real-estate studies, or the simulation of military situations. Drawings of electronic circuits, chemical structures, and mechanical configurations can be recorded in the computer's memory for patent searching or other information retrieval processing. A pair of stereographic aerial photographs can be recorded in the computer's memory and the computer programmed to produce a corresponding contour map, and so forth. As an example of picture processing, Fig. 22-27*b* shows a letter *L* processed so that only the boundary appears black. Processing this figure in the same way over and over again, we obtain the shape shown in Fig. 22-27*c*.

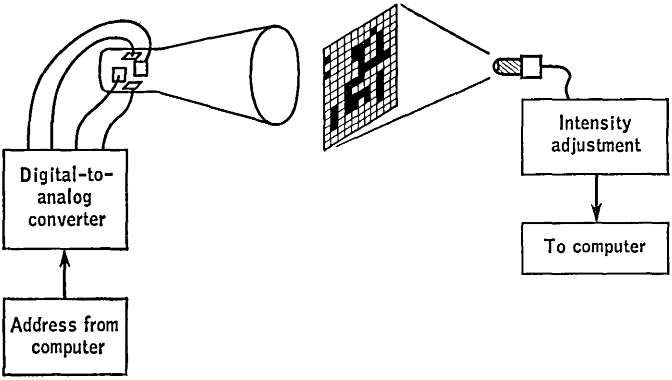


FIG. 22-28. Picture-reading machine using negative.

A second type of picture-reading machine places a negative or a positive transparency between a cathode-ray tube and a phototube. The computer is programmed to direct the electron beam scanning the picture and simultaneously to record from the phototube a unit or a zero (see Fig. 22-28).

Large Fixed Memory.† There are many applications of computers that require a fixed memory, such as a dictionary for a language-translating machine or an address-zone number correspondence for a mail-sorting machine. Of the many mechanisms available we shall describe one that depends on the transmission of light through aligned holes in plates. Consider, for example, a memory with a 3-bit address, with each of the eight addresses containing 4 bits (see Fig. 22-29). The memory consists of a cover plate and of three other plates each of which can be moved laterally into either of two possible positions, corresponding to a unit and a zero. Behind the plates four phototubes each sense one-fourth the area of the plates representing one of the 4 bits. An address is represented by the appropriate displacement of the three mov-

† Based on private communication with J. Rabinow.

able plates. Each of the four areas represents one of the four bit positions of a word. The presence of a unit (or zero) in a bit position is sensed if light passes through aligned holes (or not) in the appropriate area of all three of the plates. To insert a word initially, the plates are shifted according to the desired address, and holes are made through all three plates in only those areas representing unit bit positions. In practice holes are prepunched in all the plates and then filled by a layer of paint. Then, to insert a unit, a needle is passed through the holes, easily punching out the paint layer. *Each of the bit areas of the cover plate has as many holes in it as there are addresses.* These holes serve as guides for inserting units into the corresponding addresses. The figure shows the three plates in position for address 010; as shown, the contents of this address is 1010. A memory of this type has been built for 3,000 addresses, each of 12 bits. Frequently it is desired to use more movable plates than necessary, to give the address registration more flexibility.

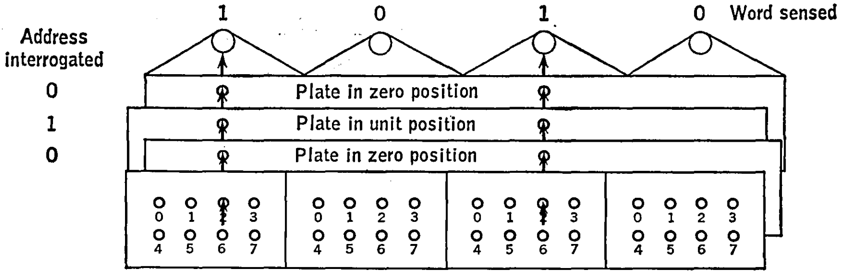


FIG. 22-29. Plate memory.

Another kind of fixed memory can be made with the cathode-ray picture reader described above. Here information can be read onto unexposed film by means of the beam spot; the processed negative can then be sensed by the same setup. An address might be a row number, and the bits in a row be the bits of the word, read out in a single scan.

High-speed Printers. Mechanical high-speed printers (see Fig. 22-30) can print 600 to 1,000 lines per minute, at 120 characters per line. A rotating drum contains 120 channels of metallic type-face forms, each channel having the complete alphabet and the digits. The paper lies between the type drum and a row of electronically controlled hammers, one for each channel. The entire line to be printed is recorded in a buffer storage unit. In its simplest form the binary code for each type face is the angle of rotation at which this type face will be opposite the hammer. The angle of the drum, converted to digital form, is compared with the character code for each channel. When a successful comparison is made, the hammer bangs the paper against this type face, printing the character. In a single revolution of the drum the proper character from each channel is printed—i.e., one line is printed per revolution of the drum.

An even higher-speed printer is based on the so-called Charactron tube. By this method over 4,500 lines of 120 characters can be printed per minute. At this rate a 300-page book can be printed in 3 min. The Charactron tube displays alphabetic and numerical symbols by shaping the electron beam. The beam is shaped by passing it through a stencil plate with holes in the shape of letters (see Fig. 22-31*a*). The characters are formed by directing the beam through the appropriate hole. Masks with 64 characters have been made. This technique is coupled with that of the xerography process as follows (see Fig. 22-31*b*): A selenium-coated rotating drum is charged and exposed to the image of the characters formed by the Charactron tube. Special ink powder can then be transferred from the exposed part of the drum to paper. There are seven stages in this transfer process: the selenium drum is charged; the Charactron image is projected on the surface; the special ink powder is put on; the blank paper is charged; the image is transferred to the paper; the image is heat-fixed on the paper; and finally the drum is cleaned for the repeat process.

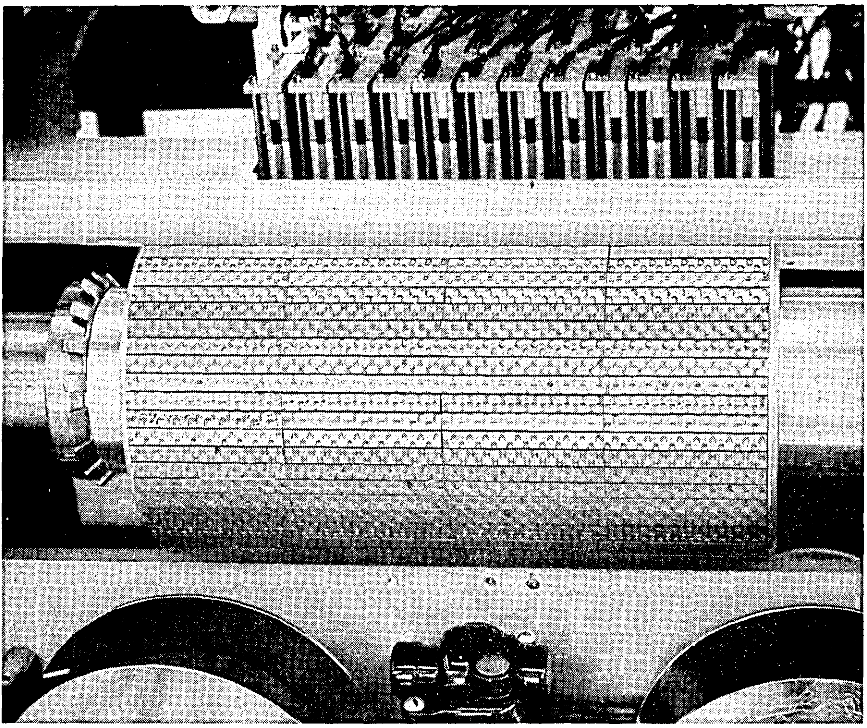


FIG. 22-30. Mechanical high-speed printer showing type-face drum. (*Photograph courtesy Potter Instrument Co., Inc.*)

Picture Displays. Cathode-ray tubes can be used to display the plots of graphs by sending the x and y coordinates of each point into a digital-to-analog converter to control the voltages for the x and y plates of the tube. In a similar fashion a television tube can be connected to a magnetic drum, so that as the drum rotates each line is displayed by the tube. In this setup the picture display is repeated each time the drum makes a revolution. To alter the picture, the programmer merely changes the pattern of bits, zeros and units, on the drum. In this way the computer can display a moving picture of the successive results of computation. Simulations carried out by the computer, such as war games, traffic queueing problems, etc., can be directly displayed. Another scheme is to use an electroluminescent surface, which emits light when a voltage is applied across it. Wires buried in the surface can be controlled by the computer to generate voltages, and hence light, as desired for picture displays.

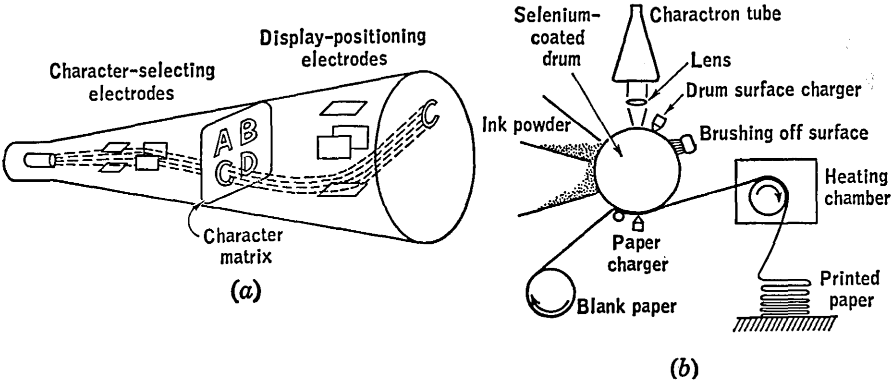


FIG. 22-31. The Charactron tube coupled with the xerography process for high-speed printing.

EXERCISES

(*a*) Write a program for the Pedagac that will enable a joy stick and foot pedal to be used as described in this section.

(*b*) Suppose in the manual write-in system that the 1 is sometimes written to the left of the two dots, that the top of the 2 may curl round too far, that the middle of the 3 does not always cross the line connecting the two dots (line *d*), and that the tails of the 7 and the 9 are occasionally placed on the wrong side of the lower dot. How can the system handle such variations?

(*c*) For the manual write-in system, using four dots and 12 lines, demonstrate a way of writing the letters of the alphabet so that they may be distinguished, as well as the digits. Develop the binary coding table.

(*d*) Logically design the two-way shift register required for the parallel-scan character-reading machine, using dynamic flip-flops.

(*e*) How would the *G-H* connections be made between flip-flops of the type described in Sec. 20-6 for the two-way shifting register of the parallel-scan reading machine?

(*f*) Write a code for the Pedagac that will develop the border, or outline, of a picture recorded in its memory.

22-7. Additional Topics

a. Selected Bibliography on Memory Systems

Alexander, M. A., and M. Rosenberg: Ferrite-core Memory Is Fast and Reliable, *Electronics*, vol. 29, no. 2, pp. 158–161, February, 1956.

Bivans, E. W.: Synchronizing Magnetic Drum Storage Speed, *Electronics*, vol. 28, no. 8, pp. 140–141, August, 1955.

Blachman, N. M.: On the Wiring of Two-dimensional Multiple-coincidence Magnetic Memories, *IRE Trans. on Electronic Computers*, vol. EC-5, pp. 19–21, March, 1956.

Booth, A. D.: A Magnetic Drum Digital Storage System, *Electronic Eng.*, vol. 21, pp. 234–238, July, 1949.

Buck, D. A., and W. I. Frank: Nondestructive Sensing of Magnetic Cores, *Communs. and Electronics*, vol. 72, no. 10, pp. 822–830, January, 1954.

Cohen, A. A.: Magnetic Drum Storage for Digital Information Processing Systems, *Mathematical Tables and Other Aids to Computation*, vol. 4, pp. 31–39, January, 1950.

Comstock, G. E.: 500,000,000-bit Random Access Memory, *Instr. and Automation*, vol. 29, pp. 2208–2211, November, 1956.

Conger, R. L.: Flat Plate Memory, *Naval Ordnance Lab. Tech. Mem.* 72-6, Nov. 14, 1955.

Digital Magnetic Circuits—Theory, Electrical Design, and Logical Design, *Burroughs Corp. Tech. Rept.* TR 59-27, vols. 1–3, April, 1959.

Eckert, J. P., Jr.: Survey of Digital Computer Memory Systems, *Proc. IRE*, vol. 41, pp. 1393–1406, October, 1953.

Everett, R. R.: Selection Systems for Magnetic Core Storage, *MIT Servomechanisms Lab. Eng. Note* E-413, August, 1951.

Forrester, J. W.: Digital Information Storage in Three Dimensions Using Magnetic Cores, *J. Appl. Phys.*, vol. 22, pp. 44–48, January, 1951.

Foss, E., and R. S. Partridge: 32,000-word Magnetic-core Memory, *IBM J. Research Develop.*, vol. 1, pp. 103–109, April, 1957.

Fuller, H. W., P. A. Husman, and R. C. Kelner: Techniques for Increasing Storage Density of Magnetic Drum Systems, *Proc. Eastern Joint Computer Conf.*, December, 1954, Philadelphia, pp. 16–21.

Guterman, S., and R. D. Kodis: Magnetic Core Selection System, *IRE Conv. Record*, pt. 4, pp. 116–123, 1954.

———, ———, and S. Ruhman: Logical and Control Functions Performed with Magnetic Cores, *Proc. IRE*, vol. 43, pp. 291–298, March, 1955.

Hagen, G. E.: Air Floating, a New Principle in Magnetic Recording of Information, *Computers and Automation*, vol. 2, no. 8, pp. 23–25, November, 1953.

Helbig, W. A., W. G. Rumble, and C. S. Warren: Transistor Operated Magnetic Core Memory, *Proc. Natl. Electronics Conf.*, vol. 12, pp. 289–299, 1957.

High-density Magnetic Recording, *Harvard Computation Lab. Progr. Rept.* 34, pt. II, August, 1954.

High-density Tape Recordings for Digital Computers, *Elec. Mfg.*, vol. 290, p. 153, November, 1955. (Work done at the National Bureau of Standards.)

Hoagland, A. S.: A Logical Reading System for Nonreturn-to-zero Magnetic Recording, *IRE Trans. on Electronic Computers*, vol. EC-4, pp. 93–95, September, 1955.

———: Magnetic Data Recording Theory: Head Design, *Trans. AIEE*, pt. I, *Communs. and Electronics*, vol. 75, pp. 506–512, November, 1956.

———: Magnetic Recording Head Design, *Proc. Western Joint Computer Conf.*, February, 1956, San Francisco, pp. 26–31.

Hoh, A. W.: A Preliminary Report on the Prototype Diode-capacitor Memory, *Proc. Argonne Conf. Digital Computers,* August, 1953. (See also *NBS Rept.* 2713, June, 1953, and *NBS Rept.* 3374, May 1, 1954.)

Lubkin, S.: An Improved Reading System for Magnetically Recorded Digital Data, *IRE Trans. on Electronic Computers,* vol. EC-3, pp. 22–25, September, 1954.

Magnetic Memory Element Exhibits 4-millimicrosecond Switching Speed, *Elec. Eng.,* vol. 78, pp. 236–237, March, 1959.

McGuigan, J. H.: Combined Reading and Writing on a Magnetic Drum, *Proc. IRE,* vol. 41, pp. 1438–1444, October, 1953.

Mee, C. D.: Magnetic Tape for Data Recording, *Proc. IEE,* vol. 105, pp. 373–380, July, 1958. (Discussion on pp. 380–382.)

Nordyke, H. W.: Magnetic Tape Recording Techniques and Performance, *Proc. AIEE-IRE-ACM Computer Conf.,* December, 1952, New York, pp. 90–95.

Noyes, T., and W. E. Dickinson: Engineering Design of a Magnetic-disk Random-access Memory, *Proc. Western Joint Computer Conf.,* February, 1956, San Francisco, pp. 42–44.

——— and ———: The Random-access Memory Accounting Machine: II. The Magnetic-disk Random-access Memory, *IBM J. Research Develop.,* vol. 1, pp. 72–75, January, 1957.

Papian, W. N.: Ferromagnetic Cores for Three-dimensional Digital Storage Arrays, *Proc. IRE,* vol. 39, pp. 292–315, March, 1951.

———: New Ferrite-core Memory Uses Pulse Transformers, *Electronics,* vol. 28, no. 3, pp. 194–197, March, 1955.

———: A Coincident Current Magnetic Memory Cell for the Storage of Digital Information, *Proc. IRE,* vol. 40, pp. 475–478, April, 1952.

Potter, J. T., and P. C. Michel: High-density Digital Recording System, *IRE Trans. on Electronic Computers,* vol. EC-1, pp. 60–72, December, 1952.

Raffel, J., and S. Bradspies: Experiments on a Three-core Cell for High-speed Memories, *IRE Conv. Record,* pt. 4, pp. 64–69, 1955.

Rajchman, J.: A Myriabit Magnetic-core Matrix Memory, *Proc. IRE,* vol. 41, no. 10, pp. 1407–1421, October, 1953.

Renwick, W. A.: A Magnetic Core Matrix Store with Direct Selection Using a Magnetic Core Switch Matrix, *Proc. IEE,* vol. 104, pt. B, suppl. no. 7, pp. 436–444, 1957.

Robinson, A. A., V. L. Newhouse, M. J. Friedman, D. G. Bindon, and I. P. V. Carter: A Digital Store Using a Magnetic Core Matrix, *Proc. IEE,* vol. 103, pt. B, suppl. 2, Convention on Digital-computer Techniques, pp. 295–301, April, 1956.

Schmidbauer, O.: The Determination of Frequency Characteristics of Recording-tape Magnetization, *Elektron. Rundschau,* vol. 11, pp. 373–375, December, 1957.

Stuart-Williams, R., M. Rosenberg, and M. A. Alexander: "Recent Advances in Coincident-current Magnetic Memory Techniques," International Telemeter Corp., Los Angeles, 1954. (This paper was presented at the annual meeting of the Association for Computing Machinery, Ann Arbor, June, 1954.)

Summarized Proceedings of the Conference on Solid State Memory and Switching Devices, *Brit. J. Appl. Phys.,* vol. 10, pp. 153–158, April, 1959.

Taylor, N. H.: Rapid-access Storage Including the Use of Magnetic Cores for Storage and Switching, *Proc. IEE,* vol. 103, pt. B, suppls. 1–3, pp. 327–330, 1956.

Thorensen, R., and W. R. Arsenault: A New Nondestructive Read for Magnetic Cores, *Proc. Western Joint Computer Conf.,* 1955, pp. 111–116.

Transistorized Magnetic Core Memory, *Computers and Automation,* vol. 6, pp. 26–27, January, 1957.

Welsh, H. F., and H. Lukoff: The Uniservo—Tape Reader and Recorder, *Proc. AIEE-IRE-ACM Computer Conf.,* December, 1952, New York, pp. 47–53.

Wier, J. W.: A High-speed Permanent Storage Device, *IRE Trans. on Electronic Computers*, vol. EC-4, pp. 16–20, March, 1955.

Wilkes, M. V., and D. W. Willis: A Magnetic-tape Auxiliary Storage System for the EDSAC, *Proc. IEE*, vol. 103, pt. B, suppl. 2, Convention on Digital-computer Techniques, pp. 337–345, April, 1956.

Williams, F. C., and J. C. West: Position Synchronization of a Rotating Drum, *Proc. IEE*, vol. 98, pt. II, pp. 29–34, February, 1951.

Younker, E. L.: A Transistor-driven Magnetic-core Memory, *IRE Trans. on Electronic Computers*, vol. EC-6, pp. 14–20, March, 1957.

b. Selected Bibliography on Thin-film Techniques

Bittmann, E. E.: Thin Film Memories, Digest of Technical Papers, p. 20, IRE-AIEE Solid-state Circuits Conference, 1959, University of Pennsylvania.

Blois, M. S.: Preparation of Thin Magnetic Films and Their Properties, *J. Appl. Phys.*, vol. 26, pp. 975–980, August, 1955.

Callen, H. B.: Rotational Remagnetization of Thin Films, Digest of Technical Papers, p. 96, IRE-AIEE Solid-state Circuits Conference, 1959, University of Pennsylvania.

Conger, R. L.: High Frequency Effects in Magnetic Films, *Proc. AIEE Conf. on Magnetism and Magnetic Materials*, 1956, pp. 610–619.

———: Magnetization Reversal in Thin Films, *Phys. Rev.*, vol. 98, pp. 1752–1754, June, 1955.

———: Thin Film Magnetization Reversal Studies, *Proc. Nonlinear Magnetics and Magnetic Amplifiers, Special Tech. Conf.*, August, 1958.

Experimental Production of Thin Ferrite Films and a Survey of the Magnetic Properties of Thin Films, *NAVORD Rept.* 3962, March, 1955. (Also available from Office of Technical Services, U.S. Department of Commerce, as PB 121177.)

Fowler, C. A., E. M. Fryer, and J. R. Stevens: Magnetic Domains in Evaporated Thin Films of Nickel-Iron, *Phys. Rev.*, vol. 104, pp. 645–649, November, 1956.

Gianola, U. F.: Nondestructive Memory Employing a Domain Oriented Steel Wire, *J. Appl. Phys.*, vol. 29, pp. 849–853, May, 1958.

Goodenough, J. B., and D. O. Smith: Magnetic Properties of Thin Films, *MIT Lincoln Lab., Tech. Rept.* 197, *ASTIA* 210730, January, 1959.

House, C. B., and R. L. Van Allen: Nondestructive Readout of Multilevel Magnetic Memory, *NRL Rept.* 5071, February, 1958. (Also available from Office of Technical Services, U.S. Department of Commerce, as PB 131475.)

Kittel, C.: Theory of the Structure of Ferromagnetic Domains in Films and Small Particles, *Phys. Rev.*, vol. 70, pp. 965–971, December, 1946.

Kudela, A. F.: The Thin Film, Flat Plate, Printed Circuit Memory, *Naval Ordnance Lab. Tech. Mem.* 72-7, Dec. 20, 1955.

Oakland, L. J., and T. D. Rossing: Coincident-current Nondestructive Readout from Thin Magnetic Films, *J. Appl. Phys.*, suppl. to vol. 30, no. 4, pp. 54S–55S, April, 1959.

Olson, C. D., and A. V. Rohm: Flux Reversal in Thin Films of 82% Ni, 18% Fe, *J. Appl. Phys.*, vol. 29, pp. 274–282, March, 1958.

