PLANNING A COMPUTER SYSTEM

PROJECT STRETCH

PLANNING A

CONTRIBUTORS

Richard S. Ballance
Robert W. Bemer
Gerrit A. Blaauw
Erich Bloch
Frederick P. Brooks, Jr.
Werner Buchholz
Sullivan G. Campbell
John Cocke
Edgar F. Codd
Paul S. Herwitz
Harwood G. Kolsky
Edward S. Lowry
Elizabeth McDonough
James H. Pomerene
Casper A. Scalzi

COMPUTER SYSTEM

PROJECT STRETCH

Edited by
WERNER BUCHHOLZ
SYSTEMS CONSULTANT
CORPORATE STAFF, RESEARCH AND ENGINEERING
INTERNATIONAL BUSINESS MACHINES CORPORATION

New York Toronto London 1962
McGRAW-HILL BOOK COMPANY, INC.

PLANNING A COMPUTER SYSTEM

Library of Congress Catalog Card Number 61-10466

THE MAPLE PRESS COMPANY, YORK, PA. 08720

FOREWORD

The electronic computer has greatly contributed to scientific research; it has reduced costs, shortened time scales, and opened new areas of investigation. Increased use of computers, in turn, has created a need for better computers. What is desired most often is a general-purpose design with the best achievable memory capacity, speed, and reliability.

User experience has shown the need for considering more than these fundamental properties in the design of a new computer. Unlike earlier machines, whose capabilities were mainly functions of the properties of individual components and units and not to any marked extent of their organization or the user's requirements, the Stretch computer is based on a comprehensive joint planning effort involving both users and designers. Their combined experience brought in many new considerations. The term *general purpose* was given a broader definition in Stretch. Areas of special concern included the vocabulary of the computer, parallel operation for greater speed and efficiency, error detection and correction, and recovery from errors and other exceptional events.

The design phase for a new-generation computer is always a difficult one. The potential user cannot predict accurately how the new tool will be used or what new areas of research will open up. The designers have to work with components for which such important data as how these components behave en masse are lacking. The Stretch project, in design as well as construction, has been successfully concluded. The degree of success, however, can only be ascertained as experience in using Stretch is accumulated.

This book forms a record of what is probably the first really comprehensive design effort for a new computer. It was written and edited by a very competent group from the technical staff of the IBM Corporation, including most of the principal designers of Stretch.

There is no doubt that still better computers will be needed. Although the Stretch computer is now solving problems that could not be solved a few months ago, many problems are known to exist for which even Stretch is inadequate. This book will be invaluable as a guide and reference source for computer development in the future.

Bengt Carlson

Los Alamos Scientific Laboratory
September 1961

PREFACE

Planning a computer system ideally consists of a continuous spectrum of activity, ranging from theoretically analyzing the problems to be solved to evaluating the technology to be used for the components. When dealing with an electronic digital computer of more than modest size that is intended to be used for fairly complex applications, one is forced to split the planning spectrum into arbitrary segments, each segment being developed with due regard for its neighbors. This book is mainly concerned with that segment that has to do with the selection of an instruction set and related functional characteristics of a computer. Except for cost and speed, these are the characteristics that do most to distinguish one computer from the next.

This book is about the planning of a specific computer. Being specific has both advantages and drawbacks. On one hand, the book reflects the thoughts of one group, not the entire state of the art. It cannot be a compendium of all the ideas, features, and approaches that have proved interesting and useful in various computers. On the other hand, concentration on one design serves to crystallize the concepts in a way that would be difficult to do with a hypothetical computer designed for the sake of exposition. Moreover, a specific computer represents compromises in bringing diverse and complex considerations together into a coherent working entity, and these practical compromises are instructive in themselves.

Although the discussion is in terms of a specific computer, the concepts discussed are quite general. The computer chosen is the IBM 7030. It is a recently developed computer incorporating many of the latest advances in machine organization, and a number of these advances are original or greatly improved over earlier versions. It is also a very large and very fast computer. There is an advantage in choosing such a large computer for examination, for it is practical to include quite a rich vocabulary in large computers, and this affords an opportunity to examine features which may not all be so readily incorporated in a single computer of smaller size. The 7030, in particular, combines computing and data-processing facilities that were previously available only in separate computers. Thus a large computer may serve as a model from which to select or adapt features for use in a smaller computer.

The 7030 computer was the outcome of *Project Stretch*, an IBM research and development project aimed at a major advance in computer technology and organization. To achieve a substantially improved computer organization required more than a mere compilation of the best features in existing machines and of new features already known. In the hope of stimulating ideas for substantial improvements it was decided to explore very thoroughly the basic structure of computers. Several of the participants in these studies published papers, from time to time, on computer organization in general and on particular conclusions drawn for Project Stretch. This book consists partly of such material, updated and edited for continuity. Much previously unpublished material has been added to fill in major gaps.

The book is intended to complement the reference manual for the 7030,[1] although enough of the details of the 7030 are summarized in the text or in the Appendix that the 7030 Reference Manual is not required for understanding the material in this book. Where the manual recites in as much detail as possible *what* the system does, this book is aimed at shedding light on *how* it is done and *why* the system was designed the way it is, as well as describing some alternative courses that were examined and rejected.

The book does not attempt to deal adequately with details of the design and construction of the computer and its components, since these might well fill another volume. Nor does it cover the programming techniques used in the extensive compiling and supervisory programs written for the system.

The book is aimed at a reader who already has a reasonably good knowledge of how a stored-program computer is organized and programmed. It may also serve as an advanced text to follow an elementary course on digital computers.

Contents of Book

Chapter 1 is a short history of Project Stretch. Chapter 2 outlines the philosophy that guided the design of the system. It emphasizes the need for a consistent point of view among those responsible for the basic plan of as complex a system as this computer.

A summary of the system in narrative form is given in Chap. 3. This is intended to give the reader a fairly complete picture of the forest before

[1] "Reference Manual, 7030 Data Processing System," IBM Data Processing Division, White Plains, N.Y.

he looks at the trees. Most of the material in this chapter is covered again in detail in later chapters.

Chapter 4 discusses different classes of data and the need for different ways of specifying each class. Chapter 5 gives the reasons for designing what is basically a binary structure, although there are provisions for both binary and decimal arithmetic.

Chapter 6 considers the choice of a new character set and code for the 7030, which provides 120 characters, including many not available before, such as those of the lower-case alphabet. It may be noted that the 7030 system is quite flexible with regard to character sets and is not tied to the set described here. One reason for writing this chapter is that the reasoning is pertinent to current industry-wide code standardization efforts, and it may be found useful as input to these important deliberations.

Chapter 7 covers the extensive variable-field-length features of the 7030, which are used for fixed-point binary and decimal arithmetic, for alphanumeric processing, and for Boolean logic. Chapter 8 describes the floating-point-arithmetic operations, which deserve much more than the routine treatment they so often receive if numerous pitfalls are to be avoided. Between them, Chaps. 7 and 8 bridge the traditionally separate domains of the "business" and "scientific" computers.

In Chap. 9, the reason for the rather complex instruction formats used in the 7030 is explained. Chapter 10 deals with various methods available to the programmer for specifying the logical sequence of instructions. (This should be distinguished from the internal rearrangement of this sequence to achieve overlapped operation, as described in Chaps. 14 and 15.) Included in Chap. 10 are original techniques for program interruption and for executing instructions outside the current instruction sequence. This *execute* feature, incidentally, is one of several examples where a new technique developed originally on Project Stretch came to light first in another IBM computer (here the 709) that happened to be built on an earlier schedule.

Chapter 11 covers a thorough investigation of indexing, which resulted in the development of the control-word technique for processing records and for controlling program loops. A general method for controlling input-output units that is independent of the precise nature of the device is discussed in Chap. 12.

Chapter 13 gives an introduction to the fairly recent subject of *multiprogramming*, which is the simultaneous execution of several problem programs. It shows how the design of the 7030 was heavily influenced by the desire to exploit multiprogramming for more efficient utilization of the computer and for better man-machine communication.

The next three chapters give a brief survey of the construction of major parts of the system to round out the picture. Chapter 14 deals with the various parts of the central processing unit, the circuits, and the method of construction. One part of the central processing unit, which has been called the *look-ahead*, receives more detailed treatment in Chap. 15, since it represents a major departure from the design of earlier computers. Chapter 16 explains the input-output *exchange* which controls the independent operation of a number of input-output channels.

Chapter 17 describes the IBM 7951 Processing Unit, which extends but is not a part of the Stretch system, having been developed under a separate contract. The 7951 introduces a completely new concept of nonarithmetical processing, which is a much more powerful tool for operating on nonnumerical data than previous techniques. The complete system includes an entire 7030 computer, all of whose facilities are available for more conventional procedures. It seemed appropriate to include in this book at least a brief account of a contemporary project related to Stretch.

Acknowledgments

As part of a contractual agreement with the Los Alamos Scientific Laboratory, the first recipient of a Stretch computer, a joint Los Alamos-IBM mathematical planning group was set up to coordinate, advise, and assist in the planning stage. Project Stretch owes a great deal to the many invaluable contributions of the Los Alamos members, a group which collectively represents as great a wealth of practical experience in all phases of the application of large computers to large mathematical problems as can be found anywhere. The Los Alamos group, ably led by B. G. Carlson, included R. M. Frank, M. Goldstein, H. G. Kolsky (now with IBM), R. B. Lazarus, E. A. Voorhees, M. B. Wells, D. F. Woods, and W. J. Worlton. A second group was formed later to work with IBM Applied Programming personnel on creating programming systems for Stretch.

It is impossible to acknowledge individually the work of all the IBM personnel who have made significant contributions to the material in this book. All except one of the fifteen coauthors of the chapters participated directly in Project Stretch.

Some other individuals should be mentioned in connection with specific chapters. The character set reported in Chap. 6 was developed jointly by a group including E. G. Law, H. J. Smith, and F. A. Williams. W. Wolensky contributed substantially to the variable-field-length system outlined in Chap. 7. A great deal of the credit for the floating-point system of Chap. 8 should go to D. W. Sweeney. Much of the early development of the control-word concept covered in Chap. 11 was stimulated by discussion with G. M. Amdahl, E. M. Boehm, J. E. Griffith, and

R. A. Rahenkamp. J. D. Calvert was in charge of the design of the input-output control system described in Chap. 12. Engineering responsibility for major units described in Chap. 14 rested with R. T. Blosk (instruction unit), J. F. Dirac (look-ahead unit), J. A. Hipp and O. L. MacSorley (arithmetic units), and L. O. Ulfsparre (memory bus unit), while R. E. Merwin and E. Bloch (the author of the chapter) had over-all engineering direction for Project Stretch. The description of the input-output exchange in Chap. 16 was based in part on an oral paper by H. K. Wild,[1] who was in charge of the design of this unit and is responsible for much of its logic. T. C. Chen contributed material for programming examples shown in the Appendix.

Important work was contributed during the early stages of the project by several whose names have not been mentioned so far, including J. W. Backus, N. P. Edwards, P. E. Fox, L. P. Hunter, J. C. Logue, and B. L. Sarahan.

The editor wishes to acknowledge particularly the encouragement and advice he received from S. W. Dunwell, who headed Project Stretch from beginning to end.

Werner Buchholz

[1] H. K. Wild, The Organization of the Input-Output System of the Stretch Computer, presented at the Auto-Math Sessions, Paris, June, 1959.

CONTENTS

General Discussion

Floating-point Features of the 7030

PLANNING A COMPUTER SYSTEM

PROJECT STRETCH

Chapter 1

PROJECT STRETCH

by W. Buchholz

The computer that is discussed in this book was developed by the International Business Machines Corporation at Poughkeepsie, N.Y., under Project Stretch. The project started toward the end of 1954. By then IBM was producing several stored-program digital computers: the IBM 650, a medium-sized computer; the IBM 704, a large-scale computer primarily for scientific applications; and the IBM 705, a large-scale computer primarily for business data processing. The 704 and 705 had already superseded the 701 and 702, which were IBM's first commercial entries into the large-computer field. Since the entire field was still new, there had been little experience on which to base the design of these machines, but by 1954 such experience was building up rapidly. This experience showed that the early computers were basically sound and eminently usable, but it was also obvious that many of the early decisions would have been made quite differently in 1954 and that many improvements had become possible.

At the same time, solid-state components were rapidly being developed to the point where it appeared practical to produce computers entirely out of transistors and diodes, together with magnetic core memories. A computer made only of solid-state components promised to surpass its vacuum-tube predecessors with higher reliability, lower power consumption, smaller size, lower cost made possible by automatic assembly, and eventually greater speed. The imminence of new technology, together with the knowledge of shortcomings in existing designs, gave impetus to a new computer project.

In 1955 the project was directed more specifically toward achieving, on very large mathematical computing problems, the highest performance possible within certain limits of time and resources. If mostly on-the-shelf components were used, a factor-of-10 improvement over the IBM 704, the fastest computer then in production, appeared feasible. Although this level of improvement would have been a respectable achievement, it was rejected as not being a large enough step. Instead, an over-all performance of 100 times that of the 704 was set as the target.

The purpose of setting so ambitious a goal was to stimulate innovation in all aspects of computer design. The technology available in 1955 was clearly not adequate for the task. New transistors, new cores, new logical features, and new manufacturing techniques were needed, which, although they did not yet exist, were known to be at least physically possible. Even though the goal might not be reached in all respects, the resultant machine would set a new standard of performance and make available the best technology that could be achieved by straining the technical resources of the laboratory. Hence the name *Project Stretch.*

The need for a computer of the power envisioned was clear. A number of organizations in the country had many important computing problems for which the fastest existing computers were completely inadequate, and some had other problems for which even the projected computer of 100 times the speed of the existing ones would not be enough. Negotiations with such organizations resulted in a contract with the U.S. Atomic Energy Commission in late 1956 to build a Stretch system for the Los Alamos Scientific Laboratory.

The early design objectives were described in 1956[1] in terms of certain technological and organizational goals:

Performance

An over-all performance level of 100 times that of the fastest machines then in existence was the general objective. (It has since become evident that speed comparisons of widely different machines are very difficult to make, so that it is hard to ascertain how well this target has been achieved. Using the IBM 704 as the reference point, and assuming problems that can easily be fitted to the shorter word size, the smaller memory, and the more limited repertoire of the 704, the speed ratio for the computer actually built falls below the target of 100. On the other hand, for large problems which strain the facilities of the 704 in one or more ways, the ratio may exceed 100.)

Reliability

Solid-state components promised the much higher reliability needed for satisfactory operation of a necessarily complex machine.

Checking

Extensive automatic checking facilities were intended to detect any errors that occurred and to locate faults within narrow limits. Storage devices were also to be equipped with error-correction facilities to ensure

[1] S. W. Dunwell, Design Objectives for the IBM Stretch Computer, *Proc. Eastern Joint Computer Conf.*, December, 1956, pp. 20–22.

that data could be recovered in spite of an occasional error. The purpose was again to increase performance by reducing the rerun time often needed in unchecked computers.

Generality

To broaden the area of application of the system and to increase the effectiveness of the system on secondary but time-consuming portions of any single job, it was felt desirable to include in one system the best features of scientific, data-processing, and real-time control computers. Furthermore, the input-output controls were to be sufficiently general to permit considerable future expansion and attachment of new input-output devices.

High-speed Arithmetic

A high-speed parallel arithmetic unit was to execute floating-point additions in 0.8 microsecond and multiplications in 1.4 microseconds. (The actual speeds are not as high, see Chap. 14.) This unit would not be responsible for instruction preparation, indexing, and operand fetching, which were to be carried out by other sections of the system whose operation would overlap the arithmetic.

Editing

A separate serial computer unit with independent instruction sequencing was visualized to edit input and output data of variable length in a highly flexible manner. (It was later found desirable to combine the serial and parallel units to a greater degree, so that they are no longer independent, but the functional capability of both units was retained.)

Memory

The main memory was to have a cycle time of only 2 microseconds. (All but the early production memories will indeed be capable of working at 2.0 μsec, but computer timing dictates a slightly longer cycle of 2.1 μsec.) The capacity was to be 8,192 (later raised to 16,384) words per unit.[1]

Input-Output Exchange

A unit resembling somewhat a telephone exchange was to provide simultaneous operation of all kinds of input-output, storage, and data-transmission devices.

[1] A second set of faster, though smaller, memory units was also postulated, but it was later omitted because the larger units were found to give about the same over-all performance with a greater capacity per unit cost. These units are still used, however, to satisfy more specialized requirements of the 7951 Processing Unit described in Chap. 17.

High-speed Magnetic Disks

Magnetic disk units were to be used for external storage to supplement the internal memory. The target was a capacity of 1 (later raised to 2) million words with a transfer rate of 250,000 (later lowered to 125,000) words per second. These disk units permit a very high data flow rate (even at the lower figure) on problems for which data cannot be contained in memory.

As the understanding of the task deepened, this tentative plan was modified in many ways. The functional characteristics of the actual computer were developed in the years 1956 to 1958. This planning phase, which is likened in Chap. 2 to the work of an architect planning a building, culminated in a detailed programmer's manual late in 1958. During the same period the basic technology was also established. A number of changes were subsequently made as design and construction progressed, but the basic plan remained as in 1958.

The Stretch computer is now called the IBM 7030. It was delivered to Los Alamos in April, 1961. Several other 7030 systems were under construction in 1961 for delivery to other organizations with a need for very large computers. We shall leave it to others to judge, on the basis of subsequent operating experience, how close the computer comes to satisfying the original objectives of Project Stretch.

Chapter 2

ARCHITECTURAL PHILOSOPHY

by F. P. Brooks, Jr.

Computer architecture, like other architecture, is the art of determining the needs of the user of a structure and then designing to meet those needs as effectively as possible within economic and technological constraints. Architecture must include engineering considerations, so that the design will be economical and feasible; but the emphasis in architecture is upon the needs of the user, whereas in engineering the emphasis is upon the needs of the fabricator. This chapter describes the principles that guided the architectural phase of Project Stretch and the rationale of some of the features of the IBM 7030 computer which emerged.

2.1. The Two Objectives of Project Stretch

High Performance

The objective of obtaining a major increase in over-all performance over previous computers had a triple motivation.

1. There were some real-time tasks with deadlines so short that they demanded very high performance.
2. There were a number of very important problems too large to be tackled on existing computers. In principle, any general-purpose computer can do any programmable problem, given enough time. In practice, however, a problem can require so much time for solution that the program may never be "debugged" because of machine malfunctions and limited human patience. Moreover, problem parameters may change, or a problem may cease to be of interest while it is running.
3. Cost considerations formed another motivation for high performance. It has been observed that, for any given technology, performance generally increases faster than cost. A very important corollary is that, for a fully utilized computer, the cost per unit of computation declines with increasing performance. It appeared that the Stretch computer would show accordingly an improved performance-to-cost ratio over

earlier computers. It appeared, further, that some computer users did indeed have sufficient work to occupy fully an instrument of the proposed power and could, therefore, obtain economic advantage by using a Stretch computer.[1]

Generality

In addition to being fast, the Stretch computer was to be truly a general-purpose computer, readily applicable to scientific computing, business data processing, and various large information-processing tasks encountered by the military. In 1955 and 1956, when the general objectives of Project Stretch were set, it was apparent that there existed a few applications for a very-high-performance computer in each of these areas. There is no question that the new computer could have been made at least twice as fast, with perhaps no more hardware, if it had been specialized for performing a very few specific computing algorithms. This possibility was rejected in favor of a general-purpose computer for four reasons, each of which would have sufficed:

1. No prospective user had all his work confined to so few programs, nor could any user be sure that his needs would not change significantly during the life of the machine.
2. If a computer were designed to perform well on the entire class of problems encountered by any one user, the shift in balance required to make it readily applicable to other users would be quite small.
3. Since there existed only a few applications in each specialized area and since the development costs of a computer of very high performance are several times the fabrication costs, each user would in fact be acquiring a general-purpose computer (containing some hardware he did not especially need) more cheaply than he could have acquired a machine more precisely specialized for his needs.
4. Since there are real limitations on the skilled manpower and other facilities available for development efforts, it would not have been possible to develop several substantially different machines of this performance class at once, whereas it was possible to meet a variety of needs for very-high-performance computers with a single machine.

In sum, then, Project Stretch was to result in a very-high-performance, general-purpose information-processing system.

2.2. Resources

A sharp increase in computer performance does not spring solely from a strong justification for it; new technology is indispensable. It appeared that expected technological advances would permit the design to be based

[1] W. C. Sangren, Role of Digital Computers in Nuclear Design, *Nucleonics*, vol. 15, no. 5, pp. 56–60, May, 1957.

upon new core memories with a 2-microsecond cycle time, new transistor circuits with delays of 10 to 20 nanoseconds (billionths of a second) per stage, and corresponding new packaging techniques. The new transistor technology offered not only high speeds but a new standard of reliability, which made it not unreasonable to contemplate a machine with hundreds of thousands of components.

In order to complete the computer within the desired time span, it was decided to accept the risks that would be involved in (1) developing the technology and (2) designing the machine simultaneously.

The new circuits would be only ten to twenty times as fast as those of the 704, and the new memories would be only six times as fast. Obviously, a new system organization was required if there was to be a major increase in performance. It was clear that the slow memory speed would be the principal concern in system design and the principal limitation on performance. This fact influenced many decisions, among them the selection of a long memory word, and prompted the devotion of considerable effort to maximizing the use of each instruction bit.

Project Stretch benefited greatly from practical experience gained with the first generation of large-scale electronic computers, such as the IBM 700 series. Decisions made in the design of these earlier computers had necessarily been made without experience in the use of such machines. At the beginning of Project Stretch the design features of earlier machines were reviewed in the light of subsequent experience. It should not be surprising that a number of features were found inadequate: some considerations had increased in significance, others had diminished. Thus it was decided not to constrain Stretch to be program-compatible with earlier computers or to follow any existing plan. A completely fresh start meant extra architectural effort, but this freedom permitted many improvements in system organization.

A wealth of intensive experience in the application of existing computers was made available by the initial customers for Stretch computers. From these groups came ideas, insight, counsel, and often, because the groups had quite diverse applications, conflicting pressures. The diversity of these pressures was itself no small boon, for it helped ensure adherence to the objective of general applicability.

2.3. Guiding Principles

The universal adoption of several guiding principles helped ensure the conceptual integrity of a plan whose many detailed decisions were made by many contributors.

Over-all Optimization

The objective of economic efficiency was understood to imply minimizing the cost of answers, not just the cost of hardware. This meant repeated consideration of the costs associated with programming, compilation, debugging, and maintenance, as well as the obvious cost of machine time for production computation. A consequent objective was to make programming easier—not necessarily for trivial problems, but for problems worthy of the computer, problems whose coding in machine language would usually be generated automatically by a *compiler* from statements in the user's language.

A corollary of this principle was the recognition that complex tasks always entail a price in information (and therefore money) and that this price is minimized by selecting the proper form of payment—sometimes extra hardware, sometimes extra instruction executions, and sometimes harder thought in developing programming systems. For example, the price of processing data with naturally diverse lengths and structures is easily recognized (see Chap. 4). This price appeared to be paid most economically in hardware; so very flexible hardware for this purpose was provided. Similarly, protection of memory locations from unwanted alteration was accomplished much more economically with equipment than it would have been with programming. A final minor example is the STORE VALUE IN ADDRESS[1] operation, which inserts index values into addresses of different lengths; by using address-length-determining hardware already provided for other reasons, this instruction performs a task that would be rather painful to program. For other tasks, such as program relocation, exception-condition fix-up, and supervisory control of input-output, hardware was considered, but programming techniques were selected as more economical.

Power instead of Simplicity

The user was given power rather than simplicity whenever an equal-cost choice had to be made. It was recognized in the first place that the new computer would have many highly sophisticated and experienced users. It would have been presumptuous as well as unwise for the computer designers to "protect" such users from equipment complexities that might be useful for solving complex problems. In the second place, the choice is asymmetric. Powerful features can be ignored by a user who wishes to confine himself to simple techniques. But if powerful features were not provided, the skillful and motivated user could not wring their power from the computer.

For these reasons, the user is given programmed access to the hardware

[1] Names of actual 7030 operations are printed in SMALL CAPS in this book. When a name is used to denote a class of operations of which this operation is a member, it is printed in *italics*; also italicized are operations that exist in some computers but not in this one. For example, operations of the *add* type built into the 7030 include ADD, ADD TO MEMORY, ADD TO MAGNITUDE, etc., but not *add absolute*, which is provided in a different manner by modifier bits.

wherever possible. He is given, for example, an interruption and address-protection system whose use can be simple or very complex. He is given an indexing system that can be used simply or in some rather complex ways. If he chooses and if his problems are simple, he can write programs using floating-point arithmetic without regard for precision, overflow, or underflow; but if he needs to concern himself with these often complex matters, he is given full facilities for doing so.

Generalized Features

Wherever specific programming problems were considered worthy of hardware, *ad hoc* solutions were avoided and general solutions sought. This principle came from a strong faith that important variants of the same problem would surely arise and that generality and flexibility would amply repay any extra cost. There was also certainty that the architects could hardly imagine, much less predict, the many unexpected uses for general operations and facilities. This principle, for example, explains the absence of special operations to edit output: the problem is solved by the general and powerful logical-connective operations. Similarly, a single uniform interruption technique is used for input-output communication, malfunction warning, program-fault indication, and routine detection of expected but rare exceptional conditions.

Specialized Equipment for Frequent Tasks

There is also an antithetical principle. For tasks of great frequency in important applications, specialized equipment and operations *are* provided in addition to general techniques. This, of course, accounts for the provision of floating-point arithmetic and automatic index modification of addresses.

To maximize instruction density, however, specialized operations of less than the highest frequency are specified by extra instructions for such operations rather than by extra bits in all instructions. In short, the information price of specifying a less usual operation is paid when it is used rather than all the time. For example, indirect addressing, multiple indexing, and instruction-counter storing on branching each require half-word instructions when they are used, but no bits in the basic instructions are used for such purposes. As a result of such detailed optimization, the 7030 executes a typical scientific program with about 20 per cent fewer instructions of 32 bits than does the 704 with 36-bit instructions on a corresponding program.

Systematic Instruction Set

Because the machine would be memory-limited, it was important to provide a very rich instruction set so that the memory accesses for an instruction and its operand would accomplish as much as possible. As it has developed, the instruction set contains several thousand distinguishable operations. Such a wealth of function could be made conceptually manageable only by strong systematization. For example, there is only one conditional *branch* instruction for testing the machine indicators, but this is accompanied by a 6-bit code to select any one of the 64 machine indicators, a bit to specify testing for either the *on* or the *off* condition, and another bit to permit resetting of the indicator. Thus there are only a few basic operations and a few modifiers. In all, the number of operations and modifiers is less than half the number of operations in the IBM 709 (or 7090), although the number of different instruction actions is over five times that of the 709.

Such systematization, of course, implies symmetry in the operation code set—each modifier can be validly used with all the operations for which it can be indicated in the instruction, and, for most operations, the logical converses or counterparts are also provided. Thus the floating-point-arithmetic set includes not only the customary DIVIDE where the addressed operand constitutes the divisor, but also a RECIPROCAL DIVIDE which addresses the dividend.

Provision for New Operating Techniques

Experience with the IBM 650 and 704 computers had demonstrated that two computers whose speeds differ by more than one order of magnitude are different in kind as well as in degree. This confirmed the suspicion that the 7030 would be more than a super-704 and would be operated in a different way. An early effort was made, therefore, to anticipate some of the operating techniques appropriate for such an instrument, so that suitable hardware could be provided.

The most significant conclusion from these investigations was that an important operating technique would be *multiprogramming*, or time-sharing of the central computer among several independent problem programs. This now familiar (but yet unexploited) concept was new in 1956 and viewed widely with suspicion.

A second conclusion was that the proposed high-capacity, high-data-rate disk storage would contribute substantially to system performance and would permit the 7030 to be operated as a scientific computer without very-high-speed magnetic tapes.

2.4. Contemporary Trends in Computer Architecture

Over the years computer designs have gone through a constant and gradual evolution shaped largely by experience gained in many active computing centers. This experience has heavily influenced the architecture of Stretch. In several instances the attack on a problem exposed by experience with existing computers differs in Stretch from the solution presently adopted in most computer installations. For example, with existing large computers the only way to meet the high cost of human intervention is to minimize such intervention; in the Stretch design the attempt has been, instead, to make human intervention much cheaper.

The effect of several of these contemporary design trends on the Stretch architecture will be examined here.

Concurrency

Most new computer designs achieve higher performance by operating various parts of the computer system concurrently. Concurrent operation of input-output and the central computer has been available for some years, but some contemporary designs go considerably beyond this and allow various elements of the central computer to operate concurrently.[1]

A distinction may be made (see Chap. 13) between local concurrency, providing overlapped execution of instructions that are immediate neighbors in the instruction stream of a single program, and nonlocal concurrency, where the overlap is between nonadjacent instructions that may belong to different programs. The usual input-output concurrency is of the nonlocal type; since the instructions undergoing simultaneous execution are not closely related to one another, the need for interlocks and safeguards is not severe and may, to a large extent, be accomplished by supervisory programming.

Local concurrency is used extensively in the central processing unit of the 7030 to achieve a high rate of instruction flow within a single instruction sequence. Unlike another scheme,[2] in which each specialized unit performs its task and returns its result to memory to await call by the next unit, the 7030 uses registers; this is because memory speed is the main limitation on 7030 computer speed. Several of these registers form a high-speed *virtual memory* (the *look-ahead unit* of Chap. 15), which receives instructions and operands from the real memory in advance of execution by the arithmetic unit and receives the results for storing while the arithmetic unit proceeds with the next operation. Up to eleven successive instructions may be in the registers of the central processing unit at various stages of execution: undergoing address modification, awaiting access to operands in memory, waiting for and being executed by the arithmetic units, or waiting for a result to be returned to memory.

Considerable effort was expended on automatic interlocks and safeguards, so that the programmer would not have to concern himself with

[1] P. Dreyfus, Programming Design Features of the GAMMA 60 Computer, *Proc. Eastern Joint Computer Conf.*, December, 1958, pp. 174–181.

[2] *Ibid.*

the intricate logic of local concurrency. The programmer writes his program as if it were to be executed sequentially, one instruction at a time.

To make a computer with automatic program-interruption facilities behave this way was not an easy matter, because the number of instructions in various stages of processing when an interrupting signal occurs may be large. The signal may have been the result of one of these instructions, requiring interruption before the next instruction is executed. Since the next several instructions may already be under way, it must be possible to go back and cancel their effects. The amount of overlap varies dynamically and may even be different for two executions of the identical instruction sequence; so it would be almost impossible for the programmer to do the backtracking. Therefore, the elaborate safeguards provided to ensure sequential results from nonsequential operation do more than satisfy a desire to simplify programming; the programmer would be lost without them.

Multiprogramming

Time-sharing (as of a computer by multiprogramming) and concurrency are two sides of one coin: to overcome imbalance in a computer system, faster elements are time-shared and slower elements are made to operate concurrently. In the 7030, for example, the single central computer uses several concurrently operating memory boxes, and the single computer-memory system may control in turn many concurrently operating input-output devices.

Even though per-operation cost tends to decrease as system performance increases, per-second cost increases, and it therefore becomes more important to avoid delaying the calculator for input-output. To take full advantage of concurrent input-output operation for a computer of very high performance demands that input data for one program be entered while a preceding program is in control of calculation and that output take place after calculation is complete. For this reason alone, it was apparent from the beginning that multiprogramming facilities would be needed for Project Stretch.

A second motivation for multiprogramming is the need for a closer man-machine relationship. As computers have become faster, the increasing cost of wasted seconds has dictated increasing separation between the problem sponsor and the solution process. This has reduced the over-all efficiency of the problem-solving process; for, in fact, the more complex problems solved on faster calculators are harder, not easier, for the sponsor to comprehend and therefore need more, not less, dynamic interaction between solution process and sponsor. There can be no doubt that much computer time and more printer time has been wasted because the problem sponsor cannot observe and react as his program is being run on large computers like the IBM 704. This difficulty promised to become more acute with the even more complex problems for which Stretch was needed.

With multiprogramming it becomes economically practical for a person seated at a console to observe his program during execution and interrupt it while considering the next step. Since the computer can immediately be switched to another waiting program, the user is not charged with the cost of an idle computer. Thus the extension of multiprogramming to manual operation offers, once the technique has been mastered, a tremendous economic breakthrough: it provides a general technique for solving the problem of loss of contact between sponsor and solution. A sponsor can now interact with his problem at his own speed, paying only the cost of delaying the problem, not that of delaying the machine. This should materially accelerate that large proportion of scientific computation which is expended on continual and perpetual refinement and debugging of mathematical models and the programs that embody them. The solution of most such problems is characterized more closely by a fixed number of interactions between computer and sponsor than by a fixed amount of computer time.

Multiprogramming also makes it economically practical to enter new data and to print or display results *on line*, that is, via directly connected input and output devices; whereas the economics of previous computers forced card-to-tape and tape-to-printer conversion *off line*, that is, with physically separate devices, so that only the fastest possible medium, magnetic tape, would be used on the computer. On-line operation of input and output is emphasized in the Stretch philosophy, because it removes much of the routine operator intervention and reduces the overall elapsed time for each run of a problem.

Multiprogramming makes several demands upon system organization. Most obvious is the requirement of ample and fast storage, both internal and external. Of equal importance is an adequate and flexible interruption system. Also, in the real world, time-sharing of a computer among users with ordinary human failings requires memory protection, so that each user can feel secure within his assigned share of the machine. Debugging is difficult enough at best, and most users would sacrifice efficiency rather than tolerate difficulties caused by the errors in other programs. It proved possible in the 7030 to provide a rudimentary but sufficient form of memory protection without affecting speed and with a modest amount of hardware.

The equipment for multiprogramming was, however, limited to two essential features: program interruption and address monitoring, and these were designed to be as flexible as possible. Other multiprogramming functions are left to the supervisory program, partly because that arrangement appeared to be efficient, but primarily because no one could be sure which further facilities would prove useful and which would prove merely expensive and overly rigid inconveniences. Several years of actual multiprogramming experience will undoubtedly demonstrate the value of other built-in features.

If multiprogramming is to be an operating technique, a radically different design is needed for the operator's console. If several independent programs are to be run, each with active operator intervention, there must be provision for multiple independent consoles. Each console must be incapable of altering any program other than the associated problem program. For active intervention by the problem sponsor (rather than by a special machine operator), the console must be especially convenient to use. Finally, if a supervisory program is to exercise complete control in scheduling programs automatically, it must be able to ignore unused console facilities. Although intelligent human intervention is prized highly, routine human intervention is to be minimized, so as to reduce delays and opportunities for error.

The operating console was designed to be simply another input-output device with a convenient assortment of switches, keys, lights, digital displays, and a typewriter. A console interpretive program assigns meaning to the bits generated by each switch and displayed by each light. There are no maintenance facilities on the operator's console, and completely separate maintenance consoles are provided.

Automatic Programming

Undoubtedly the most important change in computer application technique in the past several years has been the appearance of symbolic assemblers and problem-language compilers. Studies showed that for Stretch at least half of all computer time would be used by compiler-produced programs; all programs would be at least initially translated by an assembler.

A most important implication of symbolic-language programming is that the addressing radix and structure need not be determined for coder convenience. Fairly complex instruction formats can be used without causing coding errors, and operation sets with hundreds of diverse operations can be used effectively.

Many proposals for amending system architecture to simplify compilers were considered. The most far-reaching of these concerned the number of index registers, which should be infinity or unity for greatest ease of assignment during compilation. The alternatives were investigated in considerable detail, and both turned out to reduce computer performance rather sharply. Indeed, reduced performance was implied by most such proposals. These studies resulted in a belief which is not shared by all who construct compilers; this is that total cost to the user is minimized not by restricting system power to keep compilers simple but by enhancing facilities for the task of compilation itself, so that compilers can operate more rapidly and efficiently.

Information Processing

The arithmetic power of a computer is often only ancillary to its power of assembling, rearranging, testing, and otherwise manipulating information. To an increasing extent, bits in even a scientific computer represent things other than numerical quantities: elements of a program metalanguage, alphabetic material, representations of graphs, bits scanned from a pattern, etc. In the light of this trend, it was therefore important to match powerful arithmetical with powerful manipulative facilities. These are provided in the variable-field-length arithmetic and, in unique form, in the variable-field-length connective operations, which operate upon bits as entities rather than components of numbers. Good variable-field-length facilities are, of course, particularly important for business and military data processing.

2.5. Hindsight

As the actual shape of the 7030 began to emerge from the initial planning and design stages, it became apparent that some of the earlier thoughts had to be revised. (Some of these changes have already been noted parenthetically in Chap. 1.) The bus unit for linking and scheduling traffic between many memory boxes and many memory-using units turned out to be a key part of the design. The original algorithms for multiplication and division proved inadequate with available circuits, and new approaches were devised. It became clear that division, especially, could not be improved by the same factor as multiplication. Serial (variable-field-length) operation turned out to be considerably slower than expected; so serial multiplication and division were abandoned, and the variable-field-length multiplication and division operations were redesigned to use the faster parallel unit.

The two separate computer sections that were postulated originally were later combined (see Chap. 1), and both sets of facilities were placed under the control of one instruction counter. Although the concept of multiple computing units, closely coupled into one system, was not found practical for the 7030 system, this concept still seems promising.[1] In fact, the input-output exchange coupled to the main computer in the 7030 is a simplified example, since the exchange is really another computer, albeit a highly specialized one with an extremely limited instruction vocabulary.

[1] A. L. Leiner, W. A. Notz, J. L. Smith, and A. Weinberger, PILOT: A New Multiple Computer System, *J. ACM*, vol. 6, no. 3, pp. 313–335, July, 1959.

Some architectural features proved unworkable. Rather late in the design period, for example, it became clear that the method of handling zero quantities in floating-point arithmetic was ill-conceived; so this method was abandoned, and a better concept was devised.

Two excellent features, each of which contributes markedly to system performance, were found to have inherently conflicting requirements; their interaction prevents either feature from realizing its full potential. The program-interrupt system is intended to permit unpredicted changes in instruction sequencing. The instruction look-ahead unit, on the other hand, depends for its effectiveness on the predictability of instruction sequences; each interruption drains the look-ahead and takes time to recover. This destroyed the usefulness of the interrupt system for frequent one-instruction fix-ups and required the addition of built-in exception handling in such cases as floating-point underflow.

On the other hand, some improvements became possible as the design progressed. It turned out, for example, that the equipment for performing variable-field-length binary multiplication with the parallel arithmetic unit could easily be made to do binary-decimal and format conversions; so this facility was added.

There are in the 7030 architectural features whose usefulness is still unmeasured. A few are probably mistakes. Others seem to be innovations that will find redefinition and refinement in future computers, large and small. Still other features appear now to be wise for very-high-performance computers, but must be considerably scaled down for more modest machines. Experience has, however, reinforced the system architects' belief in the guiding principles of the design and in the general applicability of these principles to other computer-planning projects.

Chapter 3

SYSTEM SUMMARY OF IBM 7030

by W. Buchholz

3.1. System Organization

The IBM 7030 is composed of a central processing unit, one or more memory units, a memory bus unit, an input-output exchange, and input-output devices. Optionally, high-speed magnetic disk storage units and a disk control unit may be added for external storage. A typical system configuration is shown in Fig. 3.1.

Information moves between the input-output devices and the memories under control of the exchange. The central processing unit (CPU) actually consists of several units that may operate concurrently: an instruction unit, which controls the fetching and indexing of instructions and executes the instructions concerned with indexing arithmetic; a look-ahead unit, which controls fetching and storing of data for several instructions ahead of the one being executed, so as to minimize memory traffic delays; a parallel arithmetic unit, for performing binary arithmetic on floating-point numbers at very high speed; and a serial arithmetic unit, for performing binary and decimal arithmetic, alphanumeric operations, and logical-connective operations on fields of varying lengths.

Logically the CPU operates as one coordinated unit upon a succession of instructions under the control of a single instruction counter. Care is taken in the design so that the user need not concern himself with the intricacies of overlapped operations within the CPU.

The memory bus unit coordinates all traffic between the various memory units on the one side and, on the other side, the exchange, the disk control, and the various parts of the CPU.

3.2. Memory Units

The main magnetic core memory units have a read-write cycle time of 2.1 microseconds. A memory word consists of 64 information bits and 8 check bits for automatic single-error correction and double-error detection.

The address part of every instruction provides for addressing directly any of 262,144 (2^{18}) word locations. Addresses are numbered from 0 up to the amount of memory provided in a particular system, but addresses 0 to 31 refer to index words and special registers instead of general-purpose memory locations.

Each unit of memory consists of 16,384 (2^{14}) words. A system may contain one, two, or a multiple of two such units, up to a maximum of sixteen units. Each memory unit operates independently. In systems with two units or more, several memory references may be in process at the same time. In order to take better advantage of this simultaneity, successive addresses are distributed among different boxes. When a system comprises two units, successive addresses alternate between the two. When a system comprises four or more units, the units are arranged in groups of four, and successive addresses rotate to each of the four units in one group, except for the last group which may consist of only two units with alternating addresses.

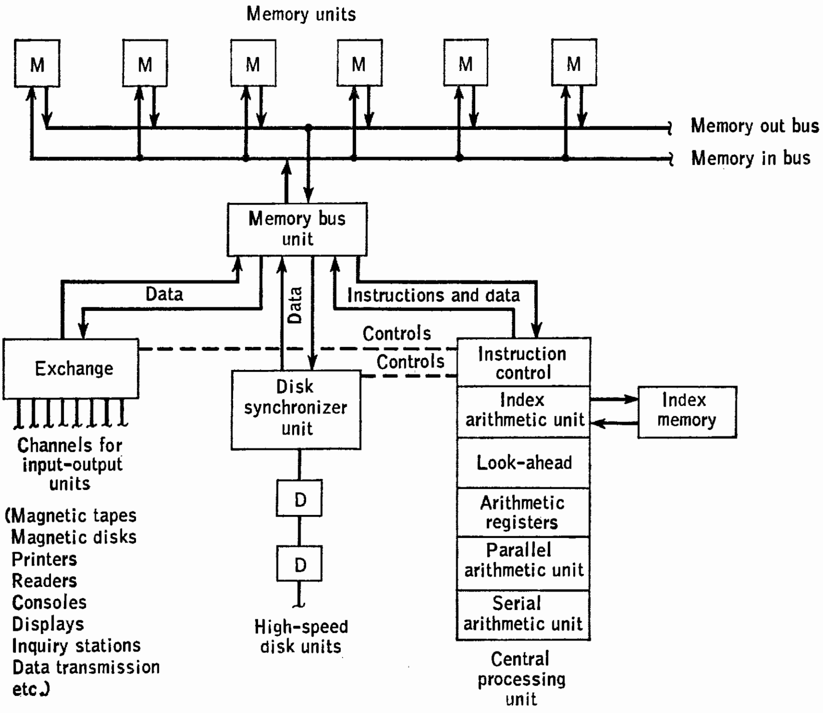


FIG. 3.1. 7030 system.

3.3. Index Memory

A separate fast magnetic core memory is used for index registers. Since index words are normally read out much more often than they are altered, this memory has a short, nondestructive read cycle of 0.6 μsec. The longer clear-and-write cycle of 1.2 μsec is taken only when needed.

The index memory is directly associated with the instruction unit of the computer. It cannot be used to furnish instructions, nor can it be used directly with input or output.

The sixteen index registers have regular addresses 16 to 31, which correspond to abbreviated 4-bit index addresses 0 to 15. The first register cannot participate in automatic address modification since an index address of 0 is used to indicate no indexing.

3.4. Special Registers

Many of the registers of the machine are directly addressable. Some of these are composed of transistor flip-flops; others are in the fast index memory or in main memory. The addressable registers are assigned addresses 0 to 15. These locations cannot be used for instructions or for input or output data.

Address 0 always contains zero. It is a bottomless pit; regardless of what is put in, nothing comes out. The program may attempt to store data at address 0, but any word fetched from there will contain only 0 data bits.[1]

The remaining fifteen addresses correspond to machine registers, time clocks, and control bits. They are listed in the Appendix.

3.5. Input and Output Facilities

Input to the system passes from the input devices to memory through the exchange. The exchange assembles successive 64-bit words from the flow of input information and stores the assembled words in successive memory locations without tying up the central processing unit. The CPU specifies only the number of input words to be read and their location in memory; the exchange then completes the operation by itself.

The exchange operates in a similar manner for output, fetching successive memory words and disassembling them for the output devices independently of the CPU. External storage devices, such as tapes and disks, are operated via the exchange as if they were input and output.

The exchange has the basic capability of operating eight independent input-output units. This eight-channel exchange can be enlarged by

[1] A distinctive type (0, 1) is used in the text for the bits of binary numbers or codes, and regular type (0, 1, 2, . . .) for decimal digits. For example, 10 is a binary number (*two*) and 10 a decimal number (*ten*).

adding more eight-channel groups. Each of these channels can handle information at a rate of over 500,000 bits per second. The exchange as a whole can reach a peak data rate of 6 million information bits per second.

A wide variety of input-output units can be operated by the exchange. These include card readers and punches, printers, magnetic tapes, operator's consoles, and typewriter inquiry stations. Several of some kinds of units can be attached to a single exchange channel; of the several units on a single channel, only one can be operated at a time.

Provisions have been made in the design of the exchange for adding up to 64 more channels operating simultaneously but at a much lower data rate per channel. This extension is intended for tying the computer economically into a large network of low-speed units, such as manually operated inquiry stations.

3.6. High-speed Disk Units

For many large problems, the amount of core storage that it is practical to provide is not nearly large enough to hold all the data needed during computation. Earlier systems have been severely limited by the relatively low data rates of magnetic tapes or the relatively low capacities of magnetic drums available for back-up storage. To avoid having the over-all 7030 performance limited by the same devices, it was essential to develop an external storage medium with high capacity and high data-transfer rates. A magnetic disk storage unit was designed for this purpose.

The disk units read or write at a rate of 125,000 words per second, or 8 million bits per second over a single channel (a rate 90 times that of the IBM 727 tape available with the 704). One or more units, each with a capacity of 2 million words, may be attached. Access to any location of any disk unit requires of the order of 150 milliseconds. Once data transmission has started it continues at top speed for as many consecutive words as desired, without further delays for access to successive tracks.

The control unit, or *disk synchronizer*, functions like the input-output exchange except that it is a single-channel device designed specifically to handle the high data rate of the disks. The exchange and the disk synchronizer can operate independently and simultaneously at full speed. An error-correcting code is used on the disks, and any single errors in data read from the disks are corrected automatically by the control unit before transfer to memory.

3.7. Central Processing Unit

The central processing unit performs arithmetical and logical operations upon operands taken from memory. The results are generally left in accumulator registers to be further operated on or to be stored in memory subsequently. Operations are specified one at a time by instructions, which are also taken from memory. Each instruction usually specifies an operation and an operand or result. The operand specification is made up of an *address* and an *index address.* Part of the index word contents are added to the address in the instruction to obtain an *effective address.* The effective address designates the actual location of the operand or result. The additions needed to derive the effective address and to modify index words are performed in an index-arithmetic unit which is separate from the main arithmetic unit.

3.8. Instruction Controls

An instruction may be one word or one half word in length. Full- and half-length instructions can be intermixed without regard to word boundaries in memory.

Instructions are taken in succession under control of an instruction counter. The sequence of instructions may be altered by branching operations, which can be made to depend on a wide variety of conditions. Automatic interruption of the normal sequence can also be caused by many conditions. The conditions for interruption and control of branching are represented by bits in an indicator register. The interrupt system also includes a *mask register* for controlling interruption and an *interrupt address register* for selecting the desired set of alternate programs. When it is needed, the address of the input or output unit causing an interruption can be read from a *channel address register* which can be set up only by the exchange.

The interpretation and execution of instructions is monitored to make sure that the effective addresses are within boundaries defined by two boundary registers.

3.9. Index-arithmetic Unit

The *index-arithmetic unit,* which is part of the instruction-control unit, contains registers for holding the instructions to be modified and the index words used in the modification. When index words themselves are operated on, some of these registers also hold the operand data. The indexing operations include loading, storing, adding, and comparing. The index-arithmetic unit has gates for selecting the necessary fields in index and instruction words and a 24-bit algebraic adder.

3.10. Instruction Look-ahead

After initiating a reference to memory for a data word, the instruction unit passes the modified instruction on to the *look-ahead unit.* This unit holds the relevant parts of the instruction until the data arrive, so that both the operation and its operand can be sent to the arithmetic unit together. Since access to the desired memory unit takes a relatively long time, the look-ahead will accept several instructions at a time and initiate their memory references, so as to smooth out the memory traffic and obtain a high degree of overlap between memory units. Thus the unit "looks" several instructions ahead of the instruction being executed and anticipates the memory references needed. This reduces delays and keeps the arithmetic unit in as nearly continuous operation as possible.

Indexing and branching instructions are completed by the instruction unit without involving the main arithmetic unit. The instruction unit receives its own operands, whereas the look-ahead receives operands for the main arithmetic unit. The look-ahead, however, is responsible for storing all results for both units, so that permanent modification of stored information is done in the proper logical sequence. Interlocks in the look-ahead unit ensure that nothing is altered permanently until all preceding instructions have been executed successfully.

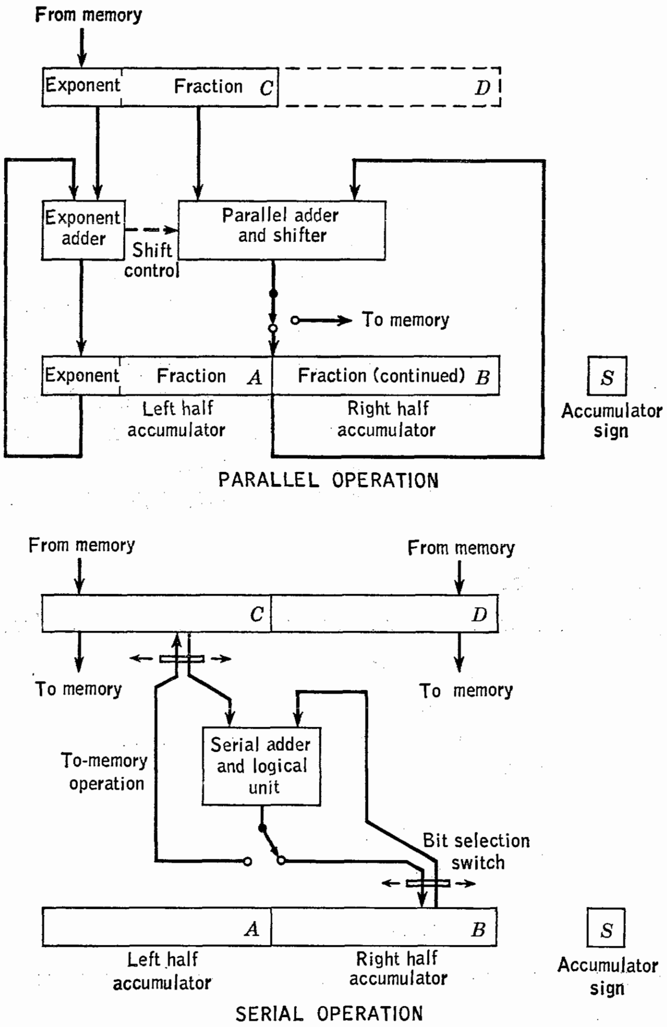
3.11. Arithmetic Unit

The arithmetic unit consists of a parallel and a serial section. The parallel section essentially performs floating-point arithmetic at high speed, and the serial section performs fixed-point arithmetic and logical operations on fields of variable length. Both sections share the same basic registers and much of the control equipment; so they may be treated as one unit.

For simplicity, the arithmetic unit may be considered to be composed of 4 one-word registers and a short register. This conceptual structure is shown in Fig. 3.2, where the full-length registers are labeled A, B, C, and D, and the short register is labeled S. The registers marked A and B constitute the left and right halves of the accumulator. The registers marked C and D serve only as temporary-storage registers, receiving words from memory and (in serial operations only) assembling results to be stored in memory. The short register S stores the accumulator sign bit and certain other indicative bits.

In floating-point addition the operand from memory is sent to register C. (Since floating-point operands will fit into register C, register D is not needed here.) This operand is then added to the contents of register A or of both registers A and B, depending on whether single- or double-length addition has been specified. The result is placed in A or in A and B. As an alternative (adding to memory), the result may be returned to the location of the memory operand instead.

In floating-point multiplication one factor is the number in accumulator register A. The other factor comes from memory and is trans-



PARALLEL OPERATION

SERIAL OPERATION

FIG. 3.2. Simplified register structure of arithmetic unit.

ferred to register C. The factors are now multiplied together, and the product is returned to the accumulator register, *replacing* the previous contents. In cumulative multiplication one factor must have been previously loaded into a separate factor register (not shown). The other factor again comes from memory and goes to C. The factors are multiplied as in ordinary multiplication, but the product is *added* to the contents of the accumulator register.

In floating-point division the dividend is in the accumulator, and the divisor is brought from memory to register C. The quotient is returned to the accumulator, and the remainder, if any, goes to a *remainder register* (not shown).

In serial variable-field-length operations the operand field may occupy parts of two adjacent memory words, and both words if necessary are fetched and placed in registers *C* and *D*. The other operand field comes from *A* and *B*. The operands are selected a few bits at a time and processed in serial fashion. The result field may replace *A* and *B*, or it may replace selected bits of *C* and *D* whose contents are then returned to memory. Binary multiplication and division operands are stepped into the parallel mechanism a few bits at a time, but the actual operation is performed in parallel.

Other registers are the *transit register*, a full-word location, which may be used for automatic subroutine entry; and two 7-bit registers, the *all-ones counter* and the *left-zeros counter*, which are used in connective operations to hold bit counts developed from the results.

All registers mentioned above, except memory registers *C* and *D*, are also addressable as explicit operands.

3.12. Instruction Set

The operations available may be divided into these categories:

Data arithmetic
  1. Floating-point arithmetic
  2. Variable-field-length arithmetic
Radix conversion
Connectives
Index arithmetic
Branching
Transmission
Input-Output

The categories are briefly described in the next few sections.

3.13. Data Arithmetic

The arithmetical instruction set includes the conventional operations LOAD, ADD, STORE, MULTIPLY, and DIVIDE. Modifier bits are available to change the operand sign. The operations *subtract* and *add absolute* are obtained by use of sign modifiers to the ADD instruction and are not provided as separate operations. The same modifiers make it possible to change the sign of a number that is to be loaded, stored, multiplied, or divided.

A convenient feature of the MULTIPLY operation is that one of the factors is taken from the accumulator rather than from a separate register, and this factor may be the result of previous computation. Similarly, DIVIDE places the quotient in the accumulator, and so the quotient is available directly for further arithmetical steps.

Extensions of the basic set of arithmetical operations permit adding and counting in memory, rounding, cumulative multiplication, comparison, and further variations of the standard ADD operation.

One of these variations is called ADD TO MAGNITUDE. This operation differs from ADD in that, when the signs and modifiers are set for subtraction, it does not allow the result sign to change. When the result sign would change, the result is set instead to zero. This operation is useful in dealing with nonnegative numbers or in computing with discontinuous rates.

The important arithmetical operations are available in the floating-point mode as well as in the (fixed-point) variable-field-length mode.

Floating-point-arithmetic Operations

Floating-point (FLP) arithmetic uses a 64-bit floating-point word consisting of a signed 48-bit binary fraction, a signed 10-bit binary exponent, and an exponent flag to indicate numbers that have exceeded the available exponent range. Arithmetic can be performed in either normalized or unnormalized form.

The 48-bit fraction (mantissa) is longer than those available in earlier computers, so that many problems can be computed in single precision, which would previously have required much slower double precision. When multiple-precision computation is required, however, it is greatly facilitated by operations that produce double-length results.

To aid in significance studies, a *noisy mode* is provided in which the low-order bits of results are modified. Running the same problem twice, first in the normal mode and then in the noisy mode, gives an estimate of the significance of the results.

Variable-field-length-arithmetic Operations

The class of variable-field-length (VFL) arithmetic is used for data arithmetic on other than the specialized floating-point numbers. The emphasis here is on versatility and on economy of storage. Arithmetic may be performed directly in either decimal or binary radix. Individual numbers, or *fields*, may be of any length, from 1 to 64 bits. Fields of different lengths may be assigned to adjacent locations in memory, even if this means that a field lies partly in one memory word and partly in the next. Each field may be addressed directly by specifying its position and length in the instruction; the computer takes care of selecting the memory words required and altering only the desired information.

Numerical data may be signed or unsigned. For unsigned data the sign is simply omitted in memory; this saves space and avoids the task of assigning signs where there are none to begin with. Unsigned numbers are treated arithmetically as if they were positive.

VFL arithmetic is sometimes called *integer arithmetic*, because in multiplication and division the results are normally aligned as if the operands were integers. It is possible, though, to specify that operands be *offset* so as to obtain any desired alignment of the radix point. An offset can be specified in every instruction, and there is no need for separate instructions to shift the contents of the accumulator.

A significant feature of the VFL DIVIDE operation is that it will produce meaningful results regardless of the magnitude of the dividend or the divisor (provided these fall within the bounds of numbers generally acceptable to the arithmetic unit). The only and obvious exception is a zero divisor. This greater freedom eliminates much of the scaling previously required before a DIVIDE instruction could be accepted.

All VFL-arithmetic operations are available in either decimal or binary form, and the choice can be made by setting 1 modifier bit. Decimal multiplication and division, however, are not built into the computer directly; instead their operation codes are used to cause an automatic entry to a standard subroutine which can take advantage of high-speed radix conversion and binary multiplication or division. Thus decimal multiplication and division are faster but just as convenient to program as if they had been built in for execution by the serial decimal circuits.

An operation is provided that causes an automatic entry to a subroutine. A field of this instruction may be used to distinguish up to 128 pseudo operations.

One use of the VFL-arithmetic operations is to perform general arithmetic on *portions* of floating-point words, instruction words, or index words. The floating-point and index-arithmetic instruction classes do contain special addition and comparison instructions for the most frequent operations on partial words of this kind, but the VFL operations provide a complete set for all purposes.

Alphabetic and alphanumeric fields of various lengths are handled by VFL-arithmetic operations as if they were unsigned binary numbers, regardless of the character code. There is actually no fixed character code built into the computer, although a certain code with many desirable features is recommended. Alphanumeric high-low comparisons are made by a simple binary subtraction of two fields. The only requirement is that the binary numbers representing each character fall into the comparing sequence desired for the application. If the code used for input or output does not conform to this comparing requirement, special provisions facilitate the translating of the code to any other form by programming a table look-up.

The number of bits used to encode individual characters may be varied. Thus a decimal digit may be compactly represented by a binary code of

4 bits, or it may be expanded to 6 or more bits when intermixed with alphabetic information.

3.14. Radix-conversion Operations

A group of radix-conversion operations is provided to convert integers between decimal and binary form in either direction. These operations are also used in implementing the decimal multiplication and division pseudo operations mentioned in the preceding section.

3.15. Connective Operations

Instructions that combine bits by logical *and*, *or*, and *exclusive or* functions have been available in earlier computers. These and many other nonarithmetical data-handling operations are here replaced in simple and orderly fashion by *connective* operations that provide many logical facilities not previously available. These operations are called CONNECT, CONNECT TO MEMORY, and CONNECT FOR TEST.

Each connective operation specifies a memory field of any length from 1 to 64 bits, as in integer arithmetic. Each bit in the memory field is logically combined with a corresponding bit in the accumulator; the resulting bit replaces the accumulator bit in CONNECT, the memory bit in CONNECT TO MEMORY, or neither in CONNECT FOR TEST. All three operations make available certain tests and counts of 0 and 1 bits.

There are sixteen possible ways in which to combine, or *connect*, two bits. Each of these logical connectives can be specified along with each of the three connective operations. Besides the connectives *and*, *or*, and *exclusive or*, there are connectives to match bits, to replace bits, and to set bits to 0 or 1. Either or both of the operands may be inverted.

Although the term *logical connectives* suggests evaluation of elaborate expressions in Boolean algebra, the connective instructions have important everyday applications, such as the assembling and converting of input-output data. Their power lies in their ability to specify fields of any length and in any position in memory, either single test bits or strings of adjacent bits.

3.16. Index-arithmetic Operations

The address part of any instruction may be modified by adding a number in a specified *index register* before the address is used. Normally both the instruction and the index register remain unchanged. To alter the index registers is the function of the *index arithmetic* operations.

These operations include loading, storing, incrementing, and comparing of index values. The index *value* is a signed number, and additions are algebraic. One of the instructions allows up to sixteen index values to be added together for use in further indexing. Another indexing instruction provides the function of *indirect addressing.*

Each index word contains a *count* to keep track of the number of times a program loop has been traversed. Counting may be coupled with incrementing of the index value. A third field in each index word specifies a *refill* address from which another index word may be loaded automatically.

Instructions generally specify one of a set of fifteen index registers for address modification, but the number of available registers may be readily supplemented by other index locations in memory through the operation RENAME. This operation identifies one designated index register with one of these memory locations and does the bookkeeping necessary to cause this memory location to reflect changes in the index register.

Although indexing instructions are provided to change index values and counts explicitly, it is possible to use another mode, called *progressive indexing*, in which the index quantities may be advanced each time they are used.

3.17. Branching Operations

The *branching* operations either conditionally or unconditionally alter the instruction counter so as to change the course of a program. The number of these operations is not large, but modifiers are available to provide a great deal of flexibility.

All machine-state indicators, such as sign, overflow, error, and input-output conditions, are collected in one 64-bit indicator register. The BRANCH ON INDICATOR instruction may specify any one of these 64 indicators as the condition to be tested. A modifier specifies whether branching is to occur when the indicator is *on* or *off*. Another modifier may cause the indicator tested to be reset.

A second operation, BRANCH ON BIT, permits the testing of a single bit anywhere in memory with one instruction. The tested bit may also be modified. This instruction places a virtually unlimited number of indicators under the direct control of the program.

A hybrid operation combines advancing of an index word with testing and branching. Thus the most common program loops may be closed with one half-length instruction, although full indexing flexibility requires two half-length instructions to specify the necessary quantities.

Branch instructions may be coupled with another operation to store the instruction-counter contents at any desired location before branching. This simplifies reentry to a program from a subprogram.

3.18. Transmission Operations

The operation TRANSMIT provides the facilities to move a block of data from one memory area to another. A second operation, SWAP, interchanges the contents of two memory areas.

3.19. Input-Output Operations

There are basically two operations for controlling input-output and external storage units: READ and WRITE. Each instruction specifies the unit desired and a memory area for the data to be read or written.

The memory area is specified by giving the address of a control word which contains the first data address in memory and a count of the number of words to be transferred. The control word also contains a refill address which can specify the address of another control word. Control words can thus be chained together to define memory areas that are not adjacent.

Control words have the same format as index words and can be used for indexing. This important feature means that the same word can be used first for reading new data, then for indexing while those data are being processed, and finally for writing the data from the same memory area.

Various modifications of READ and WRITE are provided to fit different circumstances. Other instructions perform various control functions.

All instructions for operating external units are issued by the computer program but are executed independently of the program. Several data transfers can thus take place simultaneously, all sharing access to memory. Signaling functions inform the program when each external process is completed.

All external units, regardless of their characteristics, are controlled by the same set of instructions. They are distinguished only by a number assigned to each unit.

3.20. New Features

New programming features not identified with specific instructions are summarized in this section.

Addressing

In instructions where this is meaningful, the position of a single bit in any word of memory can be addressed directly. A complete word-and-bit address forms a 24-bit number. The word address (18 bits) is on the left and the bit address (6 bits) on the right of that number. For the purpose of bit addressing, the entire memory may be regarded as a set of consecutively numbered bits. Since the number of bits in a memory word (64) is a power of 2 and all addressing is binary, the address of the rightmost bit (bit 63) of one memory word is followed immediately by the address of the leftmost bit (bit 0) of the word with the next higher word address. Memory-word boundaries may be ignored by the program.

Other instructions use only full memory words as data, and these provide space for only 18 bits of address. The bit address is assumed to be 0. Still other instructions refer to half words and use 19 bits of address. The extra bit is immediately to the right of the word address, and the remaining 5 bits of the bit address are treated as 0s.

Index words provide space for a sign and 24 bits in the value field, so that all addresses may be fully indexed to the bit level. The entire 24-bit instruction address, with 0s inserted where instructions have fewer address bits, participates in the algebraic addition during address modification. When less than 24 bits are needed in the effective address, the low-order bits are dropped.

Many internal machine registers are directly addressable as if they were memory. The accumulator may, for example, be added to itself; this is accomplished by addressing the accumulator as the operand of an ADD instruction. One important use of this facility is in preserving and restoring the contents of internal registers by transmitting them as a block to or from memory with one TRANSMIT instruction.

Instead of selecting a location from which to fetch data, the address itself may serve as data in many operations. It is then called an *immediate address*. Such data are limited to at most 24 bits. This feature is very convenient for defining short constants without having to provide the space and time for separate access to memory. Immediate addressing is not available for sending data to memory, because the address space is needed to select memory.

The term *direct address* is used to distinguish the usual type of address which gives the location of an operand or of an instruction.

The term *indirect address* refers to an address that gives the location of another address. An indirect address may select an immediate address, a direct address, or yet another indirect address. Indirect addresses are obtained in the 7030 by the instruction LOAD VALUE EFFECTIVE, which places the effective address found at the specified memory location into an index register for indexing a subsequent instruction. Multiple-level indirect addressing is obtained when LOAD VALUE EFFECTIVE finds at the selected location another instruction LOAD VALUE EFFECTIVE which causes the indirect addressing process to be repeated.

Program Interruption

A single *program-interrupt* system serves for responding to asynchronously occurring external signals and for monitoring exceptional conditions generated by the program itself. When one of the indicators in the previously mentioned indicator register comes on, the computer selects an instruction from a corresponding position in a table of fix-up instructions. This instruction is sandwiched into the program currently being executed at whatever time the interruption occurs. The extra instruction is usually one which first stores the current instruction-counter setting, to preserve the point at which the current program was interrupted, and then branches to the desired fix-up routine. The table of fix-up instructions may be placed anywhere in memory.

Means are provided to select which indicators may cause interruption and when interruption will be permitted. Priorities can thus be established. If more than one interrupt condition should occur at a time, the system will take them in order. Special provisions are made to permit interruptions to any level to occur without causing program confusion.

Address Monitoring

Address-monitoring facilities are provided to assist in the debugging of new programs and to protect already debugged programs against erroneous use of their memory locations by other programs being run simultaneously in multiprogrammed fashion. The two *address-boundary registers* are used to define the desired memory area. One register specifies the lower boundary and one the upper boundary. All effective operand addresses and all instruction addresses are compared against the two addresses in the registers to see whether the address in question falls inside or outside the boundaries. By setting a control bit, it is possible to define either the area inside the boundaries or the area outside the boundaries as the protected area. Whichever it is, any attempt to fetch an instruction or data word from the protected area or to store new information in the protected area may be suppressed, and the program may be interrupted immediately. Thus it is possible to use the address-monitoring system to make sure either that a given program does not stray *outside* its assigned area or that no program will interfere with whatever is stored *inside* the area.

The built-in monitoring system is much more effective than the alternative of screening each program in advance to make sure that all addresses are proper. It is very difficult to predict by inspection all the *effective* addresses that may be generated during execution by indexing, indirect addressing, or other procedures, especially in a program that may contain errors.

Clocks

An *interval timer* is built in to measure elapsed time over relatively short intervals. It can be set to any value at any time, and an indicator shows when the time period has ended. This indicator will cause automatic program interruption.

To provide a continuous indication of time, a *time clock* is also furnished. This clock runs continuously while the machine is in operation; its setting cannot be altered by the programmer. It may be used to time longer intervals for logging purposes or, in conjunction with an external calibrating signal, to provide a time-of-day indication.

3.21. Performance

Since high performance is so important an objective of the 7030, a summary of the system should give some examples of its internal speed. Such speeds cannot be quoted with any accuracy, however.

In earlier computers it has been a relatively simple matter to compile a list of exact times or time formulas for the execution of each operation. To determine the time taken to execute a program it was necessary only to add the times required for each instruction of the program. Describing the internal speed of the 7030 with any accuracy is a much more difficult task because of the high degree of overlap among the independently and asynchronously operating parts of the central processing unit.

A few raw arithmetic speeds are listed in Chap. 14. The list is not complete and includes only the time spent by the arithmetic unit operating on data already available. There would be little point in extending the list; instruction and data fetches, address modification, and the execution of indexing and branching instructions all overlap the arithmetic-execution times to varying degrees; so the figures could not be meaningfully added together.

Rules of thumb and approximation formulas may be developed in time, but their accuracy would depend considerably on the type of program. The degree of overlap varies widely between problems requiring a predominance of floating-point arithmetic or variable-field-length arithmetic or branching or input-output activity. A zero-order approximation, which could be off by a factor of 2 or more, might be to count 2.5 microseconds for each instruction written. To arrive at a more accurate figure it is necessary to take into account the complex timing relationships of a succession of specific instructions in considerable detail. Even then it would be difficult to measure the effect on performance of the long floating-point word, the large core memory, the very large capacity of the high-speed disk units, the overlapped input-output data transfer, or the interrupt system. The best approach is still to program a complete problem and then time the actual execution on the 7030 itself.

Chapter 4

NATURAL DATA UNITS

by G. A. Blaauw, F. P. Brooks, Jr., and W. Buchholz

4.1. Lengths and Structures of Natural Data Units

In considering automatic data-processing tasks generally, we identify five common types of operations: floating-point operations, fixed-point arithmetic, address arithmetic, logical manipulations, and editing operations. Each of these has a *natural data unit* distinct from those of the other types in length, variability of length, or internal structure. An ideal computer would permit each operation to address its natural data unit directly, and this addressing would be simplified by utilizing all properties of the natural data unit that are constant.

It should be observed that the natural data unit is associated with an individual manipulative operation, not with a whole program. In any program there will be different kinds of operations and, therefore, different natural data units. Furthermore, the same datum is generally the object of different kinds of operations. For example, a floating-point datum may be developed as a unit in a computation, its components then used in radix-conversion arithmetic, and the characters of the result finally used as units in editing for printing. The format of a datum is usually made to agree as closely as possible with the natural data unit of the operations most often performed on that datum.

The natural data unit for most technical computation has come to be the floating-point number, because the use of *floating-point arithmetic* frees the mathematician from many details of magnitude analysis. This unit has considerable internal structure: the representation of a single

Note: Sections 4.1 and 4.2 of this chapter are taken from a previously published paper by the same authors: Processing Data in Bits and Pieces, *IRE Trans. on Electronic Computers,* vol. EC-8, no. 2, pp. 118–124, June, 1959; also "Information Processing," UNESCO (Paris), R. Oldenbourg (Munich), and Butterworths (London), 1960, pp. 375–382.

number includes a number sign, a fraction, an exponent, an exponent sign, and bits for flagging numbers (Fig. 4.1). The fraction part of this unit might be made to vary widely in length, depending upon precision requirements, but the precision analysis that such variation would imply would often be as burdensome as the detailed magnitude analysis that floating-point operation eliminates. Moreover, these operations must proceed with the utmost speed, and a fixed format facilitates parallel arithmetic. For these reasons, floating-point numbers follow a rigid format. The datum is usually long—in this machine it uses 64 bits, with the fraction occupying 48 of these.

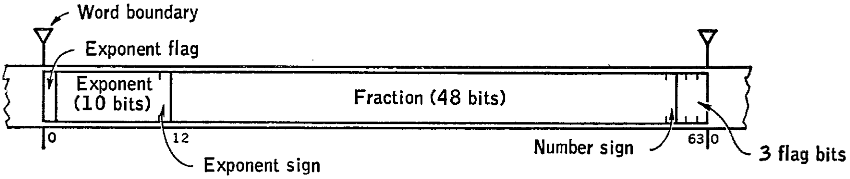
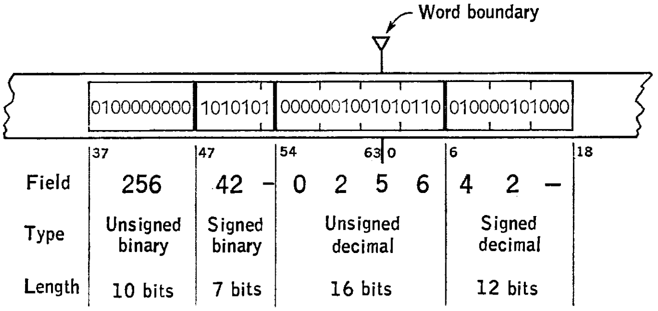


FIG. 4.1. Data unit for floating-point arithmetic.

Fixed-point arithmetic is used on problem data when magnitude analysis is trivial, such as that encountered in business or statistical calculations. Figure 4.2 shows some examples. Numbers may or may not be signed. If the arithmetic is binary, the data unit has a simple structure. If the arithmetic is decimal, the number has an inner structure of digits individually encoded in binary form. Whether the unit is simple or complex in structure, its natural length is quite variable, with typical numbers varying from 4 to 40 bits in length.

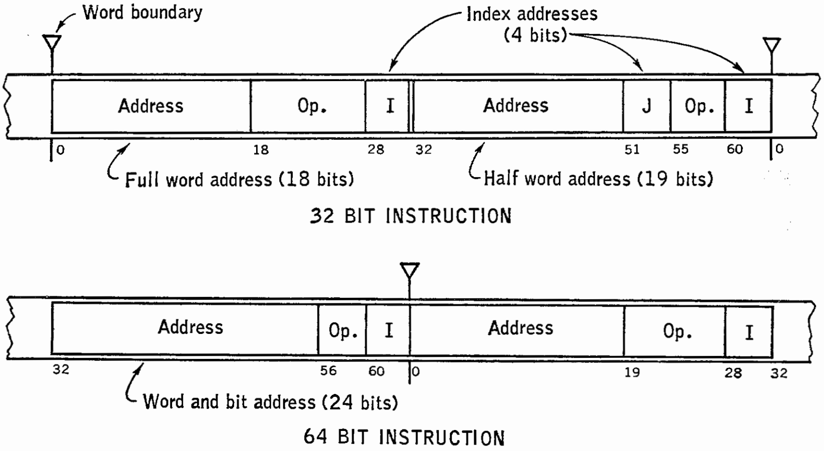


Field	256	42 −	0 2 5 6	4 2 −
Type	Unsigned binary	Signed binary	Unsigned decimal	Signed decimal
Length	10 bits	7 bits	16 bits	12 bits

FIG. 4.2. Data units for fixed-point arithmetic.

Address arithmetic operates upon a natural data unit whose structure is similar to that of unsigned fixed-point data, whether decimal or binary (Fig. 4.3). The unit has, however, one or a few standard lengths because of the fixed size of memory, and the length of the unit is relatively short.

Pure *logical manipulations*—whether used as the main part of a program, as in combinatorial analysis, or for controlling the course of the program—operate upon a very simple data-unit structure composed of a group of bits, each of which has an independent meaning (Fig. 4.4). This distinguishes such operations from arithmetic, which uses bits as components of numbers. Since in logical operations field length depends upon the number of operations that can be paralleled, lengths vary; but since no carries are propagated, restriction to fields of arbitrary lengths is not too burdensome.



32 BIT INSTRUCTION

64 BIT INSTRUCTION

FIG. 4.3. Data units for address arithmetic.

A final class, *editing operations*, includes all operations in which data are transformed from one format to another, checked for consistency with a source format, or tested for controlling the course of the program. The natural data unit for such operations varies widely (Fig. 4.5). All natural data units of the previous four types of operation undergo editing operations in the normal course of their processing, and there are other units unique to editing operations. For example, a group of data fields may be moved as a unit within memory to assemble records for output.

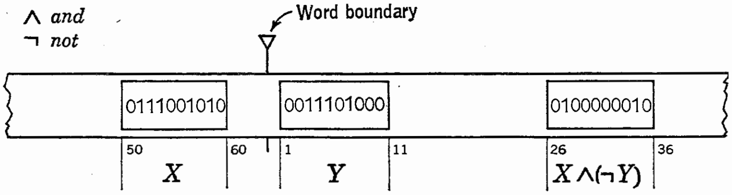


FIG. 4.4. Data units for logical manipulations.

Editing operations not only possess the most complex natural data structures, but they also use the most widely varying natural field lengths. For some manipulations, the natural unit is the single character; for other manipulations, such as comparison or transmission, the natural data unit is a field of many characters or a complete record.

Besides these five kinds of natural data units that can be identified for operations commonly built into computers, other natural data units are suitable for operations usually encoded with subroutines, such as matrix arithmetic, complex arithmetic, and multiple-precision arithmetic. As these larger units are necessarily composed of components that themselves are the data units of some built-in operation, they need not be considered separately.

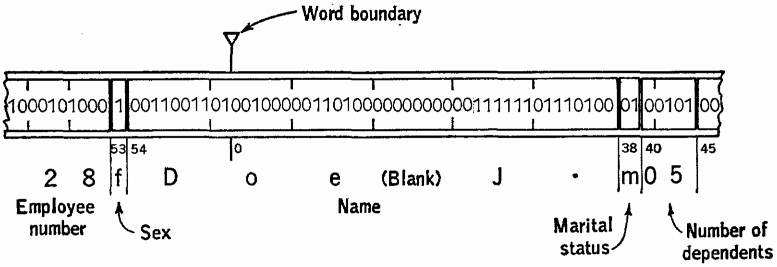


FIG. 4.5. Data units for editing operations.

4.2. Procedures for Specifying Natural Data Units

The previous section has shown how natural data units for different operations differ in structure, length, and variability of length. These diversities imply that more information is required for the specification of the natural data units than would be required if they were alike. The computer designer can choose the manner in which the user will pay this information price, but the price must be paid.

The data and instructions for any given problem may be considered to consist of a single stream of natural data units, without computer-prescribed spacers, groupings, etc. The computer designer must furnish a memory structure and an addressing system with which the individual components of a stream of natural data units are to be manipulated. The programmer must map the data-unit stream of his problem into a spaced and grouped stream suitable for the memory organization that the computer designer provides. This mapping requires some of the computer's power and necessarily introduces some inefficiencies. The more complex and difficult the mapping, the lower is the performance of the whole system.

The classical approach to this problem was to ignore it. For simplicity, early computer designers assumed (1) that provisions for handling the object data of fixed-point-arithmetic operations would suffice and (2) that the natural data unit for these operations was the single number of constant length. These two assumptions led to a simple, homogeneous, fixed-word-length memory organization. Since neither assumption was completely true, the information price of diversity was paid by the user in reduced performance and more complex programming.

When performing operations other than fixed-point arithmetic, such as editing and address arithmetic, the programmer shifted, extracted, and packed in order to get at the natural data unit of the operation. When faced with data of varying lengths, the programmer had two options as to the method of paying the information price. He could (1) place each unit in a different machine word or (2) pack several shorter units into a single word. (Since the machine word was usually picked to be a reasonable upper bound on natural data lengths, he was less often faced with the problem of manipulating units that required several words.) The price of using a different word for each data unit is reduced memory capacity and increased operating times for input-output and arithmetic units. The price of packing memory cells is paid in memory capacity for the packing instructions, in execution time, and in programming time.

Clearly, one way to improve the performance of a computer by changing its organization is to pay the price of diverse data units in the form of more complex hardware. This implies a memory structure that can be composed of variable-length cells. Several computers have been so organized. These computers have been intended primarily for business data processing, where editing operations are of great importance and where the assumption of constant-length data units is particularly poor. As the importance of nonarithmetical operations in all kinds of calculations became more apparent, a variable-cell-length memory organization became more desirable for any high-performance general-purpose computer.

There are several methods of achieving variable cell size. If the memory is to be addressed rather than scanned, the cell lengths may vary from cell to cell and from problem to problem, but the positions (and therefore the lengths) of cells must remain constant within a single computation. That is, cells at different addresses may have different lengths, but a change in the contents of a cell must not change its length. On tape, where scanning is used instead of addressing, this constraint does not hold, and some computers allow item lengths on tape to vary by deleting either leading numerical zeros or trailing alphabetic blanks.

A simple way of organizing a memory of different cell sizes is to provide a fixed complement of assorted sizes; this is done, for example, in the IBM 604 calculator. This rather inflexible arrangement was discarded for the IBM 7030 in favor of a second method, where the smallest data component is made addressable; a cell is defined by specifying both the position of one component in memory and the extent of the cell. Because of the requirements of pure logical operations and of editing operations, addressing resolution was provided all the way down to the individual bit level. Each bit in the memory has a unique address.

There are several techniques for specifying cell extent. The first is to use a unique combination of data bits as a partition between cells. This method is used to separate numerical fields in the IBM 705. The use of partition symbols implies reduced memory capacity due to the symbols themselves and, more seriously, exclusion of the partition-bit combination from the set of permissible data symbols. This difficulty alone would have precluded use of partitions between memory cells in the 7030. Arbitrary bit combinations arise in assembling instructions, reading data from external devices, and performing binary computations, and such activities could not be excluded. Furthermore, in any computer where memory speed is the limiting factor on performance, it is highly desirable that each bit fetched from memory contain 1 bit of information. Use of extra symbols and restrictions on bit combinations both reduce information content.

A variation of the partition-bit approach is to provide space for marker bits outside the data storage space. In the smaller IBM 1401 computer, for example, the cell size is variable to the character level, and the high-order end of a cell is indicated by a separate bit available in each character position. This is a simple technique to implement, and it avoids restrictions on the data symbols permissible. The obvious information price of this scheme is 1 extra bit per character. An additional price must be paid in instructions to set up and alter the positions of these marks, which, being extraterritorial in nature, are awkward to bring in from the input. Moreover, this approach becomes relatively more costly as data storage space increases in comparison to program storage space.