Olson, E. R.: Research and Development on Magnetic Films and Magnetic Matric Memory Units, *Final Rept.* AD-151 160, Servomechanisms, Inc., Contract AF 33(616)3039, WADC TR 57-707, April, 1958. (Available also from Office of Technical Services, U.S. Department of Commerce, as PB 135053.)

Raffel, J. I.: Operating Characteristics of a Thin Film Memory, *J. Appl. Phys.*, vol. 30, pp. 60S–61S, April, 1959.

Rohm, A. V., and S. M. Rubens: A Compact Coincident-current Memory, *Proc. Eastern Joint Computer Conf.*, December, 1956, pp. 120–123.

Rossing, T. D., and R. M. Sanders: Reversible Rotation in Magnetic Films, *J. Appl. Phys.*, vol. 29, pp. 288–289, March, 1958.

Smith, D. O.: Magnetization Reversal and Thin Films, *J. Appl. Phys.*, vol. 29, pp. 264–273, March, 1958.

———: Thin Magnetic Films for Digital Computer Memories, *Electronics*, June 26, 1959, p. 44.

c. Articles on Analog-to-digital and Digital-to-analog Conversion Processes

Bain, M. B.: Precision Digital-to-analog Conversion Methods for Graphical Plotters, *Proc. Natl. Electronics Conf.*, 1955, pp. 268–277.

Barker, R. H.: A Transducer for Digital Data Transmission Systems, *Proc. IEE*, vol. 103, pt. B, pp. 42–51, January, 1956.

Beckett, F.: A Rapid Digital-to-analogue Converter for Numbers Having Eleven Binary Digits, *Proc. IEE*, vol. 103, pt. B, suppl. 3, Convention on Digital-computer Techniques, pp. 427–431, April, 1956.

Brown, J. H.: Measure Motion to 0.0001 Inch without Friction or Wear, *Control Eng.*, vol. 2, no. 4, pp. 50–52, April, 1955.

Burke, H. E.: A Survey of Analog-to-digital Converters, *Proc. IRE*, vol. 41, pp. 1455–1462, October, 1953.

Cronin, F. R.: Automatic Converter to Prepare Analog Data for Digital Computation, M.S. thesis, Electrical Engineering Department, Massachusetts Institute of Technology, June, 1956.

Elliot, W. S., R. C. Robbins, and D. S. Evans: Remote Position Control and Indication by Digital Means, *Proc. IEE*, vol. 103, pt. B, suppl. 3, Convention on Digital-computer Techniques, pp. 437–446, April, 1956.

Evans, D. S.: The Digitizer—A High-accuracy Scale for Automatic Data Processing, *Brit. Communs. Electronics*, vol. 4, pp. 334–339, June, 1957.

Follingstad, H. G., J. N. Shive, and R. E. Yaeger: An Optical Position Encoder and Digit Register, *Proc. IRE*, vol. 40, pp. 1573–1583, November, 1952.

Forrand, W. A.: An Accurate Digital-analogue Function Generator, *Proc. Electronic Computer Symposium*, April–May, 1952, Los Angeles, pp. XVI-1–XVI-9.

Foss, F. A.: The Use of Reflected Code in Digital Control Systems, *IRE Trans. on Electronic Computers*, vol. EC-3, pp. 1–6, December, 1954.

Henn, W., and A. S. Robinson: A Digital Sine-Cosine Transducer, *IRE Trans. on Instrumentation*, vol. PGI-5, pp. 202–209, June, 1956.

Hollander, G. H.: Criteria for the Selection of Analog-to-digital Converters, *Proc. Natl. Electronics Conf.*, 1953, pp. 670–683.

Horn, H. S.: Multiple-input Analog-to-digital Converter with 12-bit Accuracy and Fast, Non-sequential Switching, *IRE Natl. Conv. Record*, pt. 4, pp. 259–266, March, 1959.

Jury, E. I.: Analysis and Synthesis of Sampled Data Control Systems, *Trans. AIEE*, vol. 73, pt. I, pp. 332–346, September, 1954.

Kay, A. F.: Relay-scanning-design Technique Generates High Accuracy and Speed in Analogue-to-digital Transducer Measurements, *Communs. and Electronics*, no. 36, pp. 248–250, May, 1958.

Klein, M. L.: High Speed Analog-digital Converters, *IRE Trans. on Instrumentation*, vol. PGI-5, pp. 148–154, June, 1956.

Kuder, M. L.: Anodige, an Electronic Analogue-to-digital Converter, *NBS Rept.* 1117, Aug. 24, 1951.

Libaw, W. H., and L. J. Craig: A Photoelectric Decimal-coded Shaft Digitizer, *IRE Trans. on Electronic Computers*, vol. EC-2, pp. 1–4, September, 1953.

Linvill, W. K.: Sampled Data Control Systems through Comparison of Sampling with Amplitude Modulation, *Trans. AIEE, Tech. Paper* 51-324, vol. 70, pt. 2, pp. 1779–1786, 1951. (Discussion, p. 1787.)

——— and J. M. Salzer: Analysis of Control Systems Involving Digital Computers, *Proc. IRE*, vol. 41, pp. 901–906, July, 1953.

——— and R. W. Sittler: Extension of Conventional Techniques to the Design of Sampled Data Systems, *IRE Conv. Record*, 1953, pt. 1, Radar and Telemetry, pp. 99–104.

Lippel, B.: A High Precision Analog-to-digital Converter, *Proc. Natl. Electronics Conf.*, 1951, pp. 206–215.

———: Interconversion of Analog and Digital Data in Systems for Measurement and Control, *Proc. Natl. Electronics Conf.*, 1952, pp. 636–646.

———: A Decimal Code for Analog-to-digital Conversion, *IRE Trans. on Electronic Computers*, vol. EC-4, pp. 158–159, December, 1955.

Mergler, H. W.: A Digital-analog Machine Tool System, *Proc. Western Joint Computer Conf.*, February, 1954, Los Angeles, pp. 46–59.

Meter Reads Optically, Then Digitizes, *Control Eng.*, vol. 3, no. 3, p. 107, March, 1956.

Mitchell, A. L.: Photoelectric Analog-to-digital Converters, *Electronic Design*, vol. 4, no. 9, pp. 20–23, May 1, 1956.

Mitchell, J. M.: A High-speed Multichannel Analog-digital Converter, *Proc. Western Joint Computer Conf.*, February, 1954, Los Angeles, pp. 118–127.

Morrison, J. J.: An Automatic Graph Plotter, *Trans. Soc. Instr. Tech.*, vol. 10, pp. 55–56, June, 1958.

Murphy, G. J., and R. D. Ormsby: A Survey of Techniques for the Analysis of Sampled Data Control Systems, *IRE Trans. on Automatic Control*, vol. PGAC-2, pp. 79–90, February, 1957.

Partos, P.: Industrial Data-reduction and Analogue-digital Conversion Equipment, *J. Brit. IRE*, vol. 16, pp. 651–678, December, 1956.

Putschi, H. N., J. A. Raper, and J. J. Suran: Digital/Analogue Converter Provides Storage, *Electronics*, vol. 30, pp. 148–151, Dec. 1, 1957.

Ragazzini, J. R., and G. F. Franklin: "Sampled-data Control Systems," McGraw-Hill Book Company, Inc., New York, 1958.

——— and L. A. Zadek: The Analysis of Sampled Data Systems, *Trans. AIEE, Tech. Paper* 52-161, vol. 71, pt. 2, pp. 225–232, 1952.

Resnik, I. L.: Binary Conversion for Analog-digital Converters, *Electronic Design*, vol. 4, no. 12, pp. 42–45, June 15, 1956.

Retzinger, L. P., Jr.: An Input-output System for a Digital Computer, *Proc. Western Joint Computer Conf.*, August, 1954, pp. 67–76.

Rigby, S.: Analog-to-digital Converter, *Electronics*, vol. 29, no. 1, pp. 152–155, January, 1956. (A frequency-modulation method.)

Salzer, J. M.: Frequency Analysis of Digital Computers Operating in Real-time, *Proc. IRE*, vol. 42, no. 2, pp. 457–466, February, 1954.

———: System Compensation with a Digital Computer, *IRE Conv. Record*, pt. 5, pp. 179–186, 1954.

Scarbrough, A. D.: An Analog-to-digital Converter, *IRE Trans. on Electronic Computers*, vol. EC-2, pp. 5–7, September, 1953.

Sisson, R. L., and A. K. Susskind: Devices for Conversion between Analog Quantities and Binary Pulse-coded Numbers, M.S. thesis, Department of Electrical Engineering, Massachusetts Institute of Technology, 1950.

Slaughter, D. W.: An Analog-to-digital Converter with an Improved Linear-sweep Generator, *IRE Conv. Record,* pt. 7, pp. 7–12, 1953.

Smith, B. D.: An Unusual Electronic Analog-digital Conversion Method, *IRE Trans. on Instrumentation,* vol. PGI-5, pp. 155–160, June, 1956.

Spaulding, C. P.: Sine-cosine Angular Position Encoders, *IRE Trans. on Instrumentation,* vol. PGI-5, pp. 161–167, June, 1956.

Speller, J. B.: A Digital Converter, *Proc. Western Joint Computer Conf.,* August, 1957, pp. 29–31.

Susskind, A. K.: Approaches to Design Problems in Conversion Equipment, *Proc. Western Joint Computer Conf.,* February, 1954, Los Angeles, pp. 105–112.

——— (ed.): "Notes on Analog-digital Conversion Techniques," Technology Press, Boston, 1957.

Tompkins, H. E.: Unit-distance Binary-decimal Codes for Two-track Commutation, *IRE Trans. on Electronic Computers,* vol. EC-5, p. 139, September, 1956.

Towles, W. B.: A Transistorized Analogue-digital Converter, *Electronics,* vol. 31, pp. 90–93, Aug. 1, 1958.

Villars, C. P.: Design of a Transistorized 1.5 Megabit Analog-to-digital Encoder, Digest of Technical Papers, p. 38, IRE-AIEE Solid-state Circuits Conference, 1959, University of Pennsylvania.

Walton, C. A.: A Direct-reading Printed-circuit Commutator for Analogue-to-digital Data Conversion, *IBM J. Research Develop.,* vol. 2, pp. 178–192, July, 1958.

Willis, J., and M. G. Hartley: A Cathode-ray-tube Analogue-to-serial Digital Converter, *J. Sci. Instr.,* vol. 35, pp. 197–202, June, 1958.

Wister, A. J.: A Magnetically Coupled Low-cost High Speed Shaft Position Digitizer, *Proc. Western Joint Computer Conf.,* 1953, pp. 203–207.

Zweizig, J. R.: A Digital Voltage Encoder, *IRE Trans. on Electronic Computers,* vol. EC-3, pp. 25–28, September, 1954.

d. Selected Bibliography on Input-Output Methods

Bliss, W. H., and J. E. Ruedy: An Electron Tube for High-speed Teleprinting, *RCA Rev.,* vol. 16, pp. 5–15, March, 1955.

Carroll, J. M.: Trends in Computer Input-output Devices, *Electronics,* vol. 29, no. 9, pp. 142–149, September, 1956.

David, E. E., M. V. Mathews, and H. S. McDonald: A High-speed Data Translator for Computer Simulation of Speech and Television Devices, *Proc. Western Joint Computer Conf.,* March, 1959.

Davidson, J. T., and R. L. Fortune: Automatic Translation of Printed Code to Impulses Acceptable to Computing Equipment, *Proc. Western Joint Computer Conf.,* March, 1955, Los Angeles, pp. 29–33.

Diamond, T. L.: Devices for Reading Handwritten Characters, *Proc. Eastern Joint Computer Conf.,* December, 1957, pp. 232–238.

Dineen, G. P.: Programming Pattern Recognition, *Proc. Western Joint Computer Conf.,* March, 1955, Los Angeles, pp. 94–100.

Epstein, H.: The Electrographic Recording Technique, *Proc. Western Joint Computer Conf.,* March, 1955, Los Angeles, pp. 116–118.

Gettings, H.: A Big Step in Digital Telemetry, *Missiles and Rockets,* July 13, 1959, pp. 37–40.

Greanias, E. C., C. J. Hoppel, M. Kloomok, and J. S. Osborne: The Design of a Logic for the Recognition of Printed Characters by Simulation, *Proc. IEE,* vol. 103, pt. B, suppl. 3, Convention on Digital-computer Techniques, pp. 456–462, April, 1956. (Also appears in *IBM J, Research Develop.,* vol. 1, pp. 8–18, January, 1957.)

Heasley, C. C., Jr.: Some Communication Aspects of Character Sensing Systems, *Proc. Western Joint Computer Conf.*, March, 1959, pp. 176–181.

Highleyman, W. H., and L. A. Kamentsky: Generalized Scanner for Pattern and Character Recognition Studies, *Proc. Western Joint Computer Conf.*, March, 1959.

Johnson, R. B.: Writing around Dots, Patent No. 2,141,312, Apr. 10, 1956.

Kamentsky, L. A.: Pattern and Character Recognition Systems—Picture Processing by a Net of Neuron-like Elements, *Proc. Western Joint Computer Conf.*, March, 1959.

Kirsch, R. A., et al.: Experiments in Processing Pictorial Information with a Digital Computer, *Proc. Eastern Joint Computer Conf.*, December, 1957, pp. 221–230.

McPherson, R. G., and I. A. Sonderby: Digital and Pictorial Photographic Electronic Recorder, *Communs. and Electronics*, no. 36, pp. 194–196, May, 1958.

Ogle, J.: Optical Display for Data-handling System Output, *Proc. Eastern Joint Computer Conf.*, December, 1957, pp. 230–232.

Pike, J. L., and E. F. Ainsworth: Input-output Devices for NBS Computers, *NBS Circ.* 551, Computer Development at the National Bureau of Standards, pp. 109–118, January, 1955.

Rabinow, J.: Report on DOFL First Reader, *DOFL Rept.* TR-128, Nov. 26, 1954.

Review of Input and Output Equipment Used in Computing Systems, *Proc. Joint AIEE-IRE-ACM Computer Conf.*, December, 1952, New York.

Seehof, J., et al.: The National Cash Register High-speed Magnetic Printer, *Proc. Eastern Joint Computer Conf.*, December, 1957, pp. 243–251.

Selfridge, O. G.: Pattern Recognition and Modern Computers, *Proc. Western Joint Computer Conf.*, March, 1955, Los Angeles, pp. 91–93.

Shepard, D. H., and C. C. Heasley: Photoelectric Reader Feeds Business Machines, *Electronics*, vol. 28, no. 5, pp. 134–138, May, 1955.

Smith, H. M.: The Typotron, *IRE Conv. Record*, pt. 4, pp. 129–134, 1955.

Tersoff, A. I.: Automatic Registration in High-speed Character Sensing Equipment, *Proc. Eastern Joint Computer Conf.*, December, 1957, pp. 238–243.

West, R. E.: High-speed Readout for Data Processing, *Electronics*, May 29, 1959, pp. 83–85.

CHAPTER 23

THE ELECTRONIC DESIGN OF THE PEDAGAC

23-1. Introduction

In this chapter the attempt is made to complete the thread of continuity—from the systems design of a general-purpose computer, through its logical design, to its electronic design and final wiring tables. The chapter may at first seem specific in that it illustrates the electronic design of a single, simple machine. However, the reader should realize that the purpose of discussing the Pedagac is to bring up in a natural and realistic way some very important general problems applicable to all computer and control designs. Consideration of each problem had to be post-

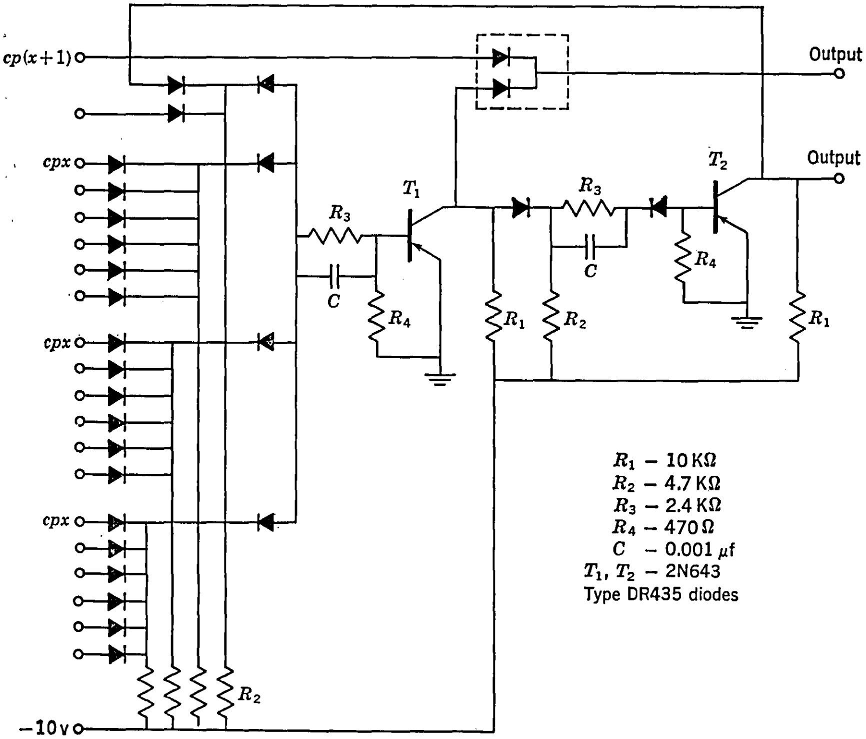


FIG. 23-1. Type C package schematic wiring diagram. (The portion of the figure enclosed in the dashed rectangle is discussed in Sec. 23-3.)

poned until all the necessary background had been developed. Now in this chapter we can finally discuss the all-important specific *timing* problem, illustrated by its solution for the Pedagac. Then there are the *delay* and *negation* problems, for which the Pedagac solutions are given. Finally we shall be in a position to appreciate the *wiring-table* problem, which in practical cases becomes far from trivial.

23-2. Problems in the Electronic Design of the Pedagac

Basic Design of Pedagac Packages. The Pedagac package is a diode-gated transistor-amplifier regeneration circuit (see Sec. 20-5). Figure 23-1 is a schematic wiring diagram of a type *C* package (see Sec. 18-2), which has three input *and* gates feeding an *or* gate. Note that the regenerative *and* gate is wired internally. The direct and inverted outputs can each drive six other packages. The time required for the leading edge of a negative input pulse to pass through both transistors, i.e., the transit time t_T, is 0.47 μsec. Figure 23-2 illustrates the output waveforms from a package.

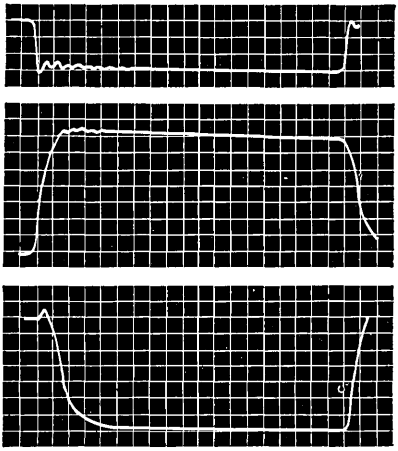


FIG. 23-2. Output waveforms of a Pedagac package driving eight direct and eight inversion inputs. (*Top*) Input at 200 kc; (*center*) inverted output; (*bottom*) direct output. Each vertical division represents 1.25 volts; each horizontal division represents 0.135 μsec.

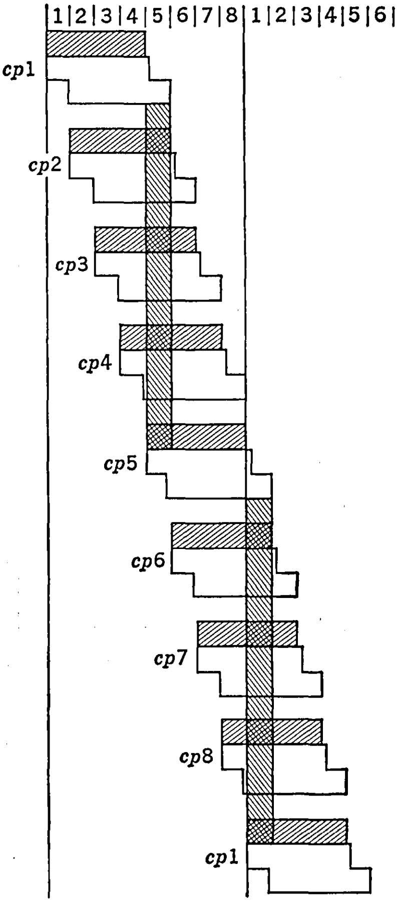


FIG. 23-3. Clock-phase system of the Pedagac: eight phases, 50 per cent duty cycle. Observe that the delay time of the trailing edge of each pulse allows skipping of three clock phases (cf. Fig. 23-5). Compare this figure with Figs. 19-5 and 19-6 on pages 632 and 633.

Packages of types A, B, D, and E differ from Fig. 23-1 only in that they contain, respectively, one, two, four, and five input *and* gates.

Timing and Repeater Problems. For the Pedagac we have chosen an eight-phase clock (see Sec. 19-3) with a unit time interval of 5 μsec. The pulse width t_W is taken to be a little more than four of the clock-phase divisions, or slightly more than half a unit time interval. Each clock-phase division is $5/8 = 0.62$ μsec in length. Since for the regeneration circuit it is sufficient to have an overlap time no shorter than the transit time (0.47 μsec for the Pedagac package), then an overlap of one clock-phase division (0.62 μsec) is adequate for the transfer of a signal from one package to the next (see Fig. 23-3). A detailed examination of the logical design diagram of the Pedagac shows that no signal (except 1CS in the current-address counter) traverses more than seven packages during a single minor cycle; the eight-phase clock is more than adequate to handle this. Figure 23-4 illustrates a signal that traverses seven packages from the memory to the accumulator in one unit time interval.