A third method of specifying cell extent is to use a Procrustean-bed technique in which data are transferred from memory to a register until the register is full. Transfers to memory likewise proceed until the register is completely copied. This technique is used for alphabetic fields in the 705. The disadvantage is that the technique requires extra instructions for changing the length of the receiving register or the use of several receiving registers of different lengths.

Pro·crus′tes (prô·krŭs′tēz), *n.* [L., fr. Gr. *Prokroustēs*, fr. *prokrouein* to beat out, to stretch, fr. *pro* forward + *krouein* to strike.] *Gr. Antiq.* A celebrated legendary highwayman of Attica, who tied his victims upon an iron bed, and, as the case required, either stretched or cut off their legs to adapt them to its length. Hence, ***the bed of Procrustes*** or ***Procrustean bed***, an idea, theory, or system to which facts, human nature, or the like, would be arbitrarily fitted.

(*By permission from Webster's "New International Dictionary," 2d ed., copyright* 1959 *by G. & C. Merriam Company, Springfield, Mass., publishers of the Merriam-Webster dictionaries.*)

A fourth technique, and that adopted, is to provide the information on cell extent in the instructions that use that cell. This can be done by specifying one of several masks, by specifying beginning and end, or by specifying beginning and length. In order to simplify indexing, the last method was selected. Each instruction that can refer to variable-length cells contains the complete address of the leftmost (high-order) bit of the cell and the length of the cell; however, instructions that do not need to refer to cells of varying length do not contain all this information.

4.3. Data Hierarchies

Most data-processing tasks involve a hierarchy of data units which, in ascending order of size, are frequently called *character*, *field*, *record*, and *file*. Each structural unit consists of one or more of the preceding units. The reason for the existence of this structure is that an association of meaningful data units may have a meaning of its own. To use a well-worn example, a payroll record consists of an employee identification number and related data, such as name, pay rate, and amounts, each of which is a field which, in turn, is made up of alphabetic or numerical characters. This record as a whole may be sorted into identification number sequence with other employees' records, if the fields remain associated with the identification; if the fields were all sorted individually, their meaning would be destroyed. Again, a file of last week's payroll records can be distinguished from a file of this week's records if they remain together.

It has been found useful to define a similar hierarchical structure for the machines that process the data, but often for different reasons. The number of bits transmitted in parallel at one time between the computer and input-output units is one such data unit; that transmitted in parallel between computer and memory is another, often different. Efficient operation of input-output units usually requires the definition of still larger groupings of data.

The distinction between the natural requirements of the data and those of the machine has often been obscured by the fact, already referred to, that the user may be forced to adapt his data to the characteristics of the machine. Thus the same terms are frequently used for both purposes. We prefer to use two sets of terms and to point out similarities by listing, side by side, terms that have a corresponding ranking:

Natural data hierarchy	*Machine data hierarchy*
Bit	Bit
Character	Byte
Field	Word
Record	Block
File	Reel of tape, tray of cards, web of paper, etc.

Bit is widely used in both contexts and, since it causes no confusion, the term will be retained for both.

Character is usually identified with a graphic symbol, such as a numerical digit, alphabetic letter, punctuation mark, or mathematical symbol.

Field denotes a group of characters processed together in a single numerical or logical operation. Examples are a number, a name, an address. A field is identified by its location in storage or in a record. (The term goes back to punched-card usage. *Item* has also been used.)

A *record* is a group of fields that are processed together. Corresponding fields in successive records normally occupy the same relative position within the record. A record is identified by one or more identifier fields or by its location in storage or in a file.

A *file* is a group of records, which are usually processed one record at a time. A file may be identified by an identifier record.

The actual usage of the above terms depends largely on the application, and many applications require additional steps in the hierarchy which may not have generic names.

Terms used here to describe the structure imposed by the machine design, in addition to *bit*, are listed below.

Byte denotes a group of bits used to encode a character, or the number of bits transmitted in parallel to and from input-output units. A term other than *character* is used here because a given character may be represented in different applications by more than one code, and different codes may use different numbers of bits (i.e., different byte sizes). In input-output transmission the grouping of bits may be completely arbitrary and have no relation to actual characters. (The term is coined from *bite*, but respelled to avoid accidental mutation to *bit.*)

A *word* consists of the number of data bits transmitted in parallel from or to memory in one memory cycle. Word size is thus defined as a structural property of the memory. (The term *catena* was coined for this purpose by the designers of the Bull GAMMA 60 computer.)

Block refers to the number of words transmitted to or from an input-output unit in response to a single input-output instruction. Block size is a structural property of an input-output unit; it may have been fixed by the design or left to be varied by the program.

4.4. Classes of Operations

Several classes of operations are provided in the 7030 to deal directly with different natural data units. In particular, the variable-field-length system to be described in Chap. 7 has been designed to overcome the limitations of the rigid word structure of the memory and permit the program to specify fields of any length, up to the rather high limit of

64 bits. This system is used for fixed-point-arithmetic, alphanumeric, and logical operations, since the data units for these classes of operations can be specified in the same way.

The floating-point operations (see Chap. 8) deal specifically with floating-point numbers. As has been mentioned, it is advantageous here to make the length of the floating-point number the same as that of the memory word.

Address arithmetic is performed primarily by indexing operations, which are discussed in Chap. 11, and these operations are designed to handle the various address lengths encountered in the 7030.

Editing operations require a combination of these classes of operations and others, like data transmission, that are not so readily classified. Data transmission and input-output operations (see Chap. 12) have the restriction that only full 64-bit words can be transmitted. Thus a record of a given natural length must be approximated by a block that is a multiple of 64 bits long. To save the few extra bits in the last word of a block would have greatly increased the amount of equipment and was not considered worth while.

Chapter 5

CHOOSING A NUMBER BASE

by W. Buchholz

5.1. Introduction

One of the basic choices the designers of a digital computer must make is whether to represent numbers in decimal or binary form. Many factors enter into this choice. Where high performance is a major goal, as in the IBM 7030, high arithmetical speed is of the essence and a proper choice of number system can contribute to arithmetical speed. But the over-all performance of a computer cannot be measured by its arithmetical speed alone; it is significantly affected by the ease with which nonarithmetical operations are performed. Equally important is the human factor. Unless the computer is programmed to assist in the preparation of a problem and in the presentation of results, false starts and waiting time can greatly dilute the effective performance of a high-speed computer. Regardless of the number system chosen for internal arithmetic, decimal numbers must be used in communicating between man and the computer.

Civilized man settled on 10 as the preferred number base for his own arithmetic a long time ago.[1] The ten digits of the decimal system had their origin when man learned to count on his ten fingers. The word *digit* is derived from the Latin word *digitus* for *finger* and remains to testify to the history of decimal numbers. Historically, several other number bases have been employed by various peoples at different times. The smaller number bases are clearly more awkward for human beings

Note: The material in Chap. 5 is taken from W. Buchholz, Fingers or Fists? (The Choice of Decimal or Binary Representation), *Communs. ACM*, vol. 2, no. 12, pp. 3–11, December, 1959.

[1] Although in most languages numbers are expressed by decimal symbols, it is a curious fact that there has been so far no standardization on multiples of 10 for units of money, length, weight, and time. We are still content to do much of our everyday arithmetic in what is really a mixed-radix system which includes such number bases as 3, 4, 7, 12, 24, 32, 60, 144, 1,760, etc.

to use because more symbols are needed to express a given number. Nevertheless, there is evidence of the use of the base 2, presumably by men who observed that they had two ears, eyes, feet, or fists.

With the decimal symbolism in universal use, it was natural that the earliest automatic digital computers, like the desk calculators and punched-card equipment that preceded them, should have been decimal. In 1946 John von Neumann and his colleagues at the Institute for Advanced Study, in their classical report describing the new concept of a stored-program computer, proposed to depart from that practice.[1] They chose the base 2 for their system of arithmetic because of its greater economy, simplicity, and speed.

Many designers have followed this lead and built binary computers patterned after the machine then proposed. Others have disagreed and pointed out techniques for obtaining satisfactory speeds with decimal arithmetic without unduly increasing the over-all cost of the computer. Since decimal numbers are easier to use, the conclusion has been drawn that decimal computers are easier to use. There have been two schools of thought ever since, each supported by the fact that both decimal and binary computers have been eminently successful.

As the Institute for Advanced Study report has long been out of print, it seems appropriate to quote at some length the reasons then given for choosing binary arithmetic:

> In spite of the longstanding tradition of building digital machines in the decimal system, we feel strongly in favor of the binary system for our device. Our fundamental unit of memory is naturally adapted to the binary system since we do not attempt to measure gradations of charge at a particular point in the Selectron [the memory device then proposed] but are content to distinguish two states. The flip-flop again is truly a binary device. On magnetic wires or tapes and in acoustic delay line memories one is also content to recognize the presence or absence of a pulse or (if a carrier frequency is used) of a pulse train, or of the sign of a pulse. (We will not discuss here the ternary possibilities of a positive-or-negative-or-no pulse system and their relationship to questions of reliability and checking, nor the very interesting possibilities of carrier frequency modulation.) Hence if one contemplates using a decimal system . . . one is forced into a binary coding of the decimal system—each decimal digit being represented by at least a tetrad of binary digits. Thus an accuracy of ten decimal digits requires at least 40 binary digits. In a true binary representation of numbers, however, about 33 digits suffice to achieve a precision of 10^{10}. The use of the binary system is therefore somewhat more economical of equipment than is the decimal.

[1] A. W. Burks, H. H. Goldstine, and J. von Neumann, "Preliminary Discussion of the Logical Design of an Electronic Computing Instrument," Institute for Advanced Study, Princeton, N.J., 1st ed. June, 1946, 2d ed., 1947, sec. 5.2; also subsequent reports by H. H. Goldstine and J. von Neumann.

> The main virtue of the binary system as against the decimal is, however, the greater simplicity and speed with which the elementary operations can be performed. To illustrate, consider multiplication by repeated addition. In binary multiplication the product of a particular digit of the multiplier by the multiplicand is either the multiplicand or null according as the multiplier digit is 1 or 0. In the decimal system, however, this product has ten possible values between null and nine times the multiplicand, inclusive. Of course, a decimal number has only $\log_{10} 2 \approx 0.3$ times as many digits as a binary number of the same accuracy, but even so multiplication in the decimal system is considerably longer than in the binary system. One can accelerate decimal multiplication by complicating the circuits, but this fact is irrelevant to the point just made since binary multiplication can likewise be accelerated by adding to the equipment. Similar remarks may be made about the other operations.
>
> An additional point that deserves emphasis is this: An important part of the machine is not arithmetical but logical in nature. Now logic, being a yes-no system, is fundamentally binary. Therefore a binary arrangement of the arithmetical organs contributes very significantly towards producing a more homogeneous machine, which can be better integrated and is more efficient.
>
> The one disadvantage of the binary system from the human point of view is the conversion problem. Since, however, it is completely known how to convert numbers from one base to another and since this conversion can be effected solely by the use of the usual arithmetic processes, there is no reason why the computer itself cannot carry out this conversion. It might be argued that this is a time-consuming operation. This, however, is not the case. . . . Indeed a general-purpose computer, used as a scientific research tool, is called upon to do a very great number of multiplications upon a relatively small amount of input data, and hence the time consumed in the decimal-to-binary conversion is only a trivial per cent of the total computing time. A similar remark is applicable to the output data.

The computer field and, along with it, the technical literature on computers have grown tremendously since this pioneering report appeared. It seems desirable, therefore, to bring these early comments up to date in the light of experience. The present discussion is also intended to widen the scope of the examination so as to reflect knowledge gained from increasing areas of application of the large computers. Mathematical computations are still important, but the processing of large files of business data has since become a major field. Computers are beginning to be applied to the control of planes in actual flight, to the collection and display of data on demand, and to language translation and systems simulation. Regardless of the application, a great deal of the time of any large computer is spent on preparing programs before they can be run on that computer. Much of this work is nonnumerical data processing. The point of view has thus shifted considerably since the days of the von Neumann report, and a reevaluation seems to be in order.

5.2. Information Content

Information theory[1,2] allows us to measure the information content of a number in a specific sense. Assume a set of N possible numbers, each of which is equally likely to occur during a computing process. The information H contained in the selection of a number is then

$$H = \log_2 N$$

Suppose, now, that a set of b binary digits (bits) represents a set of 2^b consecutive integers, extending from 0 to $2^b - 1$, each of these integers being equally probable. Then

$$\begin{aligned} H &= \log_2 2^b \\ &= b \text{ bits} \end{aligned}$$

(Because in this example the amount of information is equal to the number of bits needed to represent the integer in binary form, the bit is often chosen as the unit of information. The two uses of the term *bit* should not be confused, however. Numbers are defined independently of their representation, and the information content of a number is measured in bits regardless of whether the number is in binary, decimal, or any other form.)

Similarly, assume a set of 10^d consecutive integers from 0 to $10^d - 1$ expressed by d decimal digits. Here

$$\begin{aligned} H &= \log_2 10^d \\ &= d \log_2 10 = \frac{d}{\log_{10} 2} \\ &= 3.322d \text{ bits (approx.)} \end{aligned}$$

Thus a decimal digit is approximately equivalent in information content to 3.322 binary digits.

In the actual representation of a number N, both b and d must, of course, be integers. The ranges 10^d and 2^b cannot be compared exactly. For such pairs as $d = 3$ and $b = 10$, the values $10^3 = 1{,}000$ and $2^{10} = 1{,}024$ come very close to being equal. Here $b/d = {}^{10}\!/_3 = 3.333$ (approx.), which agrees well with the above value 3.322. This shows, at least, that the measure of information is a plausible one.

Conversely, to express a binary number requires approximately 3.322 times as many binary symbols (0 and 1) as decimal symbols (0 to 9).

[1] C. E. Shannon and W. Weaver, "The Mathematical Theory of Communication," The University of Illinois Press, Urbana, Ill., 1949.

[2] L. Brillouin, "Science and Information Theory," Academic Press, Inc., New York, 1956, pp. 3–4.

Few truly decimal switching and storage devices have found application in high-speed electronic computers; otherwise a decimal computer might be a great deal more compact than a corresponding binary computer. Generally, only binary (or on-off) devices are used; hence decimal digits must be encoded in binary form even in decimal computers.[1] Since bits cannot be split to make up the 3.322 bits theoretically required, at least 4 bits are needed to represent a decimal digit. Therefore, instead of being more compact, a decimal computer in fact requires at least $4/3.322 = 1.204$ times as many storage and switching elements in a large portion of its system. The reciprocal ratio, 3.322/4 or 83 per cent, might be considered to be the maximum storage efficiency of a decimal computer.

Four-bit coding of decimal digits is called *binary-coded decimal* (BCD) notation. Codes with more than 4 bits for each decimal digit are often used to take advantage of certain self-checking and other properties; the efficiency of such codes is correspondingly lower than 83 per cent.

The 83 per cent efficiency is only a theoretical value for even a 4-bit code. A basic assumption made in arriving at this value was that all the N possible numbers in the expression $\log_2 N$ were equally likely to occur. Nonuniform distributions are quite frequent, however. A common situation is that a set of b bits (in the binary case) is chosen to represent N integers from 0 to $N - 1$, $N < 2^b$, and the integers N to $2^b - 1$ are never encountered. The information content $\log_2 N$ may then be considerably less than b bits. Both binary and decimal computers suffer a loss of efficiency when the number range N is not a power of the number base.

For example, assume $N = 150$; that is, the numbers range from 0 to 149. Then

$$H = \log_2 150 = 7.23 \text{ bits}$$

Since 8 is the next largest integer, a binary computer requires at least 8 bits to represent these numbers, giving an efficiency of 7.23/8 or 90 per

[1] The universal use of binary elements is based on practical engineering considerations, but under certain crude assumptions it can be shown that 2 is also a near-optimum radix theoretically. Let a given number N be represented in radix r by n radix positions; that is, $N = r^n$. Assume the cost of each radix position to be proportional to the radix, so that the cost C of representing N is

$$C = krn = kr\frac{\log_e N}{\log_e r}$$

Assume further that r and n could be continuously variable; then setting $dC/dr = 0$ gives a minimum cost for $r = e$. The nearest integral radixes are 2 and 3, and their value of C is not much greater than the minimum. Although ternary arithmetic is an interesting possibility, there has been little incentive to develop ternary devices in practice.

cent. A decimal computer requires at least three decimal digits or 12 bits, with an efficiency of 7.23/12 or 60 per cent. Relative to the binary number base, the efficiency of decimal representation is only 60/90 or 67 per cent.

The loss in efficiency is greatest for the smaller integers. With binary integers the lowest efficiency of 78 per cent occurs for $N = 5$. Decimal representation has its lowest efficiency of 25 per cent at $N = 2$. Decimal representation is never more efficient than binary representation, and only for $N = 9$ and $N = 10$ are they equally efficient.

Figure 5.1 shows the storage efficiency curves for binary and decimal systems, and Fig. 5.2 shows the efficiency of the decimal representation relative to the binary system.

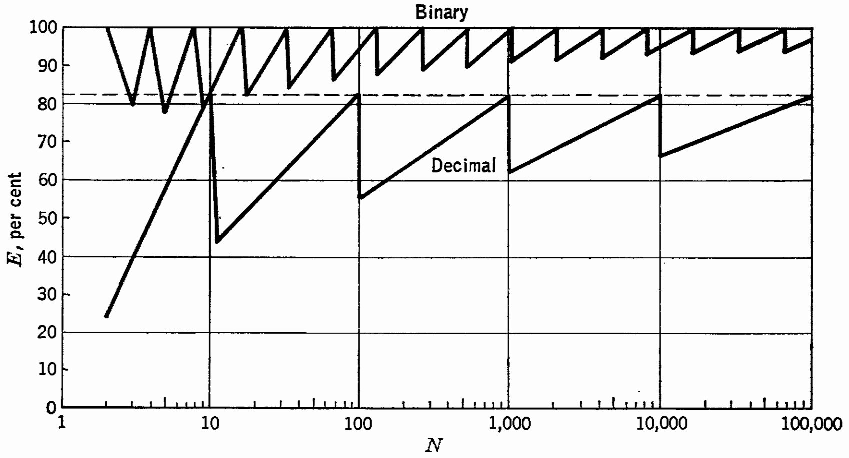


FIG. 5.1. Absolute efficiency of decimal and binary number systems. $E = (\log_2 N)/b$, where b is the least number of bits to represent N.

For the above analysis a variable-field-length operation was assumed where the least possible number of bits or decimal digits can be assigned to represent the maximum value of N. A great many computers are designed around a fixed word length, and even more space will then be wasted unless time is taken to program closer packing of data. It was also assumed that the N integers considered were distributed uniformly throughout the interval; a nonuniform distribution with numbers missing throughout the interval results in a further lowering of storage efficiency, which affects binary and decimal computers alike.

Although only integers have been considered so far, the same reasoning obviously applies to fractions truncated to a given precision, since these are treated in storage in the same manner as integers. Similarly, the sign of a number may be regarded as an integer with $N = 2$. Instructions are always made up of a number of short, independent pieces. For example, an operation code for 45 different operations may be encoded as a set of integers with $N = 45$, for which the binary efficiency is 92 per cent and the decimal efficiency only 69 per cent.

The lower information-handling efficiency of the decimal representation may reflect itself in higher cost, in lower performance, or both. If performance is to be maintained, the cost will go up, but it would be wrong to assume that the extra bits required for decimal representation mean a proportional increase in cost. The ratio of the cost of storage, registers, and associated switching circuits to the total cost of a computer depends greatly on the design. Factors other than hardware cost need to be considered in estimating the over-all cost of using a computer on a given job.

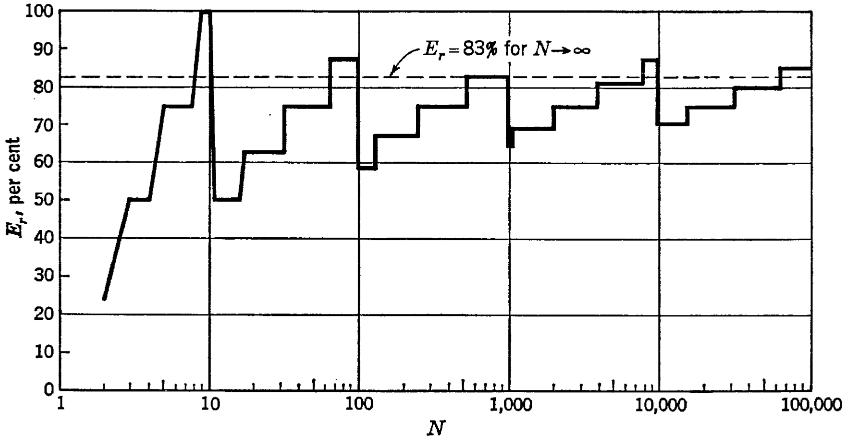


FIG. 5.2. Relative efficiency of decimal and binary number systems. $E_r = b_2/b_{10}$, where b_2 (b_{10}) is the least number of bits in the binary (decimal) representation of N.

When the cost is to be the same, a lower storage efficiency may result in lower performance. Thus the performance of many storage devices, such as magnetic tape, is limited by the bit transmission rate, so that the greater storage space occupied by decimal numbers, as compared to equivalent binary numbers, is reflected in a corresponding loss of speed. This may be important for applications in which the transmission rate to and from tape, or other *external storage*, is the limiting time factor: a binary computer is clearly at least 20 per cent faster than a corresponding decimal computer on a tape-limited job of processing numerical data.

Similarly, in many other applications the rate of information (data and instruction) flow out of and into the internal *memory* will be a major limiting factor, particularly for a computer designed to achieve the highest practicable performance with given types of components. Although it can be very misleading to compare two dissimilar computers on the basis of memory speed only, the comparison is appropriate for two computers using similar components and organization but differing mainly in their number representation.

A memory in active use may be looked on as an information channel with a capacity of

$$C = nw \qquad \text{bits per second}$$

where n is the number of bits in the memory word and w is the maximum number of words per second that the memory can handle.

This channel capacity is fully utilized only if the words represent numbers from 0 to $2^n - 1$, each of which is equally probable. If the information content is less than that, the actual performance is limited to Hw, where H is defined as before. More specifically, if a memory word is divided into k fields, of range $N_1, N_2, N_3, \ldots, N_k$, then

$$H = \sum_{i=1}^{k} \log_2 N_i$$

The maximum performance is lowered by the factor

$$\frac{H}{C} = \frac{\Sigma \log_2 N_i}{n}$$

For $k = 1$, this is the same factor as the storage efficiency described above.

Other organizational factors may reduce performance further, and memory multiplexing can be used to increase over-all performance. These matters are independent of the number representation. The fact remains that a decimal organization implies a decided lowering of the maximum performance available. By increasing the number of components this loss can be overcome only in part, because of physical and cost limitations.

In summary, to approach the highest theoretical performance inherent in a given complement of components of a given type, it is necessary to make each bit do 1 bit's worth of work.

5.3. Arithmetical Speed

A binary arithmetic unit is inherently faster than a decimal unit of similar hardware complexity operating on numbers of equivalent length. Whereas the gain in speed of binary over decimal arithmetic may not be significant in relatively simple computers, it is substantial when the design is aimed at maximum speed with a given set of components. There are several reasons why binary arithmetic is faster.

1. The cumulative delay in successive switching stages of an adder places a limit on the attainable speed, and the more complex decimal

adder requires more levels of switching than a binary adder for numbers of similar precision. Carry propagation, if any, also takes longer in a decimal adder because decimal numbers are longer.

2. With a base of 2, certain measures can be taken to speed up multiplication and division. An example is the skipping of successive 0s or 1s in the multiplier. When corresponding measures are taken with base 10 arithmetic, they are found to give a smaller ratio of improvement. Thus the average number of additions or subtractions needed during multiplication or division is greater, and this difference is compounded by the extra time needed for each addition or subtraction.

3. Scaling of numbers, which is required to keep numbers within the bounds of the registers during computation, results in a greater round-off error when the base is 10. The finest step of adjustment is 3.3 times as coarse in shifting by powers of 10 as it is with powers of 2. In large problems the greater error will require more frequent use of multiple-precision arithmetic, at a substantial loss of speed. This effect is partly offset by the fact that scaling will occur more often in binary arithmetic, and the extra shifting takes more time.

4. Multiplying or dividing by powers of the number base is accomplished by the fast process of shifting. The coefficients 2 and $\frac{1}{2}$ are found much more frequently in mathematical formulas than other coefficients, including 10 and $\frac{1}{10}$, and a binary computer has the advantage here.

To overcome the lower speed inherent in decimal arithmetic, it is, of course, possible to construct a more complex arithmetic unit at a greater cost in components. If top speed is desired, however, the designer of a binary arithmetic unit will have taken similar steps. There is a decided limit on this acceleration process. Not only does the bag of tricks run low after a while, but complexity eventually becomes self-defeating. Greater complexity means greater bulk, longer wires to connect the components, and more components to drive the longer wires. The longer wires and additional drivers both mean more delays in transmitting signals, which cannot be overcome by adding even more components. When the limit is reached there remains the substantial speed differential between binary and decimal arithmetic, as predicted by theoretical considerations in Sec. 5.1.

5.4. Numerical Data

Numerical data entering or leaving a computer system are of two kinds: (1) those which must be interpreted by humans and (2) those which come from or actuate other machines. The first are naturally in decimal form. The second class, which occurs when a computer is part of an automatic control system, could also be decimal, since machines, unlike human beings, can readily be designed either way; but binary coding is generally simpler and more efficient.

The previously cited von Neumann report considered only the important applications where the volume of incoming and outgoing data is small compared with the volume of intermediate results produced during a computation. In a fast computer any conversion of input and output data may take a negligible time, whereas the format of intermediate results has a major effect on the over-all speed. The von Neumann report did not consider the equally important data-processing applications in which but few arithmetical steps are taken on large volumes of input-output data. If these data are expressed in a form different from that used in the arithmetic unit, the conversion time can be a major burden. Any conversion time must be taken into account as reducing the effective speed of the arithmetic unit.

The choice would appear simple if different computers could be applied to different jobs, using decimal arithmetic when the data were predominantly decimal and binary arithmetic elsewhere. Experience has shown, however, that a single large computer is often used on a great variety of jobs that cannot be classified all one way or the other. Moreover, as will be shown in subsequent sections, there are strong reasons for choosing a binary addressing system even where the applications indicate the use of decimal data arithmetic. Some kind of binary arithmetic unit must then be provided anyway, if only to manipulate addresses.

A high-speed binary arithmetic unit is thus clearly desirable for all applications. To handle decimal data, the designer may choose to provide a separate decimal arithmetic unit in the same computer, or he may prefer to take advantage of the speed of his binary arithmetic unit by adding special instructions to facilitate binary-decimal conversion.

The decimal arithmetic and conversion facilities must take into account not only the different number base of decimal data but also the different format. Binary numbers usually consist of a simple string of numerical bits and a sign bit. Decimal numbers are frequently interspersed with alphabetic data, and extra *zone* bits (sometimes a separate digit) are then provided to distinguish decimal-digit codes from the codes for alphabetic and other characters. The separate treatment of numerical and zone portions of coded digits greatly adds to the difficulty of doing conversion by ordinary arithmetical instructions. Hence the decimal arithmetic and conversion instructions should be designed to process decimal data directly in a suitable alphanumeric code.

5.5. Nonnumerical Data

A computer may have to process a large variety of nonnumerical information:

1. Character codes representing alphabetic, numerical, or other symbols for recording data in human-readable form
2. Codes used to perform specified functions, such as terminating data transmission
3. Yes-no data ("married," "out of stock," etc.)
4. Data for logical and decision operations
5. Instructions (other than numerical addresses)
6. Machine-status information, such as error indications
7. Status of switches and lights

Because the storage and switching elements normally used in computers are binary in nature, all information, numerical or nonnumerical, is encoded in a binary form. This binary coding has no direct relation to the number base being used for arithmetic. The number base determines the rules of arithmetic, such as how carries are propagated in addition, but it has no meaning in dealing with nonnumerical information. Thus the binary-decimal distinction does not apply directly to the nonarithmetical parts of a computer, such as the input-output system.

Even where mathematical computation on numerical data is the major job, a great deal of computer time is usually spent on nonnumerical operations in preparing programs and reports. It is important, therefore, that the designer avoid constraints on the coding of input and output data, such as are found in many existing decimal computers. Many of these constraints are unnecessary and place extra burdens of data conversion and editing at greater cost on peripheral equipment.

5.6. Addresses

Memory addresses are subject to counting and adding and are thus proper numbers which can be expressed with any number base. Base 10 has the same advantage for addresses as for data: conversion is not required, and actual addresses can be continuously displayed on a console in easily readable form.

The compactness of binary numbers is found particularly advantageous in fitting addresses into the usually cramped instruction formats (see Chap. 9). Tight instruction formats contribute to performance by reducing the number of accesses to memory during the execution of a program as well as by making more memory space available for data. The low efficiency of decimal coding for addresses has already led designers of nominally decimal computers to introduce a certain amount of binary coding into their instruction formats. Such a compromise leads to programming complications, which can be avoided when the coding is purely binary.

Although the compactness of the binary notation is important, the most significant advantage of binary addressing is probably the ease of performing data transformation by address selection (table look-up). This is discussed in the next section.

5.7. Transformation

A single data-processing operation may be regarded as transforming one or more pieces of data into a result according to certain rules. The most general way of specifying the rules of transformation is to use a set of tables. The common transformations, such as addition, multiplication, and comparison, are mechanized inside the computer, and some others, such as code conversion, are often built into peripheral equipment; tables (sometimes called *matrixes*) may or may not be employed in the mechanization. All transformations not built into the computer must be programmed.

In a computer with a large rapid-access internal memory, the best transformation procedure, and often the only practical one, is *table look-up*. Each piece of data to be transformed is converted to an address which is used to select an entry in a table stored in memory. (This method of table look-up is to be distinguished from *table searching*, where all entries are scanned sequentially until a matching entry is found.) Table 5.1 serves to illustrate the process by a code-translation example.

Two methods of encoding the digits 0 to 9, both in current use, are shown in Table 5.1. One is a 2-out-of-5 code which requires 5 bits for every digit. Two and only two 1 bits are contained in each digit code, with all other 5-bit combinations declared invalid. This property permits checking for single errors and for common multiple errors. The second code is a 4-bit representation using codes 0001 to 1001 for the digits 1 to 9 and 1010 for the digit 0. Codes 0000 and 1011 to 1111 are not used.

For translation from the 5-bit code to the 4-bit code, a table of 32 (2^5) entries is stored in successive memory locations. Each entry contains a 4-bit code. Where the 5-bit combination is a valid code, the corresponding 4-bit code is shown. All invalid 5-bit combinations are indicated in the example by an entry of 1111, which is not a valid 4-bit code.

The example in Table 5.1 consists in adding a given 5-bit code 10001 to the address of the first entry, the table base address. The sum is the address in the table of the desired entry, which is seen to be 0111. If the entry had been 1111, the incoming code would have been known to contain an error.

The key to this transformation process is the conversion of data to addresses. A system capable of receiving, transforming, and transmitting any bit pattern can communicate readily with any other system, including equipment and codes over which the designer has no control.

The desire to accept any bit pattern as an address almost dictates binary addressing. It is true that decimal addressing does not entirely preclude transformation of arbitrary data by indirect methods, but such methods are very wasteful of time or memory space.

TABLE 5.1. EXAMPLE OF CODE TRANSLATION BY TRANSFORMATION

Two codes for decimal digits			*Translation table, code A to code B*	
Symbols	*Code A* (5 bits)	*Code B* (4 bits)	*Address*	*Entry*
1	00011	0001	...100000	1111
2	00101	0010	...100001	1111
3	00110	0011	...100010	1111
4	01001	0100	...100011	0001
5	01010	0101	...100100	1111
6	01100	0110	...100101	0010
7	10001	0111	...	...
8	10010	1000	...101110	1111
9	10100	1001	...101111	1111
0	11000	1010	...110000	1111
			...110001	0111
			...110010	1000
			...110011	1111
			...	...
			...111111	1111

Example: Translation of Symbol "7"

	...100000	Table base address
+	**10001**	Incoming 5-bit code
(Sum)	...110001	Address of table entry

5.8. Partitioning of Memory

It has already been mentioned that the binary radix makes it possible to scale numbers in smaller steps and thus reduce loss of significance during computation. Binary addresses also have this advantage of greater resolution. Shifting binary addresses to the left or right makes it easy to divide memory into different areas, or *cells*, whose sizes are adjustable by powers of 2. With decimal addressing such partitioning is easily obtained only by powers of 10.

In a core memory, for example, each address refers to a memory word consisting of the number of parallel bits that are accessible in a single memory cycle. Since binary addressing of these memory words had been chosen for reasons given in previous sections, there was then considerable advantage to choosing the number of bits in each word to be a power of 2. In the 7030 this word length was set at 2^6, or 64 bits. (This particular power of 2 gave a good compromise between speed and cost of memory and provided ample space for representing a floating-point number in one memory word. Thirty-two bits was too short and 128 bits too long.) Individual bits in a 64-bit memory word can be addressed simply by extending the address and inserting 6 bits to the right of the word address to operate a bit-selection mechanism. When increments are added to these addresses in binary form, whether by explicit instructions or by indexing, carries from the sixth to the seventh bit automatically advance the word address.

The flexibility of bit addressing may be illustrated by enlarging the example of Table 5.1. Instead of using an entire memory word to hold one 4-bit table entry, it is possible to use for the same entry a cell only 4 bits long, with sixteen cells in each memory word of 64 bits. With respect to the bit address, the incoming code is shifted 2 bits to the left to obtain increments of 4 bits of storage in memory:

$$\begin{array}{rrl} & \ldots 10000000 & \text{Table base address}\\ + & \mathbf{10001}00 & \text{Incoming 5-bit code with two 0s added}\\ \hline \text{(Sum)} & \underbrace{\ldots 11}_{\substack{\text{Address}\\ \text{of word}}}\underbrace{000100}_{\substack{\text{Address of}\\ \text{bit in word}}} & \text{Address of table entry}\end{array}$$

The example can be readily changed to translate from a 5-bit code to a 12-bit code, such as is used on punched cards. Without an actual table being shown, it is evident that the 12-bit code can be conveniently stored in successive 16-bit cells. The proper addresses are then obtained by inserting four 0 bits at the right, instead of two as before:

$$\begin{array}{rrl} & \ldots 1000000000 & \text{Table base address}\\ + & \mathbf{10001}0000 & \text{Incoming 5-bit code with four 0s added}\\ \hline \text{(Sum)} & \underbrace{\ldots 1100}_{\substack{\text{Address}\\ \text{of word}}}\underbrace{010000}_{\substack{\text{Address of}\\ \text{bit in word}}} & \text{Address of table entry}\end{array}$$

Similarly, the process can be extended to finer divisions. By using the incoming code as the address of a single bit, it is possible to look up a compact table of yes-no bits in memory to indicate, for example, the single fact of whether the code is valid or not.

Now consider these examples in terms of decimal addressing. If single bits were to be addressed, the next higher address digits would address every tenth bit. This is too large a cell size to permit the addressing of every decimal digit in a data field. To be practical in large-scale numerical computation, the code for a decimal digit cannot occupy a cell of more than 4, 5, or at most 6 bits. When the addressing is chosen to operate on cells of this size, direct addressing of single bits is ruled out. Table entries requiring more than one cell cannot occupy less than ten cells.

The designer of a binary computer may or may not choose to endow it with the powerful facility of addressing single bits (bit addressing) and provide for automatic modification of bit addresses (bit indexing). The point remains that the flexible partitioning of memory available to him would not have been available with decimal addressing.

5.9. Program Interpretation

A major task in any computer installation is the preparation and check-out of programs. Printing a portion of a program at the point where an error has been found is a common check-out tool for the programmer. Interpreting such a print-out is greatly simplified if the instructions are printed in the language that the programmer used.

At first glance this seems to be a convincing argument for decimal computers. On closer examination it becomes evident that both binary and decimal machines would be difficult to use without the assistance of adequate service programs. When good service programs are available to assist the user, it is hard to see how the number base in the arithmetic unit makes much difference during program check-out.

One reason for service programs is that in practice much programming is done in symbolic notation, regardless of the number base used internally. The programmer's language is then neither binary nor decimal; it is a set of alphanumeric mnemonic symbols. Conversion to or from the symbolic notation by means of a service program is desirable for any user of either kind of machine, with the possible exception of the programming specialist who writes programs in machine language either by choice or in order to develop new service programs.

Another and more basic reason for service programs is that most computers have more than one format for data and instructions, and a service program is needed to help interpret these formats. In binary computers it is desirable to know whether a data field is an integer or a floating-point number with its separate exponent (integer) and fraction. The instructions are normally divided differently from either kind of data field. A knowledge of the divisions of each format is required in converting from binary to decimal form.

Many decimal computers do not use purely decimal coding for the instructions, particularly those aimed at efficient processing of large amounts of nonnumerical business data. Moreover, alphanumeric character coding usually employs a convention different from that used in the coding of instructions. Again, a service program is needed to interpret the different data and instruction languages.

Table 5.2 illustrates this point with print-outs of actual computer programs. The first example is for an IBM 704, which uses binary arithmetic. The service program lists memory locations and instructions in octal form with the appropriate instruction bits also interpreted as alphabetic operation codes. The service program distinguishes floating-point numbers, which are listed in a decimal format with separate exponent, mantissa, and signs.

TABLE 5.2. EXAMPLES OF PROGRAM PRINT-OUTS

Print-out from IBM 704

Location	*Instruction or data*				
00622	FSB	0	30200	0	00637
00623	TZE	0	10000	0	00626
00624	TPL	0	12000	0	00607
00625	STO	0	60100	0	00634
00626	HTR	0	00000	0	00561
00627	−01	+	9.945		2245
00630	+03	+	4.130		0000
00631	−01	+	7.330		4100
00632	+05	+	5.301		7842

Print-out from IBM 705

Location	*Straight print-out*	*Print-out modified for instructions*		
01204	8 T L − 1	8	13301	10
01209	4 / Q R 1	4	11891	10
01214	L 1 0 9 4	L	1094	
01219	H W 5 R 4	H	16594	02
01224	7 W 6 N 5	7	16655	02
01229	1 2 4 4 9	1	2449	
. . .	. . .		. . .	
11304	I S P A	I	12 7A	14
11309	G E W A	G	35 6A	13
11314	S P R O	S	3 79O	10
11319	C E S S E	C	3522E	05
11324	D T H R	D	3 38R	07
11329	O U G H	O	1478	15

The second illustration shows a print-out from the IBM 705, a computer with decimal arithmetic and with alphanumeric coding for data. Each alphanumeric character has a unique 6-bit code. For reasons of storage efficiency, instructions in the 705 use a different code where some of the bits in a 6-bit character have independent meanings. In the example shown in Table 5.2, this dual representation is overcome by printing the program and data twice, once for ease of reading data and once for ease of interpreting instructions. A service program was needed to accomplish this.

The objection might be raised that the examples show up problems in existing machine organizations rather than a need for service programs. It is actually possible for "numerical engines" aimed at processing only numerical data to escape the problem of dual representation for instructions and data. When alphanumeric data must also be processed in a reasonably efficient manner, however, one cannot avoid the problem of dual representation.

5.10. Other Number Bases

Only binary and decimal computers have been considered here. Although it is clear that other number bases could be selected, they would all require translation to and from decimal formats, and they would be no more efficient than base 2.

5.11. Conclusion

The binary number base has substantial advantages in performance and versatility for addresses, for control data that are naturally in binary form, and for numerical data that are subjected to a great deal of arithmetical processing. Figures of merit are difficult to assign because the performance and cost of a given computer design depend on a great many factors other than the number base. It is clear, however, that decimal representation has an inherent loss in performance of at least 20 to 40 per cent as compared with binary representation and that refined design with increased cost can overcome this loss only in part. The decrease in efficiency makes itself felt in a number of ways; so the combined effect on over-all performance may be even greater than the percentage indicated.

It is equally clear, however, that a computer that is to find application in the processing of large files of information and in extensive man-machine communication must be adept at handling data in human-readable form; this form includes decimal numbers, alphabetic descriptions, and punctuation marks. Since the volume of data may be great, it is important that binary-decimal and other conversions should not become a burden greatly reducing the effective speed of the computer.

Hence it was decided to combine in the design of the IBM 7030 the advantages of binary and decimal number systems. Binary addressing has been adopted for its greater flexibility; each bit in memory has a separate address, and the length of a word in memory is a power of 2 (64 bits). Binary arithmetical operations are provided for manipulating these addresses and for performing floating-point arithmetic at extremely high speed. Efficient binary-decimal conversion instructions minimize the conversion time for input and output data intended for use in extensive mathematical computation. Decimal arithmetic is also included in the instruction repertoire, in order to permit simple arithmetical operations to be performed directly on data in binary-coded decimal form.

Such a combination of binary and decimal arithmetic in a single computer provides a high-performance tool for many diverse applications. It may be noted that a different conclusion might be reached for a computer with a restricted range of functions or with performance goals limited in the interest of economy; the difference between binary and decimal operation might well be considered too small to justify incorporating both. This conclusion does appear valid for high-performance computers, regardless of whether they are aimed primarily at scientific computing, business data processing, or real-time control. To recommend binary addressing for a computer intended for business data processing is admittedly a departure from earlier practice, but the need for handling and storing large quantities of nonnumerical data makes the features of binary addressing particularly attractive. In the past, the real obstacle to binary computers in business applications has been the difficulty of handling inherently decimal data. Binary addressing and decimal data arithmetic, therefore, make a powerful combination.

Chapter 6

CHARACTER SET

by R. W. Bemer and W. Buchholz

6.1. Introduction

Among the input and output devices of a computer system, one can distinguish between those having built-in codes and those largely insensitive to code. Thus typewriters and printers necessarily have a fixed code that represents printable symbols to be read by the human eye; a code must be chosen for such a device in some more or less arbitrary fashion, and the device must make the transformation between code and symbol. Data storage and transmission devices, on the other hand, such as magnetic tape units and telephone transmission terminals, merely repeat the coded data given to them without interpretation, except that some code combinations may possibly be used to control the transmission process. (Strictly speaking, storage and transmission devices do generally limit the code structure in some respect, such as maximum byte size, so that code sensitivity is a matter of degree.)

For the inherently code-sensitive devices to be attached to a new computer system, an obvious choice of character set and code would have been one of the many sets already established. When the existing sets were reviewed, however, none were found to have enough of the system characteristics considered desirable. In fact, it became clear that about the only virtue of choosing an already established set is that the set exists. Accordingly, it was decided, instead, to devise a new character set expressly for use throughout a modern computer system, from input to output. The chief characteristic of this set is its extension to many more different characters than have been available in earlier sets. The extended set designed for the 7030 (Fig. 6.1) contains codes for 120 different characters, but there is room for later expansion to up to 256 characters including control characters. In addition, useful subsets have been defined, which contain some but not all of these 120 characters and which use the same codes for the selected characters without translation.

It should be noted that the 7030 computer is relatively insensitive to the specific choice of code, and any number of codes could be successfully used in the system. For any particular application a specialized character code might be found superior. In practice, however, a large computer installation must deal with a mixture of widely different applications, and the designers have to choose a single character set as a compromise among conflicting requirements.

	Bits 0-1-2-3							
Bits 4-5-6-7	0000	0001	0010	0011	0100	0101	0110	0111
0000	Blank	[	&	c	k	s	0	8
0001	±	⊃	+	C	K	S	$_0$	$_8$
0010	→	]	$	d	l	t	1	9
0011	≠	°	=	D	L	T	$_1$	$_9$
0100	∧	←	*	e	m	u	2	.
0101	{	≡	(	E	M	U	$_2$	:
0110	↑	¬	/	f	n	v	3	–
0111	}	√	)	F	N	V	$_3$	?
1000	∨	%	,	g	o	w	4	
1001	∀	\	;	G	O	W	$_4$	
1010	↓	◊	'	h	p	x	5	
1011	||	|	"	H	P	X	$_5$	
1100	>	#	a	i	q	y	6	
1101	≧	!	A	I	Q	Y	$_6$	
1110	<	@	b	j	r	z	7	
1111	≦	~	B	J	R	Z	$_7$	

FIG. 6.1. 120-character set.

The purpose of this chapter is to list major requirements of a character set and code, and to point out how these requirements may or may not be met by the specific set to be described.

6.2. Size of Set

Present IBM 48-character sets consist of

1. 10 decimal digits
2. 26 capital letters
3. 11 special characters
4. 1 blank

Other manufacturers have employed character sets of similar or somewhat larger size.

Because a single set of eleven special characters is not sufficient, there exist several choices of special characters as "standard options."

Since this 48-character set is often represented by a 6-bit code, it is natural to try to extend it to 63 characters and a blank, so as to exploit the full capacity of a 6-bit code.[1] Although the extra sixteen characters would indeed be very useful, this step was thought not to be far-reaching enough to justify development of the new equipment that it would require.

As a minimum, a new set should include also:

5. 26 lower-case letters
6. The more important punctuation symbols found on all office typewriters
7. Enough mathematical and logical symbols to satisfy the needs of such programming languages as ALGOL[2,3]

There is, of course, no definite upper limit on the number of characters. One could go to the Greek alphabet, various type fonts and sizes, etc., and reach numbers well into the thousands. As set size increases, however, cost and complexity of equipment go up and speed of printing goes down. The actual choice of 120 characters was a matter of judgment; it was decided that this increment over existing sets would be sufficiently large to justify a departure from present codes and would not include many characters of only marginal value.

6.3. Subsets

Two subsets of 89 and 49 characters were chosen for specific purposes. The 89-character set (Fig. 6.2) is aimed at typewriters, which, with 44

[1] H. S. Bright, Letter to the Editor, *Communs. ACM*, vol. 2, no. 5, pp. 6–9, May, 1959 (a 64-character alphabet proposal).

[2] A. J. Perlis and K. Samelson, Preliminary Report: International Algebraic Language, *Communs. ACM*, vol. 1, no. 12, December, 1958.

[3] Peter Naur (editor), Report on the Algorithmic Language ALGOL 60, *Communs. ACM*, vol. 3, no. 5, May, 1960.

character keys, a case shift, and a space bar, can readily handle 89 characters. This subset was considered important because input-output typewriters can already print 89 characters without modification, and 44-key keyboards are familiar to many people.

The 49-character subset (Fig. 6.3) is the conventional set of "commercial" characters in a code compatible with the full set.[1] This subset is aimed at the *chain printer* mechanism used with the 7030, which can readily print character sets of different sizes but prints the larger sets at a reduced speed. The 49-character subset permits high-volume printing at high speed in a compatible code on jobs (such as bill printing) where the extra characters of the full set may not be needed. It should be noted that the 49-character set is not entirely a subset of the 89-character set.

Other subsets are easily derived and may prove useful. For example, for purely numerical work, one may wish to construct a 13-character set consisting of the ten digits and the symbols . (point) and – (minus), together with a special blank.

6.4. Expansion of Set

Future expansion to a set larger than 120 can take place in two ways. One is to assign additional characters to presently unassigned codes; allowance should then be made for certain control codes which will be needed for communication and other devices and which are intended to occupy the high end of the code sequence.

The second way is to define a shift character for "escape" to another character set.[2] Thus, whenever the shift character is encountered, the next character (or group of characters) identifies a new character set, and subsequent codes are interpreted as belonging to that set. Another shift character in that set can be used to shift to a third set, which may again be the first set or a different set. Such additional sets would be defined only if and when there arose applications requiring them.

6.5. Code

In choosing a code structure, many alternatives were considered. These varied in the basic number of bits used (i.e., the byte size) and in the number of such bytes that might be used to represent a single (print-

[1] Note that this is one character larger than the previously referred-to 48-character set. The additional special character was introduced in 1959 on the printer of the IBM 1401 system; but its use has not become firmly established, partly because it has no counterpart on the keypunch. Thus the 48- and 49-character sets are, in effect, the same set.

[2] R. W. Bemer, A Proposal for Character Code Compatibility, *Communs. ACM*, vol. 3, no. 2, February, 1960.

Bits 4-5-6-7	Bits 0-1-2-3							
	0000	0001	0010	0011	0100	0101	0110	0111
0000	Blank		&	c	k	s	0	8
0001			+	C	K	S	0	8
0010			$	d	l	t	1	9
0011			=	D	L	T	1	9
0100			*	e	m	u	2	.
0101			(	E	M	U	2	:
0110			/	f	n	v	3	–
0111			)	F	N	V	3	?
1000			,	g	o	w	4	
1001			;	G	ʘ	W	4	
1010			'	h	p	x	5	
1011			"	H	P	X	5	
1100			a	i	q	y	6	
1101			A	I	Q	Y	6	
1110			b	j	r	z	7	
1111			B	J	R	Z	7	

FIG. 6.2. 89-character set.

able) character. Among the alternatives were the following:

Single 6-bit byte with shift codes interspersed
Double 6-bit byte = single 12-bit byte[1]
Single 8-bit byte
Single 12-bit byte for "standard" characters (punched-card code) and two 12-bit bytes for other characters

Some of these codes represented attempts to retain partial compatibility with earlier codes so as to take advantage of existing equipment.

[1] R. W. Bemer, A Proposal for a Generalized Card Code for 256 Characters, *Communs. ACM*, vol. 2, no. 9, September, 1959.

Bits 4-5-6-7	Bits 0-1-2-3							
	0000	0001	0010	0011	0100	0101	0110	0111
0000	Blank		&				0	8
0001				C	K	S		
0010			$				1	9
0011				D	L	T		
0100			*				2	.
0101				E	M	U		
0110			/				3	-
0111				F	N	V		
1000		%	,				4	
1001				G	O	W		
1010		◊	'				5	
1011				H	P	X		
1100		#					6	
1101			A	I	Q	Y		
1110		@					7	
1111			B	J	R	Z		

FIG. 6.3. 49-character set.

These attempts were abandoned, in spite of some rather ingenious proposals, because the advantages of partial compatibility were not enough to offset the disadvantages.

The 8-bit byte was chosen for the following reasons:

1. Its full capacity of 256 characters was considered to be sufficient for the great majority of applications.
2. Within the limits of this capacity, a single character is represented by a single byte, so that the length of any particular record is not dependent on the coincidence of characters in that record.
3. 8-bit bytes are reasonably economical of storage space.

4. For purely numerical work, a decimal digit can be represented by only 4 bits, and two such 4-bit bytes can be packed in an 8-bit byte. Although such packing of numerical data is not essential, it is a common practice in order to increase speed and storage efficiency. Strictly speaking, 4-bit bytes belong to a different code, but the simplicity of the 4-and-8-bit scheme, as compared with a combination 4-and-6-bit scheme, for example, leads to simpler machine design and cleaner addressing logic.

5. Byte sizes of 4 and 8 bits, being powers of 2, permit the computer designer to take advantage of powerful features of binary addressing and indexing to the bit level (see Chaps. 4 and 5).

The eight bits of the code are here numbered for identification from left to right as 0 (high-order bit) to 7 (low-order bit). "Bit 0" may be abbreviated to B_0, "bit 1" to B_1, etc.

6.6. Parity Bit

For transmitting data, a ninth bit is attached to each byte for parity checking, and it is chosen so as to provide an odd number of 1 bits. Assuming a 1 bit to correspond to the presence of a signal and assuming also an independent source of timing signals, *odd parity* permits all 256 combinations of 8 bits to be transmitted and to be positively distinguished from the absence of information. The parity bit is identified here as "bit P" or B_P.

The purpose of defining a parity bit in conjunction with a character set is to establish a standard for communicating *between* devices and media using this set. It is not intended to exclude the possibilities of error correction or other checking techniques *within* a given device or on a given medium when appropriate.

6.7. Sequence

High-equal-low comparisons are an important aspect of data processing. Thus, in addition to defining a standard code for each character, one must also define a standard comparing (*collating*) sequence. Obviously, the decimal digits must be sequenced from 0 to 9 in ascending order, and the alphabet from A to Z. Rather more arbitrary is the relationship between groups of characters, but the most prevalent convention for the 48 IBM "commercial" characters is, in order:

(Low) Blank
Special characters . ⌑ & $ * − / , % # @
Alphabetic characters A to Z
(High) Decimal digits 0 to 9

Fundamentally, the comparing sequence of characters should conform to the natural sequence of the binary integers formed by the bits of that code. Thus 0000 0100 should follow 0000 0011. Few existing codes have this property, and it is then necessary, in effect, to translate to a special internal code during alphanumeric comparisons. This takes extra equipment, extra time, or both. An important objective of the new character set was to obtain directly from the code, without translation, a usable comparing sequence.

A second objective was to preserve the existing convention for the above 48 characters within the new code. This objective has not been achieved because of conflicts with other objectives.

The 7030 set provides the following comparing sequence without any translation:

(Low) Blank
Special characters (see chart)
Alphabetic characters a A b B c C to z Z
Numerical digits 0_{0} 1_{1} to 9_{9}
Special characters . : — ?
(High) Unassigned character codes

Note that the lower- and upper-case letters occur in pairs in adjacent positions, following the convention established for directories of names. (There appeared to be no real precedent for the relative position within the pair. The case shift is generally ignored in the sequence of names in telephone directories, even when the same name is spelled with either upper- or lower-case letters. This convention is not usable in general, since each character code must be considered unique.)

The difference between this comparing sequence and the earlier convention lies only in the special characters. Two of the previously available characters had to be placed at the high end, and the remaining special characters do not fall in quite the same sequence with respect to one another. It was felt that the new sequence would be quite usable and that it would be necessary only rarely to re-sort a file in the transition to the 7030 code. It is always possible to translate codes to obtain any other sequence, as one must do with most existing codes.

6.8. Blank

The code 0000 0000 is a natural assignment for the *blank* (i.e., the nonprint symbol that represents an empty character space). Not only should the *blank* compare lower than any printable character, but also absence of bits (other than the parity bit) corresponds to absence of mechanical movement in a print mechanism.

Blank differs, however, from a *null* character, such as the all-ones code found on paper tape. *Blank* exists as a definite character occupying a definite position on a printed line, in a record, or in a field to be compared.

A *null* may be used to delete an erroneous character, and it would be completely dropped from a record at the earliest opportunity. *Null*, therefore, occupies no definite position in a comparing sequence. A *null* has not been defined here, but it could be placed when needed among the control characters.

Considering numerical work only, it would be aesthetically pleasing to assign the all-zeros code to the digit zero, that is, to use 0000 as the common zone bits of the numeric digits (see below). In alphanumeric work, however, the comparing sequence for *blank* should take preference in the assignment of codes.

6.9. Decimal Digits

The most compact coding for decimal digits is a 4-bit code, and the natural choices for encoding 0 to 9 are the binary integers 0000 to 1001. As mentioned before, two such digits can be packed into an 8-bit byte; for example, the digits 28 in packed form could appear as

0010 1000

If decimal digits are to be represented unambiguously in conjunction with other characters, they must have a unique 8-bit representation. The obvious choice is to spread pairs of 4-bit bytes into separate 8-bit bytes and to insert a 4-bit prefix, or *zone*. For example, the digits 28 might be encoded as

zzzz 0010 *zzzz* 1000

where the actual value of each zone bit *z* is immaterial so long as the prefix is the same for all digits.

This requirement conflicted with requirements for the comparing sequence and for the case shift. As a result, the 4-bit byte is offset by 1 bit, and the actual code for 28 is

011**0** **010**0 011**1** **000**0

This compromise retains the binary integer codes 0000 to 1001 in adjacent bit positions, but not in either of the two positions where they appear in the packed format.

The upper-case counterparts of the normal decimal digits are assigned to italicized decimal subscripts.

6.10. Typewriter Keyboard

The most commonly found devices for key-recording input to a computer system are the IBM 24 and 26 keypunches, but their keyboards are not designed for keying both upper- and lower-case alphabetic characters. The shifted positions of some of the alphabetic characters are used to punch numerical digits. For key-recording character sets with much more than the basic 48 characters, it is necessary to adopt a keyboard convention different from that of the keypunch. The 89-character subset was established to bring the most important characters of the full set within the scope of the common typewriter, thus taking advantage of the widespread familiarity with the typewriter keyboard and capitalizing on existing touch-typing skills as much as possible.

The common typewriter keyboard consists of up to 44 keys and a separate case-shift key. To preserve this relationship in the code, the 44 keys are represented by 6 bits of the code (B_1 to B_6) and the case shift by a separate bit (B_7). The case shift was assigned to the lowest-order bit, so as to give the desired sequence between lower- and upper-case letters.

For ease of typing, the most commonly used characters should appear in the lower shift ($B_7 = 0$). This includes the decimal digits and, when both upper- and lower-case letters are used in ordinary text, the lower-case letters. (This convention differs from the convention for single-case typewriters presently used in many data-processing systems; when no lower-case letters are available, the digits are naturally placed in the same shift as the upper-case letters.) It is recognized that the typewriter keyboard is not the most efficient alphanumeric keyboard possible, but it would be unrealistic to expect a change in the foreseeable future. For purely numerical data, it is always possible to use a 10-key keyboard either instead of the typewriter keyboard or in addition to it.

It was not practical to retain the upper- and lower-case relationships of punctuation and other special characters commonly found on typewriter keyboards. There is no single convention anyway, and typists are already accustomed to finding differences in this area.

6.11. Adjacency

The 52 characters of the upper- and lower-case alphabets occupy 52 consecutive code positions without gaps. For the reasons given above, it was necessary to spread the ten decimal digits into every other one of twenty adjacent code positions, but the remaining ten positions are filled with logically related decimal subscripts. The alphabet and digit blocks are also contiguous. Empty positions for additional data and control characters are all consolidated at the high end of the code chart.

This grouping of related characters into solid blocks of codes, without empty slots that would sooner or later be filled with miscellaneous characters, assists greatly in the analysis and classification of data for editing purposes. Orderly expansion is provided for in advance.

6.12. Uniqueness

A basic principle underlying the choice of this set is to have only one code for each character and only one character for each code.

Much of the lack of standardization in existing character sets arises from the need for more characters than there are code positions available in the keying and printing equipment. Thus, in the existing 6-bit IBM character codes, the code 001100 may stand for any one of the three characters @ (at), − (minus), and ' (apostrophe). The 7030 set was required to contain all these characters with a unique code for each.

The opposite problem exists too. Thus, in one of the existing 6-bit codes, − may be represented by either 100000 or 001100. Such an embarrassment of riches presents a logical problem when the two codes have in fact the same meaning and can be used interchangeably. No amount of comparing and sorting will bring like items together until one code is replaced by the other everywhere.

In going to a reasonably large set, it was necessary to resist a strong temptation to duplicate some characters in different code positions so as to provide equal facilities in various subsets. Instead, every character has been chosen so as to be typographically distinguishable if it stands by itself without context. Thus, for programming purposes, it is possible to represent any code to which a character has been assigned by its unique graphic symbol, even when the bit grouping does not have the ordinary meaning of that character (e.g., in operation codes).

In many instances, however, it is possible to find a substitute character close enough to a desired character to represent it in a more restricted subset or for other purposes. For example, = (equals) may stand for ← (is replaced by) in an 89-character subset. Or again, if a hyphen is desired that compares lower than the alphabet, the symbol ~ (a modified tilde) is preferred to the more conventional − (minus).

A long-standing source of confusion has been the distinction between upper-case "oh" (O) and zero (0). Some groups have solved this problem by writing zero as Ø. Unfortunately, other groups have chosen to write "oh" as Ø. Neither solution is typographically attractive. Instead, it is proposed to modify the upper-case "oh" by a center dot (leaving the zero without the dot) and to write and print "oh" as ʘ whenever a distinction is desired.

Various typographic devices are used to distinguish letters (I, l, V, etc.) from other characters [| (stroke), 1 (one), ∨ (or), etc.]. It is suggested that the italicized subscripts be underlined when handwritten by themselves, for example, $_{\underline{5}}$.

6.13. Signs

The principle of uniqueness implies a separate 8-bit byte to represent a plus or a minus sign. Keying and printing equipment also require separate sign characters. This practice is, of course, rather expensive in storage space, but it was considered superior to the ambiguity of present

6-bit codes where otherwise "unused" zone bits in numerical fields are used to encode signs. If the objective is to save space, one may as well abandon the alphanumeric code quite frankly and switch to a 4-bit decimal coding with a 4-bit sign digit, or go to the even more compact binary radix.

6.14. Tape-recording Convention

As has been remarked before, data-recording media such as magnetic tape and punched cards are not inherently code-sensitive. It is obviously necessary, though, to adopt a fixed convention for recording a code on a given medium if that medium is to be used for communication between different systems.

Magnetic tape with eight, or a multiple of eight, information tracks permits a direct assignment of the 8 bits in the 7030 code to specific tracks. Magnetic tape with six information tracks requires some form of *byte conversion* to adapt the 8-bit code to the 6-bit tape format. The convention chosen is to distribute three successive 8-bit bytes over four successive 6-bit bytes on tape. This convention uses the tape at full efficiency, leaving no gaps except possibly in the last 6-bit byte, which may contain 2 or 4 nonsignificant 0 bits, depending on the length of the record.

Thus successive 8-bit bytes, each with bits B_0 to B_7, are recorded as shown in Table 6.1.

TABLE 6.1. CONVENTION FOR RECORDING 8-BIT CODE ON 6-TRACK TAPE

Track	*Bits*					
0	B_0	B_6	B_4	B_2	B_0	
1	B_1	B_7	B_5	B_3	B_1	
2	B_2	B_0	B_6	B_4	B_2	
3	B_3	B_1	B_7	B_5	B_3	etc.
4	B_4	B_2	B_0	B_6	B_4	
5	B_5	B_3	B_1	B_7	B_5	

The parity bit is not shown. The parity bits for the 6-bit tape format are, of course, different from those of the 8-bit code; so parity conversion must be provided also.

6.15. Card-punching Convention

Since 80-column punched cards are a common input medium, a card-punching convention for the 120 characters is likewise desirable. After the possibility of a separate card code for the 120 characters was considered—a code having the conventional IBM card code as a subset[1]—

[1] *Ibid.*

it was concluded that it would be better to punch the 8-bit code directly on the card. This does not preclude also punching the conventional code (limited to 48 characters) on part of the card for use with conventional equipment. Code translation is then needed only whenever the conventional card code is used; otherwise translation would be required for every column if advantage is to be taken of the new code in the rest of the system.

The punching convention is given in Table 6.2.

In addition, both hole 12 and hole 11 are to be punched in column 1 of every card containing the 7030 code, besides a regular 7030 character, so as to distinguish a 7030 card from cards punched with the conventional code. Eight-bit punching always starts in column 1 and extends as far as desired; a control code END (0 1111 1110) has been defined to terminate the 8-bit code area. Conventional card-code punching should be confined to the right end of those cards identified with 12-11 punching in column 1.

TABLE 6.2. CONVENTION FOR PUNCHING 8-BIT CODE ON CARDS

Card row	*Bit*
12	—
11	—
0	—
1	B_P
2	B_0
3	B_1
4	B_2
5	B_3
6	B_4
7	B_5
8	B_6
9	B_7

Since the parity bit is also punched, the 7030 area of a card contains a checkable code. Note that "blank" columns in this area still have a hole in the B_P row. If only part of the card is to be punched, however, it is possible to leave the remaining columns on the right unpunched.

6.16. List of 7030 Character Set

A list of the 7030 character-set codes and graphic symbols is shown for reference in Fig. 6.4, which includes the names of the characters.

Code		
P 0123 4567	Character	Name
1 0000 0000		Blank (Space)
0 0000 0001	±	Plus or minus
0 0000 0010	→	Right arrow (Replaces)
1 0000 0011	≠	Not equal
0 0000 0100	∧	And
1 0000 0101	{	Left brace
1 0000 0110	↑	Up arrow (Start superscript)
0 0000 0111	}	Right brace
0 0000 1000	∨	Or (inclusive)
1 0000 1001	⊻	Exclusive Or
1 0000 1010	↓	Down arrow (End superscript)
0 0000 1011	‖	Double lines
1 0000 1100	>	Greater than
0 0000 1101	≥	Greater than or equal
0 0000 1110	<	Less than
1 0000 1111	≤	Less than or equal
0 0001 0000	[	Left bracket
1 0001 0001	⊃	Implies
1 0001 0010	]	Right bracket
0 0001 0011	°	Degree
1 0001 0100	←	Left arrow (Is replaced by)
0 0001 0101	≡	Identical
0 0001 0110	¬	Not
1 0001 0111	√	Square root (Check mark)
1 0001 1000	%	Percent sign
0 0001 1001	\	Left slant (Reverse divide)
0 0001 1010	◊	Lozenge (Diamond)(Note)
1 0001 1011	|	Absolute value (Vertical line)
0 0001 1100	#	Number sign
1 0001 1101	!	Exclamation point (Factorial)
1 0001 1110	@	At sign
0 0001 1111	~	Tilde (Hyphen)

Code		
P 0123 4567	Character	Name
0 0010 0000	&	Ampersand
1 0010 0001	+	Plus sign
1 0010 0010	$	Dollar sign
0 0010 0011	=	Equals
1 0010 0100	*	Asterisk (Multiply)
0 0010 0101	(	Left parenthesis
0 0010 0110	/	Right slant (Divide)
1 0010 0111	)	Right parenthesis
1 0010 1000	,	Comma
0 0010 1001	;	Semicolon
0 0010 1010	'	Apostrophe (Single quote)
1 0010 1011	"	Ditto (Double quote)
0 0010 1100	a	
1 0010 1101	A	
1 0010 1110	b	
0 0010 1111	B	
1 0011 0000	c	
0 0011 0001	C	
0 0011 0010	d	
1 0011 0011	D	
0 0011 0100	e	
1 0011 0101	E	
1 0011 0110	f	
0 0011 0111	F	
0 0011 1000	g	
1 0011 1001	G	
1 0011 1010	h	
0 0011 1011	H	
1 0011 1100	i	
0 0011 1101	I	
0 0011 1110	j	
1 0011 1111	J	

Note: The character ⌑ has also been used.

FIG. 6.4. List of 7030 codes and characters. *(Continued on next page.)*

Code P 0123 4567	Character	Name
0 0100 0000	k	
1 0100 0001	K	
1 0100 0010	l	
0 0100 0011	L	
1 0100 0100	m	
0 0100 0101	M	
0 0100 0110	n	
1 0100 0111	N	
1 0100 1000	o	
0 0100 1001	⊙	
0 0100 1010	p	
1 0100 1011	P	
0 0100 1100	q	
1 0100 1101	Q	
1 0100 1110	r	
0 0100 1111	R	
1 0101 0000	s	
0 0101 0001	S	
0 0101 0010	t	
1 0101 0011	T	
0 0101 0100	u	
1 0101 0101	U	
1 0101 0110	v	
0 0101 0111	V	
0 0101 1000	w	
1 0101 1001	W	
1 0101 1010	x	
0 0101 1011	X	
1 0101 1100	y	
0 0101 1101	Y	
0 0101 1110	z	
1 0101 1111	Z	

Code P 0123 4567	Character	Name
1 0110 0000	0	Zero
0 0110 0001	$_0$	Subscript zero
0 0110 0010	1	One
1 0110 0011	$_1$	Subscript one
0 0110 0100	2	Two
1 0110 0101	$_2$	Subscript two
1 0110 0110	3	Three
0 0110 0111	$_3$	Subscript three
0 0110 1000	4	Four
1 0110 1001	$_4$	Subscript four
1 0110 1010	5	Five
0 0110 1011	$_5$	Subscript five
1 0110 1100	6	Six
0 0110 1101	$_6$	Subscript six
0 0110 1110	7	Seven
1 0110 1111	$_7$	Subscript seven
0 0111 0000	8	Eight
1 0111 0001	$_8$	Subscript eight
1 0111 0010	9	Nine
0 0111 0011	$_9$	Subscript nine
1 0111 0100	.	Period (point)
0 0111 0101	:	Colon
0 0111 0110	-	Minus sign
1 0111 0111	?	Question mark

FIG. 6.4 (*Continued*)

Chapter 7

VARIABLE-FIELD-LENGTH OPERATION

by G. A. Blaauw, F. P. Brooks, Jr., and W. Buchholz

7.1. Introduction

Chapter 4 dealt with the fact that natural data units for fixed-point-arithmetic, logical, and editing operations vary considerably in length and structure. The variable-field-length instructions of the 7030 have been designed to make it possible to specify these natural data units simply and directly, thus saving time, space, and programming effort.

The variable-field-length (VFL) data-handling operations may be divided into three classes: (1) arithmetical, (2) radix-conversion, and (3) logical-connective operations. VFL arithmetical and logical-connective operations are both used also for processing alphanumeric data.

The VFL instructions include the basic arithmetical repertoire (LOAD, STORE, ADD, COMPARE, MULTIPLY, DIVIDE) as well as interesting new operations and features. More important, however, is the method of data definition employed by all VFL instructions. Each field, regardless of length, is treated as a separate entity independent of its neighbors. Each numerical field may have its own sign, if a sign is desired. Any overflow beyond the end of the specified field is signaled, but the next adjacent field is protected from inadvertent carry propagation. Similarly, any loss of significant bits caused by storing a result in a field of limited size is signaled. A *result zero* indicator shows the state of only the desired field, no more and no less.

The flexibility needed for VFL operations is achieved most economically by a serial data-handling mechanism. Serial data handling is relatively slow, but the objective here is not high speed for individual instructions. (Where arithmetical speed is of the essence, the unnormalized floating-point mode should be used for fixed-point arithmetic—see Chap. 8.) The VFL instructions are intended for such operations on complex data structures as format conversion and arranging for printing. Such operations can be performed by a serial VFL unit faster than by a parallel fixed-length arithmetic and logic unit. Most of the serial mechanism is actually concerned with the structure of the data and relatively little with the operation itself. Thus the choice of a serial mechanism was not dictated by the cost of extra adder stages but by the complex switching that would have been needed to select an entire field of variable position, length, and structure, in parallel fashion—though it is granted that an elaborate parallel mechanism could have been designed that would do VFL operations even faster than a serial unit.

VFL operations are particularly desirable in processing large volumes of data. Here the most important element of high performance is reduction in storage space. With VFL operation more data can be held in storage units of fixed capacity (core memory, drums, or disks), which may permit a given problem to be solved more quickly or more problems to be tackled at one time by multiprogramming. With open-ended storage media (magnetic tape), over-all performance is often limited by the speed of data transmission; so the reduction in storage space obtained by varying the field length can result in a corresponding reduction in execution time.

7.2. Addressing of Variable-field-length Data

As explained in Chap. 5, the reason for choosing a memory word size of 64 bits, a power of 2, is that a binary address can be assigned to each bit in a memory word, with continuous numbering of all bits in memory. Accordingly, the VFL system has been designed so that the memory may be looked on by the programmer as if it were one continuous horizontal string of bits, extending from address 0 at the left to the last memory bit at the right. Fields can be placed anywhere in memory regardless of their lengths, overlapping memory-word boundaries when necessary. The programmer merely specifies the address of the field, which is the address of the leftmost bit (the high-order bit in a numerical field), and the length. Successive bits in the field have consecutively increasing address numbers; but these addresses are not referred to by the program, except when it is desired to operate explicitly on a portion of the field as if it were another field. The VFL system does the bookkeeping necessary to select the word or pair of adjacent words in memory and to select the desired array of bits in these words.

The left-to-right memory-addressing convention, where a byte, field, or record is addressed by the address of its leftmost bit, is followed throughout the system. For purposes of arithmetic it might be thought more convenient to address numerical fields from the right, since serial arithmetic always starts with the lowest-order digit. Keyed input and printed output data, on the other hand, must follow the left-to-right sequence to which humans are accustomed. Because nonnumerical data may consist of long strings of bits, whereas numbers are relatively short, it seemed desirable to adopt a consistent left-to-right convention and impose the burden of temporarily reversing the sequence on the arithmetical processes.[1] This convention avoids the possibility of having different operations refer to the same field by two different addresses.

The VFL instruction format (Fig. 7.1) contains a 24-bit operand address, of which the left 18 bits specify the memory word, and the right 6 bits specify the bit within that word at which the field starts. The 24-bit address is a homogeneous binary number, so that addresses may be computed by straightforward arithmetical processes. The operand address may be modified automatically by adding an index value that is also 24 bits long. Thus VFL instructions provide for indexing to the bit level. Indexing is specified by the index address *I* in the left half of the instruction word. (The second *I* field in the right half may be used for modifying the *length*, *byte size*, and *offset* fields described below.)

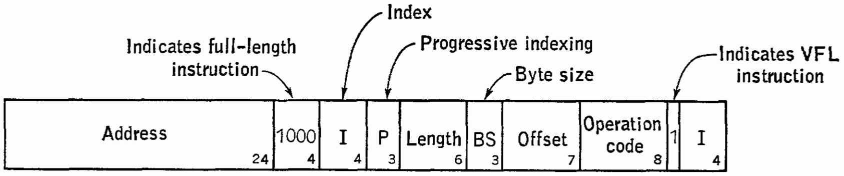


FIG. 7.1. VFL instruction format.

The address part of a VFL instruction may also be used as a data field of up to 24 bits in a mode called *immediate addressing*. Immediate addressing is useful for supplying short constants to the program.

7.3. Field Length

The length of the field is specified as a number of bits and may range from 1 to 64. It would be nicer to have an essentially unlimited field length (as in the 256-character accumulator of the IBM 705), but the cost of additional flip-flop registers (as compared with the relatively slow core storage used for the 705 accumulator) and extra controls would have outweighed their usefulness. In numerical work 64 bits are usually adequate, and multiple-precision fixed-point arithmetic should only rarely be needed. For alphanumeric comparisons, which do often deal with long fields, a special comparison operation is provided to simplify the comparing of multiple fields, so that long fields can readily be treated as

[1] This is not a great burden, because a serial arithmetic unit must be capable of progressing, or jumping, from one end of a number to the other in either direction, for several reasons. After a right-to-left subtraction, the unit may have to jump back to the right end for a second, recomplementing pass through the number. In division, the quotient must be developed digit by digit from left to right.

several shorter fields. In the other operations where long fields are occasionally encountered, there are no carries between fields, and multiple operations can again be programmed quite easily. Hence the limitation to 64 bits as the maximum field size is not onerous.

All bits of a field are counted in the field length, including the sign bits of signed numbers. Thus the field lengths are additive. In assigning memory space, adding the length of a field to its address gives the address of the next available memory space. The length of a record is the sum of the lengths of its fields.

7.4. Byte Size

Many data fields have an inner structure and are made up of a number of bytes, such as decimal digits or alphabetic characters. In some operations, primarily decimal arithmetic, the control circuits must observe the byte boundaries, since, during decimal addition for example, the carry between bits of one decimal digit has different properties from those of the carry between two adjacent decimal digits. In binary arithmetic the numerical part is homogeneous, all bits being treated alike, but the sign may require special treatment and is considered to be a separate byte. With alphabetic fields the byte boundaries are important for some functions, such as printing; other operations, such as loading, storing, and (in a well-chosen code) comparing, can be performed as if the field were a homogeneous binary number.

The natural length of bytes varies. Decimal digits are most economically represented in a 4-bit code. The commonly used 6-bit alphanumeric codes are sufficient when decimal digits, a single-case alphabet, and a few special characters are to be represented. If this list is extended to a two-case alphabet and many more special characters, a 7- or 8-bit code becomes desirable (see Chap. 6). A 3-bit octal code or a 5-bit alphabetic code is occasionally useful. There would be little use for bytes larger than 8 bits. Even with the common 12-bit code for punched cards, the first processing step is translation to a more compact code by table look-up, and during this process each column is treated as a 12-bit binary field. There would be no direct processing of longer fields in the 12-bit code.

It is common practice to employ throughout a computer a fixed byte size large enough to accommodate a 6-bit alphanumeric code. Since numerical data predominate in many applications, this simple representation is fairly inefficient: one-third of the bits in purely numerical digits are vacuous. The efficiency drops further as a larger alphabet is chosen. Another common practice is to use two different byte sizes, one to represent purely numerical fields in a relatively dense code and another for alphanumeric fields where each character is represented by two decimal digits. Assuming that 4 bits are used for a decimal digit, this 4-and-8-bit coding scheme is superior to the 6-bit code if numerical data occupy more than half the space or if a larger than 64-character alphabet is desired. A third scheme in current use allows 4-bit decimal digits and 6-bit alphanumeric characters.

The 7030 is unique in that the byte size is completely variable from 1 to 8 bits, as specified with each VFL instruction. Bytes may also overlap word boundaries.

7.5. Universal Accumulator

All VFL operations refer to an *implied operand* in the arithmetic unit. The principle was adopted in the design of both VFL and floating-point operations that the accumulator registers would always be the source of the major implied operand. Likewise, if one or more results are to be returned to the arithmetic unit, the major result is left in the accumulator ready for use as an operand in the next instruction. It should not be necessary to write extra instructions for moving operands within the arithmetic unit. Only in operations that require more than one implied operand (cumulative multiplication) or produce more than one result (division) is it necessary to load or unload an extra register; special registers are provided for these operations, and they are not used for any other purpose.

This principle of the *universal accumulator* saves housekeeping instructions, which are needed in many other computers, and simplifies exception routines, because operations follow a more uniform pattern.

7.6. Accumulator Operand

In VFL operations the implied operand in the accumulator has a maximum length of 128 bits, not counting sign bits. The right end of the accumulator operand is defined by the *offset* part of the instruction (Fig. 7.1). The offset specifies the number of bits between the right end of the accumulator and the start of the operand; i.e., a zero offset means that the operation starts at the right end of the accumulator, and an offset of 17 that the operation starts at the seventeenth bit from the right. The operation is executed in such a way that the right end of the accumulator operand lines up with the right end of the memory operand. This is done by selecting the bits from the desired register positions (not by shifting the entire contents of the register).

The main purpose of specifying an offset is to provide a shifting operation as part of every VFL instruction. No separate shift instructions are needed. Thus decimal points can be aligned without first repositioning the accumulator field.

The offset might also be looked upon as a bit address within the accumulator. Because of the nature of an offset, the accumulator bit numbering goes from right to left, in contrast with the left-to-right sequence in memory.

7.7. Binary and Decimal Arithmetic

All VFL-arithmetic operations are available in both binary and decimal modes, depending on the setting of a *binary-decimal modifier* bit in the operation code (Fig. 7.2). Strictly speaking, decimal multiplication and division are not executed directly. The instructions cause entry to a standard subroutine via the program-interrupt system, to take advantage of the higher speed of radix conversion and parallel binary arithmetic; since programs using these operations are written exactly as if they were executed directly, the distinction will not be made in this chapter.

In decimal arithmetic the accumulator operand is assumed to have a byte size of 4. The byte size of the memory operand is specified by the instruction, as mentioned before. When the result is stored, the byte size in memory is again specified; with a byte size greater than 4, *zone bits* are inserted in the high-order bit positions of every byte, these zone bits being obtained from the accumulator sign register where they are set up in advance as desired. This feature permits arithmetic to be performed directly in any alphanumeric code where the digits are encoded as binary integers in the four low-order bit positions with common zone bits in the high-order positions.[1]

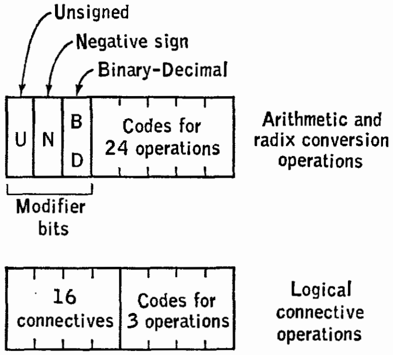


FIG. 7.2. Details of VFL operation codes.

In binary arithmetic the byte-size specification does not apply to the numerical part of binary numbers, which always have a homogeneous internal structure. Regardless of the byte-size specification (which is

[1] It should be remarked here that it was intended, at the time this feature was developed, to use such an alphanumeric code for the system. Subsequently other considerations entered the picture, and the 8-bit code described in Chap. 6 is not this kind of a code. In a compromise among conflicting requirements, the 4-bit portion representing the ten binary integers in the codes for the decimal digits was offset to the left by one bit position. Therefore, decimal arithmetic cannot be performed directly in this code. This loss is more apparent than real, however. In practice it is highly desirable to edit all numerical input data for consistency, and it is almost essential to edit numerical output data to suppress zeros, insert commas, etc. Because editing usually involves table look-up, conversion between the 8-bit 7030 code and the 4-bit decimal-arithmetic code comes free and provides, moreover, the very desirable data compression made possible by a 4-bit code.

used only to control the sign byte—see below), binary arithmetic proceeds 8 bits at a time, except that the last byte is shortened automatically if the field length is not a multiple of 8 bits.

In both forms of arithmetic the accumulator operand is considered to occupy the entire accumulator, regardless of the field length specified for the memory operand. When the accumulator is loaded, all bit positions to the left or right of the new field are set to zero. When a number is added to the accumulator contents, carries are propagated as far as necessary. Overflow occurs only in the rare case where a carry goes beyond the left end of the registers.

7.8. Integer Arithmetic

In the structure of arithmetic units, a distinction may be made between integer and fraction arithmetic according to the apparent position of the radix point. In integer arithmetic all results are lined up at the right end of the registers, as if there were a radix point at the extreme right. In fraction arithmetic all results regardless of length are lined up at the left end of the registers (except for possible overflow positions), so that the apparent radix point is at the left. The binary and decimal VFL arithmetic in the 7030 is of the integer type, whereas the floating-point arithmetic (see Chap. 8) is of the fraction type. (Among earlier computers the 705, for example, uses integer arithmetic, and the 704 fraction arithmetic; some computers have employed intermediate positions for the radix point.)

The distinction between integer and fraction arithmetic is rather subtle, because a computer must in any case have shifting facilities so as to deal with integers as well as with pure or mixed fractions. The basic arithmetical operations produce the same result digits regardless of where the point is.[1] The difference lies in the alignment of the result of one operation with the operand of a subsequent operation. For example, if the product of a multiplication is added to another number without shifting, that number will be added to the low-order part of the product in integer arithmetic and to the high-order part of the product in fraction arithmetic. A similar distinction exists in the alignment of the result of an addition for subsequent use as a dividend.