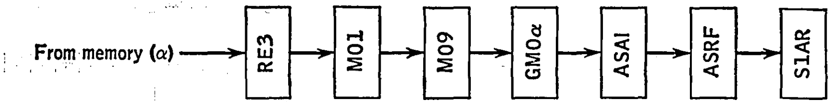
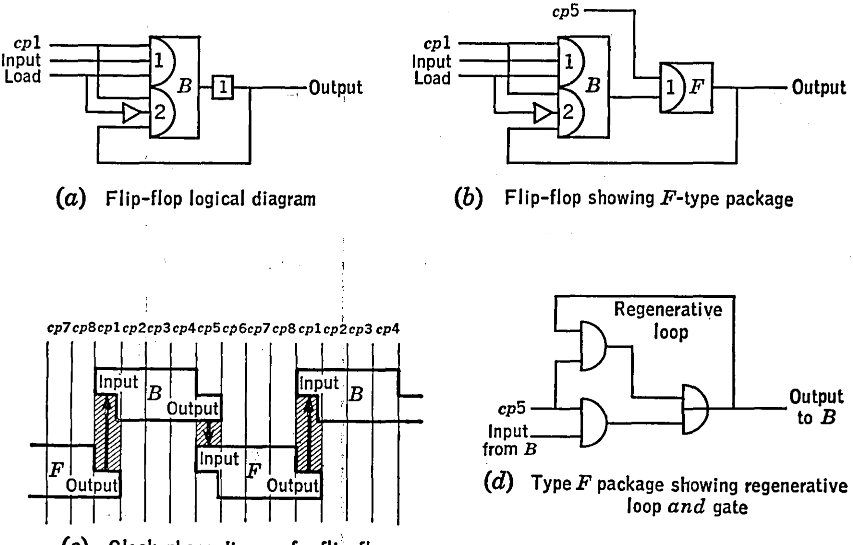


FIG. 23-4. Tracing (α) from channel 3 of the drum to the first stage of the accumulator.



(a) Flip-flop logical diagram

(b) Flip-flop showing F-type package

(c) Clock-phase diagram for flip-flop

(d) Type F package showing regenerative loop *and* gate

FIG. 23-5. Skipping phases for delay with type F package.

The 50 per cent pulse duty cycle coupled with the short necessary overlap time is desirable for skipping clock phases (see Sec. 19-3). For our purposes the most important application of the clock-phase-skipping capability is its use in connection with the electronic realization of the unit delays that so frequently appear in the logical diagrams. The delay line will be represented by a regenerative Pedagac type of package with a minimum of inputs. In Fig. 23-5 we illustrate its use for the delay in a dynamic flip-flop. Suppose that the flip-flop package B is gated with input clock phase 1. Then its output will last from clock phase 2 through and including clock phase 5. But the *delay package F* is gated with input clock phase 5, and can therefore accept the output of package B. The output of the delay package F will last from clock phase 6 through and including clock phase 1 of the next unit time interval; if its output is connected to the input of package B, then package B can sense the delayed signal. The delay package we shall call a type F package; it is constructed especially for this purpose, with a single two-input *and* gate and a regeneration gate.

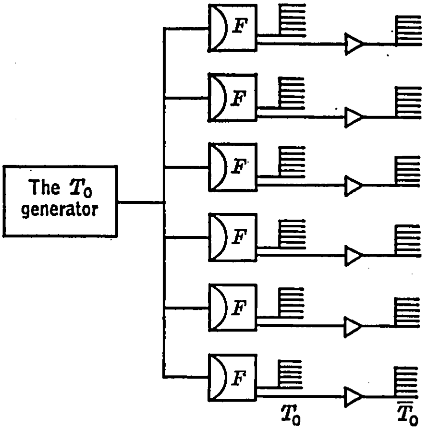


FIG. 23-6. Generating T_0 and $\bar{T}_0$ outputs, using F-type packages as repeaters.

Each Pedagac package is limited to a maximum of six outputs. But

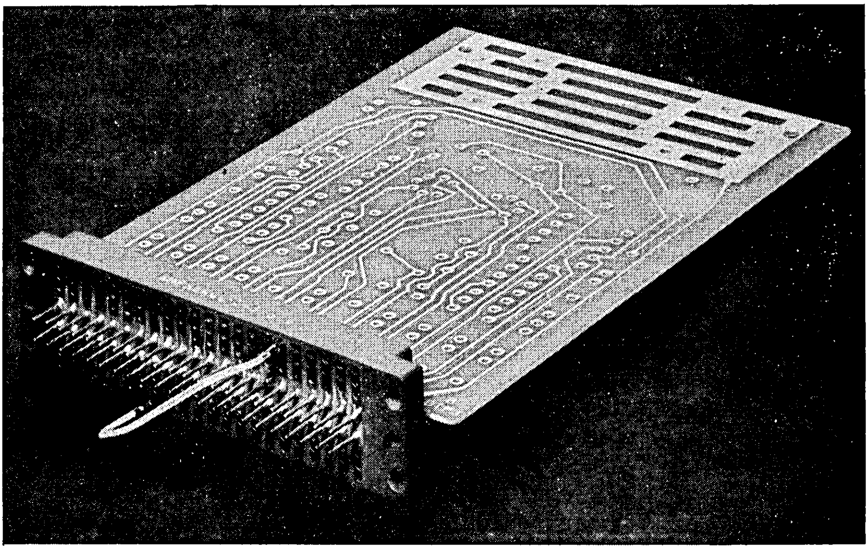


FIG. 23-7. Fifty-pin 5- by 7-in. card. The printed-circuit card is fitted into the connector as shown; electrical contact is made by spring clips above and below the card. The large holes on the sides of the connector are for mounting on the racks. The small holes, three to each pin, are for wiring, as illustrated by the jumper wire. The projecting pins are for testing.

observe from the logical diagrams that some Pedagac signals will be required as inputs many more than six times. Such signals must pass through multiplying *repeater packages* to gain more outputs. The repeater package is simply the F-type package, just described for use in delays. For example, 34 direct outputs of T_0 and 32 inverted outputs of T_0 (i.e., $\bar{T}_0$) are required. We let the direct output of the T_0 generator drive six F-type repeater packages; each output of the F-type packages is able to drive six more packages, for a total of 36 available direct T_0 outputs and 36 available inverted, or $\bar{T}_0$, outputs; these are more than adequate for this situation (see Fig. 23-6).

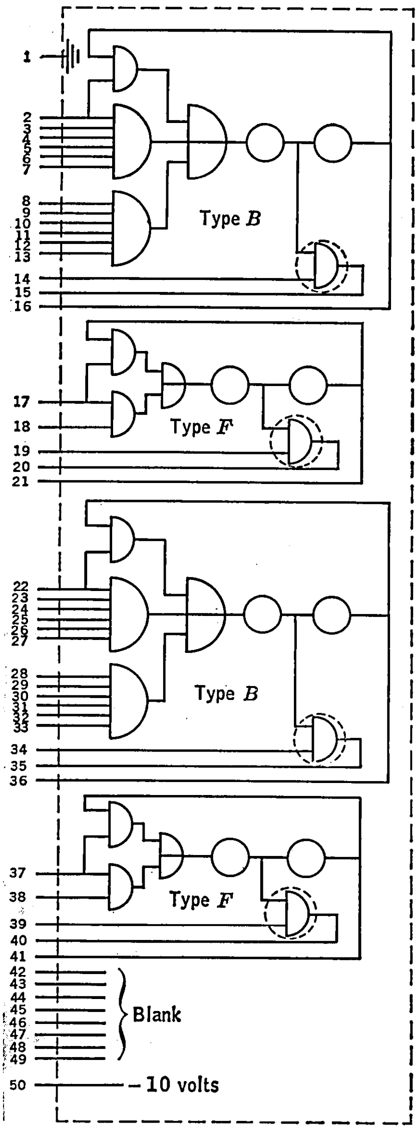


FIG. 23-8. Type 2 card. (The portions of the figure enclosed in the dashed circles are discussed in Sec. 23-3.)

From our unit-time-interval specifications we can calculate the required speed of the magnetic memory drum of the Pedagac and the average access time. Since there are $2^7 = 128$ sectors around the drum and each sector contains 23 bits per channel, there are $128 \times 23 = 2{,}944$ bits per channel, or 2,944 unit time intervals per revolution. Since a unit time interval is 5 μsec, we have a basic clock rate of $1/(5 \times 10^{-6}) = 200$ kc. The drum speed in rpm is therefore

$$\frac{60}{2{,}944 \times 5 \times 10^{-6}} = 4{,}076 \text{ rpm}$$

The average access time in milliseconds is

$$\tfrac{1}{2} \text{ (time of 1 revolution) } =$$
$$\tfrac{1}{2} \times 2.944 \times 5 \times 10^{-3} = 7.4 \text{ msec}$$

Pedagac Packages and Cards. The next step in the design of the Pedagac is a discussion of the assembly of the packages into cards, the printed-circuit modules of the Pedagac.

We have selected a 50-pin card 7 in. long and 5 in. wide (see Fig. 23-7). The pins of the card fit into a connector, on the other side of which are holes for the insertion of taper-pin tipped wires. Each type B, C, D, and

E package is paired with a type *F* package, since many of these packages are associated with unit delays (e.g., the shift-register packages, etc.). For the Pedagac we make use of six types of cards:

Type 1 card with five type *A* packages
Type 2 card with two type *B* and two type *F* packages
Type 3 card with one type *C* and one type *F* package
Type 4 card with one type *D* and one type *F* package
Type 5 card with one type *E* and one type *F* package
Type 6 card with nine type *F* packages

Figure 23-8 illustrates a type 2 card with its associated pin connections; Table 23-1 lists the pin connections for all the cards. To reduce the required number of pins per package, each Pedagac card includes a common ground and a common power bias. Thus only two voltage pins are needed, pin 1 for ground and pin 50 for the required −10-volt supply.

EXERCISES

(*a*) From the specifications of the Pedagac, and from the fact that no signal traverses more than seven packages in a unit time interval, observe that a seven-phase clock seems feasible. Perform all the calculations analogous to those described above for such a seven-phase clock system. (HINT: What duty cycle will allow the use of *F*-type packages as a delay?) What will be the average access time?

(*b*) Consider a signal that must drive 26 packages. Design a repeater system using only 4 repeater packages such that all 26 signals are available at the same time. (HINT: Skip clock phases.)

(*c*) Signal 1CS may be required to traverse up to 12 packages in a unit time interval. One method of preventing this is to design the current-address register as a parallel counter (see Exercise *f* of Sec. 18-7). Why is this unnecessary in the Pedagac? (HINT: See footnote on p. 582.)

23-3. Problems in Assigning Clock Phases

Negation Problem. With the background of the previous section we are now in a position to understand the negation problem. Consider a package with input clock pulse *cp*1. If this package is putting out a unit, the unit will appear at approximately *cp*2 through *cp*5, being delayed from the input clock pulse by one clock-phase division. During *cp*6 through *cp*8 and the next *cp*1, that is, "between" clock pulses, the output will be zero. But consider the inverted output. During *cp*2 through *cp*5 it will have the opposite polarity to the direct output and will be zero. But during *cp*6 through *cp*8 and the next *cp*1, that is, between clock pulses, the inverted output will be a unit. A problem arises because this negation output is a unit between clock pulses, whereas we have heretofore tacitly considered it to be zero. For example, consider an input *and* gate that is to form $X \cdot \bar{Y}$, gated at *cp*4, with the *X* output "on" from *cp*4 through *cp*7, but with the *Y* output "on" from *cp*2 through *cp*5. Suppose that *Y* is a unit. Then there should be a zero output from the $X \cdot \bar{Y}$ *and* gate; and there is during *cp*4

TABLE 23-1. CARD-PIN LAYOUT

Type 1 Card

Pin No.	Package and gate No.	Function
1	....	Ground
2	1*A*1	*cpx* input
3–7	1*A*1	Inputs
8	1*Aa*	*cpx* + 1 input
9	1*A*	$\overline{\text{Output}}$
10	1*A*	Output
11	2*A*1	*cpx* input
12–16	2*A*1	Inputs
17	2*Aa*	*cpx* + 1 input
18	2*A*	$\overline{\text{Output}}$
19	2*A*	Output
20	3*A*1	*cpx* input
21–25	3*A*1	Inputs
26	3*Aa*	*cpx* + 1 input
27	3*A*	$\overline{\text{Output}}$
28	3*A*	Output
29	4*A*1	*cpx* input
30–34	4*A*1	Inputs
35	4*Aa*	*cpx* + 1 input
36	4*A*	$\overline{\text{Output}}$
37	4*A*	Output
38	5*A*1	*cpx* input
39–43	5*A*1	Inputs
44	5*Aa*	*cpx* + 1 input
45	5*A*	$\overline{\text{Output}}$
46	5*A*	Output
47–49	....	Blank
50	....	−10 volts

Type 2 Card

Pin No.	Package and gate No.	Function
1	....	Ground
2	1*B*1	*cpx* input
3–7	1*B*1	Inputs
8	1*B*2	*cpx* input
9–13	1*B*2	Inputs
14	1*Ba*	*cpx* + 1 input
15	1*B*	$\overline{\text{Output}}$
16	1*B*	Output
17	1*F*2	*cpx* + 4 input
18	1*F*2	Input
19	1*Fa*	*cpx* + 5 input
20	1*F*	$\overline{\text{Output}}$
21	1*F*	Output
22	2*B*1	*cpx* input
23–27	2*B*1	Inputs
28	2*B*2	*cpx* input
29–33	2*B*2	Inputs
34	2*Ba*	*cpx* + 1 input
35	2*B*	$\overline{\text{Output}}$
36	2*B*	Output
37	2*F*2	*cpx* + 4 input
38	2*F*2	Input
39	2*Fa*	*cpx* + 5 input
40	2*F*	$\overline{\text{Output}}$
41	2*F*	Output
42–49	....	Blank
50	....	−10 volts

Type 3 Card

Pin No.	Package and gate No.	Function
1	....	Ground
2	1*C*1	*cpx* input
3–7	1*C*1	Inputs
8	1*C*2	*cpx* input

Type 4 Card

Pin No.	Package and gate No.	Function
1	....	Ground
2	1*D*1	*cpx* input
3–7	1*D*1	Inputs
8	1*D*2	*cpx* input
9–13	1*D*2	Inputs
14	1*D*3	*cpx* input

TABLE 23-1. CARD-PIN LAYOUT (*Continued*)

Type 3 Card (*Continued*)

Pin No.	Package and gate No.	Function
9–13	1*C*2	Inputs
14	1*C*3	*cpx* input
15–19	1*C*3	Inputs
20	1*Ca*	*cpx* + 1 input
21	1*C*	$\overline{\text{Output}}$
22	1*C*	Output
23	1*F*2	*cpx* input
24	1*F*2	Input
25	1*Fa*	*cpx* + 1 input
26	1*F*	$\overline{\text{Output}}$
27	1*F*	Output
28–49	....	Blank
50	....	−10 volts

Type 4 Card (*Continued*)

Pin No.	Package and gate No.	Function
15–19	1*D*3	Inputs
20	1*D*4	*cpx* input
21–25	1*D*4	Inputs
26	1*Da*	*cpx* + 1 input
27	1*D*	$\overline{\text{Output}}$
28	1*D*	Output
29	1*F*2	*cpx* + 4 input
30	1*F*2	Input
31	1*Fa*	*cpx* + 5 input
32	1*F*	$\overline{\text{Output}}$
33	1*F*	Output
34–49	....	Blank
50	....	−10 volts

Type 5 Card

Pin No.	Package and gate No.	Function
1	....	Ground
2	1*E*1	*cpx* input
3–7	1*E*1	Inputs
8	1*E*2	*cpx* input
9–13	1*E*2	Inputs
14	1*E*3	*cpx* input
15–19	1*E*3	Inputs
20	1*E*4	*cpx* input
21–25	1*E*4	Inputs
26	1*E*5	*cpx* input
27–31	1*E*5	Inputs
32	1*Ea*	*cpx* + 1 input
33	1*E*	$\overline{\text{Output}}$
34	1*E*	Output
35	1*F*2	*cpx* + 4 input
36	1*F*2	Input
37	1*Fa*	*cpx* + 5 input
38	1*F*	$\overline{\text{Output}}$
39	1*F*	Output
40–49	....	Blank
50	....	−10 volts

Type 6 Card

Pin No.	Package and gate No.	Function
1	....	Ground
2	1*F*2	*cpx* input
3	1*F*2	Input
4	1*Fa*	*cpx* + 1 input
5	1*F*	$\overline{\text{Output}}$
6	1*F*	Output
7	2*F*2	*cpx* input
8	2*F*2	Input
9	2*Fa*	*cpx* + 1 input
10	2*F*	$\overline{\text{Output}}$
11	2*F*	Output
.....	....	....
42	9*F*2	*cpx* input
43	9*F*2	Input
44	9*Fa*	*cpx* + 1 input
45	9*F*	$\overline{\text{Output}}$
46	9*F*	Output
47–49	....	Blank
50	....	−10 volts

and *cp*5. But at *cp*6 the signal Y is "off"; hence the signal $\bar{Y}$ would become a unit, and the $X \cdot \bar{Y}$ gate is turned on erroneously.

There are several solutions to the negation problem. One solution is to force the inverted output to zero between clock pulses by means of the clock pulse itself. But an inverted output unit should come out delayed, just as a direct output unit does. Hence a clock pulse *one phase division later* is used, as shown in Fig. 23-9. Then both the direct and inverted outputs will be "off" between clock pulses but will have opposite polarities during the clock pulses, as desired. A second solution to the problem is to send the inverted output into a type F package and utilize only the direct output of the type F package (why?). A third solution is to use an appropriate clock-pulse input to all *and* gates having the negation as input.

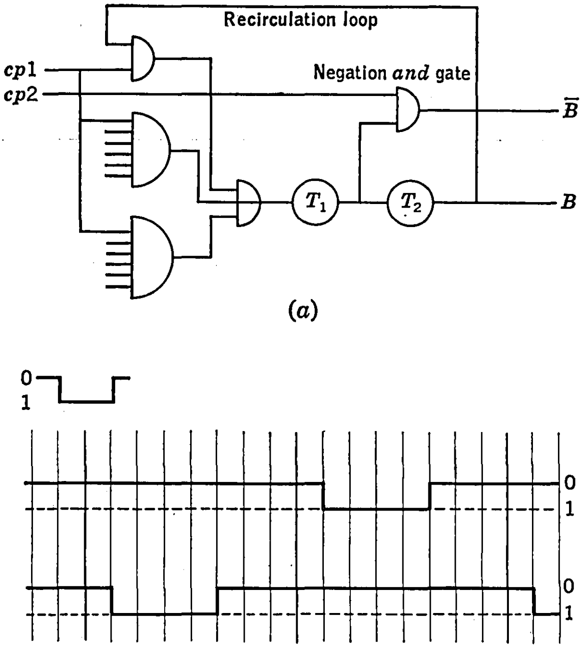


FIG. 23-9. Solving the negation problem with a delayed clock pulse. (T_1 and T_2 are transistor amplifiers.)

Assigning Clock Phases to Packages. Each package is associated with the clock pulse entering each of its input *and* gates and its regeneration gate. The phase of this clock pulse is called the *clock phase of the package.* For each package of the Pedagac an appropriate clock phase must be assigned depending on the times the input signals are available. As an example, Fig. 23-10 repeats Fig. 18-12, with the clock phases assigned to the packages. There are two steps in making such an assignment: the first is to determine the availability of each signal, the second to assign package clock phases as determined by these availabilities. For example, in Fig. 23-11 we trace (α) from the memory to S1AR, assuming that (α) leaves the memory at *cp*1. Note that any information from the memory will not be available to the arithmetic unit until *cp*5. The result of the arithmetic operations is available at *cp*7, at which time it enters S1AR for temporary storage. To assign a clock phase to a package, all the input signals are marked as to clock-phase availability; the clock phase of the package then becomes the clock phase of the *last signal* to enter the package, in order that the output will be on for the full half unit time interval. Figure 23-12 shows the availability of signals entering package S5AR. The last signal will enter at *cp*5, and so a clock pulse of *cp*5 is assigned to S5AR. Note, however, that the signal SHS is available only during *cp*1 through *cp*4; thus we must delay SHS by one or more clock-phase divisions before it enters the package. In Fig. 23-10 we have included all the type *F* packages necessary for delay purposes, in addition to assigning the proper clock phase to each package.

We have assumed above that (α) will be available from the memory at *cp*1, even during T_0. However, this requires that the selection mechanism of the IF phases begin its operation earlier than T_0. For this reason all IF packages must be adjusted to present outputs earlier than was indicated in Fig. 18-17 at T_0. Investigation of the figure indicates that this is possible. Other signals must similarly be adjusted, particularly those concerned with the instruction register and current-address counter. We shall not go into the details of this process (see the Additional Topics), except to note that the minor cycle was made three unit time intervals longer than the number of bits in a word especially to allow time for the selection processes before *cp*1 of T_0.

EXERCISES

(*a*) If the polarity of the unit and zero pulses were reversed, how would this affect the negation problem discussed above?

(*b*) Discuss the assignment of clock phases to the input-output equipment.

(*c*) Discuss other possible means of correcting for a signal cut off before the clock phase of a package it must enter.

(*d*) Assuming that information from the drum [(α)] is available at *cp*1, assign clock phases to the packages of Fig. 18-11.

23-4. Minimum-wiring Theory†

The Wiring Problem. Modern large-scale computers can require more than 100,000 wire connections between packages. (Even our Pedagac requires approximately 5,000 wire connections.) The planning of these connections turns out to be far from trivial and in fact can be accomplished feasibly only with the aid of a computer. Systematic rules must therefore be devised in order that the computer can be programmed. The wiring itself can be feasibly accomplished by hand; for a simple

† This section is based in part on H. Loberman and A. Weinberger, Formal Procedures for Connecting Terminals with a Minimum Total Wire Length, *J. Assoc. Computing Machinery,* vol. 4, no. 4, pp. 428–437, October, 1957.