As an example of the integer approach, consider a decimal multiplica-

[1] It is assumed here that the arithmetic unit, whether of the integer or the fraction form, is designed to retain all result digits from any of the basic arithmetical operations. For example, multiplication of two single-length numbers is assumed to produce a double-length product. If a designer wished to have the principal multiplication instruction produce only a single-length product, he would probably choose to keep the high-order part in fraction arithmetic or the low-order part in integer arithmetic. On the other hand, to facilitate double-precision arithmetic he would probably include a secondary operation to produce the other half of the product.

tion followed by an addition, with a field length of 2 digits:

$$\begin{array}{rl} (23.)(45.) & = 1035. \\ & +\quad 67. \\ \hline & = 1102. \end{array}$$

If the same fields are put through the same operations in fraction arithmetic, without shifting, the result will be

$$\begin{array}{rl} (.23)(.45) & = .1035 \\ & + .67 \\ \hline & = .7735 \end{array}$$

In VFL arithmetic all operands are aligned at the right if the offset is zero. The integer approach was chosen because numerical VFL operands frequently have but few digits, which are subjected to relatively few arithmetical operations, and these are mostly additions or subtractions. There is thus little concern with loss of precision (which is discussed in Chap. 8) and hence no need for carrying many low-order guard digits. Aligning numbers at the right then reduces the chances for overflow, so that rescaling is seldom needed. Moreover, in data-processing applications most of the numbers are actually integers or else have only a few places to the right of the point; the arithmetical processes for such numbers are more easily visualized in the integer form than in the fraction form. On the other hand, the alignment of VFL numbers is readily changed to any other radix-point location without extra instructions, by suitable adjustment of the offset, which is available in every VFL instruction.

The choice of fraction arithmetic for floating-point operations is discussed in Chap. 8.

7.9. Numerical Signs

Signed numbers are represented throughout the system by their absolute value and a separate sign.[1] The sign bit is 0 for + and 1 for −.

The sign bit is contained in a sign byte (Fig. 7.3) whose format depends on the byte size specified. In decimal arithmetic it is convenient to have all bytes, including the sign byte, of equal size; for uniformity the same byte-size convention is applied in binary arithmetic, but only to the sign byte.

When the byte size is 1, the sign byte just consists of the sign bit (S). When the byte size is greater than 1, the extra bit positions becoming

[1] Complements may appear as intermediate results during the execution of an instruction (see Chap. 14), but they are always converted to absolute-value form automatically.

available are utilized for independent functions. As the byte size is increased, from 1 to 3 *data flag* bits (T, U, and V) are attached to the right. These flag bits set corresponding indicators whenever an operand is fetched; the flag bit may be set by the programmer to signal, via the program interrupt system, exceptional conditions as desired. For byte sizes above 4, the previously mentioned zone bits are attached on the left of the sign bit.

VFL arithmetic may be performed on either signed numbers or unsigned numbers from memory. For unsigned numbers the sign byte is omitted and the numbers are assumed to be positive. The *unsigned* modifier bit in the instruction specifies the choice and determines whether the rightmost byte of the number is to be treated as the sign byte or as containing the low-order numerical bits.

The most important reason for providing an unsigned mode of arithmetic is the fact that in many data-processing applications most of the numerical data fields are inherently positive. For instance, a count of physical items can only be positive; quantities and prices in accounting transactions are positive, although the resulting balances may have either sign. For inherently positive quantities signs are redundant, and significant storage space can be saved by omitting sign bits.

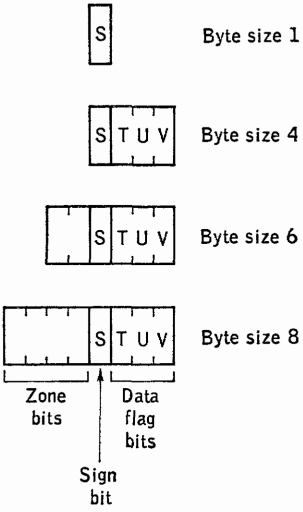


FIG. 7.3. Sign byte.

When signs are redundant they are usually omitted in the source data as well, to reduce manual recording effort. Some computers require all numbers to be signed before arithmetic can be performed. The programming effort to insert signs where none are needed can be avoided by an unsigned mode of arithmetic.

The unsigned mode is also needed in order to operate arithmetically on parts of fields, which generally do not have signs even when the entire field does.

The accumulator operand always has a sign attached. Thus it becomes possible to operate with a mixture of signed and unsigned memory operands; for example, one can add an unsigned item field to a signed total field. When the result is stored in memory it is again possible to specify whether to omit or include the sign of the result. The accumulator sign is held in the 8-bit accumulator sign-byte register, which also contains the three data flags of the accumulator operand and four zone bits, according to the byte-size-8 format of Fig. 7.3.

The VFL instructions contain another modifier bit that affects the signs, the *negative sign* modifier. If it is set to 1, this modifier causes an inversion of operand sign so that ADD becomes *subtract*, LOAD (which in some computers is called *clear and add*, or *reset add*) becomes *clear and subtract*, etc. This sign inversion is available for all arithmetical operations by virtue of the common modifier bit.

7.10. Indicators

Every VFL operation sets certain *indicators* to indicate important characteristics of the operand and the result. Operations other than comparison turn on indicators that show whether the result is less than, equal to, or greater than zero, or whether the result sign is negative (which includes the possibility of a negative zero result, as well as a result less than zero). For comparison operations there is a separate set of indicators that show whether the accumulator operand was lower than, equal to, or higher than the memory operand. Since these indicators are set only by a *compare* instruction, it is possible to insert other instructions between this instruction and the conditional *branch* that tests the comparison result, without danger of destroying the result.

A comparison may be considered to be a subtraction with the result discarded and both operands left intact; so there is a direct correspondence between the result indicators and comparison indicators:

Result indicators	*Comparison indicators*
Result less than zero	*Accumulator low*
Result zero	*Accumulator equal*
Result greater than zero	*Accumulator high*
Result negative	—

The *lost carry* indicator is set if there is an overflow beyond the left end of the accumulator, but, as was mentioned earlier, the accumulator is long enough so that this would be a rare occurrence. An overflow is more likely to become apparent when the result is stored in memory. The memory field would normally be specified just long enough to accommodate all expected results. A result overflow then means that the accumulator contains more significant bits than the memory field can hold, and the *partial field* indicator is turned on. If the *partial field* indicator remains off after a *store* operation, there is assurance that all higher-order accumulator bits were 0.

There are two *add to memory* operations which return the result of an addition to memory instead of to the accumulator. When the result goes to memory there may be a carry off the left end of the specified memory field even if there are no excess 1 bits in the accumulator. The *lost carry* indicator is then turned on.

The VFL mechanism thus protects fields adjacent to the specified field from being altered if an overflow occurs, and it signals the occurrence of overflow by the two, rather similar, result-exception indicators, *lost carry* and *partial field.* The reason for two separate indicators is that the two conditions indicated would normally be handled by different correction procedures.

Another exception indicator is *zero divisor,* which, as the name implies, indicates an attempt to divide by zero, the DIVIDE operation having been suppressed.

If the operand has been flagged with one or more data flags, the corresponding *data flag* indicators are set. The *to-memory operation* indicator distinguishes from all other operations those which return a result to memory; this is an aid in programming exception routines, since it obviates detailed testing of operation codes to see where the result, which may have to be adjusted, has been sent. Finally, the indicators *binary transit* and *decimal transit* may be used to enter subroutines after the (binary or decimal) operand has been placed in the transit register; the *decimal transit* indicator is used, for example, to enter the subroutines for decimal multiplication and division.

The result-exception, data-flag, and transit indicators may interrupt the program automatically. The result, comparison, and *to-memory operation* indicators are available only for programmed testing.

7.11. Arithmetical Operations

The various VFL-arithmetic operations will be discussed here only briefly, with emphasis on novel operations and features. The reader is referred to the summary list in the Appendix and to the 7030 Reference Manual for more complete descriptions.

LOAD (or a variant, LOAD WITH FLAG) and STORE are used to transfer operands from memory to accumulator or from accumulator to memory, respectively, replacing the previous contents. ADD and ADD TO MEMORY form the sum of the memory and accumulator operands and return the sum to the accumulator or to memory, respectively (LOAD and STORE may be considered special cases of ADD and ADD TO MEMORY, obtained by turning off one input to the adder). ADD TO MEMORY is particularly useful in single-address computers, in that it simplifies the process of adding an item to, or subtracting it from, one or more totals in memory. A variant is ADD ONE TO MEMORY, which makes it possible to develop counts in memory without disturbing the accumulator.

Further variations of the normal addition operations are ADD TO MAG-

NITUDE and ADD MAGNITUDE TO MEMORY, which are intended to be used for positive-integer arithmetic. Addition is algebraic, but the accumulator sign is taken to be positive and the result is not allowed to change sign; if the result would have been negative, it is replaced by zero.[1]

STORE ROUNDED is a novel instruction which stores a rounded result in memory while leaving the unrounded result in the accumulator for any further operations. The offset specifies the position at which rounding, by adding ½ to the absolute value, takes place, and the field is then sent to memory, dropping all positions to the right of this one.

There are several variations of COMPARE. All of them perform an algebraic subtraction and turn on a low, equal, or high indicator according to the result, but the numerical result is discarded and both operands are preserved in their original form. Comparison may be either on proper numbers, according to algebraic sign conventions, or on nonnumerical data, with fields treated as unsigned binary numbers.

One or more COMPARE IF EQUAL instructions are used following a COMPARE to continue comparison of fields longer than 64 bits. COMPARE FOR RANGE following COMPARE can be used to determine whether a quantity falls within a given range when exact equality is not desired. These three instructions are paralleled by another set of three (COMPARE FIELD, COMPARE FIELD IF EQUAL, and COMPARE FIELD FOR RANGE), which permit a *portion* of the accumulator to be compared with the memory operand.

The regular MULTIPLY instruction uses the accumulator operand as the multiplier and returns the product to the accumulator. Because it is often desired to add the product to a previous result, a cumulative multiplication operation is also provided. Here the multiplier must first have been loaded into a special factor register by the instruction LOAD FACTOR. Then MULTIPLY AND ADD forms the product of this factor with the memory operand and adds the result to the accumulator contents. The factor register remains undisturbed, and its contents are still available if the same multiplier is to be used repeatedly.

In DIVIDE, the accumulator operand is the dividend and the memory operand the divisor, with the quotient being returned to the accumulator. At the same time a signed remainder is placed in a special remainder register, where it is available any time until another division is performed. A noteworthy feature of this DIVIDE operation is that it does not

[1] This is a modification of operations independently proposed by Brooks and Murphy:

F. P. Brooks, Jr., The Analytic Design of Automatic Data Processing Systems, Ph.D. thesis, Harvard University, 1956, p. 6.42.

R. W. Murphy, A Positive-integer Arithmetic for Data Processing, *IBM J. Research and Development,* vol. 1, no. 2, pp. 158–170, April, 1957.

require adjustment of the relative magnitudes of dividend and divisor to produce a proper result. In other computers it has been necessary to make sure that division would not be halted by a dividend too large with respect to the divisor, with the possibility of error stops (or worse) if the numbers exceeded the predicted range. No scaling is needed in the 7030 for division to proceed, although sometimes it may be desired to offset the dividend relative to the divisor in order to obtain a specified number of significant quotient bits. The indeterminate case of a zero divisor is signaled by program interruption, and it is not necessary to make a test before every division.

The VFL-arithmetic instruction set may be extended by using the instruction LOAD TRANSIT AND SET for interpretive programming. The specified operand is loaded into a special register, the *transit register*, and a program interruption is initiated. A 7-bit field in the instruction can be used as a code of 128 pseudo operations by entering a table of *branch* instructions which lead to corresponding subroutines. This feature happened to be a by-product of the interpretive decimal multiplication and division scheme, but it is expected to become a useful programming tool.

7.12. Radix-conversion Operations

The radix-conversion operations provide for automatic conversion, either from decimal to binary radix and format or from binary to decimal. The numbers are treated as integers. For numbers other than integers, a multiplication by a suitable power of 10, in binary form, must be programmed.

The basic instruction LOAD CONVERTED obtains the original number from memory and places the converted result in the accumulator. All the format specifications of the VFL system are available.

Another operation, LOAD TRANSIT CONVERTED, loads the converted result into the transit register, by-passing the accumulator. Two more operations, CONVERT and CONVERT DOUBLE, take the operand from the accumulator and return the result to the accumulator; these operations are designed to convert to or from binary numbers in the floating-point format.

It is important to note that these operations combine the functions of format conversion, done efficiently by the serial arithmetic unit, and radix conversion, performed at high speed by the parallel arithmetic unit.

7.13. Logical Connectives of Two Variables

The use of Boolean algebra to express logical functions is well known, and Fig. 7.4 shows some of the commonest functions of two logical variables. The variables are called *m* and *a*, corresponding to the memory and accumulator operands. These logical connectives have found their way into the instruction repertoire of several computers.

There are sixteen ways of combining a pair of two-valued variables. By rearranging the notation of Fig. 7.4 and adding the rest of these functions, a complete table can be made, as shown in Fig. 7.5. For each connective the values of the function corresponding to the four possible combinations of bits m and a are shown under the heading *Truth tables.* The connectives are here labeled 0 to 15 according to the binary integer

m	a	$m \wedge a$
0	0	0
0	1	0
1	0	0
1	1	1

And

m	a	$m \vee a$
0	0	0
0	1	1
1	0	1
1	1	1

(Inclusive) Or

m	a	$\neg m$
0	0	1
0	1	1
1	0	0
1	1	0

Not m

m	a	$m \forall a$
0	0	0
0	1	1
1	0	1
1	1	0

Exclusive or

FIG. 7.4. Some common Boolean functions of two variables.

Connective	*Common names*	*Truth tables*				*Symbolic representation*
		m a 0 0	m a 0 1	m a 1 0	m a 1 1	
0		0	0	0	0	0
1	And	0	0	0	1	$m \wedge a$
2		0	0	1	0	$m \wedge \neg a$
3		0	0	1	1	m
4		0	1	0	0	$\neg m \wedge a$
5		0	1	0	1	a
6	Exclusive or	0	1	1	0	$m \forall a$
7	Or	0	1	1	1	$m \vee a$
8	Nor (dagger)	1	0	0	0	$m \downarrow a$
9	Identity (match)	1	0	0	1	$m \equiv a$
10	Not	1	0	1	0	$\neg a$
11	Implication	1	0	1	1	$m \supset a$
12	Not	1	1	0	0	$\neg m$
13		1	1	0	1	$\neg m \vee a$
14	Not and (stroke)	1	1	1	0	$m \mid a$
15		1	1	1	1	1

FIG. 7.5. Complete table of logical connectives of two variables.

formed by the 4 bits in the truth tables. Thus, with the particular arrangement chosen, the function *and* is connective 1 and the function *or* is connective 7. The column at the right shows a representation of each function, in terms of symbols chosen in Chap. 6.

The sixteen logical connectives include several that might be considered trivial, such as 0 and 15, which depend on neither variable, or 3 and 5, which merely reproduce one of the variables disregarding the other. Then again, connectives 4 and 13 can be obtained from 2 and 11 simply by interchanging *m* and *a*, and the second half of the table is, of course, the same as the first half inverted. Thus it might appear economically wise to restrict the connective operations in a computer to a small set, such as that of Fig. 7.4.

That all sixteen connectives be provided in the 7030 was originally proposed for the sake of completeness and as a matter of principle. It was decided to specify connectives by placing the 4 bits of the desired truth table (Fig. 7.5) directly in the operation code of the instruction (Fig. 7.2). It was then discovered that the logic unit could be implemented very simply by connecting wires corresponding to bits *m* and *a*, or their inverse, and the specifier bits to 4 three-way *and* circuits feeding a four-way *or* circuit. Thus the èxtra cost of furnishing all sixteen connectives was very low indeed. Moreover, it was found during exploratory programming that the "trivial" connectives were used much more often than connectives depending on both variables, since they provide such common functions as setting, resetting, and inverting of bits.

So far we have discussed connective operations on a single pair of binary variables with a single-bit result. To evaluate a complex logical statement with such operations, it is necessary to apply different connectives sequentially, one pair of variables at a time. In other applications, such as inverting or masking an array of bits, it is desirable to apply a single connective to a group of bits. The connective operations are designed to make possible both modes of operation by means of the VFL mechanism; the field length specifies the number of bits, from 1 to 64.

7.14. Connective Operations

The connective operations, like the other VFL operations, specify a memory operand by the address of the leftmost bit and by the field length in bits; the second operand is taken from the accumulator, and its right end is defined by the offset, as before. The connective specified by the above-mentioned 4-bit code in the instruction is applied to each pair of corresponding bits from memory (*m*) and accumulator (*a*). Some illustrative examples are shown in Fig. 7.6.

There are three operations: CONNECT, which returns the result to the accumulator; CONNECT TO MEMORY, which returns the result to memory;

and CONNECT FOR TEST, which leaves both operands intact and discards the result after recording certain tests that are always made after each of the three operations.

One test determines whether all result bits are 0 and sets the *result zero* indicator. More comprehensive tests may be made on the basis of two bit counts which are developed from the results: the *left-zeros count* indicates the number of consecutive 0 bits between the left end of the result field and the first 1 bit; the *all-ones count* gives the number of 1 bits in the result. As an example, the low-order bit of the all-ones count gives the odd-even parity of the result field.

	Operands
m	0 0 1 1 0 0 1 1
a	1 0 0 1 0 1 1 0

Connective		*Result*	*Left-zeros count*	*All-ones count*
0 0 0 1	$(m \wedge a)$	0 0 0 1 0 0 1 0	3	2
0 1 1 0	$(m \veebar a)$	1 0 1 0 0 1 0 1	0	4
0 1 1 1	$(m \vee a)$	1 0 1 1 0 1 1 1	0	6
1 0 1 0	$(\neg a)$	0 1 1 0 1 0 0 1	1	4
1 0 1 1	$(m \supset a)$	0 1 1 1 1 0 1 1	1	6
1 1 1 1	(1)	1 1 1 1 1 1 1 1	0	8

FIG. 7.6. Examples of logical connectives. Field length and byte size are 8.

Logical fields have no internal structure, each bit being independent of the others, and a byte size of 8 is specified as a rule. The accumulator operand is the same length as the memory operand, all other accumulator bits being ignored. This is unlike the other VFL operations, which treat the entire accumulator contents as the implied operand. Thus LOAD not only places the memory operand in the accumulator, but also resets all other bits to 0; CONNECT, on the other hand, changes only those accumulator bits which directly correspond to the specified memory bits, all other bits being left unchanged. This very useful property of the *connect* operations allows independent use of different parts of the accumulator. In particular, CONNECT 0011 (see Fig. 7.5) can be used for assembly of data in the accumulator, and CONNECT TO MEMORY 0101 for storing selected portions of the accumulator. These functions are especially helpful in programming table references, either by address selection or by searching.

Since the byte-size-determining mechanism is available, it has been put to use also in connective operations. When the byte size is less than 8, each memory byte is automatically filled with leading 0s to make an 8-bit byte before these are combined with 8-bit bytes from the accumulator. (The accumulator always operates with byte size 8 in connective operations, as compared with an automatic byte size of 4 in decimal arithmetic.) The result bytes, also 8 bits long, are cut to the specified size in CONNECT TO MEMORY by deleting excess bits. The byte-size controls permit expansion or contraction of bytes, or selection, interleaving, and distribution of bits.

The combined facilities of the connective operations constitute a complete, novel, and powerful system for operating upon groups of independent bits rather than numbers. They are perhaps the most significant new feature of the 7030. It has become clear that logical operations are neither modifications of arithmetic nor auxiliaries to it, but are equal to arithmetic in importance.

Chapter 8

FLOATING-POINT OPERATION

by S. G. Campbell

In this chapter we shall first discuss the reasons for going to floating-point operation and cover some general questions concerning this mode of arithmetic. Then we shall describe the implementation of floating-point arithmetic in the 7030 computer.

GENERAL DISCUSSION

8.1. Problems of Fixed-point Arithmetic

Two basic problems in large-scale scientific computation are the range and the precision of numbers. The *range* of numbers is given by the extreme values that the numbers may assume; too small a range will cause frequent overflow (or underflow) of numbers, requiring excessive intervention by the programmer. *Precision* refers to the number of digits (or bits) needed during the calculation to retain the desired number of significant digits (or bits) in the result; when the number of digits is insufficient, the progressive significance loss and the cumulative round-off errors, which usually occur as the calculation proceeds, may cause the results to be meaningless.

Most of the early computers designed for scientific computation used *fixed-point arithmetic*. A number was represented by a fixed number of digits, and the machine was designed with the decimal point (or binary point) in a fixed position, as in a mechanical desk calculator. This arrangement automatically implies a rather restricted *natural range*, which was commonly the interval from -1 to $+1$. Similarly the *natural precision* was a function of the fixed word length of, say, n digits, so that numbers within the natural range from -1 to $+1$ (any number of absolute value not exceeding unity) could be represented with a maximum error of $R^{-n}/2$, where R is the radix used (most commonly 2 or 10). If the natural precision of the machine was inadequate for a particular calculation (and in most early machines it was about 10 to 12 decimal digits, or the equivalent), additional accuracy could be obtained by programming *multiple-precision arithmetic,* that is, by using more than one word to represent a single number. Programmed multiple-precision operations were very slow relative to the corresponding single-precision operations performed on the natural unit of information, and they were wasteful of both data storage and instruction storage.

The problem of range was handled by a completely different technique, commonly called *scaling.* There were several approaches to scaling, depending upon the problem and upon the persuasion of those who analyzed and programmed it. Sometimes it was possible to scale the problem rather than the arithmetic. Obviously, numbers used in scientific calculations do not fall naturally within the unit interval, but such problems may be transformed into problems in the unit interval, solved there, and the results related back to the real world. For example, if we are integrating some function $f(x)$ between limits a and b, we may translate and compress uniformly by some factor R^p the interval (a,b) into the unit interval (0,1) on the x axis, and compress $f(x)$ uniformly on the y axis by some factor R^q greater than the maximum absolute value of $f(x)$ in the interval (a,b). The resulting integral is clearly less than unity in absolute value, as are all the quantities involved in calculating it; so the entire calculation can be performed in fixed-point arithmetic, and the unscaled value of the integral can be obtained by simply multiplying the scaled result by the factor R^{p+q}. Even in this simple example it is necessary to know the maximum value of the integrand, to perform a linear transformation on the function, and to scale it properly.

For more complicated problems more and deeper analysis may be required; it may become impractical to scale the problem, but it is still possible to scale the arithmetic. Such scaling simply takes advantage of the fact that, with n digits in radix R, we can represent any number whose absolute value does not exceed R^p with a maximum error of $R^{p-n}/2$. (In the special case of $p = n$, the quantity represented is an integer.) The quantity p, which may be any integer, is sometimes called the *scale factor* and may be either implicit or explicit—that is, it may exist only in the mind of the programmer, who takes it into account in his calculation, or it may appear explicitly in the computer memory. If the scale factor is explicit, *scaling loops* may be used to modify the scale factor as circumstances dictate. In either case, a common scale factor p is shared by an entire set of numbers, the only condition being that no number in the set can be as large as R^p in magnitude.

The weaknesses of scaling the arithmetic are twofold: a considerable amount of mathematical analysis as well as side computation is involved in determining and keeping track of the scale factor; and the scale factor for an entire set of numbers is determined by the maximum value that any of them can achieve. The first difficulty has become more acute as the number of computers has increased relative to the number of analysts and programmers. The second introduces a significance problem: given a common scale factor p, the actual difference k between the scale factor p and the order of magnitude of a given scaled fixed-point number causes k leading zeros to occur in the fixed-point number, leaving a maximum of $n - k$, instead of n, significant digits. It is thus possible for k information digits to be permanently lost.

8.2. Floating-point Arithmetic

To avoid difficulties of limited range and scaling in fixed-point arithmetic, G. R. Stibitz in the early 1940's proposed an automatic scaling procedure, called *floating-point arithmetic*, which was incorporated in the Bell Telephone Laboratories' Model V Relay Computer.[1] A similar procedure was developed, apparently independently, for the Harvard Mark II computer.[2] Automatic scaling was a controversial subject for quite a few years. Many opposed it on the basis that the programmer could not be relieved of the responsibility of knowing the size of his numbers and that programmed scaling would give him better control over significance. Nevertheless, as early as 1950, users began to incorporate automatic scaling on fixed-point computers by means of subroutines, first on the plugboard-controlled CPC (Card Programmed Calculator) and later on stored-program machines. Then, after it had thus proved its usefulness, floating-point arithmetic was provided as a built-in feature, starting with the IBM 704 and NORC computers, and this gave an enormous increase in speed over the subroutines. Today floating-point operation is available, at least as an option, on all computers intended to be used full- or part-time on substantial scientific computing applications. In view of the almost universal use of floating-point arithmetic, it is remarkable that there is very little literature on the subject.

In floating-point (FLP) arithmetic each number has its own *exponent* (or scale factor) E, as well as a numerical part, the *fraction* F. The pair (E,F) represents the floating-point number

$$FR^E$$

[1] Engineering Research Associates, W. W. Stifler, Jr., editor, "High-speed Computing Devices," p. 188, McGraw-Hill Book Company, Inc., New York, 1950.

[2] *Ibid.*, p. 186.

where E is a signed integer, and F is a signed fraction.[1] The exponent is variable and determines the true position of the decimal or binary point of the number; whence the name *floating point.*

The rules for combining FLP numbers follow directly from elementary arithmetic and the law of exponents.[2]

Multiplication:

$$(E_1,F_1) * (E_2,F_2) = (E_1 + E_2, F_1 * F_2) \tag{8.1}$$

Division:

$$(E_1,F_1) \,/\, (E_2,F_2) = (E_1 - E_2, F_1 \,/\, F_2) \tag{8.2}$$

Addition-Subtraction:

$$(E_1,F_1) \pm (E_2,F_2) = \begin{cases} [E_1, (F_1 \pm F_2R^{-(E_1-E_2)})] & \text{if } E_1 \geqq E_2 \\ [E_2, (F_1R^{-(E_2-E_1)} \pm F_2)] & \text{if } E_1 < E_2 \end{cases} \tag{8.3}$$

Multiplication [Eq. (8.1)] and division [Eq. (8.2)] are straightforward—the fractions are multiplied or divided, and the exponents are added or subtracted, respectively. Fractions and exponents can be manipulated simultaneously; so these operations take essentially the same amount of time as corresponding operations on fixed-point numbers of the same lengths as the fractions. (It should be noted, however, that fixed-point multiplication and division are often accompanied by extra scaling instructions, which are avoided with floating point. Thus the built-in FLP operations actually take less over-all time than fixed-point multiplication and division.)

Additions and subtractions [Eq. (8.3)] are more complex, because the radix points must be lined up first. This is done, quite automatically, by comparing the exponents and shifting the fraction with the smaller exponent to the right by an amount equal to the difference in exponents. The addition or subtraction of the fractions then proceeds, and the larger exponent is attached to the result. These steps are essentially sequential; so FLP addition and subtraction generally take more time than fixed-

[1] The term *mantissa* is often used instead of *fraction,* by a rather loose analogy with the terminology of logarithms. It is not necessary for the numerical part to be a proper fraction; it could just as well be made an integer or a mixed fraction by adjusting the exponent. This is largely a design choice. The *exponent* has been represented in many machines by an unsigned number obtained by adding an arbitrary constant; this unsigned number has been called the *characteristic.* The signed-exponent notation is more natural and simpler to use, especially when fixed-point arithmetic is to be performed on the exponent separately.

[2] Following a convention established by the FORTRAN programming system, the symbols * and / are used here for explicitly stated *multiply* and *divide* operations, in preference to other common symbols that are harder to type and write, such as ×, ·, and ÷.

point addition and subtraction. (The speed relation, therefore, is the reverse of that for multiplication and division.)

The basic rules of FLP arithmetic are thus stated quite easily, but they lead to several difficulties, of which some are fundamental and some can be resolved by more or less arbitrary decisions. One difficulty arises from the semilogarithmic nature of FLP numbers. If multiplication and division were the only arithmetical operations, the fraction part would not be necessary and high-speed addition of the logarithms (noninteger exponents) would suffice. Addition and subtraction, however, require the fraction parts, with the exponents restricted to integers, so as to permit the associated shifting operation. Hence FLP numbers are a mixture of rational numbers and logarithms, but the representation of a given number is not unique. For example, in decimal notation,

$$0.600*10^2 = 0.060*10^3 = 0.006*10^4$$

More important problems are presented by the singularities. Like fixed-point arithmetic, FLP arithmetic must provide for the occurrence of two quasi *infinities* (numbers whose absolute value is greater than the largest representable number; that is, the exponent exceeds its largest positive value) and of *zero* (the result of subtracting equal numbers), but the lack of a unique FLP representation introduces subtle questions. Thus a zero with a large exponent may represent a more significant quantity than a zero, or even nonzero, number with a small exponent. FLP arithmetic, unlike fixed-point arithmetic, must also allow for the possibility of two *infinitesimals* (numbers whose absolute value is less than the smallest representable number; that is, the exponent exceeds its largest negative value). Whereas in fixed-point notation the infinitesimals are indistinguishable from zero, a zero in FLP notation may have a true value quite different from an infinitesimal. (The ambiguity of zeros and infinitesimals does occur also in *scaled* fixed-point arithmetic, where the individual programmer has had to find his own way of programming around the difficulty. Built-in floating-point arithmetic removes the means of detecting singularities from the programmer's direct control; so the problem must now be faced by the designer.)

Among the situations that may be corrected by decision making, the most glaring concerns the treatment of division. Since there is no guarantee that $F_1 < F_2$, there is no guarantee that the quotient fraction will have a magnitude within the allowable range. This may be treated by ruling that, if $F_1 < F_2$, the division will proceed as in Eq. (8.2); but if $F_1 \geqq F_2$, assuming $F_1 \neq F_2 \neq 0$, the quotient will be

$$(E_1 - E_2 + p, F_1R^{-p} / F_2)$$

where p is an integer such that

$$F_2R^{-1} \leqq F_1R^{-p} < F_2$$

The result will always be arithmetically correct; in fact, it will be as precise as possible whenever $F_1 \geqq F_2$.

A different problem can arise in the case of a true addition (an addition involving operands of the same sign or a subtraction involving operands of different signs) whenever the resulting fraction exceeds the allowable range. This is a version of the familiar fixed-point-overflow problem and may be treated in the same way— by turning on an indicator to indicate that a 1 has been lost off the high-order end of the fraction, leaving any desired corrective action to the programmer. Another solution is to replace the result (E,F) automatically by $(E + 1,\ R^{-1}F)$, which is done in normalized arithmetic (below).

Solutions to these difficulties of FLP arithmetic will be discussed in subsequent sections.

8.3. Normalization

To improve precision it is desirable to maintain as many significant digits as possible. To this end all leading zeros may be removed from the result of an operation by shifting the fraction to the left and decreasing the exponent accordingly. Thus the decimal floating-point number (2, 0.006) when adjusted becomes (4, 0.600). Such numbers are called *normalized*, whereas numbers whose fractions are permitted to have leading zeros are called *unnormalized*. Floating-point arithmetic is called normalized or unnormalized depending on whether the normalization step is performed at the end or not. The operands of normalized operations do not as a rule have to be normalized numbers themselves.

Another function of normalization is to correct for overflow after an addition by shifting the result fraction to the right until the most significant digit is again in the high-order position and then appropriately increasing the exponent. Such a right shift to preserve the most significant digit may cause the loss of the least significant digit, but this is unavoidable.

The singular quantity $(E,0)$ cannot be normalized, since its fraction is all zeros; it is useful to regard $(E,0)$ as both a normalized and an unnormalized FLP number, since it may serve to indicate the order of magnitude of a result. Except for this and any other specially defined singularity, a normalized FLP number satisfies the inequality

$$R^{-1} \leqq |F| < 1$$

8.4. Floating-point Singularities

First-order singularities may occur when legitimate FLP operations are performed upon legitimate FLP operands with nonzero fractions. Singular results fall into three categories:

1. *Exponent overflow.* The exponent of the result exceeds the allowable exponent range. This result is outside the allowable number representation and may be likened to a positive or negative *infinity*, the sign being that of the fraction. The symbol $\pm\infty$ will be used to represent such a number.

2. *Exponent underflow.* The exponent of the result is negative and exceeds the allowable exponent range in magnitude. This result may be likened to a positive or negative *infinitesimal*, since it is outside (or inside!) the allowable number representation, is smaller than any legitimate quantity, and is definitely not zero (unless the fraction is zero). It has the same sign as the fraction. The symbol $\pm\epsilon$ will be used.

3. *Zero fraction.* This result can occur as a first-order singularity only from a true subtraction with equal operands:

$$(E,F) - (E,F) = (E,0)$$

The result is thus an indeterminate quantity with unknown sign, about which all that is known is that it satisfies the inequality

$$-R^{E-n} < (E,0) < R^{E-n}$$

where n is the number of fraction digits, and R is the radix. $(E,0)$ may cover a wide range of values including the true zero. The exponent E and the n zeros of the fraction indicate the maximum order of magnitude correctly; hence the name *order-of-magnitude zero* is often used.

In dealing with first-order singularities of the FLP number system, there are two points of primary importance: provision for unambiguous indication that a singularity has been created, and automatic tagging of the result. The zero fraction is suitable as a tag for an order-of-magnitude zero, but special tags are needed to distinguish exponent underflow and overflow from legitimate operands.

Second-order singularities—those created by performing arithmetical operations upon one or more first-order-singular floating-point quantities—cannot in general be handled automatically (and blindly) by the computer without creating serious problems. Nevertheless, it is reasonable to provide straightforward and fairly safe procedures for the standard automatic treatment of such cases, provided that the operands are again automatically tagged and that interrupt signals are available to permit programming of any other corrective action to take place, either immediately after the singular result is produced or later.

8.5. Range and Precision

Problems of range and problems of precision are often confused· Programmers sometimes go to FLP arithmetic when they actually require multiple precision, and even to multiple precision when what they actually need is more range.

Since the purpose of FLP arithmetic is to gain a vast increase in the range of representable numbers, range is seldom exceeded, but even when it is, range is not so serious a problem as precision. The exponent of a FLP number always indicates the range exactly, as long as the number is representable; one can, for instance, determine that a number is approaching, but has not exceeded, one of the limits of representation. If the exponent does overflow or underflow, the nature of the singularity may be indicated, or, if necessary, the range can be extended by using a multiple-precision exponent.

There is no corresponding mechanism to record loss of precision. The fraction always contains the same number of digits, and it is not immediately evident which digits are no longer significant—unless an order-of-magnitude zero is created by a single operation, so that all precision is lost at once. When serious precision loss takes place, it does not usually occur so dramatically. Rather, precision is lost by a process of gradual attrition, and its departure remains unnoticed unless some sort of running significance check is made. More of this later.

All numerical calculation reduces ultimately to the question of precision. Precision is, so to speak, limited at both ends of the calculation—limited at one end by the given precision of the input data and at the other end by the required precision of the result. Subject to considerations of time and cost, the gap between these limits must be adequately bridged by method and machine. If the machine is inadequate, the method used must make up for it; and if the method is inadequate (as often happens through insufficient time, insufficient analysis, or poor definition of a problem), the machine must be designed to take up as much slack as possible. Insufficiencies of method can be partially compensated for by machine checks of exceptional conditions, just as programming difficulties can be lessened by provision of a more powerful instruction set.

The two mechanisms that combine and interact to produce loss of precision in normalized FLP calculations are significance loss and round-off error. Volumes have been written about round-off error (perhaps more has been written about it than has been done about it), but only a few papers have been written about significance loss[1] (though it has possibly caused more noise to be accepted as pure signal). Most of the important work done on round-off error has in fact referred to fixed-point round-off and does not apply at all to the problems of normalized FLP round-off. Furthermore, it is doubtful that a valid FLP error analysis can be made without information on significance loss. The only procedure that limits the effect of both significance loss and round-off error is to increase the number of fraction digits used, with considerations of cost, size, and speed dictating how far it is practical to go in this direction.

8.6. Round-off Error

Performing any of the four basic FLP-arithmetic operations upon FLP operands with n-digit fractions gives a result fraction of from n to $2n$ digits. In multiplication the product always has $2n$ digits. In division there are two results, the quotient and the remainder, each with an n-digit fraction. In an addition or subtraction the result may range from n to $2n$ digits, depending upon the amount of preshift; *preshift* refers to the right shift of the fraction of the operand with the smaller exponent. (This shift may vary from no shift to a shift of $2n$ places; if the shift is more than $2n$ places, we define the two quantities as incommensurate and take the quantity with the larger exponent as the result, with suitable sign manipulation.) In normalized FLP arithmetic any operation may be followed by a normalizing left shift of less than $2n$ places to eliminate the leading zeros of the result fraction or by a normalizing right shift of one place to correct for overflow of the fraction. These shifts are referred to as *postshifts*. (Binary normalized FLP operations involve always at least one preshift or one postshift or both.)

In the interest of speed, economy of storage, and programming directness, the result of a FLP-arithmetic operation is ordinarily reduced to the same number of digits n as are possessed by the operands from which the result was produced. The simplest and fastest way to accomplish this is to shorten the result by merely dropping all except the high-order digits; this produces results that are consistently somewhat smaller in magnitude than the true value.

To avoid the downward bias of the simplest method it is common practice to *round* the result by adding $R^{-n}/2$ to the magnitude of the fraction before dropping the excess digits; this procedure also tends to reduce the magnitude of the error. This form of rounding poses difficulties: it

[1] J. W. Carr, III, Error Analysis in Floating Point Arithmetic, *Communs. ACM*, vol. 2, no. 5, pp. 10–15, May, 1959; R. L. Ashenhurst and N. Metropolis, Unnormalized Floating Point Arithmetic, *J. ACM*, vol. 6, no. 3, pp. 415–428, July, 1959; W. G. Wadey, Floating-Point Arithmetics, *ibid.*, vol. 7, no. 2, pp. 129–139, April, 1960.

must follow normalization, is therefore postponed until the operation is otherwise complete, and requires extra time and an extra register position to boot. A simpler but more artificial form of rounding is to force a 1 in the remaining least significant bit of the shortened result (in binary machines); although this decreases the bias, it does not decrease the maximum error, and it leads to logical problems.

Rounding is, therefore, not necessarily the best way to remove excess digits. In fact, automatic rounding on all FLP operations can lead to serious problems of error analysis, and it gives multiple-precision arithmetic a nightmarish quality. (How do you unround a number?) The most prudent approach is to give the user his choice of how to control *round-off error*—this term being used for the error resulting from the loss of the extra digits, whether true rounding takes place or not.

There are two important cases in which more than n digits are kept:

1. The extra digits, which are normally discarded, may be required for some special purpose—e.g., the remainder may have to be kept and tested for zero in order to know whether the divisor was a perfect divisor.

2. Multiple-precision arithmetic may be required because the natural precision of the machine is inadequate for the particular computation; so all $2n$ possible digits of the result must be made available.

Higher precision is actually obtainable at little extra cost for some important activities even in single-precision calculation. For example, one of the most frequently occurring activities in scientific or statistical problems is the calculation of the inner product $\Sigma a_i b_i$. This may be accomplished by *cumulative multiplication,* in which $2n$-digit products of n-digit factors are repeatedly added to the $2n$-digit partial sum, thus minimizing the effect of both round-off error and significance loss.

8.7. Significance Checks

Programmed significance checks have been used by programmers in a number of installations for many years and have proved effective in trapping many actual cases of total significance loss. When used with built-in FLP arithmetic, however, such a programmed significance check slows down effective arithmetic speeds by a considerable factor, for the significance check takes much more time than the actual arithmetic.

The significance check may be built in. There are two possibilities: either the check may be made in parallel with the operation, in which case there is no time loss, but roughly $\log_R n$ extra digits are required to keep the significance check (and such extra digits are required in all positions of data memory); or else a record of lost significance is encoded into the area of the fraction normally occupied by nonsignificant digits, requiring at least one extra flag digit and a relatively long time for encoding and decoding. Most users would rather keep any extra positions of storage to maintain more precision and use any extra equipment to improve the FLP instruction set itself.

Another approach involves the injection of deliberate *noise* into the computation, so that results affected by significance loss will have a very high probability of indicating the loss by differences between normal runs and "noisy" runs of the same problem. This approach, which requires little extra hardware and no extra storage, was chosen for the 7030. After an extensive search, the most effective technique turned out to be both elegant and remarkably simple.

By definition of ordinary normalized FLP operations, numbers are frequently extended on the right by attaching zeros. During addition the n-digit operand that is not preshifted is extended with n zeros, so as to provide the extra positions to which the preshifted operand can be added. Any operand or result that is shifted left to be normalized requires a corresponding number of zeros to be shifted in at the right. Both sets of zeros tend to produce numbers smaller in absolute value than they would have been if more digits had been carried. In the *noisy mode* these numbers are simply extended with 1s instead of zeros (1s in a binary machine, 9s in a decimal machine). Now all numbers tend to be too large in absolute value. The true value, if there had been no significance loss, should lie between these two extremes. Hence, two runs, one made without and one made with the noisy mode, should show differences in result that indicate which digits may have been affected by significance loss.