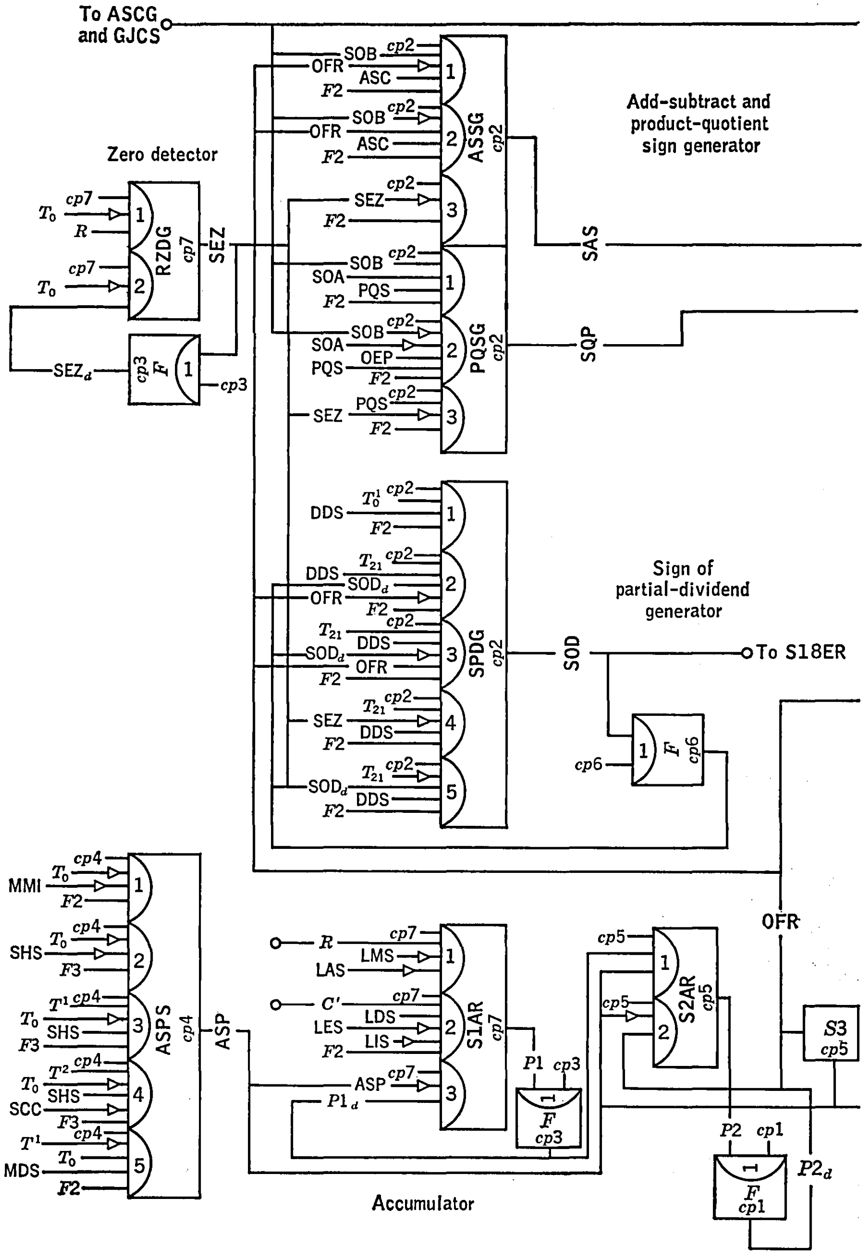
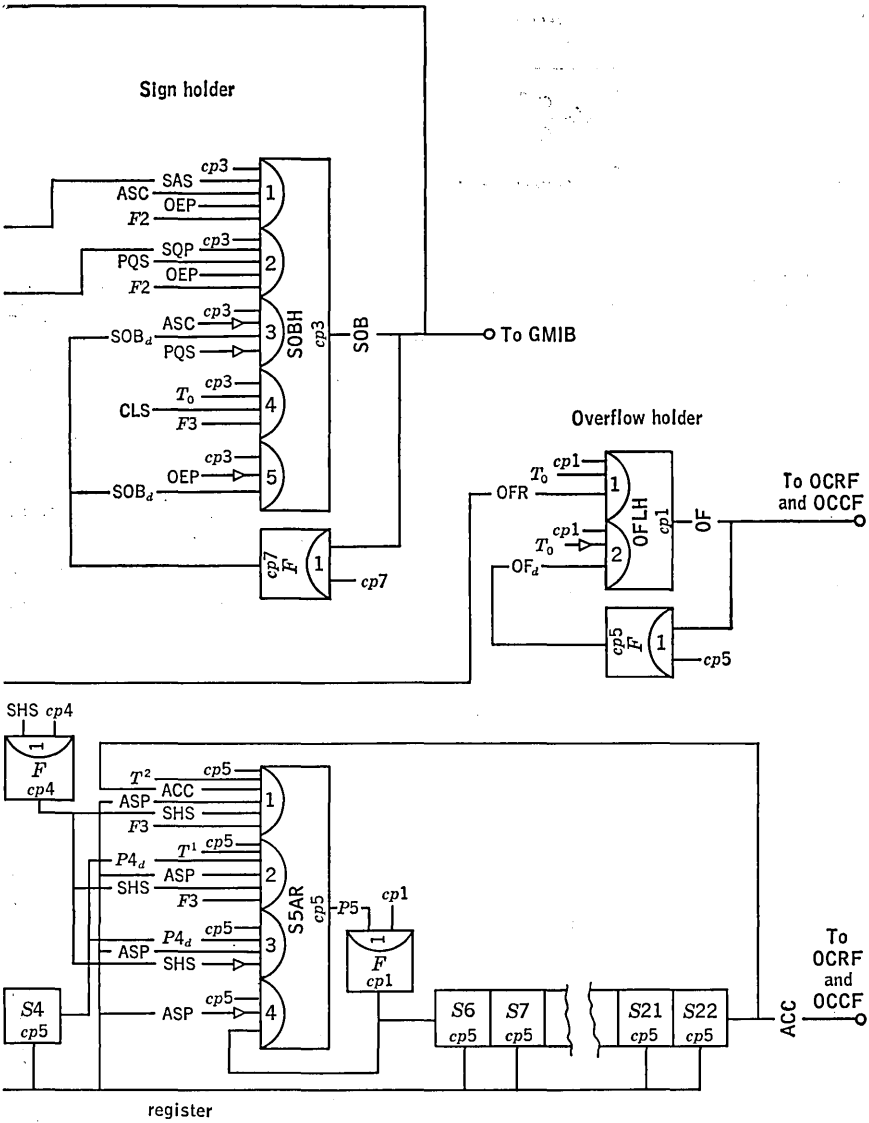


FIG. 23-10. Logical diagram of the arithmetic unit: sign generator



and accumulator—with associated clock phases and delay packages.

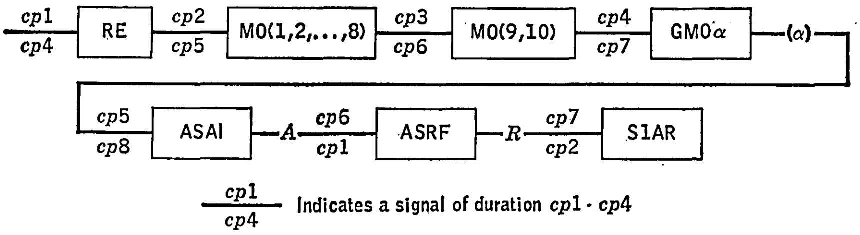


FIG. 23-11. Tracing (α) from memory to S1AR, with clock-phase availability times.

calculation based on the assumption that a technician using taper-pin connectors (no soldering required) can place one wire per minute indicates that four men working a 40-hr week would require approximately 3 months to wire such a computer.

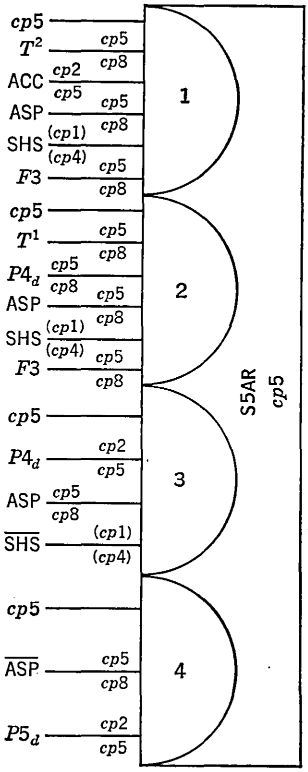


FIG. 23-12. Assignment of clock phase to package S5AR.

The wiring problem takes on specific form as follows: The computer cards will be assembled into female connectors mounted on racks. For the Pedagac only one rack is needed, as illustrated in Fig. 23-13. The cards are assigned row and column positions in the rack, which means that each pin number on the other side of the female connector (i.e., the holes for taper pins) now is associated with a specific signal (see Fig. 23-7). The technician connecting the wires must be supplied with a table giving the row and column of the card and the pin number for each end of every wire to be connected. It is highly desirable, for efficiency as well as for electronic purposes, to plan the wiring table so that a minimum length of wire is used. For example, suppose in Fig. 23-14 that pin 1 must be connected to pins 2, 3, 4, 5, 6, and 7, where for ease of access wires can be strung only along horizontal and vertical paths. The left-hand picture shows a bad connection; the right-hand picture shows the minimum connections.

We shall present below two algorithms for determining the minimum wire length. The algorithms are applied separately to each set of pins that is associated with a particular signal, including both its source and destination. The inputs to the algorithm are the distances between all possible pairs of this set of pins, calculated according to the wire-path rules determined by the rack setup. If there are n pins, then there are $\binom{n}{2} = \frac{n(n-1)}{2}$ distances to be specified.

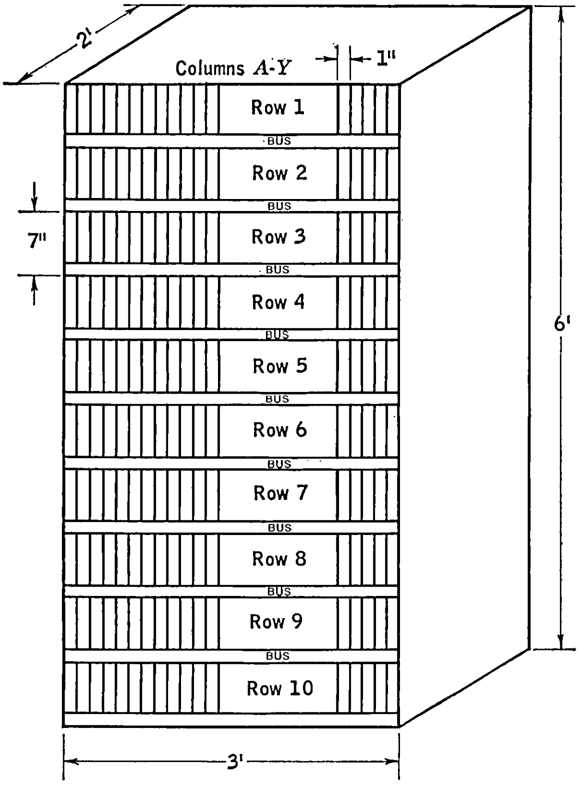


FIG. 23-13. Pedagac card rack.

Definitions. Before proceeding with the theory we must introduce further terminology, derived in part from mathematical topology and in part from electrical-circuit theory. A *proper pattern* of connections is one in which there is only one path from each terminal to every other terminal and there are no loops formed by redundant connections. Figure 23-15 shows two possible proper patterns for connecting five terminals. All terminals are referred to as *nodes.* The direct connection between two nodes is called a *branch,* the *magnitude* of which is the distance between its end nodes. A *path* between two nodes is a connection of one or more branches. A *graph* is a structure consisting of a number of

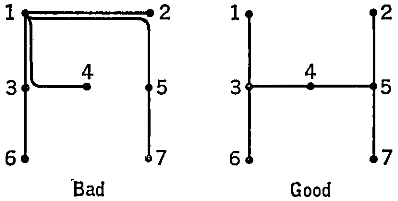


FIG. 23-14. Minimum and nonminimum wiring connections.

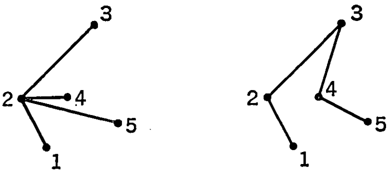


FIG. 23-15. Proper-pattern connections.

nodes connected pairwise by one or more branches. A *tree* is a graph having only one path between every two nodes (a proper pattern). A proper pattern where the length of wire is minimum is a *minimum tree.* A *subtree* is a tree consisting of m of n nodes, where $m < n$.

Two Algorithms. To connect n nodes into a tree, it is obvious that exactly $n - 1$ branches are required. If more than $n - 1$ branches are used, there will be redundant connections forming loops. On the other hand, if fewer than $n - 1$ branches are used, the tree will be incomplete. If $n - 1$ branches are used, but incorrectly, then both loops and unconnected nodes will result. It can be shown that the number of possible trees for n nodes is n^{n-2}; obviously, as n becomes great, the number of possible trees becomes overwhelming. A system which would generate all trees and select the minimum tree therefore appears impractical. However, since the total number of branches between each node and *every other node is* $n(n - 1)/2$, then for $n > 5$ the total number of branches is considerably smaller than the total number of trees. Solutions to systems which have a large number of nodes must make use of algorithms which work on the branches.

For the first of the two algorithms we shall describe, the branches are *sorted* according to length, from short to long. The first branch is always used, its nodes being recorded as belonging to a subtree. Then each branch is examined, in order, for one of four possible conditions. The rules governing these four cases are as follows (see the flow chart of Fig. 23-16):

1. Neither of the nodes is present in a recorded subtree: the branch is accepted, and the nodes are recorded as constituting a new subtree.

2. Only one node is present in a recorded subtree: the branch is accepted, and the *new* node is added to the subtree containing the other node of this branch.

3. Each of the nodes is recorded, but in different subtrees: the branch is accepted, and two subtrees containing the nodes are combined into a single subtree.

4. Both nodes are present in the same subtree: the branch is rejected.

The process is completed when it is determined (1) that all the n nodes are in the recorded tree or (2) that a total of $n - 1$ branches have been recorded.

Using the above algorithm, let us now construct the minimum tree for signal SOB of the Pedagac. Signal SOB goes to seven different locations: we shall therefore have eight nodes in our tree (one output and seven input), with 28 possible branches $[(8)(8 - 1)/2 = 28]$. The seven destinations of signal SOB are SOBH gate 3, SOBH gate 5, ASSG gate 1, PQSG gate 1, ASCG gate 2, ASCG gate 4, and GMIB gate 3. In order to simplify the writing of the 28 possible branches, let us assign code numbers to the various packages as follows: SOB = 1; SOBH-3 = 2, SOBH-5 = 3; ASSG-1 = 4; PQSG-1 = 5; ASCG-2 = 6; ASCG-4 = 7; and GMIB-3 = 8.

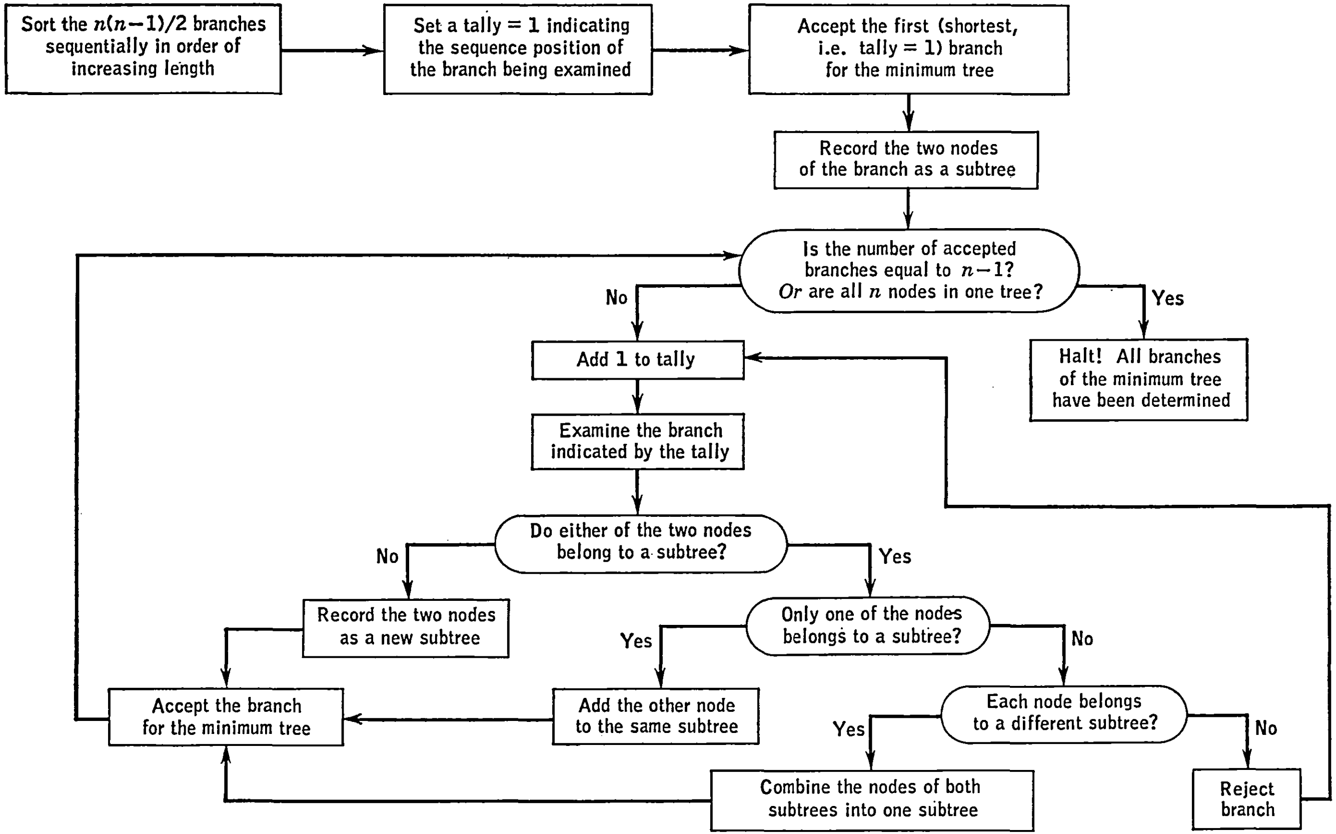

Sort the $n(n-1)/2$ branches sequentially in order of increasing length
Set a tally = 1 indicating the sequence position of the branch being examined
Accept the first (shortest, i.e. tally = 1) branch for the minimum tree
Record the two nodes of the branch as a subtree
Is the number of accepted branches equal to $n-1$? *Or* are all n nodes in one tree?
No
Yes
Halt! All branches of the minimum tree have been determined
Add 1 to tally
Examine the branch indicated by the tally
Do either of the two nodes belong to a subtree?
No
Yes
Record the two nodes as a new subtree
Only one of the nodes belongs to a subtree?
Yes
No
Accept the branch for the minimum tree
Add the other node to the same subtree
Each node belongs to a different subtree?
Yes
No
Combine the nodes of both subtrees into one subtree
Reject branch

FIG. 23-16. Flow chart for forming minimum tree by the first algorithm.

The branches (*br*) and their associated lengths (*len*) are shown below. (The method of determining the lengths of the branches will be indicated in the following section.)

br	*len*	*br*	*len*	*br*	*len*	*br*	*len*	*br*	*len*	*br*	*len*	*br*	*len*
1–2	**3**												
1–3	**3**	2–3	**3**										
1–4	**8**	2–4	**8**	3–4	**8**								
1–5	**7**	2–5	**7**	3–5	**7**	4–5	**6**						
1–6	**15**	2–6	**15**	3–6	**15**	4–6	**12**	5–6	**13**				
1–7	**15**	2–7	**15**	3–7	**15**	4–7	**12**	5–7	**13**	6–7	**3**		
1–8	**46**	2–8	**46**	3–8	**46**	4–8	**49**	5–8	**48**	6–8	**56**	7–8	**56**

Ordering the list according to the length of the branches, we obtain

1–2	**3**	3–5	**7**	5–7	**13**	1–8	**46**
1–3	**3**	1–4	**8**	1–6	**15**	2–8	**46**
2–3	**3**	2–4	**8**	1–7	**15**	3–8	**46**
6–7	**3**	3–4	**8**	2–6	**15**	5–8	**48**
4–5	**6**	4–6	**12**	2–7	**15**	4–8	**49**
1–5	**7**	4–7	**12**	3–6	**15**	6–8	**56**
2–5	**7**	5–6	**13**	3–7	**15**	7–8	**56**

Rather than go through a detailed description of the procedure to form this particular tree, we have elected to illustrate the procedure by means of a diagram (Fig. 23-17) which indicates the various steps in forming the tree. Note that in the tree we have disregarded the branch from the output node of the generating package to the input node of the associated *F*-type package. (Recall that the connection is necessary whenever a signal is recirculated within a *card* for dynamic flip-flop.) Since this branch will always be equal to the shortest possible length, it will always be incorporated in the tree. In order to reduce the size of the tree and the associated computations, we have chosen to ignore this branch in the present treatment. However, in the next section such branches will be included when the wiring tables are formed. Note further that the branch from SOB to SOBH gate 3 is actually measured from the output of the type *F* package to gate 3 of the SOBH package and that the branches SOB to PQSG gate 1 and SOB to GMIB gate 3 are made from the output of the SOBH package.

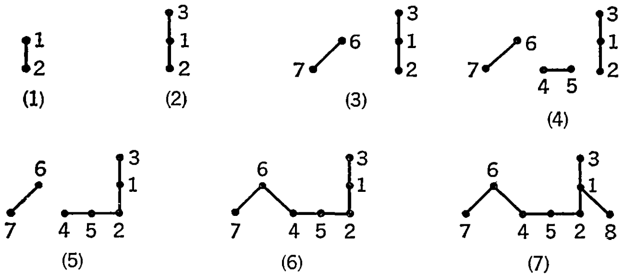


FIG. 23-17. Seven stages in the construction of the eight-node SOB tree (not to scale).

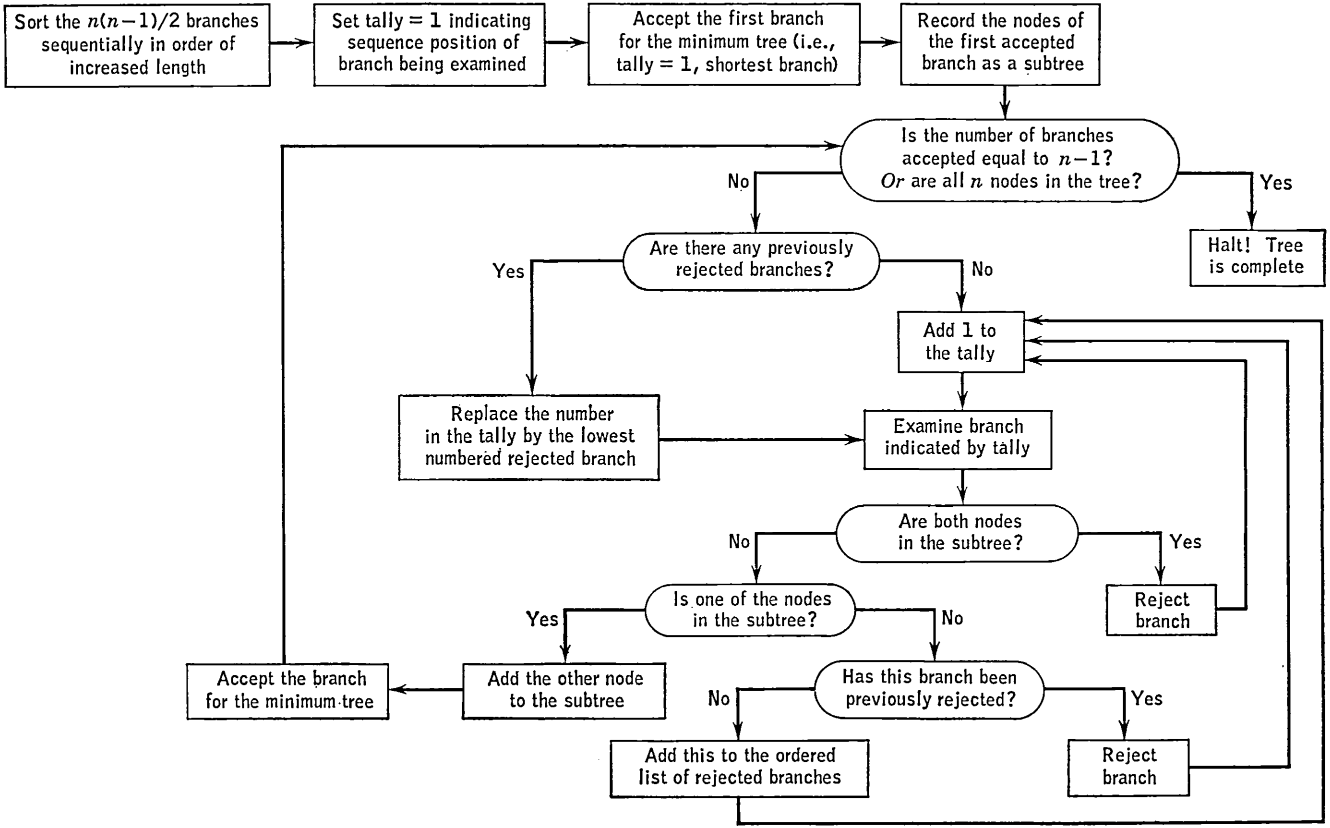

Sort the $n(n-1)/2$ branches sequentially in order of increased length
Set tally = 1 indicating sequence position of branch being examined
Accept the first branch for the minimum tree (i.e., tally = 1, shortest branch)
Record the nodes of the first accepted branch as a subtree
Is the number of branches accepted equal to $n-1$? *Or* are all n nodes in the tree?
Yes
Halt! Tree is complete
No
Are there any previously rejected branches?
Yes
Replace the number in the tally by the lowest numbered rejected branch
No
Add 1 to the tally
Examine branch indicated by tally
Are both nodes in the subtree?
Yes
Reject branch
No
Is one of the nodes in the subtree?
Yes
Add the other node to the subtree
Accept the branch for the minimum tree
No
Has this branch been previously rejected?
Yes
Reject branch
No
Add this to the ordered list of rejected branches

FIG. 23-18. Flow chart for forming minimum tree by the second algorithm.

The flow chart for the second algorithm is shown in Fig. 23-18, which is self-explanatory. In the first algorithm several disconnected subtrees can be formed, and these subtrees must be checked as each new branch is considered. In the second algorithm a single tree is formed. However, the branches which were previously rejected must be rechecked each time a new node is added to the tree.

EXERCISES

(*a*) Prove that the maximum number of trees possible is n^{n-2}.

(*b*) Write a Pedagac program for the first algorithm (see Fig. 23-16).

(*c*) Form the minimum tree for signal SOB, using the second algorithm. Use the data given above for the example of the first algorithm.

(*d*) Write a Pedagac program for the second algorithm (see Fig. 23-18).

(*e*) Compare the minimum-tree problem with the traveling-salesman problem (see J. B. Kruskal's article mentioned in the Additional Topics, Sec. 23-6).

23-5. Wiring Table for the Pedagac

We are now in a position to develop the minimum wiring table for the Pedagac. Since the Pedagac requires about 5,000 wires, we shall consider here only a portion of the arithmetic unit, specifically that shown in Fig. 18-12.

Figure 23-13 shows the Pedagac rack, Fig. 23-19 the card locations within the rack. The bus shown in Fig. 23-13 serves as a common source, from which are tapped the various clock-phase signals, the ground connection, and the -10-volt supply. Situated immediately below the bus (not shown in the figure) are the channels through which the wires are strung. For the Pedagac we shall assume that all wires are to run along horizontal and vertical paths. With the location of the cards and the distance between cards known, the formula for the length of wire required to connect two terminals is

$$\begin{aligned}\text{Length of wire} = K_1 &\times \text{(difference in row numbers)} \\ &+ K_2 \times \text{(difference in column number)} + K_3\end{aligned}$$

where K_1 = distance between rows = 7 in.
K_2 = between columns = 1 in.
and K_3 is a constant compensating for differences in pin numbers. Since a wire may be required between pin 8 of one card and pin 47 of another, we assign a length of 5 in. (the card height) to K_3; but for connections to be made between pins of a single card we have arbitrarily assigned K_3 = 3 in.

Referring to Fig. 18-12, we see that there are 33 output signals (direct or inverted). Accordingly we must form 33 separate, distinct trees for the outputs in this figure. The majority of these trees will be trivial, consisting of only two or three nodes. The wiring table, based on these 33 minimum trees, can take on several forms. First, we may form a table in which the branches of the various trees are ordered according to

Row 10 Sector selector (*A*-*E*) Sector counter (*F*-*K*) Minor-cycle counter (*K*-*P*) Unit-interval counter (*Q*-*Y*)	Row 9 Channel selector	Row 8 In-out phase gen. (*A*-*H*) Buffer register (*J*-*T*)	Row 7 icand register	Row 6 Accumulator register (*A*-*M*) ier register (*N*-*X*)	Row 5 Adder-subtractor Sign generators Output complementer	Row 4 Operations signal gen. (*A*-*O*) Operations phase gen. (*S*-*Y*)	Row 3 Instruction register (*A*-*J*) Instruction decoder (*L*-*Q*)	Row 2 Memory selector	Row 1 Current-address counter	
SS1 SS2 2	CH0- CH4 1	EWOG EWIG 2	*S*1IR 3	ASPS 5	SOAH CIAS 2	GASC GJCS 2	α1 α2 2	α*1 3	α'1 3	*A*
SS3 SS4 2	CH5- CH9 1	RIBG ROBG 1	*S*2IR *S*3IR 2	*S*1AR 3	ASCG 5	GLMS GLAS 2	α3 α4 2	α*2 3	α'2 3	*B*
SS5 SS6 2	CH10- CH14 1	STBG SPBG 2	*S*4IR *S*5IR 2	*S*2AR *S*3AR 2	ASAI 4	GLES GLIS 2	α5 α6 2	α*3 3	α'3 3	*C*
SS7 2	CH15- CH19 1	RUBG 2	*S*6IR *S*7IR 2	*S*4AR *S*6AR 2	ASBI 5	GDCS GSCC 2	α7 α8 2	α*4 3	α'4 3	*D*
SS*a* SS*b* GCOI 1	CH20- CH24 1	IFOG IFIG 2	*S*8IR *S*9IR 2	*S*5AR 4	IFCF 3	GMCS 3	α9 α10 2	α*5 3	α'5 3	*E*
SC1S SC2S 2	CH25- CH29 1	FOOG FOIG 2	*S*10IR *S*11IR 2	*S*7AR *S*8AR 2	ASRF 4	GPQT 3	α11 α12 2	α*6 3	α'6 3	*F*
SC3S SC4S 2	CH30 CH31 1	FBOG 3	*S*12IR *S*13IR 2	*S*9AR *S*10AR 2	ASCF 3	GPQC 4	ω1 ω2 2	α*7 3	α'7 3	*G*
SC5S SC6S 2	WR0- WR4 1	FBIG 4	*S*14IR *S*15IR 2	*S*11AR *S*12AR 2	RZDG OFLH 2	GOEP 4	ω3 ω4 2	α*8 3	α'8 3	*H*
SC7S 2	WR5- WR9 1		*S*16IR *S*17IR 2	*S*13AR *S*14AR 2	ASSG 3	GLDS GMMI GMDS GSIC 1	ω5 ω6 2	α*9 3	α'9 3	*I*
SC1*a*S- SC5*a*S 1	WR10- WR14 1	BSPG 4	*S*18IR 2	*S*15AR *S*16AR 2	PQSG 3	GPQS GDRI GDDS 1	IRSG 1	α*10 3	α'10 3	*J*
SC6*a*S MC1*a*S- MC8*a*S 1	WR15- WR19 1	BU1 3		*S*17AR *S*18AR 2	SPDG 5	SHA1 2		α*11 3	α'11 3	*K*
MCCL 3	WR20- WR24 1	BU2 BU3 2		*S*19AR *S*20AR 2	SOBH 5	SHA2 3	0 1 2 3 4 1	α*12 3	α'12 3	*L*
MC1S 3	WR25- WR29 1	BU4 BU5 2		*S*21AR *S*22AR 2	OCCF 2	SHA3 4	00 20 30 40 50 1		α'1a- α'5a 1	*M*
MC2S MC4S 2	WR30 WR31 1	BU6 BU7 2		ISPS 4	OCRF 3	SHA4 5	60 70 G00*D* G21*D* G32*D* 1		α'6a- α'10a 1	*N*
MC8S MC16S 2	RE0- RE4 1	BU8 BU9 2		*S*1ER 4	MUSG 3	SHA5 3	G41*D* G42*D* G43*D* G44*D* G50*D* 1		α'11a α''12a 1	*O*
T^1 T^2 T^{19} 1	RE5- RE9 1	BU10 BU11 2		*S*2ER *S*3ER 2			G52*D* G53*D* G54*D* G60*D* G70*D* 1		JCLG 1CSG STCG 1	*P*
UTSG UC1S 2	RE10- RE14 1	BU12 BU13 2		*S*4ER *S*5ER 2			G71*D* G72*D* G73*D* G74*D* 1			*Q*
UC2S UC4S 2	RE15- RE19 1	BU14 BU15 2		*S*6ER *S*7ER 2						*R*
UC8S UC16S 2	RE20- RE24 1	BU16 BU17 2		*S*8ER *S*9ER 2		F1SG F2SG 2				*S*
UC1aS- UC8aS T_{22} 1	RE25- RE29 1	BU18 BU19 2		*S*10ER *S*11ER 2		F3SG 4				*T*
T_0-T_4 1	RE30 RE31 1			*S*12ER *S*13ER 2		IF1G 3				*U*
T_5 T_{18} T_{19}-T_{21} 1	MO1- MO4 MO9 1			*S*14ER *S*15ER 2		IF2G 5				*V*
$T^1_{0\text{-}18}$ $T_{1\text{-}22}$ 2	MO5- MO8 MO10 1			*S*16ER *S*17ER 2		IF3G I'F1G 2				*W*
$T_{1\text{-}18}$ $T_{1\text{-}19}$ 2	GMOα 3			*S*18ER 4		I*F1G 5				*X*
$T_{1\text{-}17}$ $T_{0\text{-}22}$ 2	GMIB 3					ISFG 5				*Y*

FIG. 23-19. Location of cards in the Pedagac rack shown in Fig. 23-13. Numbers at tops of cards indicate their types. Packages for each card are as indicated.

TABLE 23-2. SCHEMATIC WIRING TABLE FOR FIG. 18-12†

	1	2	3	4		1	2	3	4
1	4-*A*-25	$\overline{\mathrm{SOB}}$	5-*B*-4	13	41	6-*A*-31	$\overline{\mathrm{ASP}}$	6-*B*-12	6
2	5-*B*-4	SOB	5-*I*-8	12	42	6-*A*-33	ASP	6-*C*-4	7
3	5-*B*-4	$\overline{\mathrm{SOB}}$	5-*B*-16	3	43	6-*B*-12	$\overline{\mathrm{ASP}}$	6-*C*-7	6
4	5-*B*-10	SOB	5-*I*-2	12	44	6-*B*-17	*P*1	6-*B*-26	3
5	5-*B*-10	SOB	5-*B*-22	3	45	6-*B*-22	*P*1	6-*B*-24	3
6	5-*H*-4	OFR	6-*C*-21	17	46	6-*B*-22	*P*1	6-*C*-3	6
7	5-*H*-4	OFR	5-*J*-10	7	47	6-*C*-4	ASP	6-*C*-24	3
8	5-*H*-10	SEZ	5-*H*-21	3	48	6-*C*-7	$\overline{\mathrm{ASP}}$	6-*C*-28	3
9	5-*H*-15	$\overline{\mathrm{SEZ}}$	5-*I*-14	6	49	6-*C*-10	*P*2	6-*C*-21	3
10	5-*H*-16	SEZ	5-*H*-18	3	50	6-*C*-16	*P*2	6-*C*-18	3
11	5-*H*-30	OF	5-*H*-40	3	51	6-*C*-21	*P*2	6-*C*-23	3
12	5-*H*-34	$\overline{\mathrm{OF}}$	5-*N*-4	11	52	6-*C*-24	ASP	6-*D*-4	6
13	5-*H*-35	OF	5-*H*-38	3	53	6-*C*-28	$\overline{\mathrm{ASP}}$	6-*D*-7	6
14	5-*H*-35	OF	5-*M*-4	10	54	6-*C*-30	*P*3	6-*C*-40	3
15	5-*I*-8	$\overline{\mathrm{SOB}}$	5-*L*-31	8	55	6-*C*-35	*P*3	6-*C*-38	3
16	5-*I*-8	$\overline{\mathrm{SOB}}$	5-*J*-8	6	56	6-*C*-40	*P*3	6-*D*-3	6
17	5-*I*-14	$\overline{\mathrm{SEZ}}$	5-*J*-15	6	57	6-*D*-4	ASP	6-*D*-24	3
18	5-*I*-21	SAS	5-*L*-2	8	58	6-*D*-7	$\overline{\mathrm{ASP}}$	6-*D*-28	3
19	5-*J*-2	SOB	5-*L*-33	7	59	6-*D*-10	*P*4	6-*D*-21	3
20	5-*J*-2	SOB	5-*I*-2	6	60	6-*D*-16	*P*4	6-*D*-18	3
21	5-*J*-4	$\overline{\mathrm{OFR}}$	6-*C*-19	19	61	6-*D*-20	*P*4	6-*E*-10	6
22	5-*J*-4	$\overline{\mathrm{OFR}}$	5-*K*-12	6	62	6-*D*-21	*P*4	6-*E*-15	6
23	5-*J*-10	OFR	5-*K*-18	6	63	6-*D*-23	*P*5	6-*E*-32	6
24	5-*J*-15	$\overline{\mathrm{SEZ}}$	5-*K*-24	6	64	6-*D*-24	ASP	6-*E*-5	6
25	5-*J*-22	SQP	5-*L*-8	7	65	6-*D*-28	$\overline{\mathrm{ASP}}$	6-*E*-17	6
26	5-*K*-11	SOD	5-*K*-37	3	66	6-*D*-30	*P*6	6-*D*-40	3
27	5-*K*-11	SOD	5-*K*-27	3	67	6-*D*-35	*P*6	6-*D*-38	3
28	5-*K*-16	$\overline{\mathrm{SOD}}$	5-*K*-36	3	68	6-*D*-40	*P*6	6-*F*-3	7
29	5-*K*-33	SOD	6-*X*-4	25	69	6-*E*-4	ACC	6-*M*-40	13
30	5-*K*-33	SOD	5-*K*-35	3	70	6-*E*-5	ASP	6-*E*-10	3
31	5-*L*-15	SOB	5-*L*-27	3	71	6-*E*-7	*P*5	6-*E*-30	3
32	5-*L*-27	SOB	5-*L*-37	3	72	6-*E*-10	ASP	6-*E*-15	3
33	5-*L*-33	SOB	5-*L*-35	3	73	6-*E*-15	ASP	6-*F*-4	6
34	5-*L*-33	SOB	9-*Y*-13	46	74	6-*E*-17	$\overline{\mathrm{ASP}}$	6-*F*-7	6
35	5-*M*-4	ACC	6-*M*-40	12	75	6-*E*-22	*P*5	6-*E*-32	3
36	5-*M*-4	ACC	5-*N*-4	6	76	6-*F*-4	ASP	6-*F*-24	3
37	5-*M*-4	OF	5-*M*-10	3	77	6-*F*-7	$\overline{\mathrm{ASP}}$	6-*F*-28	3
38	5-*M*-10	OF	5-*N*-16	6	78	6-*F*-10	*P*7	6-*F*-21	3
39	5-*N*-4	ACC	5-*N*-10	6	79	6-*F*-16	*P*7	6-*F*-18	3
40	5-*N*-16	$\overline{\mathrm{ACC}}$	6-*M*-39	13	80	6-*F*-21	*P*7	6-*F*-23	3

TABLE 23-2. SCHEMATIC WIRING TABLE FOR FIG. 18-12† (*Continued*)

	1	2	3	4		1	2	3	4
81	6-*F*-24	ASP	6-*G*-4	6	118	6-*J*-10	*P*15	6-*J*-21	3
82	6-*F*-28	$\overline{\text{ASP}}$	6-*G*-7	6	119	6-*J*-16	*P*15	6-*J*-18	3
83	6-*F*-30	*P*8	6-*F*-40	3	120	6-*J*-21	*P*15	6-*J*-23	3
84	6-*F*-35	*P*8	6-*F*-37	3	121	6-*J*-24	ASP	6-*K*-4	6
85	6-*F*-40	*P*8	6-*G*-3	6	122	6-*J*-28	$\overline{\text{ASP}}$	6-*K*-7	6
86	6-*G*-4	ASP	6-*G*-24	3	123	6-*J*-30	*P*16	6-*J*-40	3
87	6-*G*-7	$\overline{\text{ASP}}$	6-*G*-28	3	124	6-*J*-35	*P*16	6-*J*-37	3
88	6-*G*-10	*P*9	6-*G*-21	3	125	6-*J*-40	*P*16	6-*K*-3	6
89	6-*G*-16	*P*9	6-*G*-18	3	126	6-*K*-4	ASP	6-*K*-24	3
90	6-*G*-21	*P*9	6-*G*-23	3	127	6-*K*-7	$\overline{\text{ASP}}$	6-*K*-28	3
91	6-*G*-24	ASP	6-*H*-4	6	128	6-*K*-10	*P*17	6-*K*-21	3
92	6-*G*-28	$\overline{\text{ASP}}$	6-*H*-7	6	129	6-*K*-16	*P*17	6-*K*-18	3
93	6-*G*-30	*P*10	6-*G*-40	3	130	6-*K*-21	*P*17	6-*K*-23	3
94	6-*G*-35	*P*10	6-*G*-37	3	131	6-*K*-24	ASP	6-*L*-4	6
95	6-*G*-40	*P*10	6-*H*-3	6	132	6-*K*-28	$\overline{\text{ASP}}$	6-*L*-7	6
96	6-*H*-4	ASP	6-*H*-24	3	133	6-*K*-30	*P*18	6-*K*-40	3
97	6-*H*-7	$\overline{\text{ASP}}$	6-*H*-28	3	134	6-*K*-35	*P*18	6-*K*-37	3
98	6-*H*-10	*P*11	6-*H*-21	3	135	6-*K*-40	*P*18	6-*L*-3	6
99	6-*H*-16	*P*11	6-*H*-18	3	136	6-*L*-4	ASP	6-*L*-24	3
100	6-*H*-21	*P*11	6-*H*-23	3	137	6-*L*-7	$\overline{\text{ASP}}$	6-*L*-28	3
101	6-*H*-24	ASP	6-*I*-4	6	138	6-*L*-10	*P*19	6-*L*-21	3
102	6-*H*-28	$\overline{\text{ASP}}$	6-*I*-7	6	139	6-*L*-16	*P*19	6-*L*-18	3
103	6-*H*-30	*P*12	6-*H*-40	3	140	6-*L*-21	*P*19	6-*L*-23	3
104	6-*H*-35	*P*12	6-*H*-37	3	141	6-*L*-24	ASP	6-*M*-4	6
105	6-*H*-40	*P*12	6-*I*-3	6	142	6-*L*-28	$\overline{\text{ASP}}$	6-*M*-7	6
106	6-*I*-4	ASP	6-*I*-24	3	143	6-*L*-30	*P*20	6-*L*-40	3
107	6-*I*-7	$\overline{\text{ASP}}$	6-*I*-28	3	144	6-*L*-35	*P*20	6-*L*-37	3
108	6-*I*-10	*P*13	6-*I*-21	3	145	6-*L*-40	*P*20	6-*M*-3	6
109	6-*I*-16	*P*13	6-*I*-18	3	146	6-*M*-4	ASP	6-*M*-24	3
110	6-*I*-21	*P*13	6-*I*-23	3	147	6-*M*-7	$\overline{\text{ASP}}$	6-*M*-28	3
111	6-*I*-24	ASP	6-*J*-4	6	148	6-*M*-10	*P*21	6-*M*-21	3
112	6-*I*-28	$\overline{\text{ASP}}$	6-*J*-7	6	149	6-*M*-16	*P*21	6-*M*-18	3
113	6-*I*-30	*P*14	6-*J*-40	6	150	6-*M*-21	*P*21	6-*M*-23	3
114	6-*I*-35	*P*14	6-*I*-37	3	151	6-*M*-24	ASP	6-*N*-4	6
115	6-*I*-40	*P*14	6-*J*-3	6	152	6-*M*-28	$\overline{\text{ASP}}$	6-*N*-7	6
116	6-*J*-4	ASP	6-*J*-24	3	153	6-*M*-30	*P*22	6-*M*-40	3
117	6-*J*-7	$\overline{\text{ASP}}$	6-*J*-28	3	154	6-*M*-35	*P*22	6-*M*-37	3

† Column 1 indicates row, column, and pin number of first node of the branch. Column 2 indicates name of tree of which the branch is part. Column 3 indicates row, column, and pin number of second node of the branch. Column 4 indicates the length of the branch.

node locations, i.e., ordered by rack, row, column, and pin number. This type of table is referred to as the *schematic wiring table;* Table 23-2 is the schematic wiring table for the outputs of Fig. 18-12. This is the table to be used by the wiring technicians. A second type of table, the alphabetic table, is formed directly from the minimum trees, by ordering the signal names alphabetically. Table 23-3 is a portion of the alphabetic table for Fig. 18-12.