The principal weakness of the noisy-mode procedure is that it requires two runs for the same problem. A much less important weakness is that the loss of significance cannot be guaranteed to show up—it merely has a very high probability of showing up—whereas built-in significance checks can be made slightly pessimistic, so that actual significance loss will not be greater than indicated. On the other hand, little extra hardware and no extra storage are required for the noisy-mode approach. Furthermore, significance loss is relatively rare, so that running a problem twice when significance loss is suspected does not pose a serious problem. What is serious is the possibility of *unsuspected* significance loss.

In discussions of significance two points are often overlooked. The first of these is trivial: the best way of ensuring significant results is to use an adequate number of fraction digits. The second is almost equally mundane: for a given procedure, normalized FLP arithmetic will ordinarily produce the greatest precision possible for the number of fraction digits used. Normalized FLP arithmetic has been criticized with respect to significance loss, because such loss is not indicated by the creation of leading zeros, as it is with fixed-point arithmetic. In other words, the contention is not that normalized FLP arithmetic is more prone to significance loss than equivalent fixed-point arithmetic, which would be untrue, but that an equivalent indication of such loss is not provided. Loss of significance, however, is also a serious problem in fixed-point arithmetic; multiplication and division do not handle it at all correctly by means of leading zeros. (In particular, fixed-point multiplication may lead to serious or even total significance loss, which would not have occurred with normalized FLP arithmetic; and although leading zeros in addition and subtraction of fixed-point operands do give correct significance indications, the use of other operations and of built-in scaling loops frequently destroys entirely the leading-zeros method of counting significance.)

There are other points of common confusion between fixed- and floating-point calculation. For example, given a set of fixed-point numbers with a common scale factor, the most significant number is the one with the largest absolute value; accordingly, many optimal procedures depend upon selecting this element. Frequently, the equivalent normalized FLP procedure would be to select the element with most significance rather than the element of largest absolute value. In the absence of any information about significance, however, it is statistically best to pick the element of largest absolute value, since loss of significance is associated with a corresponding decrease in the exponent and so the element of largest absolute value does have the greatest probability of being also the most significant number. Similarly, fixed-point error analysis ordinarily concentrates on some statistical characterization of the absolute error, whereas in normalized FLP operations it is the relative error that is important. Thus a polynomial approximation should be chosen to minimize the appropriate statistical function of the relative error, rather than the absolute error. (The relative error in FLP calculations is analogous to the noise-to-signal ratio in information theory.)

8.8. Forms of Floating-point Arithmetic

It is difficult to formulate a single set of floating-point operations that would satisfy all requirements. Normalized operations are required for most of the heavy calculation, but there are uses for unnormalized operations that cannot be ignored. Unnormalized arithmetic is needed, for instance, to program multiple-precision operations; it may also be used for fixed-point calculation in lieu of separate high-speed fixed-point-arithmetic facilities that would otherwise be essential. (Thus the 7030 has high-speed floating-point arithmetic as basic equipment, and it was decided to omit high-speed fixed-word-length fixed-point operations. This is the inverse of the situation with the early scientific computers, which had only fixed-point arithmetic until a floating-point set was grafted on.) Again, in order to permit extended precision whenever necessary, double-length sums, products, and dividends (i.e., numbers with $2n$-digit fractions) should be available, but this would slow down all operations and penalize most applications, which require only single-length numbers (with n-digit fractions for operands and results). Hence both single- and double-length operations are desirable.

Another decision, which only the user can make, is whether to round the results or not. As mentioned before, true rounding tends to reduce errors but consumes extra time. Moreover, in actual practice, it is often desired to store the accumulator contents rounded to n digits while leaving the complete $2n$-digit result in the accumulator for further calculation.

The various procedures that result from decisions about normalization, rounding, and the treatment of extra precision and of singular quantities in reality define various FLP "arithmetics." A primary task in large-scale computation is determining which of these numerous "arithmetics" is really desired.

8.9. Structure of Floating-point Data

To each form of FLP arithmetic there corresponds a particular FLP data structure. Sometimes the same data structure can be used for different forms of arithmetic; normalized and unnormalized arithmetic are an example. In other cases different formats are required (as is obviously true for single- and double-precision arithmetic). The machine designer must decide which arithmetics and corresponding data formats to build into the machine and which to leave to programming. In a given machine environment it is not usually practical to implement all forms of FLP arithmetic and all formats that any potential user might possibly desire. The designer must, therefore, determine what facilities are needed to assist in programming the others.

The FLP number itself may be regarded as composed of at least two partially independent parts (the exponent and the fraction); this becomes four parts if we consider the signs attached to each and increases to five or six parts if we flag the exponent, the fraction, or the entire number. In many situations it is desirable to manipulate one or more of these parts independently of the others, and such manipulation has been a source of much added programming complexity on earlier computers.

The most fundamental question of numerical data structure is that of the radix. This has been considered in general terms in Chap. 5. The high storage efficiency of the binary system, as opposed to the decimal, is particularly important in extending both the range and the precision of the FLP number: a 10-bit exponent gives an exponent range of 1,023, whereas the same bits used in the 4-bit coded decimal representation will handle a maximum exponent of only 399.

FLP arithmetic really involves three radixes: the radix R_E used in the exponent representation, the radix R_F used in the fraction representation, and the FLP radix R used in the representation $(E,F) = FR^E$. In principle these three radixes are independent; in practice they are not. If we were doing only unnormalized multiplication and division, all three radixes could be arbitrary integers greater than unity. But the necessity of preshifting before addition and subtraction and of postshifting for normalized operations implies that the FLP radix R must be some positive, integral, nonzero power of the fraction radix R_F, since only shifts by integer amounts are meaningful.

The exponent radix R_E is still arbitrary. As a matter of fact, it would make perfectly good engineering sense in a decimal floating-point machine to make the FLP radix and the fraction radix both 10 and to let the exponent radix be 2. Thus, using the previous example of a 10-bit exponent, the range would be enlarged from 10^{399} for $R_E = 10$ to 10^{1023} for $R_E = 2$ (a factor of 10^{624}!), and the decoding circuits for driving the preshifter would be simplified. On the other hand, proponents of either radix are likely to extend their reasoning to the exponent as well; so the exponent radix is ordinarily chosen to be the same as the fraction radix.

Several binary floating-point machines have been designed to use the floating-point radix $R = 2^k$, where k is an integer greater than unity. If $k = 3$, the radix is octal; if $k = 4$, it is hexadecimal. The Los Alamos MANIAC II computer uses $k = 8$, that is, a FLP radix R of 256. The advantages of a larger FLP radix are twofold: the maximum range is extended from, say, R^m to R^{km}; and the number of times that pre- and postshifts occur is drastically reduced, with a corresponding decrease in the amount of equipment required for equivalent performance. There is just one disadvantage: precision is lost through increased round-off and significance loss, because, with FLP radix 2^k, normalized fractions may have up to $k - 1$ leading zeros. Such precision loss may partly be compensated for by decreasing the number of exponent bits and using the extra bits in the fraction instead. This reduces the gain in range in order to limit the loss in precision, but the advantage of reduced shifting is retained. It should also be noted that special procedures are available to reduce the actual amount of shifting, particularly for the binary radix; the average amount of postshifting needed with normalized FLP arithmetic and $R = 2$ may be reduced, at the cost of extra equipment, until it approximates that of $R = 8$.

In practice, the use of a larger FLP radix results in an operation more nearly resembling scaled fixed-point calculation, except that it is automatic. The designers of a particular FLP system must consider the radix problem in the light of the machine environment and the expected problem mix. There is no substitute for a careful statistical analysis of the various available procedures to determine the specific implementation.

FLOATING-POINT FEATURES OF THE 7030

8.10. Floating-point Instruction Format

The floating-point instructions in the 7030 use a tightly packed half-word format (Fig. 8.1), as do the indexing and branching instructions commonly associated with them in high-speed computing loops.

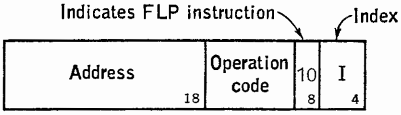


FIG. 8.1. FLP instruction format.

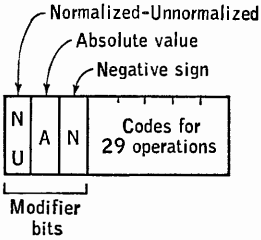


FIG. 8.2. Details of FLP operation code.

The operation code (Fig. 8.2) consists of 5 bits to encode 29 different FLP operations and 3 modifier bits which apply uniformly to any of the 29 operations. The three modifiers are:

1. *Normalization* modifier. This specifies whether postnormalization is to take place (normalized) or not (unnormalized).

2. *Absolute value* modifier. If set to 1, this specifies that the memory operand is to be considered positive, ignoring the actual sign in memory. (This modifier is analogous to the VFL *unsigned* modifier, except that in the fixed-length FLP format the sign position is always there, whether used or not.)

3. *Negative sign* modifier. If set to 1, this inverts the sign of the unreplaced operand, that is, the memory operand in a from-memory operation or the accumulator operand in a to-memory operation. It is applied after the *absolute value* modifier. Thus ADD and related operations are changed to *subtract* operations, etc. (This is the same as the corresponding VFL modifier.)

8.11. Floating-point Data Formats

The FLP number occupies a full 64-bit memory word. The reasons for choosing as the length of the memory word a number of bits that is a power of 2 are discussed in Chap. 5. Considerations of speed dictated that a FLP number be located in a single memory word, so as to avoid the time penalty of crossing word boundaries. This soon restricted the choice to 64 bits; experience had shown that the 36-bit word of the 704 would be too tight for a much more powerful machine but that lengths in the range of 50 to 60 bits would be adequate for most applications.

Sixty-four bits certainly seemed to be a liberal amount. A number longer than really necessary carries some penalty in extra equipment and possibly lower speed. (The possibility of a variable FLP number length, giving the user his choice of speed or storage efficiency, was discarded as impractical for reasons of both speed and cost.) Offsetting this penalty is the greater range and precision of single-length numbers, which reduces the amount of exception handling and permits fast single-precision operations to be retained in many large jobs that would otherwise require much slower multiple precision.

The basic data format is shown in Fig. 8.3. It consists of a 12-bit exponent field and a 52-bit fraction field including a 4-bit sign field. The exponent field consists of 10 numerical bits, an exponent sign bit, and an exponent flag to signal a previous overflow or underflow. The sign field contains the fraction sign bit (the sign of the number) and three data flags which, at the programmer's option, may be used to mark exceptional data, such as boundary values. It should be noted that the 11-bit signed exponent and the 52-bit signed fraction are each compatible with VFL data formats, so that VFL instructions can be used directly to execute those operations on parts of a FLP number for which there are no specialized FLP instructions. One example is multiplication or division of exponents.

The format of Fig. 8.3 is used for all FLP numbers in memory. The format in the accumulator is somewhat different (Fig. 8.4). For single-length numbers, the 12-bit exponent field and the 48-bit fraction field occupy corresponding positions in the left half of the accumulator. The 4-bit sign field, however, is stored in a separate sign-byte register (as in VFL operations). The low-order 4 bits in the left half of the accumulator are not used, and neither is the right half of the accumulator.

For double-length FLP numbers, that is, numbers with a 96-bit fraction, an additional 48 positions of the accumulator are activated; so the double-length fraction in the accumulator forms a homogeneous 96-bit number. The exponent and sign remain the same. Since the accumulator is 128 bits long, this leaves 20 bits unused in the right half. It should be noted that the unused parts of the accumulator (shown shaded in Fig. 8.4 for the two classes of operations) are always left undisturbed during FLP operations and may be used for temporary storage of other kinds of data.

Symbolically we can represent a single-precision FLP number as

$$(Ef,E,F,S,T,U,V)$$

where Ef is the exponent flag, E the (signed) exponent, F the (unsigned) fraction, S the fraction sign, and T, U, V the data flags. Then the single-length format in the accumulator is given by (Ef,E,F) with S, T, U, V in the sign-byte register. The double-precision FLP format in memory becomes the pair $(Ef_H,E,F_H,S,T_H,U_H,V_H)$, $(Ef_L,E-48,F_L,S,T_L,U_L,V_L)$. The exponent flags are usually, but not always, the same; the exponents differ by 48, except when one part is singular and the other part is not; the fractions are independent, F_L being a continuation of the fraction F_H; the sign bits are identical, but the data flags may be independent. The double-length FLP number in the accumulator, however, is quite different: it is (Ef_H,E,F_H,F_L), with the sign-byte register containing S, T, U, V.

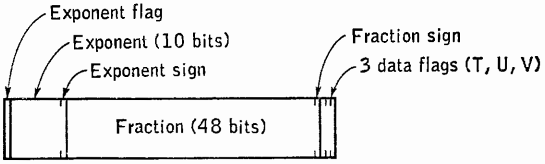


FIG. 8.3. FLP data format.

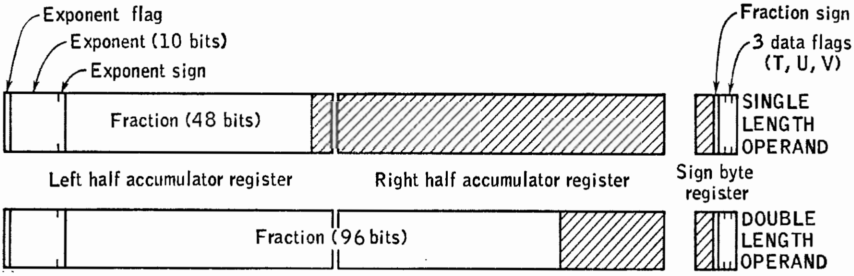


FIG. 8.4. FLP accumulator formats. Shaded areas are left undisturbed.

A special *store* instruction is available to convert the low-order part of a double-length number in the accumulator to a proper FLP number in memory with correct exponent and sign.

It should be noted that a word may have a nonsingular representation in the double-length accumulator, although the corresponding number in memory is singular (i.e., the low-order exponent has an exponent flag).

8.12. Singular Floating-point Numbers

The range of numbers representable by the above format is indicated schematically in Fig. 8.5. Normal numbers ($\pm N$) are bounded by infinities ($\pm\infty$) and infinitesimals ($\pm\epsilon$). Not shown is the previously discussed order-of-magnitude zero (OMZ), which may result from subtracting numbers in the N range and may thus have a true value anywhere in this range. (An OMZ is different from the true zero, shown as the dividing line between positive and negative numbers.)

The representation of singular numbers in the 7030 is straightforward:

Infinity (∞). The exponent flag is set to 1, and the exponent sign is positive. Hence this is also called an *exponent flag positive* condition (XFP).

Infinitesimal (ϵ). The exponent flag is set to 1, and the exponent sign is negative. Hence this is also called an *exponent flag negative* condition (XFN).

Zero fraction, or *order-of-magnitude zero* (OMZ). All 48 bits of the fraction (or all 96 bits for results of double-length operations in the accumulator) are 0.

The rules for doing arithmetic with infinities or infinitesimals as operands follow the notion that an infinity is larger in magnitude than any normal number and an infinitesimal is smaller in magnitude than any normal number. All infinitesimals behave arithmetically like zeros, but an infinitesimal with a zero fraction (an *XFN zero*) is the closest to a true zero. The sign of a singular number is the fraction sign and is manipulated like the sign of a normal number.

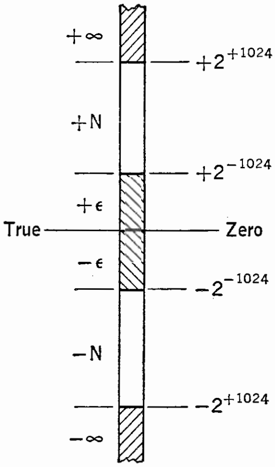


FIG. 8.5. FLP number range. Representable numbers *N* lie in unshaded areas.

Thus the rules for arithmetically combining a normal number *N* with an infinity or infinitesimal are evident from the definitions. For addition and subtraction these rules are

$$\begin{array}{cc} \infty \pm N = \infty & N - \infty = -\infty \\ N \pm \epsilon = N & \epsilon - N = -N \end{array} \tag{8.4}$$

For multiplication and division the usual rule of signs determines the fraction sign of the result, and the magnitude is given by

$$\begin{array}{cc} \infty * N = \infty & \epsilon * N = \epsilon \\ \infty / N = \infty & \epsilon / N = \epsilon \\ N / \infty = \epsilon & N / \epsilon = \infty \end{array} \tag{8.5}$$

Some of the operations on two singular numbers likewise follow from their definition:

$$\begin{array}{cc} \infty + \infty = \infty & \infty * \infty = \infty \\ \infty \pm \epsilon = \infty & \epsilon * \epsilon = \epsilon \\ \epsilon \pm \epsilon = \epsilon & \infty / \epsilon = \infty \\ \epsilon - \infty = -\infty & \epsilon / \infty = \epsilon \end{array} \tag{8.6}$$

Other operations have indeterminate results (since in the discrete number system of a digital computer there is no satisfactory substitute for L'Hôpital's rule). It was thought important to propagate singularities through the course of calculation, and, of the two possibilities, infinity and infinitesimal, infinity was chosen arbitrarily because the programmer would consider it more alarming:

$$\begin{array}{cc} \infty - \infty \equiv \infty & \infty / \infty \equiv \infty \\ \infty * \epsilon \equiv \infty & \epsilon / \epsilon \equiv \infty \end{array} \tag{8.7}$$

[The purist may argue that the results in (8.7) should have a zero fraction part as well as a positive flagged exponent, which would indicate that the number is both indeterminate and outside the normal range. This distinction may be programmed in the rare case when it is important.]

In comparing infinities and infinitesimals, the inequality relations are self-evident:

$$+\infty > +N > +\epsilon > -\epsilon > -N > -\infty \tag{8.8}$$

When infinities of like sign are compared, they are considered equal; similarly, infinitesimals of like sign are equal:

$$\begin{array}{cc} +\infty = +\infty & +\epsilon = +\epsilon \\ -\infty = -\infty & -\epsilon = -\epsilon \end{array} \tag{8.9}$$

[Definition (8.9) is consistent with some but not all of the rules (8.4) to (8.7). For example, $\epsilon - \epsilon = \epsilon$ implies that infinitesimals are equal, but $\infty - \infty = \infty$ implies that infinities are different. This problem arises because no consistent logic applies when both operands are singular.]

In the case of order-of-magnitude zero (OMZ), the operation takes its normal course. So long as only one operand is an OMZ, this gives a reasonable result. Since an OMZ represents a range of indeterminacy, multiplication or division by a legitimate number simply increases or decreases the size of the range of indeterminacy appropriately. Division by an OMZ is suppressed and, when it would occur, the *zero divisor* indicator is turned on. Addition of an OMZ to either a legitimate operand or another OMZ produces either a legitimate result or an OMZ, depending upon the relative magnitudes of the quantities involved. (However, comparison operations call equal all OMZs whose exponents differ by less than 48.)

The single-length product of two OMZs raises a particularly difficult problem. We define

$$(E_1,0) * (E_2,0) = (E_1 + E_2,\ 0) \tag{8.10}$$

The double-precision product of the two zero fractions was a 96-bit zero and correctly represented the result of the multiplication. When the number is cut to single-precision length, however, 48 meaningful 0s are thrown away.

In a sense the product has been "normalized" 48 places. This may be seen by considering that $(E,0)$ may be approximately represented by $(E,2^{-48})$, and Eq. (8.10) may be replaced, to within a small error, by

$$(E_1,2^{-48}) * (E_2,2^{-48}) = (E_1 + E_2,\ 2^{-96})$$

After truncation the result will henceforth be indistinguishable within 48 bits from $(E_1 + E_2,\ 2^{-48})$, a number that is too large by a factor of 2^{48}.

Thus (8.10) is the correct definition for the double-length product in the accumulator, whereas for storing in memory the correct answer should be $(E_1 + E_2 - 48,\ 0)$. Since only the programmer can decide when to store a result, the exponent adjustment can only be made by programming. For this purpose a *zero multiply* indicator is turned on whenever multiplication results in a zero fraction. The programmer may then define any desired exponent adjustment or choose to ignore the condition.

The zero problem in multiplication would perhaps not be so serious, were it not for the fact that OMZs are frequently successively squared, which can lead to an unrestricted growth of the exponent, creating a large indeterminacy that can wipe out legitimate numbers.

For the square root we have automatically $(E,0)^{1/2} = (E/2,\ 0)$ if E is even, or $[(E + 1)/2,\ 0]$ if E is odd. To be compatible with the foregoing, the root should really be $[(E/2) - 24,\ 0]$ or $[(E + 1)/2 - 24,\ 0]$; otherwise squaring and square-rooting are not inverse procedures. In this case, however, the magnitude of the result is made too small. It loses its ability to grow without bound and hence most of its ability to damage the calculation. For this reason no indicator is set for the square root. (If an indication is desired, it may be obtained by setting the fraction sign negative on all OMZs and using the *imaginary root* indicator.)

As both computers and computations have increased in complexity, the amount of analysis per instruction written must decline; so automatic treatment of FLP singularities becomes more important. The absence of test instructions also leads to cleaner programs, making coding and debugging much easier. In some physical problems, not only zeros and infinitesimals but also OMZs are common: a steady-state condition may prevail with everything initially at rest, and the difference equations used to move out in time are likely to create OMZs during the early part of the calculation. OMZs must either be handled by the system or circumvented at the cost of considerable extra analysis and programming. In the 7030 these are handled automatically and may die out during the course of the calculation, so that no special starting procedures are required. A different situation, in which the automatic handling of singular quantities is important, is that in which they are produced unexpectedly as intermediate quantities in a calculation, but have no effect on the result. The fact that such singularities may arise infrequently, and may not even arise at all, does not obviate the necessity for dealing with them when they do occur.

8.13. Indicators

The FLP indicators fall into three categories: (1) those which are set by both VFL and FLP operations and have analogous meaning for both; (2) those which are set only by FLP operations; and (3) the *noisy mode* indicator.

Indicators Common to VFL and FLP Operations

The following indicators are shared by VFL and FLP operations:

1. *Arithmetic result indicators.* They show whether the result is less than zero, zero, or greater than zero, or whether the result sign is negative.
2. *Comparison indicators.* They indicate after a comparison operation whether the accumulator operand was low, equal, or high relative to the memory operand.
3. *Lost carry* and *partial field.* These apply only to unnormalized operations because the conditions are otherwise taken care of by normalization.
4. *Zero divisor.* It indicates an attempt to divide by a zero fraction.
5. *Data flag indicators.* They signal flagged operands.
6. *To-memory operation.* This indicator distinguishes between *store* and *fetch* operations, for easier exception programming.

FLP Indicators

The indicators that are private to FLP operations are listed below:

1. *Exponent range indicators.* These indicators signal that the result exponent E lies in a certain range; they are as follows:

a. *Exponent overflow.* $E \geqq +2^{10}$. The exponent flag *Ef* is turned on. This indicator shows that an overflow has been *generated* during the current operation.

b. *Exponent range high.* $+2^9 \leqq E < +2^{10}$.

c. *Exponent range low.* $+2^6 \leqq E < +2^9$.

d. *Exponent underflow.* $E \leqq -2^{10}$. *Ef* is turned on. This indicator shows that an underflow has been *generated* during the current operation.

e. *Exponent flag positive.* $E \geqq +2^{10}$ and *Ef* was already on. This indicator shows that an overflow has been *propagated;* that is, the overflow was forced because the operand was an infinity.

The *exponent overflow* and *exponent underflow* indicators signal that the number has already gone out of range. The *exponent range high* and *exponent range low* indicators may be used as a warning that numbers have entered a larger range than anticipated *before* the damage has been done, since the result is still a representable number. The last indicator warns that the operand was an *infinity*, in case corrective action other than the built-in procedure is desired. A corresponding indicator for infinitesimals is not provided, since these are less likely to cause serious damage; if flagging is desired, the programmer could turn on a data flag after detecting the original exponent underflow.

2. *Lost significance.* Adding or shifting nonsingular operands has resulted in a zero fraction, leaving no significant bits.

3. *Zero multiply.* A multiplication has resulted in a zero fraction; so the result may not indicate the proper order of magnitude.

4. *Preparatory shift greater than* 48. During addition the exponent difference is found to be greater than 48; so some or all of the bits of the number with the smaller exponent have been shifted off the right end of the double-length result and are lost. In a single-precision sense, the operands are incommensurate.

5. *Imaginary root.* The operand for a square-root operation is negative.

6. *Remainder underflow.* Same as *exponent underflow*, except that it applies to the remainder produced after a double-length division, whereas exponent underflow after division applies to the quotient.

Noisy Mode Indicator

This indicator, when on, causes all normalized FLP operations to be performed in the *noisy mode,* where 1s replace 0s at the right.

The *noisy mode* indicator is a programmed switch, which can be turned on and off only by the programmer. It is placed among the other indicators in order to simplify program interruption. When interruption occurs, the indicator register is stored in memory and subsequently reloaded. Thus the *noisy mode* and other indicators are restored to the same state they were in at the point of interruption.

8.14. Universal Accumulator

The principle of the universal accumulator, where the accumulator is the source of the major implied operand and the destination of the major result of every arithmetical operation, was stated already in Chap. 7. It deserves restating here because it is an important factor in reducing the housekeeping burden of floating-point calculations and increasing their speed.

8.15. Fraction Arithmetic

The distinction between integer and fraction arithmetic has already been discussed in Chap. 7, where reasons are given for choosing integer VFL arithmetic. Fraction arithmetic, on the other hand, was preferred for floating-point operations in the 7030.

The fraction notation is a natural choice for numbers that approximately represent continuously variable mathematical quantities to a given number of significant digits, the remaining low-order digits being discarded. This is especially so when the numbers are normalized for maximum precision. In multiplication, for example, it is desirable to have available either a single-length or a double-length product for single- or double-precision work. If fraction arithmetic is used, the high-order part of a normalized double-length product is the same as the corresponding (unrounded) single-length product. With integer arithmetic the two have different positions and exponents, which makes this convention a little more awkward, although one can readily formulate a consistent set of rules for integer FLP arithmetic. In most respects the practical difference between fraction and integer FLP numbers is just a matter of changing all exponents by an additive constant.

8.16. Floating-point-arithmetic Operations

The FLP operations may be placed in three categories: (1) single-length operations (which produce a result with a 48-bit fraction), (2) double-length operations (which produce a 96-bit fraction), and (3) special operations.

Internally, operations are actually performed in double-length form. Thus the parallel adder for the fractions is 96 bits long, and 48-bit operand fractions are extended with 0s (or 1s in single-length noisy mode) after shifting, to make up 96 bits at the input of the adder. A full 96-bit result is produced. The difference between single- and double-length operations is primarily whether the fraction part of the accumulator operand is taken to be 48 or 96 bits long and whether the result in the accumulator, after normalization if specified, is truncated to 48 bits or not.

The fraction arithmetic takes place in 96-bit registers which are different from the accumulator registers. Thus it becomes possible, in single-length operations, to leave unmolested all bits to the right of the 48th fraction bit in the accumulator, even though intermediate results may require more than 48 bits of register space.

Since the bulk of the computing was expected to be in single precision, the design of the arithmetic unit was biased in favor of performing single-length operations at high speed, sometimes at the sacrifice of speed for double-length operations. Thus no time is taken to preserve the rarely needed remainder in single-length DIVIDE, even though this remainder is obviously generated, leaving the dressing up and storing of the remainder in the remainder register to DIVIDE DOUBLE.

Many of the basic FLP operations are analogous to the VFL operations of the same name (Chap. 7):

LOAD
LOAD WITH FLAG
STORE
STORE ROUNDED
ADD
ADD TO MAGNITUDE
ADD TO MEMORY
ADD MAGNITUDE TO MEMORY
COMPARE
COMPARE FOR RANGE
MULTIPLY
LOAD FACTOR
MULTIPLY AND ADD
DIVIDE (except that no remainder is kept in FLP)

The nature of these operations is indicated by their names and follows from what has been said in previous sections. A summary of all operations is given in the Appendix. If more detail is desired, the reader is referred to the 7030 Reference Manual. A few comments will be made here on certain specific features that will be important in subsequent discussion.

STORE ROUNDED provides a means of storing a rounded single-precision number in memory while leaving the original, unrounded, double-precision number in the accumulator for any further calculation. There is no automatic rounding in any other operation. Rounding is performed only when and where desired. Rounding is done by adding a 1 to the 49th fraction bit of the absolute value of the accumulator operand; rounding is followed by normalization, if specified, and storing of the high-order 48 bits.

The unnormalized *add* operations are interpreted to mean that there is no normalizing right or left shift after the addition. Consequently, any carry out of the high-order position of the fraction is lost, and the *lost carry* indicator is turned on. This feature is important in pseudo fixed-point arithmetic. There is no lost carry in normalized addition, of course; a right shift with exponent adjustment takes care of the matter.

MULTIPLY AND ADD is designed for cumulative multiplication. The product of the memory operand and of the operand in the factor register (previously loaded with a LOAD FACTOR instruction) is formed and then added to the accumulator contents. The sum is a double-precision number. Thus in important calculations—like forming the inner product $\Sigma A_i B_i$, which can be done with a three-instruction loop—the double-precision sum avoids round-off error until a single STORE ROUNDED is given at the end. MULTIPLY AND ADD is the only double-length operation in the above list of basic operations; all others are single-length.

Not shown in the above list are two instructions, COMPARE MAGNITUDE and COMPARE MAGNITUDE FOR RANGE, which correspond to the VFL operations COMPARE FIELD (FOR RANGE) in that the accumulator sign is ignored in the comparison; the difference in nomenclature arose because the VFL operations may include only a partial accumulator field, whereas the FLP operations always deal with the entire operand.

Two other single-length operations occur only in the FLP repertoire, since they did not seem so important for VFL use. One is RECIPROCAL DIVIDE, which is the same as DIVIDE but with dividend and divisor interchanged; the other is STORE ROOT, which extracts the square root of the accumulator operand and stores it in memory.

The double-length operations (we intentionally avoid the term *double-precision* because only the accumulator operand is really of double precision, the memory operand necessarily being of single precision, and so the operations are at best of "one and a half precision") include the following variations of the single-length operations:

LOAD DOUBLE
LOAD DOUBLE WITH FLAG
ADD DOUBLE
ADD DOUBLE TO MAGNITUDE
MULTIPLY DOUBLE
DIVIDE DOUBLE
STORE LOW ORDER

The double *load* operations reset all 96 fraction bit positions in the accumulator to 0 before loading the single-length memory operand, whereas the single *load* operations affect only the high-order 48 fraction positions. The double *add* operations combine a single-length memory operand with a double-length accumulator operand and return a double-length result to the accumulator. To store a double-length accumulator operand in memory, it is necessary to create a pair of single-length operands; this is done by using STORE, for the high-order part, and STORE LOW ORDER, which attaches the correct exponent ($E - 48$) and the sign to the low-order part to form a proper FLP number. Normalization may be specified if desired. Loading a double-precision number pair may be accomplished by LOAD DOUBLE followed by ADD DOUBLE, specifying the operand in either order since the exponents take care of themselves.

Multiplication, whether single or double, operates only on single-length factors from memory and from the accumulator. MULTIPLY and MULTIPLY DOUBLE differ in whether a single-length or double-length product is returned to the accumulator.

As might be expected, division is the most complex of the FLP operations to implement, because there are many exceptional conditions to be considered if they are not to be a burden on the programmer. The principles followed were that (1) no scaling should be required in advance, and (2) the quotient should be developed with maximum precision. We must distinguish here between normalized and unnormalized division.

In normalized division the first step is to normalize both the dividend and the divisor. The quotient is then developed. Since it is still possible for the normalized dividend fraction to be greater than the normalized divisor fraction, the quotient may have an overflow bit and require a single right shift for normalization; otherwise the quotient will be already normalized.

Even for unnormalized division the divisor is fully normalized, so as to guarantee the greatest quotient precision. The dividend, however, is normalized only to the extent that the amount of the left shift does not exceed the left shift of the divisor. If the dividend has as many or more leading zeros than the divisor, both will have been shifted by the same amount; the difference between dividend and divisor exponents is then still the correct quotient exponent, but the quotient fraction may have leading zeros as in any other unnormalized operation. If the dividend has fewer leading zeros than the divisor, it cannot be shifted so far. In the fixed-point sense the division is illegitimate, since the quotient will overflow (which also happens when the number of leading zeros in the dividend and the divisor are the same and the dividend fraction is equal to or greater than the divisor fraction). So as not to require the programmer to test and scale his numbers beforehand to avoid this situation, the division is carried out and the scale factor is made available for adjustments only if and when overflow occurs. The procedure is as follows.

The dividend is normalized either as far as it will go or as far as the divisor, whichever requires the lesser amount of shift. Division then proceeds as in the normalized operation, and the quotient exponent is adjusted for the amount of shift that occurred. The difference between the amount of left shift of the divisor and the left shift of the dividend is entered into a counter, the left-zeros counter, which is conveniently available for this purpose; to this a 1 is added if the quotient had to be shifted right once to remove the overflow. If the final counter setting in unnormalized division is greater than zero, the *partial field* indicator is turned on as a signal. The counter contains the proper scale factor. If the left-zeros counter contents are zero, the dividend was shifted as far as the divisor, the quotient did not overflow, and no scaling is required. (The counter contents cannot be negative.)

DIVIDE DOUBLE differs from DIVIDE in several respects. A double-length dividend in the accumulator is used. A correct 48-bit remainder corresponding to a 48-bit quotient is produced and deposited in a separate remainder register (whereas DIVIDE produces no remainder). The quotient is left in the accumulator; it is a 48-bit number in DIVIDE, but a 49-bit number in DIVIDE DOUBLE. The 49th quotient bit is intended to be used with STORE ROUNDED to obtain a rounded 48-bit quotient in memory, but it does *not* affect the magnitude of the remainder. Thus the remainder has the correct value for programming extended precision. (Strictly speaking, the remainder also has 49 bits when the normalized dividend fraction equals or exceeds the normalized divisor fraction. Only the high-order 48 remainder bits are preserved. If a low-order 1 is thus dropped in unnormalized division, the *lost carry* indicator is turned on, so that a correction may be programmed when desired.)

Four special operations on the accumulator operand, which alter the fraction or exponent part independently except for possible normalization after an addition, complete the FLP set:

ADD TO FRACTION
SHIFT FRACTION
ADD EXPONENT
ADD IMMEDIATE TO EXPONENT

The question naturally arises why these special operations are provided in the FLP set if the same functions could be performed by VFL instructions. An important reason is that FLP instructions are faster and take up only a half word each. More decisive is the fact that VFL operations would not set the special FLP indicators.

8.17. Fixed-point Arithmetic Using Unnormalized Floating-point Operations

As has been mentioned before, there are two ways of performing binary fixed-point arithmetic in the 7030. One way, which is fast but relatively wasteful of storage, is to use unnormalized FLP operations. The other way is to perform binary VFL operations; this uses storage efficiently but is slower.

With unnormalized FLP arithmetic a fixed-point fraction f is ordinarily represented by the FLP number $(0,f)$. It is clear from definitions (8.1) to (8.3) that addition, subtraction, and multiplication of such numbers result in numbers of the same kind, so long as the fraction has enough bits to avoid overflow. Division produces such numbers only if the divisor fraction is greater in magnitude than the dividend fraction. Otherwise, the quotient is (k,f), where $k > 0$; this is a correct quotient, but it is no longer of the pseudo fixed-point form $(0,f)$. As discussed earlier, the quantity k is available in the left-zeros counter for use by the program in scaling results after the *partial field* indicator signals the condition.

Treatment of singularities is indicated by Eqs. (8.4) to (8.7). It should be noted that multiplication and division of singular quantities, as executed automatically in the 7030, are not always inverse operations.

8.18. Special Functions and Forms of Arithmetic

In planning the FLP instruction set, consideration was given to the implementation of several common functions other than the basic arithmetical operations, such as logarithmic and trigonometric functions, complex-number arithmetic, polynomial evaluation, and the vector inner product. It was found that the high degree of concurrent operation within the CPU reduced the time spent on housekeeping instructions so much that built-in macro-instructions would not be appreciably faster than programmed macro-instructions, and they would be much less flexible.

The square-root function is an exception. It was built in because it could be carried out economically by an algorithm quite similar to the division algorithm chosen.

8.19. Multiple-precision Arithmetic

Built-in double-precision operations were among the special forms of FLP arithmetic that were considered but rejected because of insufficient speed advantage. A second reason for not providing such operations directly was the greater fraction length of the 7030, which would minimize the need for double-precision arithmetic. (Double-precision accuracy on the 7030 is more than 3.5 times single-precision accuracy on the 704.)

When the occasion for extending precision arises, furthermore, double precision is not necessarily sufficient; so triple- or higher-precision programs would have to be written anyway. The step from double to triple or quadruple precision will be as important as the step from single to double, and there is little justification for favoring the latter to the detriment of the former. Accordingly, the objective in the 7030 was to facilitate the programming of any multiple-precision arithmetic. The facilities provided include the double-length accumulator, appropriately defined unnormalized instructions, and exception indicators.

Tables 8.1 and 8.2 illustrate programs for double-precision addition and multiplication, respectively. These examples assume that a double-precision operand A is in a pseudo accumulator at memory addresses 200.0 (high-order part) and 201.0 (low-order part). The second operand B is at memory addresses 202.0 and 203.0. The result is to be returned to the pseudo accumulator.

The addition program illustrated takes six instructions, and the

Table 8.1. Double-precision Addition

Form $C = A + B = a_H + a_L + b_H + b_L$ where the subscripts H and L indicate high-order and low-order parts of each double-precision number.

Location	*Statement*	*Notes*
100.0	LOAD DOUBLE (FU), 201.0	
100.32	ADD DOUBLE (FN), 203.0	(1)
101.0	ADD DOUBLE (FN), 200.0	
101.32	ADD DOUBLE (FN), 202.0	(2)
102.0	STORE (FU), 200.0	
102.32	STORE LOW ORDER (FU), 201.0	(3)
200.0	DATA, AH	Pseudo accumulator
201.0	DATA, AL	
202.0	DATA, BH	
203.0	DATA, BL	

Notes: (1) Add low-order parts.
(2) Add high-order parts last for greatest precision.
(3) Result in pseudo accumulator.
(FU): unnormalized floating-point.
(FN): normalized floating-point.
100.32: bit 32 of word 100, that is, the right half word.

Table 8.2. Double-precision Multiplication

Form $C = A * B = a_H b_H + a_L b_H + a_H b_L$ (approximately). (Omitting the product term $a_L b_L$ may cause an error of 2^{-96} in the fraction magnitude.)

Location	*Statement*	*Notes*
100.0	LOAD (FU), 200.0	
100.32	MULTIPLY DOUBLE (FU), 203.0	(1)
101.0	LOAD FACTOR (FU), 202.0	
101.32	MULTIPLY AND ADD (FN), 201.0	(2)
102.0	MULTIPLY AND ADD (FN), 200.0	(3)
102.32	STORE (FU), 200.0	
103.0	STORE LOW ORDER (FU), 201.0	
200.0	DATA, AH	Pseudo accumulator
201.0	DATA, AL	
202.0	DATA, BH	
203.0	DATA, BL	

Notes: (1) Form $a_H b_L$.
(2) Add $a_L b_H$.
(3) Add high-order term $a_H b_H$ last.

multiplication program takes seven. For double-precision addition only, it is possible to hold the implied operand in the real accumulator, and no more than two ADD DOUBLE instructions are needed in that case. This compares with at least twelve and sixteen instructions, respectively, needed for double-precision FLP addition and multiplication in the IBM 704, which has no special facilities for multiple precision. The IBM 704 figures are a minimum; they allow for testing only once for accumulator overflow and quotient overflow. A practical 704 program may require more instructions for additional tests and sign adjustments, the actual number being a matter of individual needs. The length and intricacy of double-precision programming for the 704 make it advisable to use subroutines; whereas the 7030 programs are short enough to justify either writing the few instructions needed into the main program or using macro-instructions to compile them. The net result is a substantial reduction in the ratio of execution times for double- and single-precision arithmetic.

Triple- and higher-precision arithmetic is more complex for both machines, but the 7030 facilities again provide an advantage.

8.20. General Remarks

The key problems in planning and implementing a normalized floating-point instruction set in a digital computer involve, first of all, attaining the highest performance consistent with the required precision and range, and, second, a really adequate instruction set. By its very nature, the FLP instruction set is highly specialized and will always be incomplete. For this reason the accent must be on very high performance for specialized operations. Insofar as completeness and generality of the FLP instruction set have any meaning at all, it is in the facilities for the important FLP "arithmetics" and representations and for conversion among them. Symmetry of the instruction set is important, both because the instructions added for symmetry are likely to be important on their own and because symmetry simplifies the programming system.

The goal of highest possible performance must also be viewed in the context of the total operating system within which the computer proper is to perform: the automatic programming system, the programmers, the operators, and all the rules that they must follow. The goal is maximum total throughput, rather than maximum performance on any particular operation or set of operations. Nevertheless, it is obvious that the total throughput of a large-scale scientific computer will not be very high unless it possesses a fast, powerful FLP instruction set that performs very well all those operations which we know must be performed well by such an installation and performs adequately those operations which are only sometimes important.

Chapter 9

INSTRUCTION FORMATS

by W. Buchholz

9.1. Introduction

The importance of devising a good instruction set for a stored-program computer has been recognized by computer designers from the beginning. Most designers lavish a great deal of care on this aspect of their job, and so the instruction set contains the most easily distinguishable characteristics of a computer. It is not surprising, therefore, that different schools of thought have existed as to the best format for instructions. An especially popular subject for debate—more in private than in print—used to be whether it was easier to program with single-address or multiple-address instructions. By now this question has become rather academic. The importance of machine language programming is decreasing rapidly with the advent of problem-oriented programming languages. More attention is now focused on efficiency in the compiling of programs and on speed in running the finished product.

This is just one of several changes in environment which have resulted in a trend, over the years, away from the simple instruction formats of early computers. It may be instructive to illustrate the trend by some examples before considering the choice of formats for the 7030.

9.2. Earlier Instruction Languages

The instruction formats of some earlier computers are reviewed in Fig. 9.1.

The MIT Whirlwind computer represented the simplest of single-address instruction formats. It specified the operation and the address of one of the operands. The other operand was implied to be in a working register.

Note: Chapter 9 is an updated version of an earlier paper: W. Buchholz, Selection of an Instruction Language, *Proc. Western Joint Computer Conf.*, May, 1958, pp. 128–130.

The UNIVAC 1103 scientific computer, made by Remington Rand, uses a two-address scheme where two operands may be specified. The result may be returned to one of the two addresses.

The IBM 650 employs a different two-address scheme. Only one address specifies an operand, the other operand residing in an implied working register. The second address specifies the next instruction. This technique is advantageous in association with a revolving storage device, for it permits instructions to be located so that access time is minimized.

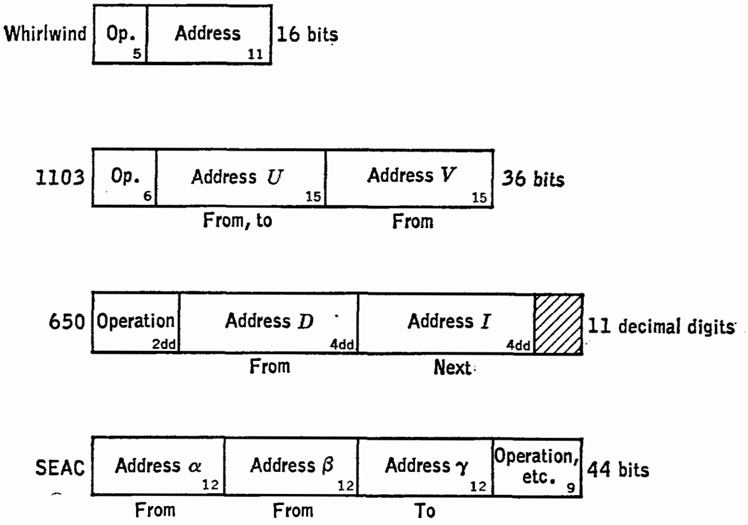


FIG. 9.1. Some classical instruction formats with one, two, and three addresses.

The National Bureau of Standards SEAC computer had available two instruction formats, one with three addresses and another with four. The three-address format is shown. Two operands and a result could be specified.

In retrospect one wonders whether, in each choice, fitting instruction words to a desired data-word length was not just as strong a factor as the intrinsic merit of the instruction format which gave rise to so much discussion. The distinction is mainly in whether one chooses to write related pieces of information vertically on a sheet of paper or horizontally. There was remarkably little difference among most of the early computers with respect to the operations that they performed.

In the early computers, simplicity was an important engineering consideration. After all, no one was quite sure in those days that the complex electronic devices parading under the imposing name of large-scale electronic data-processing machines would actually work.

The computers, however, turned out to be really usable and productive. They provided valuable experience for the designers of later computers. They clearly showed a need for much higher speed and much larger storage. At the same time, it became evident that speed could be gained and storage space saved by providing more built-in operations. A larger vocabulary can mean a quite drastic reduction in the number of instructions written and executed to do a given job. Floating-point arithmetic and automatic address modification, or indexing, are two features that have become standard equipment on scientific computers. Alphabetic representation and variable field length have similarly become accepted as built-in functions for business data processors. The instruction set has been growing steadily in size and complexity.

The desire to specify more things with one instruction has left no room in most instructions for more than one major address. The debate over multiple addresses has thus been settled by a process of evolution.

9.3. Evolution of the Single-address Instruction

The illustrations for this evolutionary process will be taken from experience gathered at IBM over a number of years. The experience is not unique, and similar examples could be chosen from other designs.

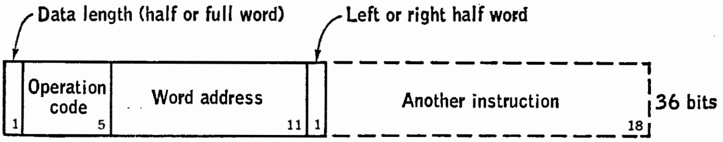


FIG. 9.2. Instruction format for IBM 701.

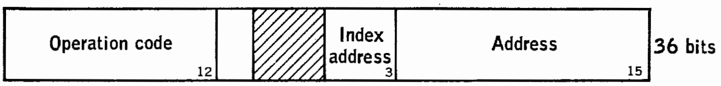


FIG. 9.3. Typical instruction format for IBM 704, 709, and 7090.

The IBM 701 followed the simple single-address pattern (Fig. 9.2). To make efficient use of the word length selected for data representation, two instructions are packed in each word.

The 704 and, later, the 709 and 7090 are all direct descendants of the 701, but they have a much bigger repertoire of instructions and features. As a result, the instruction has grown to fill the entire word (Fig. 9.3).

Bigger computing problems were found to require much larger memories. The address part of the instruction, therefore, was increased from 11 to 15 bits, giving sixteen times the capacity of the 701 memory. Three bits were added to specify indexing. The portion of the instruction that specifies the operation was increased from 5 to about 12 bits.

Part of this increase was needed because several times as many operations were made available to the user. Some bits were added to govern the interpretation of other bits, thus permitting more than one instruction format. For instance, there is a format in which two 15-bit quantities can be specified to provide a limited two-address repertoire in the 704.

For Project Stretch the evolution was carried a step further. More functions and more addressing capacity were desired. For other reasons, a much greater basic word length was chosen: 64 bits, or almost twice that of the 704. On the other hand, it became clear that extra memory accesses resulting from inefficient use of instruction bits would significantly reduce performance; so the more frequent instructions were compressed into a 32-bit format, which is shorter than the 704 instruction format. Since it was decided not to impose the restriction of compatibility with earlier machines, the 7030 instruction set could be made much more systematic and also more efficient than that of its predecessors.

9.4. Implied Addresses

We have already seen that single-address instructions differ from multiple-address instructions not in the number of operands required for a given operation but in that only one of the operands is located at an explicitly specified address, any other operands being located at *implied addresses.* A single-address *add* instruction, for instance, may have one implied operand in the accumulator, to which an explicitly specified operand is added. The sum replaces either the implied operand or the specified operand. Of the three addresses required by the operation, only one is stated explicitly. This gain in efficiency is nullified when *add* is preceded by *load* and followed by *store.* Therefore, implied addresses provide a gain in instructon-bit efficiency only when repeated reference is made to the same implied operand. In arithmetical operations repeated reference to the same implied operand occurs sufficiently often to justify the single-address instruction format.

As will be seen in the following section, the instruction formats for the 7030 still follow primarily the single-address pattern with an implied accumulator operand, but each format has one or more secondary addresses, such as index addresses. Some less frequently used instructions have two complete addresses, each accompanied by its own index address; and these do not require the accumulator.

It may be noted here that an accumulator may be designed to hold more than one implied operand. An interesting version of a multiple-operand accumulator has been called a *nesting store* by its originators,[1] or

[1] G. M. Davis, The English Electric KDF 9 Computer System, *The Computer Bulletin,* vol. 4, no. 3, pp. 119–120, December, 1960.

more descriptively a *push-down accumulator*. It may be pictured as a (theoretically infinite) stack of operands, with the most current operand on top. If a new operand is loaded at the top, the remaining ones are pushed down. If the topmost operand is removed and stored in main memory, all others below it are pushed up automatically. By avoiding instructions for transferring intermediate results to and from temporary storage locations, this scheme may show a gain in efficiency when a calculation can be arranged so that the order of using operands is: last in, first out (or any other prespecified rule of accession). The push-down scheme appeared too late to be evaluated for its effectiveness in the 7030.

9.5. Basic 7030 Instruction Formats

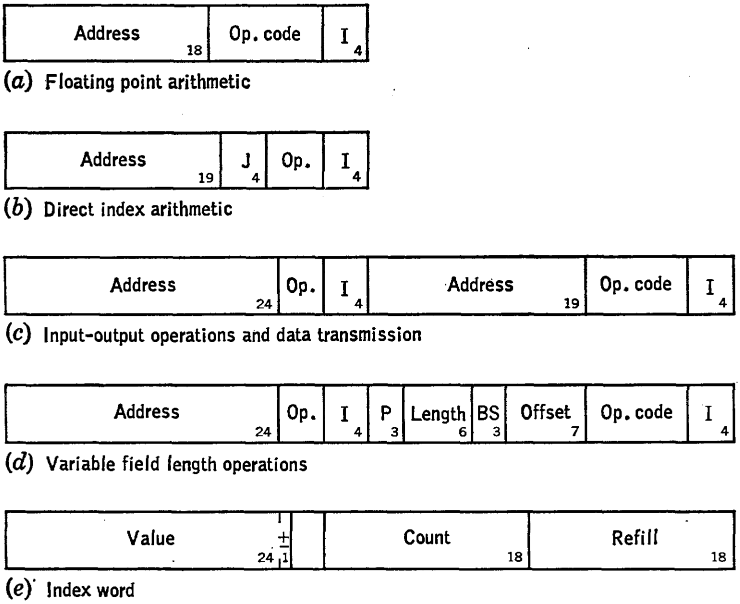
The basic pattern of instruction formats is shown in Fig. 9.4. A simple half-word format (Fig. 9.4*a*) consists of an address, an index address *I* to specify the index register to be used for automatic address modification, and a code *OP* that defines the operation to be executed. The 4-bit *I* address specifies either one of *fifteen* index registers (1 to 15) for address modification, or *no* address modification (0).

A second index address, *J*, is added to the format for index arithmetic to designate the index register on which the operation is to be performed (Fig. 9.4*b*). The 4-bit *J* address may specify one of *sixteen* index registers (0 to 15), including one register (0) that cannot participate in automatic address modification.

A full-word instruction consists essentially of two half-word formats, each half having an address, a modifier index address *I*, and an operation code *OP*. The operation code in the left half is merely a unique code to distinguish it from all the half-word instructions and to ensure proper interpretation of the right half. Full-word instructions may occupy a full memory word; or they may overlap the memory-word boundary, the left half being in one memory word and the right half in the next higher word. Thus full-word and half-word instructions may be freely intermixed.

A good example of a full-word instruction (Fig. 9.4*c*) is TRANSMIT, which may be used to transmit a word (or a block of words) from the memory area starting at the address in the left half of the instruction to the memory area starting at the address in the right half. Another instruction, SWAP, interchanges the contents of the two memory areas. Input-output transmission instructions use a similar format, except that the left address gives the number of an input-output channel, and the right address is used in an indirect fashion, specifying a control word which in turn defines the memory area (see Chap. 12).

The fourth example (Fig. 9.4*d*) is the format of the variable-field-length (VFL) operations. The left half contains a memory address, but the corresponding part of the right half is occupied by additional specifications. *P* is a modifier to indicate different kinds of address manipulation, including progressive indexing (Chap. 11). *Length* and *byte size* (*BS*) further define the operand in memory (Chap. 7). The second operand is implied to be in the accumulator; separate specifications are not essential, but an *offset* is provided as a partial address of the second operand for greater flexibility. It designates the starting position within the accumulator, thus avoiding extra shift instructions to line up the operands. The *I* address in the right half is there primarily for consistency with other formats; automatic modification of the bits in fields *length, BS,* and *offset,* as if they were an address, is possible and occasionally useful.



(*a*) Floating point arithmetic

(*b*) Direct index arithmetic

(*c*) Input-output operations and data transmission

(*d*) Variable field length operations

(*e*) Index word

FIG. 9.4. Basic instruction formats for IBM 7030. The index-word format (*e*) is shown for comparison.

A complete list of instruction formats is given in the Appendix.

9.6. Instruction Efficiency

It was pointed out in Chap. 4 that different natural data units require different amounts of specification. The most complex data unit, the floating-point number, has a rigid format. Its specification is built into the arithmetic circuits for greatest speed. There is relatively little left for the instruction to specify: an address, an index register, and an operation. Hence a simple instruction format suffices (Fig. 9.4*a*).

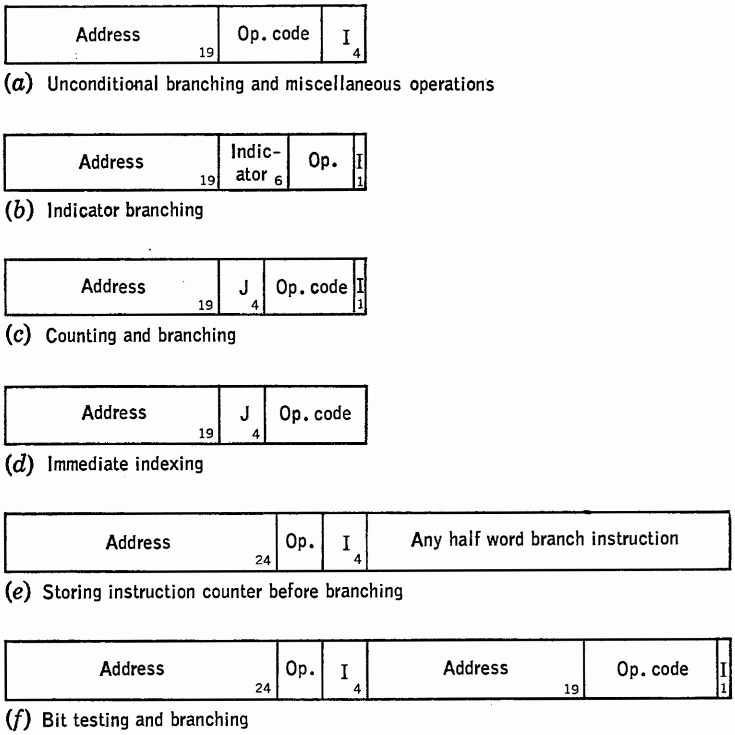
The most complex instruction format (Fig. 9.4*d*) is provided to operate on variable-field-length data, which are the most flexible data units. VFL data are, to the computer circuits, a mere collection of bits until their structure is specified in the instruction. The intent here is to give the programmer a very versatile tool with which, despite relatively low speed, certain important tasks can be performed more expeditiously than they could be with faster but more restricted operations.

It is obvious from information theory that instructions of varying information content can be expressed by a varying number of bits. It is not so obvious that the saving in memory space for programs, which results from having multiple instruction formats, would alone pay for the additional equipment cost of decoding these formats. What really prompted the introduction of multiple instruction formats was the observation that the speed of the 7030 was in danger of becoming severely limited by the time taken to fetch instructions from memory during the execution of the all-important inner loops of arithmetical programs. At that point in the design, it was found that almost all the instructions usually needed in the inner loops (floating-point arithmetic, indexing, and branching) could be expressed in terms of 32-bit half words and that, if they were so expressed, the number of accesses to memory for instructions could be cut almost in half.

Completely variable instruction lengths, though desirable in theory, are not practical. Either instructions would have to be scanned serially, which would be slow, or they would have to be passed through a complex parallel switching network with cumulative circuit delays, which would again slow down the computer. In practice, with binary addressing, instruction lengths must be kept to binary submultiples of the memory-word length. Half-length, quarter-length, and even eighth-length instruction vocabularies were actually tried. It was found that, although short instructions saved space, the saving could be quickly eaten up by the extra bits needed to define each format. The greatest economy of memory space and memory references was gained in a mixture of half-length and full-length instructions.

The 32 bits are rather tight for some of the short instructions. Since it was not possible to add a bit or two to an instruction when needed, it was necessary to vary the length of fields within the 32-bit space in order to provide all the functions that were thought desirable. These measures resulted in multiple 32-bit formats, which required additional decoding equipment as well as certain compromises.

Some of the additional formats are shown in Fig. 9.5. It will be seen that a 4-bit *I* address for indexing is not available in all formats. In particular, the conditional branching operations have only a 1-bit *I* address, permitting choice between no indexing and indexing against a single index register. It was felt that full indexing facilities, though desirable, were less important than, for example, being able to specify any of the 64 indicators as a test for indicator branching (Fig. 9.5*b*). The unconditional branching operations, however, have a complete index address, so that indexed branch tables may be readily constructed. Immediate indexing operations have no *I* address at all, since there seemed to be little use for automatic address modification when the address was itself the operand.



(*a*) Unconditional branching and miscellaneous operations

(*b*) Indicator branching

(*c*) Counting and branching

(*d*) Immediate indexing

(*e*) Storing instruction counter before branching

(*f*) Bit testing and branching

FIG. 9.5. Other 7030 instruction formats.

Operand addresses also vary in length for different formats. 18-, 19-, and 24-bit addresses are used depending on whether addressing is to be carried to the word, half-word, or bit level. The index-word format, shown in Fig. 9.4*e* for comparison with the instruction formats, has a full 24-bit value field as well as a sign; no sign bit could be provided in any of the instruction formats. To simplify indexing, all addresses line up against the left boundary of the word (or half word) in such a manner that the significant bits fall into corresponding positions in every format. Missing bits, including a 0 (+) sign bit, are automatically supplied to the right as the instruction is decoded, so that indexing always results in an effective address 24 bits long (Fig. 9.6).

The operation codes of different classes of instructions, especially half-length instructions, differ in length, position within the format, and variability. There are 76 distinct operation codes among the half-length instructions; at least 7 bits are required to specify them. Up to 8 more bits are used as modifiers common to all operations in the same class, so as to make the set as systematic as possible. For example, all arithmetical instructions have a modifier bit to indicate whether the operand sign is to be inverted, which eliminates the need for separate *add* and *subtract* codes. Thus adding 7 operation bits and 8 modifier bits to the 19 address bits and 4 index-address bits required by many instructions gives a total of at least 38 bits that would have been needed to encode these operations in a simple and straightforward manner. By eliminating redundancy, it was possible to compress the format to 32 bits. The only functional sacrifice was the reduced index address in some of the branching operations, as noted before.

Word address	Bit address	Sign	
xxx xxx xxx xxx xxx xxx	000 000	0	18-bit address
xxx xxx xxx xxx xxx xxx	x00 000	0	19-bit address
xxx xxx xxx xxx xxx xxx	xxx xxx	0	24-bit address
xxx xxx xxx xxx xxx xxx	xxx xxx	x	Index address

x Indicates a bit which may be 0 or 1

FIG. 9.6. Expansion of addresses of various lengths.

An analysis of the operation codes shows that only 0.05 bit of information is left unused in the 32-bit formats.[1] The 0.05 bit actually represents, at the time of writing, unallocated space in the formats for three more floating-point operations with their modifiers and nine more miscellaneous operations, each with a full 4-bit *I* address, which is not a trivial amount of space. The full-word formats are not so closely packed. The 64 bits are found to contain almost 6 bits of redundancy.

Yet another technique for increasing instruction efficiency is to use extra half words to define important but less frequently needed functions. This arrangement raises the instruction information content, because it uses one out of many operation codes, instead of tying up 1 bit in every instruction. Also, the efficiency with which a program can be stated is improved, since the infrequent use of an extra instruction is easily offset by the greater information content of each frequent instruction, whereas omitting the instruction entirely from the repertoire would require use of a subroutine each time the need arose.

[1] This assumes that all defined combinations are equally probable and all 18 bits of memory address are fully justified from the start to permit future expansion in a clean way.

As an example, indirect addressing is a powerful tool when needed, but its use is not very common; hence a separate half-word instruction is used as a kind of prefix for the instruction to which indirect addressing is to be applied (see Chap. 11 for more details). Figure 9.5*e* shows another example. A half-word prefix is attached directly to any half-word branching instruction, to make what is actually a complete set of full-length branching instructions; these permit the current setting of the instruction counter to be stored anywhere in memory before the instruction counter is changed to its new setting. The significance of making this a single full-length instruction is that, for conditional branching, the instruction counter setting is stored only when the branching actually takes place, thus saving valuable time. A final example is the very flexible full-length bit-testing instruction (Fig. 9.5*f*). This allows any addressable bit in memory or in the computer registers to be tested and set, and branching occurs if the test is satisfied. A more limited test of only the indicator bits (such as zero and overflow indications) satisfies the most frequent demands for testing, and the half-length indicator branching operation of Fig. 9.5*b* was provided for this reason, even though it is logically redundant.

These rather elaborate measures to increase instruction efficiency do not come cheaply in terms of decoding equipment and program-compiling time, but they do help materially to shorten the program-running time. Compared with the 704, for instance, the typical instruction length has gone down from 36 to 32 bits, and the power of the instruction has been increased. As a rule, the number of 7030 instruction half words to be executed is substantially less than the number of 704 instruction words for an equivalent program. This gain, of course, is to be added to the large gain in speed of corresponding individual instructions.

9.7. The Simplicity of Complexity

One may ask whether a more complex instruction set does not lead to more difficult programming. One answer is that programming can be simplified by adding instructions to complete a set (*branch on plus*, as well as *branch on minus*) and arranging them systematically. Another answer can be obtained by looking at the other extreme.

Van der Poel has shown[1] that the simplest instruction set theoretically consists of just one instruction. This instruction contains no operation code, only an address. Every instruction causes a combination of *subtract* and *store* to be executed; the difference replaces the contents of both

[1] W. L. van der Poel, The Essential Types of Operations in an Automatic Computer, *Nachrichtentechnische Fachberichte,* vol. 4, 1956, p. 144 (proceedings of a conference on Electronic Digital Computing and Information Processing held at Darmstadt, Germany, October, 1955); also, "The Logical Principles of Some Simple Computers," a monograph by the same author, Excelsior, The Hague, Netherlands, p. 100.

the accumulator and the specified memory address. All other computing operations, including conditional branching, can be built up from this one instruction, which is a very easy instruction to learn. But the programs needed to simulate no more than the elementary instruction set of early computers would be enormous. It would be quite a task just to estimate the size of the program for a real job. It seems safe to say that the storage required would be gigantic, and a desk calculator would probably be faster.

A complex, but appropriate, language will in fact simplify the programmer's task as the problems to be solved become more complex.

9.8. Relationship to Automatic Programming Languages

In tracing the development of instruction sets, we have found that the advent of more powerful computers designed to tackle larger problems is accompanied by more elaborate and versatile instruction vocabularies. Programs to do the same job require considerably fewer instructions and fewer references to memory. Or, to look at it another way, sequencing of simpler instructions stored in a relatively slow memory is replaced by internal sequencing with high-speed control circuits. This is a form of microprogramming using the fastest available memory, one made of transistor flip-flops.

Such an instruction set is still a long way from the "superlanguages" being developed under the heading of automatic programming. These languages are intended to simplify the task of the problem coder, not to raise the performance of the machine. The instruction set is an intermediate level between the programmer's language and the language of the elementary control steps inside the machine.

A two-step process of translation is thus required. One is the programmed assembly of machine instructions from the statements in the superlanguage. The other is the internal translation of instructions to control sequences. The two-step process is a matter of necessity at this stage of development to keep the complexity of the computer within bounds. It has the advantage that each language can be developed independently of the other to be most effective for its own purpose.

At the level of the user, there may be a need for developing specialized languages that facilitate programming of different jobs with varying emphasis on arithmetic, logical operations, data manipulation, and input-output control. At the machine level, where all these jobs come together, the need is clearly for a versatile and relatively unspecialized language. Perhaps the greatest demand on versatility is made by the process of translating from an automatic programming language to machine language. The performance of a computer in translating its own programs is a significant measure of how effective a tool the instruction set really is.

Chapter 10
INSTRUCTION SEQUENCING

by F. P. Brooks, Jr.

10.1. Modes of Instruction Sequencing

It is possible to distinguish four modes of instruction sequencing, which define the manner in which control may or may not pass from an original instruction sequence A to a new sequence B:

1. Normal sequencing. A keeps control.
2. Branching. A gives control to B.
3. Interruption. B takes control from A.
4. Executing. A lends control to B.

The first two are the basic modes of instruction sequencing found in the earliest automatically sequenced computers. Each instruction normally has a single successor, which may be defined by an instruction counter or by a next-instruction address within the instruction itself. Selection of an alternative sequence or modification of the original sequence may be accomplished at a point defined in the original sequence by conditional branching (also called *jumping*, or *transfer of control*), by indexed branching, or by the skipping or suppressing of one or more of the operations in the original sequence. In computers in which the normal sequence is defined principally by a counter, an unconditional *branch* instruction is used to specify a normal successor that does not occupy the next address.

Some conditions that may demand a change in instruction sequence arise either very rarely or at arbitrary times with respect to the program being executed. Testing for such conditions may be unduly awkward and time-consuming. Facilities for program interruption allow sequence changes to be set up in advance of the occurrence of the exceptional

Note: The major part of Chap. 10 has been adapted from two papers by the same author: A Program-controlled Program Interruption System, *Proc. Eastern Joint Computer Conf.*, December, 1957, p. 128; The Execute Operations: A Fourth Mode of Instruction Sequencing, *Communs. ACM*, vol. 3, no. 3, pp. 168–170, March, 1960.

condition, which is monitored continuously; when the exception occurs, the current program is interrupted and the new sequence is started.

A rudimentary form of interruption upon the occurrence of an exception condition during an instruction execution (such as overflow) was provided in as early a computer as the UNIVAC I. A more general system, which monitored external, independently timed conditions, first appeared more recently.[1]

The fourth mode allows the original sequence to execute instructions from another sequence, without changing the normal sequencing control to specify the second sequence. Implementations of this mode of operation are found in two earlier computers.[2,3]

The instruction-sequencing modes of the 7030 are described in the following sections, with emphasis on the *interrupt* and *execute* features, which go considerably beyond those found in earlier computers.

10.2. Instruction Counter

The normal instruction sequence in the 7030 is determined by an instruction counter which is stepped up automatically by one or two half-word addresses for each instruction, depending on whether the instruction is a half word or full word long. A full-length instruction may begin at any half-word boundary; *branch* instructions specify a half-word branch address. Any instruction may alter its successor, even if both are located in the same memory word, and the successor will be executed correctly.

For entry to a closed subroutine it is necessary to preserve the current setting of the instruction counter. There are several known techniques. One is a programming trick, called after its originator the *Wheeler subroutine linkage*,[4] where an instruction is written to load itself into some available register (the accumulator or an index register) before branching into the subroutine takes place. This technique always takes time and a register, whether the branch is actually taken or not. Another solution is to employ more than one instruction counter; but if nesting of subroutines to any number of levels is desired, it is still necessary for the program to store the original counter contents after the branching to the

[1] Jules Mersel, Program Interruption on the Univac Scientific Computer, *Proc. Western Joint Computer Conf.*, February, 1956, p. 52.

[2] Reference Manual, IBM 709 Data Processing System.

[3] U. A. Machmudov, LEM-1, Small Size General Purpose Digital Computer Using Magnetic (Ferrite) Elements, *Communs. ACM*, vol. 2, no. 10, pp. 3–9, October, 1959, translated from the Soviet publication *Radiotechnika*, vol. 14, no. 3, March, 1959.

[4] M. V. Wilkes, D. J. Wheeler, and S. Gill, "The Preparation of Programs for an Electronic Computer," p. 22, Addison-Wesley Publishing Company, Cambridge, Mass., 1951.

subroutine. A more economical method, where the instruction-counter contents are stored in a fixed location at every branch point automatically, was discarded because it takes time in the many cases when the contents are not needed after branching.

The method adopted in the 7030 requires the programmer to specify when and where the instruction-counter contents are to be stored before branching. This is done by inserting ahead of any of the half-length *branch* instructions, to be described below, a half-word prefix, called STORE INSTRUCTION COUNTER IF. The "if" signifies that the counter contents are stored only if branching actually takes place, thus saving time. Since the counter contents can be stored at any memory address, it is not necessary to tie up a register for this purpose.

The ability to use the instruction counter to index addresses, which would make program relocation easier, is not provided in the 7030. The main reason for the omission was the lack of index-address bits in the tight instruction formats (see Chap. 9). Most instructions can refer to one of fifteen index registers, but the most important conditional *branch* instructions can specify only one index register. It seemed undesirable to restrict that one register permanently to be the instruction counter. It was even questioned whether the instruction counter should use one of the other fourteen index addresses; some felt that fifteen index registers was still not a large number and would have found 31 more comfortable for large problems. Without these format restrictions, however, the instruction counter could have been profitably included among the index registers. As it is, for simple unconditional branching only, a separate instruction BRANCH RELATIVE achieves the desired effect; for other branching operations, an extra half word is needed to store the instruction counter first in an index register for subsequent indexing of a normal *branch* instruction.

10.3. Unconditional Branching

The unconditional BRANCH instruction is accompanied by several variations. BRANCH DISABLED and BRANCH ENABLED are used to turn the program-interrupt mechanism off and on, as will be discussed later; these functions are combined with unconditional branching because they are frequently needed during entry to and exit from the subroutine that takes care of the interrupting condition. BRANCH ENABLED AND WAIT is the nearest equivalent to a *stop* instruction in the 7030: program execution is suspended while waiting for an interrupt signal. This conditional *stop* instruction allows the computer program to get back into step with external operations when they take longer than the internal operations. The built-in interval timer may also restart the computer when it is waiting. An unconditional *stop* instruction is neither necessary nor desirable, since its presence would permit one program inadvertently to kill other programs that might be sharing the machine.

BRANCH RELATIVE creates a branch address by adding the current contents of the instruction counter to the specified address. NO OPERATION is a pseudo *branch* instruction that does nothing. (The 7030 actually contains several ways of doing nothing—at very high speed, of course.) As in some earlier computers, the operation code of NO OPERATION differs from BRANCH by the state of a single bit. This makes possible a convenient remotely controlled program switch: the bit may be set to 0 or 1 on the basis of a test at one point of a program, thus preselecting one of two alternative paths to be taken at a later point when the test condition may no longer be available.

10.4. Conditional Branching

Conditional branching in the 7030 is distinguished by the functional richness of a small number of unified instructions. This is made possible by the technique of gathering most machine-set test conditions into a single 64-bit indicator register. (A 48-bit subset of these indicators is continually monitored for program interruption.) A list of indicators with a short description of each is given in the Appendix. The indicator word has an address and thus may be used as a regular instruction operand.

A single half-length BRANCH ON INDICATOR instruction is used to test any one of the 64 indicators. The indicator desired is specified by a 6-bit field. A further bit specifies branching either when the indicator is *on* (1) or when it is *off* (0). Yet another bit specifies whether the indicator is to be reset to zero after testing.

The full-length instruction BRANCH ON BIT extends this testing facility to all bits in memory. Any bits, including those in the addressable registers and thus the indicators, can be tested for either the *on* or the *off* condition. There are 2 bits to specify whether the test bit is to be (1) left alone, (2) reset to 0, (3) set to 1, or (4) inverted. With this instruction the programmer can set up, alter, and test individual bits as he wishes.

Because of their frequent occurrence, certain elementary indexing and associated indicator-branching operations have been combined into the two half-length instructions COUNT AND BRANCH, and COUNT, BRANCH, AND REFILL. These are discussed further in Chap. 11.

10.5. Program-interrupt System

There are two quite distinct purposes for a *program-interrupt* system. The first of these is to provide a means by which a computer can make very rapid response to extra-program circumstances that occur at arbitrary times and perform a maximum amount of useful work while waiting for such circumstances. These circumstances will most often be signals from an input-output exchange: that some interrogation has been received or that an input-output operation is complete. For efficiency in real-time operation, the computer must respond to these forthwith. This requires a system by which such signals cause a transfer of control to a suitable special program.

The second purpose is to permit the computer to make rapid and facile selection of alternative instructions when the execution of an instruction causes an exceptional condition. For example, to avoid frequent and uneconomical programmed testing or extremely costly machine stops, it is desirable to have an interrupt system for arithmetical overflow or attempted division by zero.

These two purposes—response to asynchronously occurring external signals and monitoring of exceptional conditions generated by the program itself—are quite distinct, and it would be conceivable to have systems for handling them independently. However, a single system serves both purposes equally well, and provision of a single uniform system permits more powerful operating techniques. Moreover, the interrupt system has also been integrated with conditional branching, as mentioned before.

A satisfactory program-interrupt system must obey several constraints. The most important is that programming must be straightforward, efficient, and as simple as the inherent conceptual complexities allow. Second, the special circuits should be cheap because their use is relatively infrequent. Third, the computer must not be retarded by the interrupt system, except when interruptions do in fact occur. Finally, since there is still little experience in the use of interrupt techniques, the interrupt system should be as flexible as possible.

10.6. Components of the Program-interrupt System

The first question to be answered in designing a program-interrupt system is: When to interrupt? What is required is (1) a signal when there is a reason for interruption and (2) a designation whether interruptions are to be permitted.

Providing the signal is straightforward. For each condition that may require attention, there is an indicator that can be interrogated by the control mechanism. When the condition arises, the indicator is set *on,* and it may be turned *off* when the condition disappears or when the program has cared for it. As mentioned before, there are 64 indicators altogether, arranged in the form of an addressable machine register whose contents can be loaded or unloaded in one instruction.

Designation when interruption is permitted can be made in several ways. It is possible to organize a system so that any condition arising at any time can cause interruption. Alternatively, one can provide a bit in each instruction to designate whether interruptions shall be permitted at the end of that instruction or not. These methods make no distinction among the interrupting conditions. It is highly desirable to permit selective control of interruptions, so that at any given time one class of conditions may be permitted to cause interruptions and another class prevented from causing interruptions.

Therefore, each of the interrupt indicators is provided with a *mask* bit. When the mask bit is *on*, the indicator in question is allowed to cause interruption. When the mask bit is *off*, interruption cannot be caused by the condition indicated. Twenty-eight of the mask bits can be set *on* or *off* by the program. Twenty other mask bits are permanently set *on;* these correspond to conditions so urgent that they should always cause interruption when the system is enabled. The remaining sixteen indicators, which never interrupt and can be tested only by programming, may be regarded as having mask bits that are permanently set to *off*. Like the indicators, the mask bits are assembled into a single register with an address, so that they can all be loaded and stored as a unit, as well as individually.

A second major question that the designer must answer is: What is to be done when an interruption occurs? In the simplest systems the program transfers to some fixed location, where a *fix-up* routine proceeds to determine which condition caused the interruption and what is to be done. This is rather slow. In order to save time, the 7030 provides branching to a different location for each of the conditions that can cause interruption. The particular location is selected by a *leftmost-one identifier*. This device generates a number giving the position within the indicator register of the bit that defines the condition causing the interruption. This bit number is used to generate a full-word instruction address that contains the operation to be performed next. Since it was anticipated that the 7030 would often be operated in a multiprogrammed manner, the bit address is not used directly as the instruction address, for this would require the whole table of fix-up instructions to be changed each time the computer switched to a different program. Instead, the bit address is added to a base address held in an *interrupt address register*. The sum is used as the next instruction address. One can easily select among several interrupt instruction tables by setting the base address in the interrupt address register.

A third major question is: How shall control return to the main program when the fix-up routine is complete? One might cause the current instruction-counter contents to be stored automatically in a fixed location and then change the instruction-counter setting to the address of the appropriate entry in the interrupt table. The solution preferred was to execute immediately the instruction specified in the interrupt table without disturbing the contents of the instruction counter. (Only one such instruction, whether half- or full-length, may be placed at each location in the interrupt table.)

If the interrupting instruction is one that does not alter the instruction counter, the program automatically returns to the interrupted program and proceeds. This permits exceptionally simple treatment of the conditions that can be handled with a single instruction. More complex conditions are handled by a combination of a *store instruction counter* prefix with a *branch* to a suitable subroutine; this subroutine is entered just like any other.

A fourth question concerning any program-interrupt system is: How are the contents of the accumulator, index registers, etc., to be preserved in case of interruption? Automatic storage of these is both time-consuming and inflexible. As with respect to the instruction counter, it appeared better to use the standard subroutine philosophy: the fix-up routine is responsible for preserving and restoring any of the central registers, but full flexibility is left with the subroutine programmer. He needs to store and retrieve only what he intends to corrupt.

The fifth question that must be answered is: How are priorities to be established among interrupting conditions, and what allowance is to be made for multiple interruptions? Provision of the masking facility answers this problem, since any subset of the conditions may be permitted to cause interruption. Each fix-up subroutine can use a mask of its own, thereby defining the conditions that are allowed to cause interruption during that routine. There is also provided a means of disabling the whole interrupt mechanism for those short intervals when an interruption would be awkward. One such interval is that which occurs between the time when a subroutine restores the interrupt base address appropriate for the main program and the time when it effects return to the main program. The mechanism is disabled or enabled by means of the instruction BRANCH DISABLED or BRANCH ENABLED, typically during entry to or exit from the interrupt fix-up routine.

Simultaneous conditions are taken care of by the leftmost-one identifier, which selects the condition with the lowest bit address in the indicator register for first treatment. This is satisfactory because the fix-up routines for the several conditions are largely independent of one another. The positioning of conditions within the indicator register defines a built-in priority, but this priority can readily be overridden by suitable masking whenever the programmer desires. In fact, it might be said that the leftmost-one identifier solves the problem of simultaneity, while the selectivity provided by the mask solves the problem of over-all and longer-term priorities.

10.7. Examples of Program-interrupt Techniques

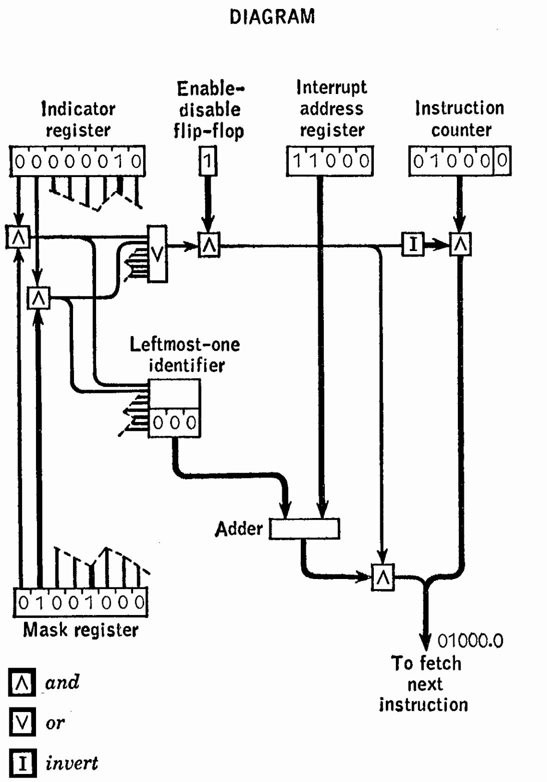
Figure 10.1 shows the system organization of a simplified interrupt system with only eight interrupt conditions and 32 words of memory. The abbreviated addresses consist of 5 bits for numbering full words and a sixth bit for selecting the left or right half word. The numbers 00111.1 in binary and 7.32 in decimal notation are used to refer to the right half (starting with bit position 32) of word 7.

The example starts with condition 6 in the indicator register *on.* The mask register is set up to allow only conditions 1 and 4 to cause interruption. Instruction 7.32 has just been executed, and the instruction counter has been stepped up to 8.0. There is no interruption; so the next instruction is taken from location 8.0 in the normal manner.

In Fig. 10.2 the execution of instruction 8.0 is accompanied by the occurrence of condition 1. The leftmost-one identifier generates the number 1 which is added to the 24 contained in the interrupt address register. The result, 25, is used as the address of the next instruction rather than the 8.32 contained in the instruction counter, which is unchanged.

In Fig. 10.3 is shown the case when the instruction at location 25.0 does not change the instruction counter. The interrupt mechanism has turned off condition 1 which caused the interruption. No other condition and mask bits coincide. After the instruction at location 25.0 is completed, the next instruction is taken from the location specified by the instruction counter, which still contains 8.32. This one-instruction fix-up routine might be used to reset the interval timer at the end of an interval.