TABLE 23-3. PORTION OF ALPHABETIC TABLE FOR FIG. 18-12

Signal name	Origin	Destination
ACC	6-*M*-40	5-*M*-4
ACC	5-*M*-4	5-*N*-4
ACC	5-*N*-4	5-*N*-10
ACC	6-*M*-40	6-*E*-4
$\overline{\text{ACC}}$	6-*M*-39	5-*N*-16
SOB	5-*L*-33	5-*L*-35
SOB	5-*L*-33	9-*Y*-13
SOB	5-*L*-37	5-*L*-27
SOB	5-*L*-27	5-*L*-15
SOB	5-*L*-33	5-*J*-2
SOB	5-*J*-2	5-*I*-2
SOB	5-*I*-2	5-*B*-10
SOB	5-*B*-10	5-*B*-22

EXERCISES

(*a*) Complete the alphabetic table for Fig. 18-12.
(*b*) Form the trees for Fig. 18-11, using the first algorithm.
(*c*) Form the trees for Fig. 18-11, using the second algorithm.
(*d*) Form the schematic wiring table for Fig. 18-11.
(*e*) Form the alphabetic table for Fig. 18-11.

23-6. Additional Topics

a. Articles Concerning Wiring Diagrams

Cayley, A.: A Theorem on Trees, "Mathematical Papers," vol. 13, pp. 26–28, Cambridge University Press, London, 1897.

Kruskal, J. B., Jr.: On the Shortest Spanning Subtree of a Graph and the Traveling Salesman Problem, *Proc. Am. Math. Soc.*, vol. 7, pp. 48–50, 1956.

Loberman, H., and A. Weinberger: Formal Procedures for Connecting Terminals with a Minimum Total Wire Length, *J. Assoc. Computing Machinery,* vol. 4, no. 4, pp. 428–437, October, 1957.

Weinberger, A., and H. Loberman: Symbolic Designations for Electrical Connections, *J. Assoc. Computing Machinery,* vol. 4, no. 4, pp. 420–427, October, 1957.

b. With the information given in Chaps. 18 and 23 assign clock-phase signals to all the *and* gates of the Pedagac packages.

c. Prepare complete schematic and/or alphabetic wiring tables for the Pedagac.

APPENDIX

TABLE A-1. ARITHMETIC-UNIT EXTERNALLY GENERATED OPERATIONS SIGNALS
(See Fig. 18-10)

Name of signal	Equation	Time period present	Destination		Generation location	Remarks
			Package	Gate		
A_1	$[\alpha 1] \cdot [T^2] \cdot [T_5]$ $+ [\bar{A}_{1d}] \cdot [T^2]$ $\cdot [\bar{T}_5]$	As generated	GSIC SHA1 SHA2 SHA3 SHA4 SHA5	$\bar{1}$ $\bar{2}_d$ $2_d, \bar{3}_d$ $2_d, \bar{4}_d$ $2_d, \bar{5}_d$ 2_d	SHA1	Least significant bit of shift count-down
A_2	$[\alpha_2] \cdot [T^2] \cdot [T_5]$ $+ [T^2] \cdot [\bar{T}_5]$ $\cdot [A_{1d} \cdot A2_d$ $+ \bar{A}_{1d} \cdot \bar{A}_{2d}]$	As generated	GSIC SHA2 SHA3 SHA4 SHA5	$\bar{1}$ $2_d, \bar{3}_d$ $3_d, \bar{4}_d$ $3_d, \bar{5}_d$ 3_d	SHA2	Fourth significant bit of shift count-down
A_3	$[\alpha 3] \cdot [T^2] \cdot [T_5]$ $+ [T^2] \cdot [\bar{T}_5]$ $\cdot [A_{1d} \cdot A_{3d}$ $+ A_{2d} \cdot A_{3d}$ $+ \bar{A}_{1d} \cdot \bar{A}_{2d}$ $\cdot \bar{A}_{3d}]$	As generated	GSIC SHA3 SHA4	$\bar{1}$ $2_d, 3_d,$ $\bar{4}_d$ 4_d	SHA3	Third significant bit of shift count-down
A_4	$[\alpha 4] \cdot [T^2] \cdot [T_5]$ $+ [T^2] \cdot [\bar{T}_5]$ $\cdot [A_{1d} \cdot A_{4d}$ $+ A_{2d} \cdot A_{4d}$ $+ A_{3d} \cdot A_{4d}$ $+ \bar{A}_{1d} \cdot \bar{A}_{2d} \cdot A_{5d}]$	As generated	GSIC SHA4	$\bar{1}$ $2_d, 3_d,$ 4_d	SHA4	Second significant bit of shift count-down
A_5	$[\alpha 5] \cdot [T^2] \cdot [T_5]$ $+ [T^2] \cdot [\bar{T}_5]$ $\cdot [A_{5d}] \cdot [A_{1d}]$ $+ [A_{2d}]$	As generated	GSIC SHA4 SHA5	$\bar{1}$ 5_d $2_d, 3_d$	SHA5	Most significant bit of shift count-down
ADD	$53D$	As generated	ASCG IF2G (GASC 1)	3, 4 1	($53D$)	Addition decode signal

TABLE A-1. ARITHMETIC-UNIT EXTERNALLY GENERATED OPERATIONS SIGNALS (*Continued*)

Name of signal	Equation	Time period present	Destination		Generation location	Remarks
			Package	Gate		
ASC	$[53D] \cdot [\bar{T}_0] + [54D] \cdot [\bar{T}_0]$	$\bar{T}_0$	ASSG SOBH	1, 2 1, $\bar{3}$	GASC	Add-subtract control signal, on during addition and subtraction instruction
CLS	$50D$	As generated	ASBI SOBH F3SG	$\bar{2}$ 4 2	($50D$)	Clear instruction control signal; this signal inhibits contents of accumulator from entering adder, thereby clearing accumulator; it also ensures that sign of accumulator is positive after clearing
DCS	$[41D] \cdot [\overline{T^{19}}] \cdot [T_1] + [\text{DCS}_d] \cdot [\bar{T}_{19}]$	T^{1-18}_{1-18}	ASAI ASBI GDCS	2 3 2_d	GDCS	Division control signal
DDS	$[41D] \cdot [\overline{T^{19}}]$	T^{1-18}	ASCG SPDG S1ER S18ER ISPS	$\bar{1}, \bar{2}, \bar{3}, \bar{4}, \bar{5}$ 1, 2, 3, 4, 5 3 1, 3, $\bar{4}$ 3	GDDS	Division decode signal
DRI	$[41D] \cdot [T_{21}]$	T_{21}	S1ER	$\bar{4}$	GDRI	Division recirculation inhibit signal; it kills recirculation in S1 of ier
JCS	$[43D] \cdot [\overline{\text{SOB}}] \cdot [T_{22}] \cdot [F1] + [44D] \cdot [T_{22}] \cdot [F1]$	$F1$, T_{22}	JCLG 1CSG	1 $\bar{1}$	GJCS	Jump control signal
LAS	$[72D] \cdot [T_1] + [\text{LAS}_d] \cdot [\bar{T}_{19}]$	T_{1-18}	S1AR GLAS	$\bar{1}$ 2_d	GLAS	Logical-add signal
LDS	$71D + 72D + 73D + 74D$	As generated	ITCF CIAS S1AR IF2G	3 1, $\bar{2}$ 2 3	GLDS	Logical decode signal; this signal is on when any of the four logical instructions is present

TABLE A-1. ARITHMETIC-UNIT EXTERNALLY GENERATED OPERATIONS SIGNALS (*Continued*)

Name of signal	Equation	Time period present	Destination		Generation location	Remarks
			Package	Gate		
LES	$[73D]\cdot[T_1]$ $+ [\mathrm{LES}_d]\cdot[\bar{T}_{19}]$	T_{1-18}	ITCF S1AR GLES	$\overline{3}$ $\overline{2}$ 2_d	GLES	Logical-equivalence signal
LIS	$[74D]\cdot[T_1]$ $+ [\mathrm{LIS}_d]\cdot[\bar{T}_{19}]$	T_{1-18}	ITCF CIAS S1AR GLIS	$\overline{3}$ $\overline{1}$ $\overline{2}$ 2_d	GLIS	Logical-inequivalence signal
LMS	$[71D]\cdot[T_1]$ $+ [\mathrm{LMS}_d]\cdot[\bar{T}_{19}]$	T_{1-18}	CIAS S1AR GLMS	$\overline{1}$ $\overline{1}$ 2_d	GLMS	Logical-multiply signal
MCS	$[42D]\cdot[\overline{T^{19}}]\cdot[T_1]$ $+ [32D]\cdot[\overline{T^{19}}]$ $\cdot[T_1]$ $+ [\mathrm{MCS}_d]\cdot[\bar{T}_{19}]$	T_{1-18}^{1-18}	ASAI ASBI S1ER ISPS GMCS	3, 4 4 1 1 3_d	GMCS	Multiply control signal
MDS	$32D + 42D$	As generated	ASCG ASAI ASPS MUSG S1ER ISPS	5 $\overline{1}$ 5 1, 2 2 2	GMDS	Multiplication decode signal
MMI	$[42D]\cdot[T^{19}]$	T^{19}	ASPS	$\overline{1}$	GMMI	Major-multiplication inhibit signal; prevents contents of accumulator from recirculating during 19th minor cycle
OEP	$[\overline{\mathrm{PQS}}]\cdot[T^1]\cdot[T_{22}]$ $\cdot[F2] + [\mathrm{PQS}]$ $\cdot[T^{19}]\cdot[T_{22}]$ $\cdot[F2] + [\mathrm{SHS}]$ $\cdot[T^2]\cdot[T_{22}]\cdot[F3]$ $+ [\overline{\mathrm{SHS}}]\cdot[T^1]$ $\cdot[T_{22}]\cdot[F3]$	T_{22} at end of operation	SOAH SOBH PQSG MCCL F2SG F3SG I*F1G	$\overline{2}$ 1, 2, $\overline{5}$ 2 2_d, 3_d $\overline{2}_d$ 4_d 3_d, 4_d	GOEP	Operation-ending pulse
PQC	$[32D]\cdot[\overline{T^{19}}]\cdot[T_1]$ $+ [41D]\cdot[\overline{T^{19}}]$ $\cdot[T_1] + [42D]$ $\cdot[\overline{T^{19}}]\cdot[T_1]$ $+ [\mathrm{PQC}_d]\cdot[\bar{T}_{19}]$	T_{1-18}^{1-18}	S1IR S2IR S3IR GPQC	1, 2, $\overline{3}$ 1, $\overline{2}$ 1, $\overline{2}$ 4_d	GPQC	Product-quotient control signal

TABLE A-1. ARITHMETIC-UNIT EXTERNALLY GENERATED OPERATIONS SIGNALS (*Continued*)

Name of signal	Equation	Time period present	Destination		Generation location	Remarks
			Package	Gate		
PQS	$32D + 41D + 42D$	As generated	PQSG SOBH IF2G GOEP ASBI	1, 2, 3 2, $\bar{3}$ 4 $\bar{1}$, 2 $\bar{1}$	GPQS	Product-quotient sign read-in; gates SOA and SOB into product-quotient sign generator
PQT	$[32D] \cdot [T^{19}] \cdot [T_1]$ $+ [41D] \cdot [T^{19}]$ $\cdot [T_1]$ $+ [\mathrm{PQT}_d] \cdot [\bar{T}_{19}]$	T^{19}_{1-18}	ASBI ISPI GPQT	5 4 3_d	GPQT	Product-quotient transfer signal
SCC	$[\mathrm{SIC}] \cdot [\bar{T}_1] \cdot [\bar{T}_2]$ $\cdot [\bar{T}_3] \cdot [\bar{T}_4]$ $+ [\mathrm{SCC}_d] \cdot [\bar{T}_1]$ $\cdot [F3]$	$T^2_{\mathrm{SIC}-22}$	ASPS GSCC	$\bar{4}$ 2_d	GSCC	Shift control signal
SHS	$70D$	As generated	ASPS GOEP F3SG S5AR	$\bar{2}$, 3, 4 3, $\bar{4}$ 1 1, 2, $\bar{3}$	($70D$)	Shift decode signal
SIC	$[\bar{A}_1] \cdot [\bar{A}_2] \cdot [\bar{A}_3]$ $\cdot [\bar{A}_4] \cdot [\bar{A}_5]$	As generated	GSCC	1	GSIC	Shift inhibit control signal
SUB	$54D$	As generated	ASCG IF2G (GASC 2)	1, 2 2	($54D$)	Subtraction decode signal
TRS	$52D$	As generated	GMIB IF3G	2, 3 1	($52D$)	Transfer decode signal

TABLE A-2. ARITHMETIC-UNIT INTERNALLY GENERATED SIGNALS
(See Figs. 18-11 to 18-13)

Name of signal	Equation	Time period present	Destination		Generation location	Remarks
			Package	Gate		
A	$[(\alpha)] \cdot [\overline{\mathrm{MDS}}] \cdot [T^1]$ $\cdot [F2] + [1\mathrm{CN}]$ $\cdot [\mathrm{DCS}] \cdot [\overline{T^1}]$ $\cdot [F2] + [(\alpha)]$ $\cdot [\mathrm{MCS}] \cdot [\mathrm{MUS}]$ $\cdot [T^1] \cdot [F2]$ $+ [1\mathrm{CN}] \cdot [\mathrm{MCS}]$ $\cdot [\mathrm{MUS}] \cdot [\overline{T^1}]$ $\cdot [F2]$	$F2$	ASRF ASCF	1, $\bar{2}$, $\bar{3}$, 4 1, 3	ASAI	α input to adder-subtractor

TABLE A-2. ARITHMETIC-UNIT INTERNALLY GENERATED SIGNALS (*Continued*)

Name of signal	Equation	Time period present	Destination		Generation location	Remarks
			Package	Gate		
(ACC)	$[P21_d] \cdot [\text{ASP}] + [P22_d] \cdot [\overline{\text{ASP}}]$	$F2$ or $F3$	OCRF OCCF S5AR	$1, 2, \bar{3}$ 1 1	Accumulator	Contents of accumulator entering output complementer
ASP	$[\overline{\text{MMI}}] \cdot [\bar{T}_0] \cdot [F2] + [\overline{\text{SHS}}] \cdot [\bar{T}_0] \cdot [F3] + [\text{SHS}] \cdot [\overline{\text{SCC}}] \cdot [\bar{T}_0] \cdot [T^2] \cdot [F3] + [\text{MDS}] \cdot [T_0] \cdot [\overline{T^1}] \cdot [F2] + [\text{SHS}] \cdot [T^1] \cdot [\bar{T}_0] \cdot [F3]$	$F2$ or $F3$	S1AR S2AR through S22AR except: S5AR	$\bar{3}$ $1, \bar{2}$ $1, 2, 3, \bar{4}$	ASPS	Accumulator shift pulse signal
B	$[\bar{T}_0] \cdot [\overline{\text{OF}}] \cdot [\text{ACC}] + [\bar{T}_0] \cdot [\overline{\text{CC}_d}] \cdot [\text{ACC}] + [\bar{T}_0] \cdot [\text{OF}] \cdot [\overline{\text{ACC}}] \cdot [\text{CC}_d]$	$F2$ or $F3$	S1ER MUSG ASBI GMIB	1, 2 1 $1, 2, 3_d, 4$ 2	OCRF	Contents of accumulator out of output complementer
B'	$[B] \cdot [\overline{\text{PQS}}] \cdot [F2] \cdot [T^1_{0-18}] \cdot [\bar{T}_0] + [B] \cdot [\overline{\text{CLS}}] \cdot [F3] \cdot [\bar{T}_0] \cdot [T^1] + [B_d] \cdot [\text{DCS}] \cdot [F2] + [B] \cdot [\text{MCS}] \cdot [F2] \cdot [\overline{T^1}] + [(\text{IER})] \cdot [\text{PQT}] \cdot [F2]$	$F2$ or $F3$	IFCF ASRF	$1, \bar{2}, 3$ $\bar{1}, 2, \bar{3}, 4$	ASBI	B input into adder-subtractor
B''	$[B'] \cdot [\text{OPS}] + [\overline{B'}] \cdot [\overline{\text{OPS}}] \cdot [F2] + [B'] \cdot [\text{LDS}] \cdot [\overline{\text{LES}}] \cdot [\overline{\text{LIS}}] \cdot [F2]$	$F2$	ASCF	2, 3	ITCF	B input into carry generator
C	$[\text{LDS}] \cdot [\overline{\text{LMS}}] \cdot [\overline{\text{LIS}}] \cdot [F2] + [\overline{\text{LDS}}] \cdot [C'_d] \cdot [\bar{T}_0] \cdot [F2]$	$F2$	ASCF ASRF	1, 2 $\bar{1}, \bar{2}, 3, 4$	CIAS	Carry input into adder-subtractor
C'	$[A] \cdot [C] \cdot [\bar{T}_0] + [B''] \cdot [C] \cdot [\bar{T}_0] + [A] \cdot [B''] \cdot [\bar{T}_0]$	$F2$	CIAS S1AR	2_d 2	ASCF	Carry signal from adder-subtractor carry generator

TABLE A-2. ARITHMETIC-UNIT INTERNALLY GENERATED SIGNALS (*Continued*)

Name of signal	Equation	Time period present	Destination		Generation location	Remarks
			Package	Gate		
CC	$[(\mathrm{ACC})] \cdot [\mathrm{OF}] \cdot [\bar{T}_0] + [\mathrm{CC}_d] \cdot [\mathrm{OF}] \cdot [\bar{T}_0]$	$F2$ or $F3$	OCCF OCRF	2_d $\bar{2}_d$, 3_d	OCCF	Carry from output complementer
(ICN)	$[P17_d] \cdot [\mathrm{PQC}] + [P18_d] \cdot [\overline{\mathrm{PQC}}]$	$F2$	S1IR ASAI	2 2, 4	S18IR	Contents of icand register
(IER)	$[P17_d] \cdot [\mathrm{SPI}] + [(\mathrm{IER})] \cdot [\overline{\mathrm{SPI}}] \cdot [F2] \cdot [\overline{\mathrm{DDS}}] + [\bar{T}_{21}] \cdot [(\mathrm{IER})] \cdot [\overline{\mathrm{SPI}}] \cdot [F2] \cdot [\mathrm{DDS}] + [\mathrm{SOD}] \cdot [F2] \cdot [\mathrm{DDS}] \cdot [T_{21}]$	$F2$	S1ER MUSG ASBI	3 2 5	S18ER	Contents of ier register
MUS	$[B] \cdot [\mathrm{MDS}] \cdot [T^1] \cdot [T_1] \cdot [F2] + [(\mathrm{IER})] \cdot [\mathrm{MDS}] \cdot [\overline{T^1}] \cdot [T_1] \cdot [F2] + [\mathrm{MUS}_d] \cdot [\bar{T}_0] \cdot [F2]$	$F2$, T_{1-22}^{1-18}	MUSG ASAI	3_d 3, 4	MUSG	Multiply control signal; this signal is used to gate contents of α and contents of icand register into the adder during multiplication instructions
OF	$[\mathrm{OFR}] \cdot [T_0] + [\mathrm{OF}_d] \cdot [\bar{T}_0]$	T_{0-22}	OFLH OCRF OCCF	2_d $\bar{1}$, 3 1, 2	OFLH	Overflow signal to output complementer
OFR	$\{[P1_d] \cdot [\mathrm{ASP}] + [P2_d] \cdot [\overline{\mathrm{ASP}}]\}_d$	T_{21}, T^{OEP}	ASSG SPDG OFLH	$\bar{1}$, 2 $\bar{2}$, 3 1	S2AR	Overflow due to result of arithmetic operation; this signal is used to determine sign of B as well as to determine value of quotient bit

TABLE A-2. ARITHMETIC-UNIT INTERNALLY GENERATED SIGNALS (*Continued*)

Name of signal	Equation	Time period present	Destination		Generation location	Remarks
			Package	Gate		
OPS	$[\text{SOA}] \cdot [\overline{\text{SOB}}]$ $\cdot [\text{SUB}] \cdot [\overline{\text{DDS}}]$ $\cdot [F2] + [\overline{\text{SOA}}]$ $\cdot [\text{SOB}] \cdot [\text{SUB}]$ $\cdot [\overline{\text{DDS}}] \cdot [F2]$ $+ [\overline{\text{SOA}}] \cdot [\overline{\text{SOB}}]$ $\cdot [\text{ADD}] \cdot [\overline{\text{DDS}}]$ $\cdot [F2] + [\text{SOA}]$ $\cdot [\text{SOB}] \cdot [\text{ADD}]$ $\cdot [\overline{\text{DDS}}] \cdot [F2]$ $+ [\text{MDS}]$ $\cdot [\overline{\text{DDS}}] \cdot [F2]$	$F2$	ITCF	$1, \bar{2}$	ASCG	Operation selector signal; the presence of this signal causes adder to add; absence causes adder to subtract
P5AR	$[P5_d] \cdot [\overline{\text{ASP}}]$ $+ [P4_d] \cdot [\text{ASP}]$ $\cdot [\overline{\text{SHS}}]$ $+ [P4_d][\text{ASP}]$ $\cdot [\text{SHS}] \cdot [T^1]$ $\cdot [F3] + [(\text{ACC})]$ $\cdot [\text{ASP}] \cdot [\text{SHS}]$ $\cdot [T^2] \cdot [F3]$	As generated	S5AR S6AR	4_d 1_d	S5AR	Contents of stage 5 of accumulator register
Pn	$[P(n - 1)_d]$ $\cdot$ [shift pulse signal] $+ [Pn_d]$ $\cdot [\overline{\text{shift pulse sig-}}$ $\overline{\text{nal}}]$	As generated	SnIR or SnER or SnAR	[Highest gate number in package]$_d$	SnIR SnER SnAR	Recirculation of contents of nth stage of icand, ier, or accumulator register
R	$[A] \cdot [\bar{B}'] \cdot [\bar{C}]$ $\cdot [\bar{T}_0] + [\bar{A}] \cdot [\bar{B}']$ $\cdot [\bar{C}] \cdot [\bar{T}_0] + [\bar{A}]$ $\cdot [B'] \cdot [C] \cdot [\bar{T}_0]$ $+ [A] \cdot [B'] \cdot [C]$ $\cdot [\bar{T}_0]$	$F2$, $F3$	RZDG S1AR	1 1	ASRF	Results from adder-subtractor
SAS	$[\text{SOB}] \cdot [\overline{\text{OFR}}]$ $\cdot [\text{ASC}] \cdot [F2]$ $+ [\overline{\text{SOB}}] \cdot [\text{OFR}]$ $\cdot [\text{ASC}] \cdot [F2]$ $+ [\overline{\text{SEZ}}] \cdot [F2]$	$F2$	SOBH	1	ASSG	Sign of R as result of addition or subtraction
SEZ	$[R] \cdot [\bar{T}_0]$ $+ [\text{SEZ}_d] \cdot [\bar{T}_0]$	$F2$	RZDG ASSG PQSG SPDG	2_d $\bar{3}$ $\bar{3}$ $\bar{4}$	RZDG	Sum equals zero signal
SOA	$[(\alpha)] \cdot [T^1] \cdot [T_0]$ $\cdot [F2] + [\text{SOA}_d]$ $\cdot [\overline{\text{OEP}}]$	T_0^1 to OEP, $F2$	SOAH ASCG PQSG	2_d $1, \bar{2}, \bar{3}, 4$ $1, \bar{2}$	SOAH	Sign of α

TABLE A-2. ARITHMETIC-UNIT INTERNALLY GENERATED SIGNALS (*Continued*)

Name of signal	Equation	Time period present	Destination		Generation location	Remarks
			Package	Gate		
SOB	$[\mathrm{SAS}]\cdot[\mathrm{ASC}]$ $\cdot[\mathrm{OEP}]\cdot[F2]$ $+[\mathrm{SQP}]\cdot[\mathrm{PQS}]$ $\cdot[\mathrm{OEP}]\cdot[F2]$ $+[\mathrm{SOB}_d]\cdot[\overline{\mathrm{PQS}}]$ $+[\mathrm{CLS}]\cdot[T_0]$ $\cdot[F3]$ $+[\mathrm{SOB}_d]\cdot[\overline{\mathrm{OEP}}]$	$F2$, $F3$	SOBH ASSG PQSG ASCG GMIB GJCS	3_d, 5_d $1, \overline{2}$ $1, \overline{2}$ $\overline{1}, 2, \overline{3}, 4$ 3 $\overline{1}$	SOBH	Sign of contents of accumulator
SOD	$[\mathrm{DDS}]\cdot[T_0{}^1]\cdot[F2]$ $+[\mathrm{SOD}_d]$ $\cdot[\overline{\mathrm{OFR}}]\cdot[\mathrm{DDS}]$ $\cdot[T_{21}]\cdot[F2]$ $+[\overline{\mathrm{SOD}}_d]$ $\cdot[\mathrm{OFR}]\cdot[\mathrm{DDS}]$ $\cdot[T_{21}]\cdot[F2]$ $+[\overline{\mathrm{SEZ}}]\cdot[\mathrm{DDS}]$ $\cdot[T_{21}]\cdot[F2]$ $+[\mathrm{SOD}_d]$ $\cdot[\mathrm{DDS}]\cdot[T_{21}]$ $\cdot[F2]$	$F2$	SPDG S18ER	2_d, $\overline{3}_d$, 5_d 1	SPDG	Quotient signal
SPI	$[\mathrm{MCS}]\cdot[T^1]\cdot[F2]$ $+[\mathrm{MDS}]\cdot[T_0]$ $\cdot[\overline{T^1}]\cdot[F2]$ $+[\mathrm{DDS}]\cdot[T_{1-17}]$ $\cdot[F2]$ $+[\mathrm{PQT}]\cdot[F2]$	$F2$	S1ER S2ER through S17ER S18ER	$\overline{4}$ $1, \overline{2}$ $2, \overline{3}, \overline{4}$	ISPS	Ier shift pulse signal
SQP	$[\mathrm{SOA}]\cdot[\mathrm{SOB}]$ $\cdot[\mathrm{PQS}]\cdot[F2]$ $+[\overline{\mathrm{SOA}}]\cdot[\overline{\mathrm{SOB}}]$ $\cdot[\mathrm{OEP}]\cdot[\mathrm{PQS}]$ $\cdot[F2]+[\overline{\mathrm{SEZ}}]$ $\cdot[\mathrm{PQS}]\cdot[F2]$	$F2$	SOBH	2	PQSG	Sign of quotient or product

TABLE A-3. CURRENT-ADDRESS AND INSTRUCTION SIGNALS
(See Fig. 18-14)

Name of signal	Equation	Time period present	Destination		Generation location	Remarks
			Package	Gate		
$00D$	$[\overline{\omega 6}\cdot\overline{\omega 5}\cdot\overline{\omega 4}]$ $\cdot[\overline{\omega 3}\cdot\overline{\omega 2}\cdot\overline{\omega 1}]$	As generated	FBIG	1	G00D	Input-unit read-in decode signal
$21D$	$[\overline{\omega 6}\cdot\omega 5\cdot\overline{\omega 4}]$ $\cdot[\ \overline{\omega 3}\cdot\overline{\omega 2}\cdot\omega 1]$	As generated	IFOG	1	G21D	Output-unit read-out decode signal

TABLE A-3. CURRENT-ADDRESS AND INSTRUCTION SIGNALS (*Continued*)

Name of signal	Equation	Time period present	Destination		Generation location	Remarks
			Package	Gate		
32*D*	$[\overline{\omega6} \cdot \omega5 \cdot \omega4] \cdot [\overline{\omega3} \cdot \omega2 \cdot \overline{\omega1}]$	As generated	GMCS GMDS GPQC GPQS GPQT	2 $\overline{1}$ 1 $\overline{1}$ 1	G32D	Minor-multiplication decode signal
41*D*	$[\omega6 \cdot \overline{\omega5} \cdot \overline{\omega4}] \cdot [\overline{\omega3} \cdot \overline{\omega2} \cdot \omega1]$	As generated	GDCS GDDS GDRI GPQC GPQS GPQT	1 1 1 3 $\overline{1}$ 2	G41D	Division decode signal
42*D*	$[\omega6 \cdot \overline{\omega5} \cdot \overline{\omega4}] \cdot [\overline{\omega3} \cdot \omega2 \cdot \overline{\omega1}]$	As generated	GMDS GMMI GPQC GPQS GMCS	$\overline{1}$ 1 2 $\overline{1}$ 1	G42D	Major-multiplication decode signal
43*D*	$[\omega6 \cdot \overline{\omega5} \cdot \overline{\omega4}] \cdot [\overline{\omega3} \cdot \omega2 \cdot \omega1]$	As generated	GJCS I*F1G	1 1	G43D	Conditional jump decode signal
44*D*	$[\omega6 \cdot \overline{\omega5} \cdot \overline{\omega4}] \cdot [\omega3 \cdot \overline{\omega2} \cdot \overline{\omega1}]$	As generated	GJCS I*F1G	2 2	G44D	Normal-jump decode signal
50*D*	$[\omega6 \cdot \overline{\omega5} \cdot \omega4] \cdot [\overline{\omega3} \cdot \overline{\omega2} \cdot \overline{\omega1}]$	As generated	Directly to CLS signal		G50D	Clear-accumulator decode signal
52*D*	$[\omega6 \cdot \overline{\omega5} \cdot \omega4] \cdot [\overline{\omega3} \cdot \omega2 \cdot \overline{\omega1}]$	As generated	Directly to TRS signal		G52D	Transfer-contents-of-accumulator decode signal
53*D*	$[\omega6 \cdot \overline{\omega5} \cdot \omega4] \cdot [\overline{\omega3} \cdot \omega2 \cdot \omega1]$	As generated	GASC Directly to ADD signal	1	G53D	Addition decode signal
54*D*	$[\omega6 \cdot \overline{\omega5} \cdot \omega4] \cdot [\omega3 \cdot \overline{\omega2} \cdot \overline{\omega1}]$	As generated	GASC Directly to SUB signal	2	G54D	Subtraction decode signal
60*D*	$[\omega6 \cdot \omega5 \cdot \overline{\omega4}] \cdot [\overline{\omega3} \cdot \overline{\omega2} \cdot \overline{\omega1}]$	As generated	ISFG	2	G60D	Stop decode signal
70*D*	$[\omega6 \cdot \omega5 \cdot \omega4] \cdot [\overline{\omega3} \cdot \overline{\omega2} \cdot \overline{\omega1}]$	As generated	Directly to SHS signal		G70D	Shift decode signal
71*D*	$[\omega6 \cdot \omega5 \cdot \omega4] \cdot [\overline{\omega3} \cdot \overline{\omega2} \cdot \omega1]$	As generated	GLDS GLMS	$\overline{1}$ 1	G71D	Logical-multiply decode signal
72*D*	$[\omega6 \cdot \omega5 \cdot \omega4] \cdot [\overline{\omega3} \cdot \omega2 \cdot \overline{\omega1}]$	As generated	GLAS GLDS	1 $\overline{1}$	G72D	Logical-addition decode signal

TABLE A-3. CURRENT-ADDRESS AND INSTRUCTION SIGNALS (*Continued*)

Name of signal	Equation	Time period present	Destination		Generation location	Remarks
			Package	Gate		
$73D$	$[\omega6 \cdot \omega5 \cdot \omega4] \cdot [\overline{\omega3} \cdot \omega2 \cdot \omega1]$	As generated	GLDS GLES	$\bar{1}$ 1	G73D	Logical-equivalence decode signal
$74D$	$[\omega6 \cdot \omega5 \cdot \omega4] \cdot [\omega3 \cdot \overline{\omega2} \cdot \overline{\omega1}]$	As generated	GLDS GLIS	$\bar{1}$ 1	G74D	Logical-inequivalence decode signal
1CS	$[\overline{\text{JCS}}] \cdot [F1] \cdot [T_{22}]$	$F1$, T_{22}	$\alpha ia\left\{\begin{matrix} i \text{ from} \\ 1 \text{ to } 12 \end{matrix}\right.$ $\alpha i\left\{\begin{matrix} i \text{ from} \\ 1 \text{ to } 12 \end{matrix}\right.$	1 $\bar{1}$, 2	1CSG	Current-address-register stepping signal
JCL	$[\text{JCS}] \cdot [F1] \cdot [T_{22}]$	$F1$, T_{22}	$\alpha i\left\{\begin{matrix} i \text{ from} \\ 1 \text{ to } 12 \end{matrix}\right.$	$\bar{1}$, 3	JCLG	Jump load signal (this signal is used to load current-address register with the α portion of the instruction)
STC	$[\text{STB}] \cdot [\text{ISF}]$	As generated	$\alpha i\left\{\begin{matrix} i \text{ from} \\ 1 \text{ to } 12 \end{matrix}\right.$ $\alpha' i$ $\omega j\{j = 1\text{–}6$	$\bar{1}$ $\bar{1}$, $\bar{2}$ $\bar{1}$	STCG	Start clear signal
IRS	$[F1] \cdot [T_{1-18}]$	$F1$, T_{1-18}	$\omega j\{j = 1\text{–}6$ $\omega j\{j = 1\text{–}5$ $\alpha i\{i = 1\text{–}12$	$\bar{1}$ 2 $\bar{1}$,2	IRSG	Instruction register shift signal
$\alpha1$	$\{[\alpha2] \cdot [\text{IRS}] + [\overline{\text{STC}}] \cdot [\overline{\text{IRS}}] \cdot [\alpha1]\}_d$	As generated	$\alpha1$ α^*1 $\alpha'1$ SHA1	1 3 3 1	$\alpha1$	$\alpha1$ bit of instruction register
$\alpha2$	$\{[\alpha3] \cdot [\text{IRS}] + [\overline{\text{STC}}] \cdot [\overline{\text{IRS}}] \cdot [\alpha2]\}_d$	As generated	$\alpha1$ $\alpha2$ α^*2 $\alpha'2$ SHA2	2 1 3 3 1	$\alpha2$	$\alpha2$ bit of instruction register
$\alpha3$	$\{[\alpha4] \cdot [\text{IRS}] + [\overline{\text{STC}}] \cdot [\overline{\text{IRS}}] \cdot [\alpha3]\}_d$	As generated	$\alpha2$ $\alpha3$ α^*3 $\alpha'3$ SHA3	2 1 3 3 1	$\alpha3$	$\alpha3$ bit of instruction register
$\alpha4$	$\{[\alpha5] \cdot [\text{IRS}] + [\overline{\text{STC}}] \cdot [\overline{\text{IRS}}] \cdot [\alpha4]\}_d$	As generated	$\alpha3$ $\alpha4$ α^*4 $\alpha'4$ SHA4	2 1 3 3 1	$\alpha4$	$\alpha4$ bit of instruction register

TABLE A-3. CURRENT-ADDRESS AND INSTRUCTION SIGNALS (*Continued*)

Name of signal	Equation	Time period present	Destination		Generation location	Remarks
			Package	Gate		
$\alpha 5$	$\{[\alpha 6]\cdot[\mathrm{IRS}] + [\overline{\mathrm{STC}}]\cdot[\overline{\mathrm{IRS}}]\cdot[\alpha 5]\}_d$	As generated	$\alpha 4$	2	$\alpha 5$	$\alpha 5$ bit of instruction register
			$\alpha 5$	1		
			α^*5	3		
			$\alpha'5$	3		
			SHA5	1		
$\alpha 6$	$\{[\alpha 7]\cdot[\mathrm{IRS}] + [\overline{\mathrm{STC}}]\cdot[\overline{\mathrm{IRS}}]\cdot[\alpha 6]\}_d$	As generated	$\alpha 5$	2	$\alpha 6$	$\alpha 6$ bit of instruction register
			$\alpha 6$	1		
			α^*6	3		
			$\alpha'6$	3		
$\alpha 7$	$\{[\alpha 8]\cdot[\mathrm{IRS}] + [\overline{\mathrm{STC}}]\cdot[\overline{\mathrm{IRS}}]\cdot[\alpha 7]\}_d$	As generated	$\alpha 6$	2	$\alpha 7$	$\alpha 7$ bit of instruction register
			$\alpha 7$	1		
			α^*7	3		
			$\alpha'7$	3		
$\alpha 8$	$\{[\alpha 9]\cdot[\mathrm{IRS}] + [\overline{\mathrm{STC}}]\cdot[\overline{\mathrm{IRS}}]\cdot[\alpha 8]\}_d$	As generated	$\alpha 7$	2	$\alpha 8$	$\alpha 8$ bit of instruction register
			$\alpha 8$	1		
			α^*8	3		
			$\alpha'8$	3		
$\alpha 9$	$\{[\alpha 10]\cdot[\mathrm{IRS}] + [\overline{\mathrm{STC}}]\cdot[\overline{\mathrm{IRS}}]\cdot[\alpha 9]\}_d$	As generated	$\alpha 8$	2	$\alpha 9$	$\alpha 9$ bit of instruction register
			$\alpha 9$	1		
			α^*9	3		
			$\alpha'9$	3		
$\alpha 10$	$\{[\alpha 11]\cdot[\mathrm{IRS}] + [\overline{\mathrm{STC}}]\cdot[\overline{\mathrm{IRS}}]\cdot[\alpha 10]\}_d$	As generated	$\alpha 9$	2	$\alpha 10$	$\alpha 10$ bit of instruction register
			$\alpha 10$	1		
			α^*10	3		
			$\alpha'10$	3		
$\alpha 11$	$\{[\alpha 12]\cdot[\mathrm{IRS}] + [\overline{\mathrm{STC}}]\cdot[\overline{\mathrm{IRS}}]\cdot[\alpha 11]\}_d$	As generated	$\alpha 10$	2	$\alpha 11$	$\alpha 11$ bit of instruction register
			$\alpha 11$	1		
			α^*11	3		
			$\alpha'11$	3		
$\alpha 12$	$\{[\omega 1]\cdot[\mathrm{IRS}] + [\overline{\mathrm{STC}}]\cdot[\overline{\mathrm{IRS}}]\cdot[\alpha 12]\}_d$	As generated	$\alpha 11$	2	$\alpha 12$	$\alpha 12$ bit of instruction register
			$\alpha 12$	1		
			α^*12	3		
			$\alpha'12$	3		
α^*1	$[F1]\cdot[\alpha'1] + [\mathrm{IF}1]\cdot[\alpha'1] + [\overline{F1}]\cdot[\overline{\mathrm{IF}1}]\cdot[\alpha 1]$	As generated	To memory selectors		α^*1	Memory-selector address
α^*2	$[F1]\cdot[\alpha'2] + [\mathrm{IF}1]\cdot[\alpha'2] + [\overline{F1}]\cdot[\overline{\mathrm{IF}1}]\cdot[\alpha 2]$	As generated	To memory selectors		α^*2	Memory-selector address
α^*3	$[F1]\cdot[\alpha'3] + [\mathrm{IF}1]\cdot[\alpha'3] + [\overline{F1}]\cdot[\overline{\mathrm{IF}1}]\cdot[\alpha 3]$	As generated	To memory selectors		α^*3	Memory-selector address

TABLE A-3. CURRENT-ADDRESS AND INSTRUCTION SIGNALS (*Continued*)

Name of signal	Equation	Time period present	Destination		Generation location	Remarks
			Package	Gate		
α^*4	$[F1] \cdot [\alpha'4] + [\text{IF1}] \cdot [\alpha'4] + [\overline{F1}] \cdot [\overline{\text{IF1}}] \cdot [\alpha 4]$	As generated	To memory selectors		α^*4	Memory-selector address
α^*5	$[F1] \cdot [\alpha'5] + [\text{IF1}] \cdot [\alpha'5] + [\overline{F1}] \cdot [\overline{\text{IF1}}] \cdot [\alpha 5]$	As generated	To memory selectors		α^*5	Memory-selector address
α^*6	$[F1] \cdot [\alpha'6] + [\text{IF1}] \cdot [\alpha'6] + [\overline{F1}] \cdot [\overline{\text{IF1}}] \cdot [\alpha 6]$	As generated	To memory selectors		α^*6	Memory-selector address
α^*7	$[F1] \cdot [\alpha'7] + [\text{IF1}] \cdot [\alpha'7] + [\overline{F1}] \cdot [\overline{\text{IF1}}] \cdot [\alpha 7]$	As generated	To memory selectors		α^*7	Memory-selector address
α^*8	$[F1] \cdot [\alpha'8] + [\text{IF1}] \cdot [\alpha'8] + [\overline{F1}] \cdot [\overline{\text{IF1}}] \cdot [\alpha 8]$	As generated	To memory selectors		α^*8	Memory-selector address
α^*9	$[F1] \cdot [\alpha'9] + [\text{IF1}] \cdot [\alpha'9] + [\overline{\text{F1}}] \cdot [\overline{\text{IF1}}] \cdot [\alpha 9]$	As generated	To memory selectors		α^*9	Memory-selector address
α^*10	$[F1] \cdot [\alpha'10] + [\text{IF1}] \cdot [\alpha'10] + [\overline{F1}] \cdot [\overline{\text{IF1}}] \cdot [\alpha 10]$	As generated	To memory selectors		α^*10	Memory-selector address
α^*11	$[F1] \cdot [\alpha'11] + [\text{IF1}] \cdot [\alpha'11] + [\overline{F1}] \cdot [\overline{\text{IF1}}] \cdot [\alpha 11]$	As generated	To memory selectors		α^*11	Memory-selector address
α^*12	$[F1] \cdot [\alpha'12] + [\text{IF1}] \cdot [\alpha'12] + [\overline{F1}] \cdot [\overline{\text{IF1}}] \cdot [\alpha 12]$	As generated	To memory selectors		α^*12	Memory-selector address
$\alpha'1$	$[\overline{\text{1CS}}] \cdot [\overline{\text{STC}}] \cdot [\alpha'1_d] \cdot [\overline{\text{JCL}}] + [\overline{\text{STC}}] \cdot [\text{1CS}] \cdot [\overline{\alpha'1_d}] + [\text{JCL}] \cdot [\alpha 1]$	As generated	$\alpha'1\text{a}$ $\alpha'1$ α^*1	1_d $1_d, \overline{2}_d$ 1, 2	$\alpha'1$	1st bit of current-address register
$\alpha'2$	$[\overline{\alpha'1\text{a}}] \cdot [\overline{\text{STC}}]$ $[\alpha'2_d] \cdot [\overline{\text{JCL}}] + [\overline{\text{STC}}] \cdot [\alpha'1\text{a}] \cdot [\overline{\alpha'2_d}] + [\text{JCL}] \cdot [\alpha 2]$	As generated	$\alpha'2\text{a}$ $\alpha'2$ α^*2	1_d $1_d, \overline{2}_d$ 1, 2	$\alpha'2$	2d bit of current-address register

TABLE A-3. CURRENT-ADDRESS AND INSTRUCTION SIGNALS (*Continued*)

Name of signal	Equation	Time period present	Destination		Generation location	Remarks
			Package	Gate		
$\alpha'3$	$[\overline{\alpha'2a}] \cdot [\overline{STC}]$ $\cdot [\alpha'3_d] \cdot [\overline{JCL}]$ $+ [\overline{STC}] \cdot [\alpha'2a]$ $\cdot [\overline{\alpha'3_d}]$ $+ [JCL] \cdot [\alpha 3]$	As generated	$\alpha'3a$ $\alpha'3$ α^*3	1_d $1_d, \overline{2}_d$ 1, 2	$\alpha'3$	3d bit of current-address register
$\alpha'4$	$[\overline{\alpha'3a}] \cdot [\overline{STC}]$ $\cdot [\alpha'4_d] \cdot [\overline{JCL}]$ $+ [\overline{STC}] \cdot [\alpha'3a]$ $\cdot [\overline{\alpha'4_d}]$ $+ [JCL] \cdot [\alpha 4]$	As generated	$\alpha'4a$ $\alpha'4$ α^*4	1_d $1_d, \overline{2}_d$ 1, 2	$\alpha'4$	4th bit of current-address register
$\alpha'5$	$[\overline{\alpha'4a}] \cdot [\overline{STC}]$ $\cdot [\alpha'5_d] \cdot [\overline{JCL}]$ $+ [\overline{STC}] \cdot [\alpha'4a]$ $\cdot [\overline{\alpha'5_d}]$ $+ [JCL] \cdot [\alpha 5]$	As generated	$\alpha'5a$ $\alpha'5$ α^*5	1_d $1_d, \overline{2}_d$ 1, 2	$\alpha'5$	5th bit of current-address register
$\alpha'6$	$[\overline{\alpha'5a}] \cdot [\overline{STC}]$ $\cdot [\alpha'6_d] \cdot [\overline{JCL}]$ $+ [\overline{STC}] \cdot [\alpha'5a]$ $\cdot [\overline{\alpha'6_d}]$ $+ [JCL] \cdot [\alpha 6]$	As generated	$\alpha'6a$ $\alpha'6$ α^*6	1_d $1_d, \overline{2}_d$ 1, 2	$\alpha'6$	6th bit of current-address register
$\alpha'7$	$[\overline{\alpha'6a}] \cdot [\overline{STC}]$ $\cdot [\alpha'7_d] \cdot [\overline{JCL}]$ $+ [\overline{STC}] \cdot [\alpha'6a]$ $\cdot [\overline{\alpha'7_d}]$ $+ [JCL] \cdot [\alpha 7]$	As generated	$\alpha'7a$ $\alpha'7$ α^*7	1_d $1_d, \overline{2}_d$ 1, 2	$\alpha'7$	7th bit of current-address register
$\alpha'8$	$[\overline{\alpha'7a}] \cdot [\overline{STC}]$ $\cdot [\alpha'8_d] \cdot [\overline{JCL}]$ $+ [\overline{STC}] \cdot [\alpha'7a]$ $\cdot [\overline{\alpha'8_d}]$ $+ [JCL] \cdot [\alpha 8]$	As generated	$\alpha'8a$ $\alpha'8$ α^*8	1_d $1_d, \overline{2}_d$ 1, 2	$\alpha'8$	8th bit of current-address register
$\alpha'9$	$[\overline{\alpha'8a}] \cdot [\overline{STC}]$ $\cdot [\alpha'9_d] \cdot [\overline{JCL}]$ $+ [\overline{STC}] \cdot [\alpha'8a]$ $\cdot [\overline{\alpha'9_d}]$ $+ [JCL] \cdot [\alpha 9]$	As generated	$\alpha'9a$ $\alpha'9$ α^*9	1_d $1_d, \overline{2}_d$ 1, 2	$\alpha'9$	9th bit of current-address register
$\alpha'10$	$[\overline{\alpha'9a}] \cdot [\overline{STC}]$ $\cdot [\alpha'10_d] \cdot [\overline{JCL}]$ $+ [\overline{STC}] \cdot [\alpha'9a]$ $\cdot [\overline{\alpha'10_d}]$ $+ [JCL] \cdot [\alpha 10]$	As generated	$\alpha'10a$ $\alpha'10$ α^*10	1_d $1_d, \overline{2}_d$ 1, 2	$\alpha'10$	10th bit of current-address register
$\alpha'11$	$[\overline{\alpha'10a}] \cdot [\overline{STC}]$ $\cdot [\alpha'11_d] \cdot [\overline{JCL}]$ $+ [\overline{STC}] \cdot [\alpha'10a]$ $\cdot [\overline{\alpha'11_d}]$ $+ [JCL] \cdot [\alpha 11]$	As generated	$\alpha'11a$ $\alpha'11$ α^*11	1_d $1_d, \overline{2}_d$ 1, 2	$\alpha'11$	11th bit of current-address register

Table A-3. Current-address and Instruction Signals (*Continued*)

Name of signal	Equation	Time period present	Destination		Generation location	Remarks
			Package	Gate		
$\alpha'12$	$[\overline{\alpha'11a}] \cdot [\overline{STC}]$ $\cdot [\alpha'12_d] \cdot [\overline{JCL}]$ $+ [\overline{STC}] \cdot [\alpha'11a]$ $\cdot [\overline{\alpha'12}_d]$ $+ [JCL] \cdot [\alpha 12]$	As generated	$\alpha'12a$ $\alpha'12$ α^*12	1_d 1_d, $\overline{2}_d$ 1, 2	$\alpha'12$	12th bit of current-address register
$\alpha'1a$	$[\alpha'1_d] \cdot [1CS]$	As generated	$\alpha'2$ $\alpha'2a$	$\overline{1}$, 2 1	$\alpha'1a$	1st auxiliary bit
$\alpha'2a$	$[\alpha'2_d] \cdot [\alpha'1a]$	As generated	$\alpha'3$ $\alpha'3a$	$\overline{1}$, 2 1	$\alpha'2a$	2d auxiliary bit
$\alpha'3a$	$[\alpha'3_d] \cdot [\alpha'2a]$	As generated	$\alpha'4$ $\alpha'4a$	$\overline{1}$, 2 1	$\alpha'3a$	3d auxiliary bit
$\alpha'4a$	$[\alpha'4_d] \cdot [\alpha'3a]$	As generated	$\alpha'5$ $\alpha'5a$	$\overline{1}$, 2 1	$\alpha'4a$	4th auxiliary bit
$\alpha'5a$	$[\alpha'5_d] \cdot [\alpha'4a]$	As generated	$\alpha'6$ $\alpha'6a$	$\overline{1}$, 2 1	$\alpha'5a$	5th auxiliary bit
$\alpha'6a$	$[\alpha'6_d] \cdot [\alpha'5a]$	As generated	$\alpha'7$ $\alpha'7a$	$\overline{1}$, 2 1	$\alpha'6a$	6th auxiliary bit
$\alpha'7a$	$[\alpha'7_d] \cdot [\alpha'6a]$	As generated	$\alpha'8$ $\alpha'8a$	$\overline{1}$, 2 1	$\alpha'7a$	7th auxiliary bit
$\alpha'8a$	$[\alpha'8_d] \cdot [\alpha'7a]$	As generated	$\alpha'9$ $\alpha'9a$	$\overline{1}$, 2 1	$\alpha'8a$	8th auxiliary bit
$\alpha'9a$	$[\alpha'9_d] \cdot [\alpha'8a]$	As generated	$\alpha'10$ $\alpha'10a$	$\overline{1}$, 2 1	$\alpha'9a$	9th auxiliary bit
$\alpha'10a$	$[\alpha'10_d] \cdot [\alpha'9a]$	As generated	$\alpha'11$ $\alpha'11a$	$\overline{1}$, 2 1	$\alpha'10a$	10th auxiliary bit
$\alpha'11a$	$[\alpha'11_d] \cdot [\alpha'10a]$	As generated	$\alpha'12$ $\alpha'12a$	$\overline{1}$, 2 1	$\alpha'11a$	11th auxiliary bit
$\omega 1$	$\{[\omega 2] \cdot [IRS]$ $+ [\omega 1] \cdot [\overline{IRS}]$ $\cdot [\overline{STC}]\}_d$	As generated	$\alpha 12$ $\omega 1$ To instruction decoder	2 1	$\omega 1$	1st bit of instruction register
$\omega 2$	$\{[\omega 3] \cdot [IRS]$ $+ [\omega 2] \cdot [\overline{IRS}]$ $\cdot [\overline{STC}]\}_d$	As generated	$\omega 1$ $\omega 2$ To instruction decoder	2 1	$\omega 2$	2d bit of instruction register
$\omega 3$	$\{[\omega 4] \cdot [IRS]$ $+ [\omega 3] \cdot [\overline{IRS}]$ $\cdot [\overline{STC}]\}_d$	As generated	$\omega 2$ $\omega 3$ To instruction decoder	2 1	$\omega 3$	3d bit of instruction register
$\omega 4$	$\{[\omega 5] \cdot [IRS]$ $+ [\omega 4] \cdot [\overline{IRS}]$ $\cdot [\overline{STC}]\}_d$	As generated	$\omega 3$ $\omega 4$ To instruction decoder	2 1	$\omega 4$	4th bit of instruction register

TABLE A-3. CURRENT-ADDRESS AND INSTRUCTION SIGNALS (*Continued*)

Name of signal	Equation	Time period present	Destination		Generation location	Remarks
			Package	Gate		
$\omega 5$	$\{[\omega 6] \cdot [\text{IRS}]$ $+ [\omega 5] \cdot [\overline{\text{IRS}}]$ $\cdot [\overline{\text{STC}}]\}_d$	As generated	$\omega 4$ $\omega 5$ To instruction decoder	2 1	$\omega 5$	5th bit of instruction register
$\omega 6$	$\{[F1] \cdot [(\alpha)]$ $+ [\omega 6] \cdot [\overline{\text{IRS}}]$ $\cdot [\overline{\text{STC}}]\}_d$	As generated	$\omega 5$ $\omega 6$ To instruction decoder	2 1	$\omega 6$	6th bit of instruction register

TABLE A-4. CONTROL-UNIT SIGNALS, CHANNEL SELECTOR, SECTOR SELECTOR (See Fig. 18-15)

Name of signal	Equation	Time period present	Destination		Generation location	Remarks
			Package	Gate		
B^*	$[\text{BUF}] \cdot [T^1_{0-18}]$ $\cdot [\text{FOI}] + [B]$ $\cdot [\text{TRS}] \cdot [T_{1-18}]$ $\cdot [F3] + [\text{SOB}]$ $\cdot [\text{TRS}] \cdot [T_0] \cdot [F3]$	FOI, T^1_{0-18}	To memory		GMIB	One word from buffer or accumulator being transferred to memory
COI	$\{[C1] \cdot [\alpha^*1]$ $+ [\overline{C1}] \cdot [\overline{\alpha^*1}]\}$ $\cdot \{[C2] \cdot [\alpha^*2]$ $+ [\overline{C2}] \cdot [\overline{\alpha^*2}]\}$ $\cdot \{[C3] \cdot [\alpha^*3]$ $+ [\overline{C3}] \cdot [\overline{\alpha^*3}]\}$ $\cdot \{[C4] \cdot [\alpha^*4]$ $+ [\overline{C4}] \cdot [\overline{\alpha^*4}]\}$ $\cdot \{[C5] \cdot [\alpha^*5]$ $+ [\overline{C5}] \cdot [\overline{\alpha^*5}]\}$ $\cdot \{[C6] \cdot [\alpha^*6]$ $+ [\overline{C6}] \cdot [\overline{\alpha^*6}]\}$ $\cdot \{[C7] \cdot [\alpha^*7]$ $+ [\overline{C7}] \cdot [\overline{\alpha^*7}]\} \cdot [T_0]$	T_0	F1SG F2SG F3SG FOIG FOOG IF1G IF2G IF3G IFIG IFOG	1 1 3 1 1 $\overline{3}$ $\overline{5}$ $\overline{2}$ $\overline{2}$ $\overline{2}$	GCOI	Coincidence signal from memory sector selector
(α)	$[\alpha] \cdot [T^1_{0-18}] \cdot \{[F1]$ $+ [F2] + [\text{FOO}]\}$	$F1$, T^1_{0-18} $F2$, T^1_{0-18} FOO, T_{0-18}	ASAI BUI S1IR SOAH $\omega 6$	1, 3 2 1 1 2	From memory through GMOα	Contents of address α of memory

TABLE A-5. UNIT-TIME-INTERVAL COUNTER SIGNALS
(See Fig. 18-16)

Name of signal	Equation	Time period present	Destination		Generation location	Remarks
			Package	Gate		
cp	Drum clock-pulse signals	All time	To every package	To every *and* gate	Drum clock track	Clock-pulse signal
DSS	Drum synchronizer signal	See Sec. 18-7	UTSG	1	Drum synchronizer track	Drum synchronizer signal
UTS	$[cp] \cdot [\text{DSS}] + [cp] \cdot [\text{UTS}_d]$	As generated	UTSG UC1aS UC1S UC2aS UC4aS UC8aS ISFG	2 1 $1, \bar{2}$ 1 1 1 $1, \bar{1}_d$	UTSG	Unit time signal; this signal acts as the unit-time-count signal

TABLE A-6. OPERATIONS PHASE-GENERATOR SIGNALS
(See Figs. 18-17 and 18-18)

Name of signal	Equation	Time period present	Destination		Generation location	Remarks
			Package	Gate		
BSP	$[\text{SIU}] \cdot [\overline{\text{EWI}}] \cdot [\text{FBI}] + [\text{SOU}] \cdot [\overline{\text{EWO}}] \cdot [\text{FBO}] + [T^1_{0-18}] \cdot [\text{FOI}] + [T^1_{0-18}] \cdot [\text{FOO}]$	T^1_{0-18}	BU1 BU2 through BU19	$\bar{3}$ $1, \bar{2}$	BSPG	Buffer shift pulse signal
BUF	$[\text{BU19}_d] \cdot [\overline{\text{BSP}}] + [\text{BU18}_d] \cdot [\text{BSP}]$	FOI or FBO	GMIB to output unit or to drum	1	BU19	Out of buffer
EWI	$[\text{EWI}'] \cdot [\bar{T}_0] \cdot [\overline{\text{IF1}}_d] + [\text{EWI}_d] \cdot [\overline{\text{IF1}}_d]$	$\bar{T}_0 \cdot \text{FBI}$	BSPG EWIG FBIG IFIG	$\bar{1}$ 2_d $\bar{4}$ 1	EWIG	External word-in signal indicating that a complete word has been sent into buffer from input unit

TABLE A-6. OPERATIONS PHASE-GENERATOR SIGNALS (*Continued*)

Name of signal	Equation	Time period present	Destination		Gener- ation location	Remarks
			Package	Gate		
EWO	$[\mathrm{EWO}'] \cdot [\bar{T}_0]$ $\cdot [\overline{\mathrm{IF1}}_d]$ $+ [\mathrm{EWO}_d]$ $\cdot [\overline{\mathrm{IF1}}_d]$	$\bar{T}_0 \cdot \mathrm{FBO}$	BSPG	$\bar{4}$	EWOG	External word-out signal indicating that a complete word has been read out of buffer into output unit
			EWOG	2_d		
			FBOG	$\bar{3}$		
			I'F1G	1		
$F1$	$[\mathrm{COI}] \cdot [\overline{\mathrm{SPB}}]$ $\cdot [\mathrm{IF1}_d]$ $+ [\bar{T}_0] \cdot [F1_d]$	$F1$	1CSG	1	F1SG	Phase 1 signal
			F1SG	2_d		
			F3SG	1_d, 2_d		
			FBIG	1_d		
			GJCS	1, 2		
			GMOα	$\bar{3}$		
			IFOG	1_d		
			IF2G	1_d, 2_d, 3_d, 4_d		
			IF3G	1_d		
			IRSG	1		
			ISFG	2_d		
			I*F1G	1_d, 2_d		
			JCLG	1		
			α*1 through α*12	1, $\bar{3}$		
			ω6	2		
$F2$	$[\mathrm{COI}] \cdot [\mathrm{IF2}_d]$ $+ [\overline{\mathrm{OEP}}_d] \cdot [F2_d]$	$F2$	ASAI	1, 2, 3, 4	F2SG	Phase 2 signal
			ASBI	1, 3, 4, 5		
			ASCG	1, 2, 3, 4, 5		
			ASPS	1, 5		
			ASSG	1, 2, 3		
			CIAS	1, 2		
			F2SG	2_d		
			GOEP	1, 2		
			GMOα	$\bar{3}$		
			ISPS	1, 2, 3, 4		
			ITCF	2, 3		
			I*F1G	3_d		
			MCCL	$\bar{1}_d$, 2_d		
			MUSG	1, 2, 3		
			PQSG	1, 2, 3		
			S1AR	2		
			S1ER	1, 2, 3, 4		

TABLE A-6. OPERATIONS PHASE-GENERATOR SIGNALS (*Continued*)

Name of signal	Equation	Time period present	Destination		Generation location	Remarks
			Package	Gate		
$F2$			S18ER S1IR SOAH SOBH SPDG	1, 3, 4 1, 2 1 1, 2 1, 2, 3, 4, 5		
$F3$	$[\text{SHS}] \cdot [T_0] \cdot [F1_d] + [\text{CLS}] \cdot [F1_d] \cdot [T_0] + [\text{COI}] \cdot [\text{IF3}_d] + [\overline{\text{OEP}}_d] \cdot [F3_d]$	$F3$	ASBI ASPS F3SG GMIB GOEP GSCC I*F1G MCCL SOBH S5AR	2 2, 3, 4 4_d 2, 3 3, 4 2 4_d $\overline{1}_d$, 3_d 4 1, 2	F3SG	Phase 3 signal
FBI	$[\text{00D}] \cdot [T_0] \cdot [F1_d] + [\text{STB}] \cdot [T_0] \cdot [\text{ISF}_d] + [\bar{T}_0] \cdot [\text{FBI}_d] + [\overline{\text{EWI}}] \cdot [\text{FBI}_d]$	FBI	BSPG BU1 FBIG RIBG STBG	1 1 3_d, 4_d 1 $\overline{2}_d$	FBIG	Phase FBI signal; during this phase one word from input unit is read into buffer
FBO	$[T_0] \cdot [\text{FOO}_d] + [\bar{T}_0] \cdot [\text{FBO}_d] + [\overline{\text{EWO}}] \cdot [\text{FBO}_d]$	FBO	BSPG FBOG ROBG	4 2_d, 3_d 1	FBOG	Phase FBO signal; during this phase one word from buffer is read into output unit
FOI	$[\text{COI}] \cdot [\text{IFI}_d] + [\bar{T}_0] \cdot [\text{FOI}_d]$	FOI	BSPG FOIG GMIB I′F1G	2 2_d 1 1_d	FOIG	Phase FOI signal; during this phase one word from buffer is read into drum
FOO	$[\text{COI}] \cdot [\text{IFO}_d] + [\bar{T}_0] \cdot [\text{FOO}_d]$	FOO	BSPG BU1 FBOG FOOG GMOα	3 2 1_d 2_d $\overline{3}$	FOOG	Phase FOO signal; during this phase one word is read from drum into buffer
IFO	$[21D] \cdot [T_0] \cdot [F1_d] + [\overline{\text{COI}}] \cdot [\text{IFO}_d]$	IFO	FOOG IFOG	1_d 2_d	IFOG	Phase IFO signal; this is an idle phase; drum section is being selected

TABLE A-6. OPERATIONS PHASE-GENERATOR SIGNALS (*Continued*)

Name of signal	Equation	Time period present	Destination		Generation location	Remarks
			Package	Gate		
IF1	$[\text{I}'\text{F1}] + [\text{I*F1}] + [\overline{\text{COI}}] \cdot [\text{IF1}_d] \cdot [\text{ISF}]$	IF1	EWOG F1SG IF1G ISFG RUBG α*1 through α*12	$\overline{1}_d$, $\overline{2}_d$ 1_d 3_d 3_d $\overline{2}_d$ 2, $\overline{3}$	IF1G	Phase IF1 signal; idle phase signal
I′F1	$[\text{EWO}] \cdot [T_0] + [\overline{\text{FOI}}_d] \cdot [T_0]$	IF1	IF1G	2	I′F1G	Phase I′F1 signal; auxiliary idle phase signal
I*F1	$[43D] \cdot [F1_d] + [44D] \cdot [F1_d] + [\text{OEP}_d] \cdot [F2_d] + [\text{OEP}_d] \cdot [F3_d] + [\text{RUB}] \cdot [T_0] \cdot [\text{ISF}_d]$	IF1	IF1G	1	I*F1G	Phase I*F1 signal; auxiliary idle phase signal
IF2	$[\text{ADD}] \cdot [T_0] \cdot [F1_d] + [\text{SUB}] \cdot [T_0] \cdot [F1_d] + [\text{LDS}] \cdot [T_0] \cdot [F1_d] + [\text{PQS}] \cdot [T_0] \cdot [F1_d] + [\overline{\text{COI}}] \cdot [\text{IF2}_d]$	IF2	F2SG IF2G	1_d 5_d	IF2G	Phase IF2 signal; idle phase signal
IF3	$[\text{TRS}] \cdot [T_0] \cdot [F1_d] + [\overline{\text{COI}}] \cdot [\text{IF3}_d]$	IF3	F3SG IF3G	3_d 2_d	IF3G	Phase IF3 signal; idle phase signal
IFI	$[\text{EWI}] \cdot [T_0] + [\overline{\text{COI}}] \cdot [\text{IFI}_d]$	IFI	EWIG FOIG IFIG	$\overline{1}_d$, $\overline{2}_d$ 1_d 2_d	IFIG	Phase IFI signal; idle phase signal
ISF	$[\text{UTS}] \cdot [\overline{\text{UTS}}_d] + [60D] \cdot [T_0] \cdot [F1_d] + [\text{SPB}] \cdot [T_0] \cdot [\text{IF1}_d] + [\bar{T}_0] \cdot [\text{ISF}_d] + [\overline{\text{STB}}] \cdot [\overline{\text{RUB}}] \cdot [\text{ISF}_d]$	ISF	FBIG ISFG I*FIG RUBG SPBG STBG STCG IF1G	2_d 4_d, 5_d 5_d 1 $\overline{1}_d$, $\overline{2}_d$ 1 1 3	ISFG	Phase ISF signal; this is the stopped idle phase signal
RIB	$[T_0] \cdot [\text{FBI}]$	T_0	Input-unit synchronizer		RIBG	Signal to input-unit synchronizer, allowing input unit to take over control

TABLE A-6. OPERATIONS PHASE-GENERATOR SIGNALS (*Continued*)

Name of signal	Equation	Time period present	Destination		Generation location	Remarks
			Package	Gate		
ROB	$[T_0] \cdot [\mathrm{FBO}]$	T_0	Output unit synchronizer		ROBG	Signal to output-unit synchronizer, allowing output unit to take over control
RUB	$[\mathrm{RUB}'] \cdot [\bar{T}_0] \cdot [\mathrm{ISF}] + [\mathrm{RUB}_d] \cdot [\overline{\mathrm{IF1}}_d]$	As generated	ISFG I*F1G RUBG	$\bar{5}$ 5 2_d	RUBG	Run-button signal
SPB	$[\mathrm{SPB}'] \cdot [\bar{T}_0] \cdot [\overline{\mathrm{ISF}}_d] + [\mathrm{SPB}_d] \cdot [\overline{\mathrm{ISF}}_d]$	As generated	F1SG ISFG SPBG	$\bar{1}$ 3 2_d	SPBG	Stop-button signal
STB	$[\mathrm{STB}'] \cdot [\bar{T}_0] \cdot [\mathrm{ISF}] + [\overline{\mathrm{FBI}}_d] \cdot [\mathrm{STB}_d]$	As generated	FBIG ISFG STBG STCG	2 $\bar{5}$ 2_d 1	STBG	Start-button signal

TABLE A-7. SIGNALS FROM LINE SYNCHRONIZERS (EXTERNAL EQUIPMENT)

Name of signal	Time period present	Destination		Generation location	Remarks
		Package	Gate		
EWO′	As generated	EWOG	1	Buffer read-out synchronizer	External signal from buffer, indicating a word has been completely read out of buffer
EWI′	As generated	EWIG	1	Buffer read-in synchronizer	External signal from buffer, indicating a word has been completely read into buffer
IEQ	As generated	BU1	1	Input-unit synchronizer	This signal is the *word* from the input unit being read into buffer
RUB′	As generated	RUBG	1	Run-button synchronizer	External run-button signal
SIU	As generated	BSPG	1	Input-unit synchronizer	Shift-pulse signal from input unit
SOU	As generated	BSPG	4	Output-unit synchronizer	Shift-pulse signal from output unit
SPB′	As generated	SPBG	1	Stop-button synchronizer	External stop-button signal
STB′	As generated	STBG	1	Start-button synchronizer	External start-button signal

NAME INDEX

SUBJECT INDEX