Figure 10.4 shows a different sequence that might have followed Fig. 10.2. Suppose indicator 1 represents an end-of-file condition on a tape and several instructions are needed to take care of the condition. In this case the instruction at location 25.0 disables the interrupt mechanism, stores the instruction-counter contents (8.32) as a branch address in the instruction at location 21.32, and then branches to location 19.0. The fix-up routine proper consists of the instructions between 19.0 and 21.0 (it might be of any length and might include testing and scheduling of further input-output operations). During the routine no more interruptions can occur. Instruction 21.32 is a BRANCH ENABLED instruction, the address part of which was set to 8.32. This returns control to the interrupted program at location 8.32 and reenables the mechanism so that further interruptions are possible. If another interrupt

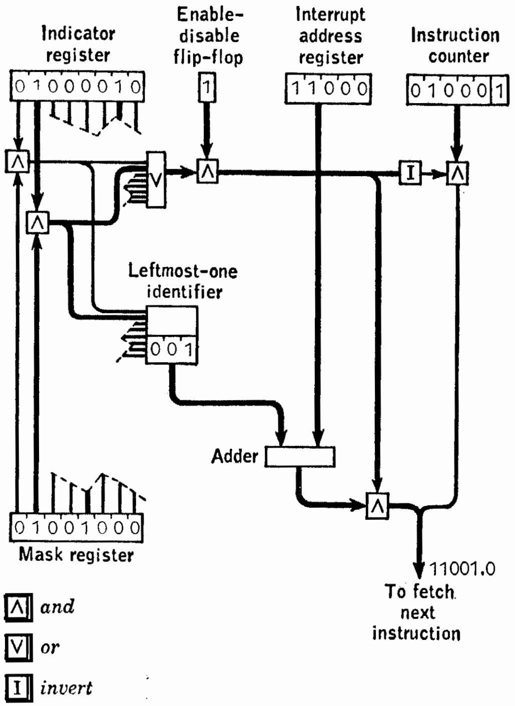


PROGRAM SEQUENCE

Instruction location		Operation	First address	Second address
Binary	Decimal			
00111.1	7.32	ADD	x	
01000.0	8.0	MULTIPLY	y	

FIG. 10.1. Program-interrupt example. Condition masked *off:* no interrupt.

DIAGRAM

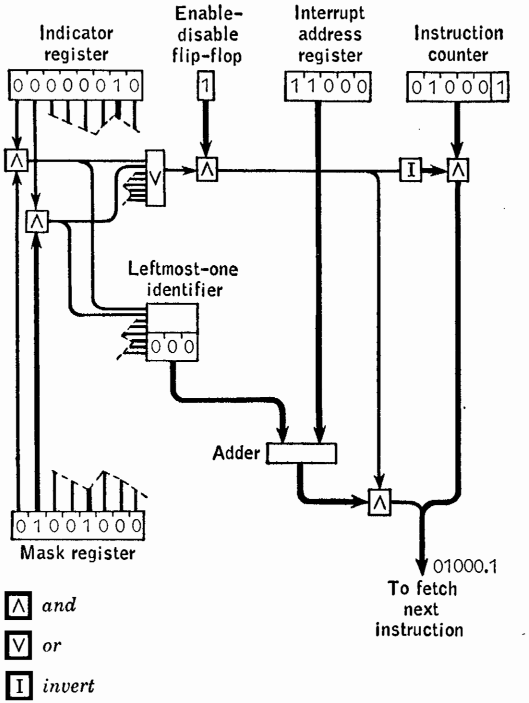


PROGRAM SEQUENCE

Instruction location		Operation	First address	Second address
Binary	Decimal			
00111.1	7.32	ADD	x	
01000.0	8.0	MULTIPLY	y	
➡				
11001.0	25.0	TRANSMIT	m	n

Fig.10.2. Program-interrupt example. Condition masked *on:* interrupt occurs.

DIAGRAM

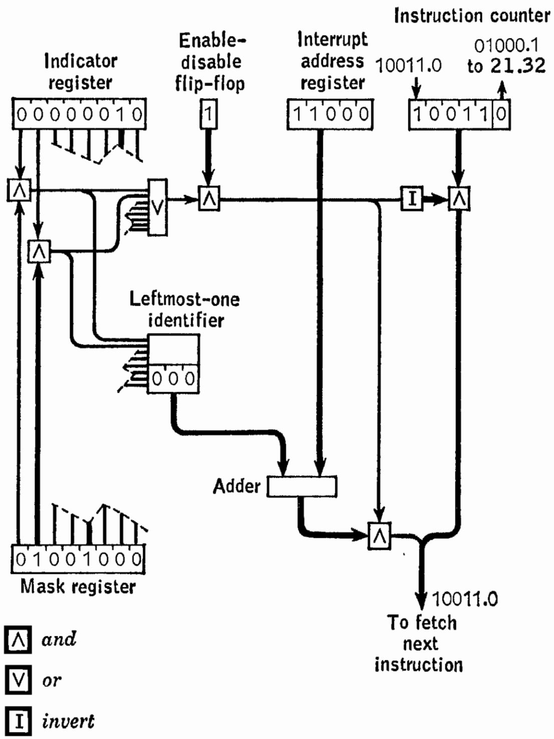


PROGRAM SEQUENCE

Instruction location		Operation	First address	Second address
Binary	Decimal			
00111.1	7.32	ADD	x	
01000.0	8.0	MULTIPLY	y	
11001.0	25.0	TRANSMIT	m	n
➡				
01000.1	8.32	STORE	z	

FIG. 10.3. Program-interrupt example. One-instruction fix-up.

DIAGRAM



PROGRAM SEQUENCE

Instruction location		Operation	First address	Second address
Binary	Decimal			
00111.1	7.32	ADD	x	
01000.0	8.0	MULTIPLY	y	
11001.0	25.0	STORE INSTRUCTION COUNTER IF BRANCH DISABLED	21.32	19.0
➡				
10011.0	19.0	LOAD	w	
⋮	⋮			
10101.1	21.32	BRANCH ENABLED	—	
01000.1	8.32	STORE	z	

FIG. 10.4. Program-interrupt example. Interrupt subroutine with further interrupts inhibited.

condition is already waiting, another interruption will take place immediately, even before the instruction at location 8.32 is executed.

The program in Fig. 10.4 assumes that it is desired to prevent further interruptions during the fix-up routine. If further interruptions were to be allowed during the routine and the same mask still applied, the programmer would use only a STORE INSTRUCTION COUNTER IF BRANCH instruction at location 25.0 and a simple BRANCH instruction at location 21.32. This procedure is appropriate when and only when the programmer is certain that condition 1 cannot arise again either during the fix-up routine or during any other routine that might interrupt it.

Instruction location		*Operation*	*First address*	*Second address*
Binary	*Decimal*			
00111.1	7.32	ADD	x	
01000.0	8.0	MULTIPLY	y	
11001.0	25.0	STORE INSTRUCTION COUNTER IF BRANCH DISABLED	23.32	19.0
10011.0	19.0	SWAP	Mask register	Temporary storage
10100.0	20.0	BRANCH ENABLED	20.32	
10100.1	20.32	LOAD	w	
.	.			
.	.			
.	.			
10110.0	22.0	BRANCH DISABLED	22.32	
10110.1	22.32	SWAP	Mask register	Temporary storage
10111.1	23.32	BRANCH ENABLED	—	
01000.1	8.32	STORE	z	

Fig. 10.5. Program-interrupt example. Interrupt subroutine permitting further interrupts.

In the most sophisticated use of the program-interrupt mechanism, where it is desired to employ a long fix-up routine that is to be interrupted under a different set of conditions, the program in Fig. 10.5 is appropriate. The mechanism is disabled at the time of the first instruction after interruption. The new mask is loaded and the old preserved. The mechanism is then enabled. At the end of the routine the mechanism is disabled, the old mask restored, and the mechanism is reenabled as control is transferred to the originally interrupted routine at location 8.32.

This procedure is clearly suitable for any number of levels of interruptions upon interruptions, each of which may have a different set of causing conditions. Each level of routine is under only the usual subroutine constraint of preserving the contents of the registers it uses.

Full program control simplifies programming and multiprogramming, as does the refusal to assign special functions to fixed memory locations. The task of the programmer of fix-up routines is simplified by the provision of special operations and by the adoption of the same conventions and requirements for interruption routines as for ordinary subroutines.

An especially important feature of the program-interrupt system just described is that it makes almost no demands upon the writer of the lowest-level program. He need only set up the interrupt address register and the mask register. He need not even understand what he puts there or why, but may follow the local ground rules of his installation. Priorities, preservation of data, and other programming considerations that are inherent in program interruption concern only the author of the fix-up routines. In open-shop installations it is important that any programming burden caused by such sophisticated operation fall upon the full-time utility programmer rather than upon the general user.

10.8. Execute Instructions

In an *execute* instruction the address part specifies, directly or indirectly, an object instruction to be executed, but does not set the instruction counter to the location of the object instruction, as a *branch* instruction would do. The next instruction to be executed, therefore, is the successor of the *execute* instruction rather than the successor of the object instruction. This is illustrated below.

Location	*Operation*	*Address*	*Comments*
100.0	EXECUTE	1715.0	Instruction counter steps to 100.32
(1715.0)	LOAD	x	Interpolated object instruction
100.32	(Next instruction)		

With the instruction counter at location 100.0, the instruction EXECUTE 1715.0 is fetched. This instruction now causes the word at address 1715.0, the object instruction LOAD x, to be loaded into the instruction decoding circuits and to be executed just as if it had occurred in the program at address 100.0. The instruction counter meanwhile has advanced to location 100.32, where the next instruction to be executed will be found. (Note that EXECUTE in the 7030 is a half-length instruction.)

In effect, an *execute* operation calls in a one-instruction subroutine and specifies immediate return to the main routine. This is similar to *indirect addressing* (see Chap. 11), except that the whole instruction, not just the address part, is selected from the specified location.

The uses of the *execute* operations arise directly from the fact that the object instruction does not imply its own successor. In the IBM 709, for example, *execute* simplifies modification of nonindexable and non-indirect-addressable operations such as those for input-output. In the Soviet LEM-1 computer,[1] there are 1,024 words of erasable storage and 7,167 words of read-only storage; here the *execute* operations permit programs in the read-only storage to use isolated modifiable instructions in the regular storage.

The one-instruction subroutines provided by the *execute* operations are especially useful in linkages between a main program and ordinary subroutines. For instance, a subroutine may need several parameters, such as character size, field length, index specification, etc. The calling sequence may include these parameters in actual machine instructions which the subroutine treats as second-order subroutines. This ability to lend control back and forth, between calling sequence and subroutine, should permit many new subroutine linkage techniques to be developed.

One useful special case of this form of subroutine technique occurs in interpretive routines where machine-language instructions can be intermixed with pseudo instructions in the argument program. The interpreter can then execute the machine-language instructions directly without transplanting them into itself.

The one-instruction subroutine techniques provided by *execute* operations permit counter-sequenced computers to use the efficient programming tricks of the IBM 650, in which instructions are executed directly from an accumulator.

For all the foregoing purposes it is preferable for the *execute* operation to have any machine instruction as its object. Thus one may desire to execute an arithmetic instruction, a *branch* instruction, or even another *execute* instruction. Actually the occurrence of a *branch* instruction as the object instruction of an *execute* operation would be rare in any of these applications. This fact makes it possible to add the restriction of not permitting *execute* to perform *branch* operations—a very useful restriction for other major applications.

One of these applications is program monitoring, where the object instruction of an *execute* operation should be prevented from changing the instruction counter that controls the monitoring routine. Consider, for example, a supervisory program A, such as a tracing routine, which is to monitor the execution of an object program B, perhaps with testing or printing of the instructions of B as they are executed. With an ordinary set of operations, the programming to effect such monitoring is quite clumsy. Each instruction of B must be moved from its normal place in memory to a place in the sequence of A. Then it must be tested

[1] Machmudov, *op. cit.*

to ensure that it is not a *branch* instruction or, if it is, that the branching condition is not met; for the execution of such an operation would transfer control of the machine from the supervisory program to some point within the object program. Finally, after the transplanted B instruction has been executed, A must update a pseudo instruction counter that keeps track of the progress of B, and repeat the whole process with the next B instruction. If the B instruction is a successful *branch*, A must appropriately revise the pseudo instruction counter. This programmed machinery is common to all monitoring routines and must be executed in addition to the actual monitoring desired.

10.9. Execute Operations in the 7030

The two *execute* operations in the 7030 are designed so that they can be used for one-instruction subroutines and for program monitoring. They are called EXECUTE and EXECUTE INDIRECT AND COUNT. Each causes a single instruction to be fetched from an addressed location and executed, except that execution may not change the instruction counter. If the object instruction specifies a *branch* operation (which would cause such a change), branching is suppressed and the *execute exception* indicator is actuated, which may interrupt the (monitoring) program. Moreover, the object instruction is not allowed to change the state (enabled or disabled) of the interrupt system.

In the EXECUTE operation, the address specifies the object instruction directly. In the EXECUTE INDIRECT AND COUNT operation the address specifies a pseudo instruction counter in memory, whose contents are the location of the object instruction. After the object instruction is performed, the pseudo instruction counter is incremented according to the length of the object instruction. This last feature is particularly convenient in a computer that has instructions of different lengths, and it uses equipment that the computer must have anyway. Any *execute* operation may have another *execute* operation as its object. This useful function makes it possible, however, for a programmer's error to initiate an endless loop of *execute* operations and thus never reach the end of the instruction. Since the ordinary interrupt system can interrupt only between instructions, a special signal forces an interruption after several hundred repeated operations, so that the computer will not be tied up indefinitely. (The same signal is used to terminate an endless indirect-addressing loop.)

The 7030 *execute* operations, then, not only provide the ability to execute an isolated instruction, with automatic return of control to the monitoring routine, but also provide for (1) suppression of branching, and (2) signaling to the monitoring routine when branching is attempted. These properties considerably simplify monitoring routines. The automatic return obviates the need for transplanting the instructions of the object program into the monitor. The suppression of branching ensures that the monitor can retain control without detailed testing of the object instruction. The notification of attempted branching permits the monitoring program to update the pseudo instruction counter for the object program without detailed testing. Since this detailed testing of the object instruction for branching and skips occupies a large part of conventional monitoring programs, the *execute* operations make such programs much more efficient. The EXECUTE INDIRECT AND COUNT operation gives further efficiency because it automatically increments the pseudo instruction counter.

A simple monitoring loop for performing a control trace in the 7030 computer reduces to:

Location	*Operation*	*Address*
100.0	EXECUTE INDIRECT AND COUNT	Pseudo instruction counter
100.32	BRANCH	100.0

When a *branch* occurs in the object program, this loop is interrupted, and a suitable routine records the tracing data and changes the pseudo instruction counter.

The *execute* operations can in theory be put into any stored-program computer. Their mechanization is somewhat simpler and more justifiable in computers that use an instruction counter for normal sequencing. Provision of the safeguards that permit the operation to be used for monitoring is greatly simplified in computers that have program-interruption systems. In other computers, attempts by the object program to change the sequence must be signaled by setting conditions that stop the machine or are tested by *branch* instructions.

An obvious extension of the *execute* operations would be to have the EXECUTE INDIRECT AND COUNT operation automatically change the pseudo instruction counter when the object instruction is a *branch*. There would still need to be an alarm to the monitoring program, however, so this function was not incorporated in the 7030.

Chapter 11
INDEXING

by G. A. Blaauw

11.1. Introduction

A basic requirement for a computer is that writing a program should take less effort than performing the desired operations without the computer. Most computer applications, therefore, use programs that can be repeated with different sets of data. There are several possible techniques.

In the earliest machines the technique employed was to change the contents of a few storage locations between successive executions of the program. A later method of achieving the same result was to change not the data at a given address but the addresses used by the program. This procedure permitted many more storage locations to be used and widened the scope of computer applications considerably. Early computers, whose programs were specified by pluggable wiring, paper tape, or cards, permitted little or no address alteration. The invention of stored-program computers provided a major advance because it allowed programs to be treated as data, so that any instruction of a program could be modified by the program itself. The main application of this general facility was for the modification of addresses.

Subsequently, it became apparent that programmed address computation, though sufficient in theory, was cumbersome in practice. Too much computing time and program space were required to perform these auxiliary operations. A remedy was provided by an address register, also called *index register* or *B-line*,[1] whose contents could automatically

Note: Chapter 11 is a reprint with minor changes of G. A. Blaauw, Indexing and Control-word Techniques, *IBM J. Research and Development*, vol. 3, no. 3, pp. 288–301, July, 1959. A condensed version was published previously under the title, Data Handling by Control Word Techniques, *Proc. Eastern Joint Computer Conf.*, December, 1958, pp. 75–79.

[1] T. Kilburn, The University of Manchester High-speed Digital Computing Machine, *Nature*, vol. 164, no. 684, 1949.

be added to the specified operand address to obtain the actual address of the operand. In recent machines several index registers—up to 100—have been made available. Thus address computation has partly taken the place of data transmission between storage locations and has subsequently been simplified by the introduction of index registers.

Providing specialized machine functions, such as indexing, for operations that could also be programmed was not new. In theory, all machine instructions but one are redundant; as noted in Chap. 9, an instruction repertoire can be replaced by a single, well-chosen instruction. In practice, a repertoire of more than one instruction is justified by the operating time and program space that are saved. Similarly, special-purpose registers, such as index registers, may be justified when they increase the effective speed and capacity of the computer enough so that the gain in performance offsets the expense of the added equipment and improves the performance-to-cost ratio. This type of performance gain should be accompanied by greater programming ease. Programming ease greatly affects the form that an added function should take, but, because programming ease is hard to express in a cost figure, it is rarely used as the sole justification for added equipment.

In the design of the IBM 7030, an attempt has been made to achieve great flexibility and generality in machine functions. The indexing functions and the associated instruction set, consequently, were examined carefully. The general principles that were considered in this examination will be discussed first. The built-in functions that were developed for the 7030 as a result of the examination will be described subsequently and illustrated by examples.

11.2. Indexing Functions

Indexing functions may be divided into four groups: (1) address modification, (2) index arithmetic, (3) termination, and (4) initialization. The first group is used in addressing operands and provides the justification for the existence of index quantities. The other groups concern the task of changing the index quantities, the tests for end conditions, and the set-up procedures. These operations are often termed *housekeeping.*

Address Modification

The common use of an index register is the addition of its contents, the *index value,* to the address part of an instruction, which will be called the *operand address,* in order to address memory with the sum, the *effective address.* This operation is called *address modification.* The operand address and the index value remain unchanged in storage.

Address modification is used in general to address successively the elements of an array. An array may be one-dimensional or multidimensional, and its elements may be single-valued or multivalued. The address of a value that is part of an array can be subdivided into three distinct parts. The first part, the *base address,* identifies the location of the array within memory. The second part, the *element address,* identifies the location within the array of the element currently being used in computation. The element address is specified relative to the base address and is independent of the location of the array in memory. The third part, the *relative address,* specifies the location of the array value relative to the current element. The relative address is independent of the location of the array and of the selection of the current element. The array value may be part of the current element or it may be part of another element. A well-known case in technical computation is the addressing of right, left, upper, and lower neighbors of an element in a two-dimensional array. Figure 11.1 illustrates this case and shows how the address of a particular array value is formed as the sum of base address, element address, and relative address.

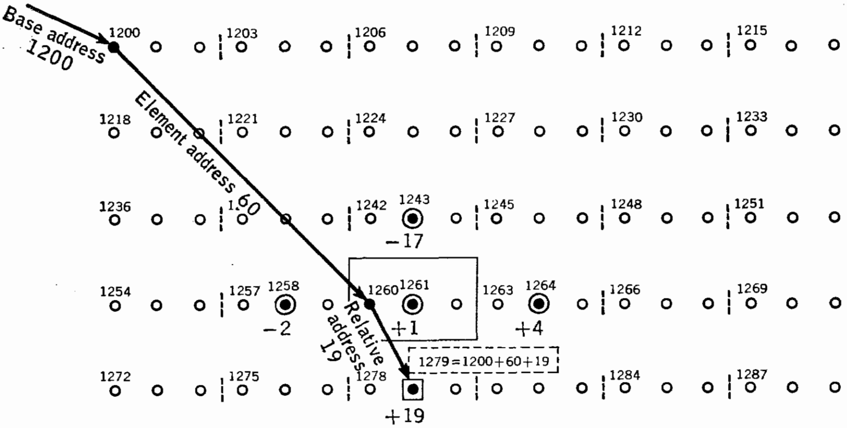


FIG. 11.1. Example of addressing of nearest neighbors in two-dimensional array. Example shows a 6 × 5 array of three-valued elements, with relative addressing of the second value of one element and of its four nearest neighbors.

The base address and relative address are constant throughout the execution of the program. The base address is determined as part of the task of memory allocation. The relative address is determined as part of the programming task by the characteristics of the computation to be performed. The element address, on the other hand, is not constant. It changes as the computation proceeds from one element to the next.

All three components, base, element, and relative address, must be available during address modification. Therefore, each of these addresses must be found either in the operand-address part of the instruction or in the index values of index registers. In order to make address modification effective, the variable part of the array address, the element address, should be part of an index value. The relative address is used to address different values for a given element address. In order to preserve the identity of the selected element, the index value, which contains the element address, must remain unchanged. Therefore, the relative address should be part of the operand address. The base address may be part either of the operand address or of an index value. In the first case, it is added to the relative address; in the second case, it may be added to the element address.

Index Arithmetic

As computation proceeds, successive elements of an array are addressed. The element addresses are generated by the algorithm appropriate for the use of the array in the computation. Since the element address is part of an index value, the address computation may be accomplished by *index arithmetic.* Very often the algorithm is a simple recurrent process in which a new index value is obtained by addition of an *increment* to the old index value.

There are several algorithms that cannot be described as simple incrementing processes. In particular, some algorithms make use of variables that are data or instructions rather than known parameters of an array. The use of data in index arithmetic occurs in *table reference* techniques. The use of instructions in index arithmetic occurs in *indirect addressing.* In this mode the effective address is used not as the address of an operand but as the address of an instruction whose effective address is the address of the operand (see Sec. 11.11).

The conventional use of the effective address as the operand address is called *direct addressing,* in contrast to the indirect addressing mode. In a simple incrementing process another addressing mode, *immediate addressing,* is often used. In this mode the effective address is used as an operand rather than as the address of an operand.

Termination

Each time an index is altered by index arithmetic, a test may be performed to determine when the last element of the array has been addressed. This process is called *termination.* Some of the forms of the test are (1) limit comparison, (2) length subtraction, and (3) counting. In limit comparison, the current index value is compared with a given constant, the *limit.* In length subtraction, a given variable, the *length,* is reduced by the value of the increment and tested for zero. In counting, a given variable, the *count,* is reduced by 1 and tested for zero.

The three methods of test are closely interrelated. When the base address is part of the index value, the limit is the sum of the base address and the length; the limit has the advantages that it stays fixed and that a comparison is rather simple to implement. The length, in turn, is the product of increment and count and so is independent of any base address to be added to the index value. Counting permits the test for completion to be independent of both base address and increment, so that even an "increment" of zero is possible.

Instead of a separate quantity, such as limit, length, or count, the index value itself can be used to determine the end of the process. In that case, the index value serves as a length, or, in other words, a limit of zero is implied. This approach, which is followed in the IBM 704, 709, and 7090, requires a minimum of information, but the base address cannot be part of the index value, and address modification must be subtractive rather than additive. A greater degree of freedom in specifying index values and tests is very desirable. Therefore, independence of index value and termination test is preferred. In the 7030, counting has been chosen as the primary means for determining the end of an index-modification sequence. The conclusions reached in the course of the discussion are, however, equally valid when a limit or length is used.

Initialization

After the last element of the array is addressed, the index value and count must each be changed to the initial setting for the array to be addressed next, which may be either the same array or another. This housekeeping operation is called *initialization.* Of course, initialization also occurs prior to the entire array-scanning operation. This case is the least frequent; it is usually part of a more general loading and resetting procedure, and its characteristics influence the indexing procedures to a lesser degree.

A summary of the indexing functions that have been described is shown in Table 11.1. The quantities that occur in the indexing procedure for a simple array are listed in the second column. The operations that make use of these quantities are listed in the third column.

TABLE 11.1. SUMMARY OF INDEXING FUNCTIONS

Function	*Quantity*	*Operation*
Index use	Index value	Address modification
Index change	Increment	Incrementing
Index test	Count	Counting and zero testing
Index reset	Next initial: Index value Count	Replacement of: Index value Count

Of the quantities listed, the index value is, of course, in the index register. This leaves four quantities that must reside somewhere. Earlier approaches have relied on storing these quantities in general memory locations. Of the four operations listed, address modification was usually the only built-in machine operation. In most earlier machines the other three operations were performed by separate instructions. For the 7030 it was decided to combine three of the quantities into one 64-bit *index word,* consisting of the index value, a count, and the address of another index word (Fig. 11.2). These three quantities may be used either independently or together by means of built-in combined indexing operations. When the three quantities in an index word belong to the same indexing or data-transmission operation, the word is often referred to as a *control word.* The terms "index word" and "control word" are largely synonymous, but the latter is intended to imply a certain mode of operation that will be discussed in subsequent sections.

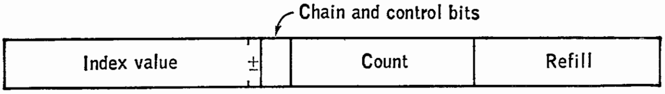


FIG. 11.2. Index or control-word format.

11.3. Instruction Format for Indexing

A general discussion of instruction formats was given in Chap. 9. We shall here consider instruction formats more specifically as they are affected by indexing.

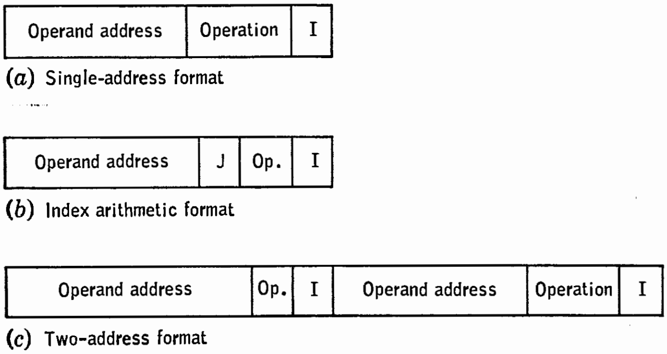
Relative addressing requires at least one field in the instruction format for direct-operand designation, called the *operand address* field, and one field for indirect-operand designation, called the *index address* field. The index-address field specifies the index used in operand-address modification. It is, of course, desirable to have a uniform system of operand addressing, where address modification is available for each operand.

It would serve no purpose to provide more than one direct designation for each operand. More index-address fields would be used rather infrequently. They might find application in *multiple indexing,* an index-arithmetic algorithm which forms the sum of two or more independently computed index values. It was decided not to burden every operand designation with added index-address fields, but to provide instead a separate instruction, LOAD VALUE WITH SUM, to be used only when needed. This instruction adds any selected number of index values and places the sum in another selected index value.

With an operand-address field and an index-address field required to specify each operand and with several operands necessary for most operations, the instruction format would become inefficient unless implied addresses or truncated addresses were used.

When two operands are needed for the usual single-address arithmetical instruction, one of the operands comes from an *implied address*, the accumulator, and the result is returned to an implied address, often again the accumulator. In the index-arithmetic operations of the 7030 the use of such implied addresses has been extended by specifying more than one operation in one instruction, as will be described in the following sections.

The use of a *truncated address*, containing fewer bits than normal operand addresses contain, saves instruction space, but it also reduces the number of available address locations and consequently makes the instruction set less general. A truncated address for index registers may be justified, however, (1) because a program usually needs only a limited number of index registers, and a complete address would therefore be inefficient; (2) because limiting the number of index registers permits preferred treatment for these registers to speed up index-arithmetic operations and address modification; (3) because truncation of the index address makes it practical to include a second index address in index-arithmetic instructions, which greatly improves the efficiency of these instructions.



(*a*) Single-address format

(*b*) Index arithmetic format

(*c*) Two-address format

FIG. 11.3. Instruction formats.

Nevertheless, some applications require complete generality for index addresses. For these cases an instruction RENAME effectively expands an index address to full capacity; it loads an index register from any desired memory location, retaining the address of the memory location, and the contents are automatically stored back at the original location before the index register is loaded by a subsequent RENAME instruction.

It would have been possible to improve the efficiency of operand specification by truncating the operand address. This method was not used, however, since the size of relative addresses would have been restricted and the base address could not then be part of the operand address.

In referring to the basic single-address format of the 7030, such auxiliary truncated addresses as the index address *I* used in address modification are not counted. The *I* address is considered part of the operand specification. Index-arithmetic instructions use a single-address format to which a second index-address field *J* has been added so that the second operand can be addressed explicitly. Some operations, for which two complete explicit operand addresses are desired, use a two-address format consisting of two single-address formats, each with an *I* address. Figure 11.3 shows the three basic formats. Figure 11.4 shows, in schematic form, the basic address-modification function of indexing.

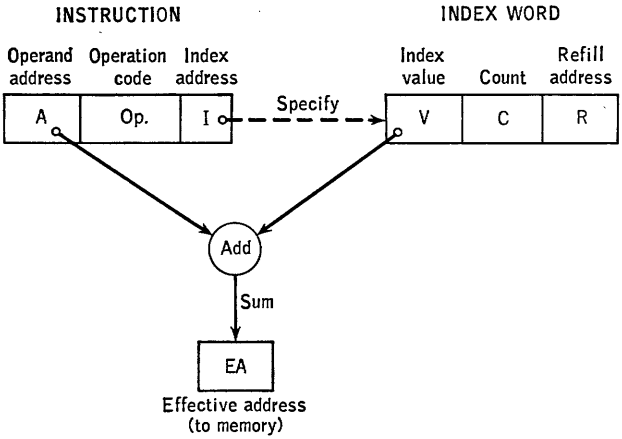


FIG. 11.4. Address modification.

11.4. Incrementing

Index incrementing could be performed in the accumulator by a series of three single-address instructions, which add the increment to the index value and return the result to the index register. Actually, only the increment and the index register to be modified need specification, and the short index-arithmetic format can be used to specify an entire incrementing operation, called ADD TO VALUE. This operation makes use of the index adder provided for address modification. The main arithmetical process for data is thus separated from the housekeeping process, and data registers are not altered. In the ADD TO VALUE operation the operand address specifies the address of the increment to be added to the value part of the index register specified by the index address *J*. The operand address can itself be indexed by the index value specified by the index address *I*, just like any other operand address. This gives indexable index arithmetic. A schematic diagram of the incrementing operation is shown at the top of Fig. 11.5. Several variations of the basic ADD TO VALUE instruction, permitting sign inversion and immediate addressing, are also available.

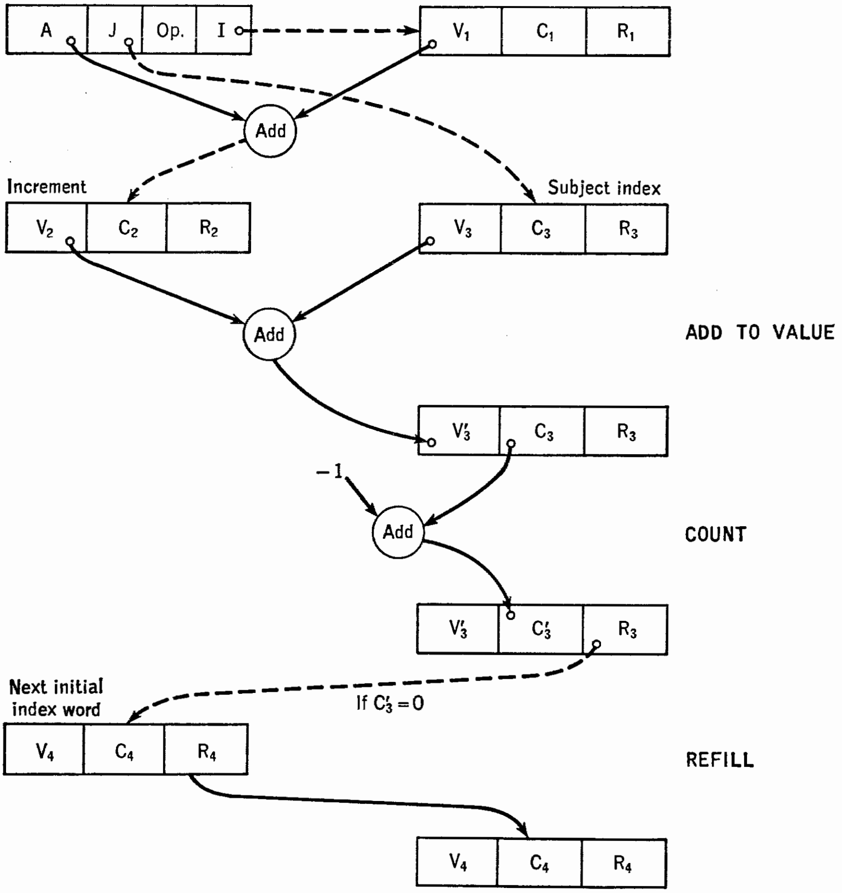


FIG. 11.5. Incrementing, counting, and refilling. Operations may be performed separately or in combination.

The quantity used in incrementing is specified explicitly in the incrementing instruction of the 7030. A different approach is possible. The increment could be associated with the index value such that the address of the increment would be known whenever the index was addressed. As pointed out in Chap. 9, an advantage is obtained from implied addressing only when the implied operand, here the increment, remains unchanged during repeated references. Furthermore, the incrementing operation could then be combined with another operation that uses the same index address. For instance, it would be possible to specify in one single-address instruction the use and the subsequent incrementing of an index. This method, however, loses its value when several different increments must be used to change an index value or when the incrementing and index use must occur in different parts of the program. In order to achieve greater generality, the separate ADD TO VALUE instruction has been chosen in preference to a combined instruction.

11.5. Counting

In the termination of array scanning, more than one count may be used, just as several increments may be used in index arithmetic. A single count is most frequent, however. It is, therefore, profitable to associate the count used in the termination with the index value to which the process applies and to use implied addressing. Since counting normally occurs when the index value is changed, it is logically consistent to specify incrementing and counting in one index-arithmetic instruction, ADD TO VALUE AND COUNT. This instruction is available in addition to ADD TO VALUE. It becomes equivalent to *count* when the increment is zero.

An implied address for the count can be obtained in various ways. A solution, economical in time and space, is to place both index value and count as separate fields into the index register. These are two of the three quantities that make up a control word. The instruction ADD TO VALUE AND COUNT adds the addressed increment to the index value, reduces the count by 1, and provides a signal when the count becomes zero. Counting is shown schematically in the center of Fig. 11.5. (The *refill* operation, indicated at the bottom of the figure, will be discussed in a later section.)

The choice of counting as a test for termination and the use of an implied address for the count do not preclude other termination tests. In particular, a COMPARE VALUE instruction is made available to allow limit tests, and instructions to add to or subtract from the count can be used for the equivalent of length subtraction. Such extra instructions add flexibility to the instruction set, but they are less efficient than ADD TO VALUE AND COUNT.

The following example, to be expanded later, illustrates the use of counting in a simple technical computation. It is required to multiply vectors A and B. Each vector has n elements. Vector A has its first element at a_0. Vector B has its first element at b_0. The product is to be stored at c_0. A is stored in successive memory locations. B is actually a column vector of a matrix, whose rows have p elements and are stored in successive memory locations. Therefore, the elements of B have locations that are p apart. The program is shown in Table 11.2.

TABLE 11.2. VECTOR MULTIPLICATION USING COUNT

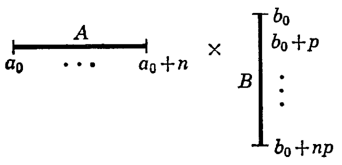
		Instructions
Initial setup	→ f	Load i from i_0
	$f + 1$	Load j from j_0
	$f + 2$	Set accumulator to zero
Vector multiply, inner loop	$f + 3$	Load cumulative multiplicand from a_0, indexed by i
	$f + 4$	Multiply cumulatively by b_0, indexed by j
Housekeeping, inner loop	$f + 5$	Increment j by p
	$f + 6$	Increment i by 1, count
	$f + 7$	Branch to $f + 3$ if count did not reach zero
Vector multiply, outer loop	$f + 8$	Store cumulative product c_0

Control words

Contents after executing the inner loop x times:

Address	*Index value*	*Count*
i	x	$n - x$
i_0	0	n
j	xp	...
j_0	0	...

Diagram of vector dimensions



Multiplicand and multiplier are specified in instructions $f + 3$ and $f + 4$. Their product is added to the accumulator, which contains the sum of the previous products. This operation is called *cumulative multiplication.* The count in control word i terminates the cumulative multiplication. The count in control word j is not used. The example shows that the use of the control words i and j in two instructions requires five added instructions in order to change, test, and initialize these control words. Three of the latter instructions are in the *inner loop.* Although the simplicity of the arithmetical process in this elementary example tends to overemphasize the housekeeping burden, it is clear that further simplification of the indexing procedure would be desirable.

11.6. Advancing by One

An array in which elements have consecutive addresses, such as vector A in Table 11.2, requires an increment of 1 to be added to the index value. The frequent occurrence of a value increment of 1, often coupled with counting, suggests the definition of an *advance and count* operation, which is the same as ADD TO VALUE AND COUNT with an implied increment of 1. Because the increment is implied, the operand address is free for other use; so the *advance and count* operation can be combined with still another single-address operation. A suitable candidate for such combination is the conditional *branch* operation that refers to the zero-count test. The new instruction, which also has several variations, is called COUNT AND BRANCH. The variations add no new indexing concepts and will not be discussed in detail.

In the example of Table 11.2, instructions $f + 6$ and $f + 7$ can be replaced by a single COUNT AND BRANCH operation.

11.7. Progressive Indexing

In discussing index use it was pointed out that a base address can be part of either the operand address or the index value. When the base address is part of the index value and the relative address is zero, the operand address is not used at all. The main operation can then be combined with an ADD TO VALUE AND COUNT operation. The index value is first used as an effective address to address memory; subsequently it is incremented by the operand address, which acts as an immediate increment. This is the same order of events that occurs when two separate instructions are used. The operation part of the instruction,

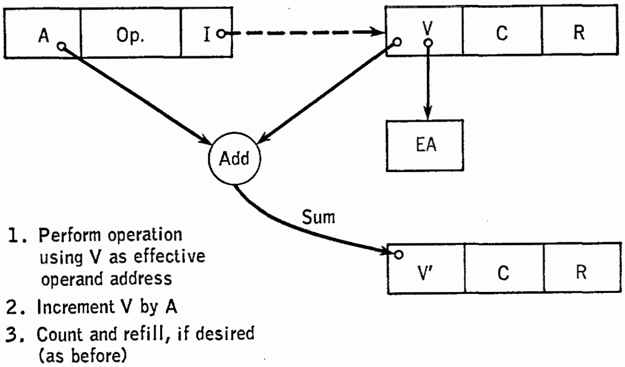
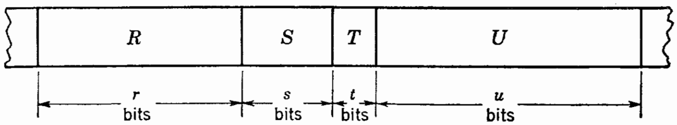

A
Op.
I
V
C
R
EA
Add
Sum
V′
C
R
1. Perform operation using V as effective operand address
2. Increment V by A
3. Count and refill, if desired (as before)

FIG. 11.6. Progressive indexing.

besides specifying the arithmetical operation, also specifies: *Use the index value as the effective address, and subsequently increment and count.* This type of indexing will be called *progressive indexing* and is shown in Fig. 11.6. Simple arrays that permit progressive indexing are frequently encountered both in data processing and in technical computations.

In the vector-multiplication problem of Table 11.2, the base addresses a_0 and b_0 could have been placed in the value field of i_0 and j_0, respectively. If progressive indexing were used, instruction $f + 5$ could be combined with $f + 4$ and, instead of using the COUNT AND BRANCH operation suggested in the previous section, instruction $f + 6$ could be combined with $f + 3$. As a result, the program would be shortened both in instructions and in execution.



e Load element R, length r bits, from location specified by i, and increment i by r.
$e + 1$ Compute with element R.
$e + 2$ Load element S, length s bits, from location specified by i.
$e + 3$ Compute with element S.
$e + 4$ Store new element S, length s bits, at location specified by i, and increment i by s.
$e + 5$ Add 1 to element T, length t bits, in location specified by i, and increment i by t.
$e + 6$ Load accumulator with a constant.
$e + 7$ Compare accumulator to element U, length u bits, in location specified by i, and increment i by u.

FIG. 11.7. Progressive indexing on elements of varying length.

The use of progressive indexing in a data-processing operation is illustrated in Fig. 11.7. A series of elements of different lengths is processed. During the computation appropriate for each element, addressing of the element is combined with progressive indexing. As a result, processing can proceed from one element to the next without extra index arithmetic. The example also shows the use of indexing words and bits within a word, as provided in the 7030.

11.8. Data Transmission

When an increment of 1 is implied, as in the COUNT AND BRANCH operation, the count becomes the equivalent of a length and represents the number of adjacent words in the addressed memory area. When, furthermore, the index value is used as an effective address, as in progressive indexing, the initial index value is the base address that refers to the first word of the memory area. A memory area can, therefore, be specified in position and length by the value field and count field of a *control word.* This makes it convenient to specify the memory areas involved in data transmission by means of control words and gives the control word the characteristic of a shorthand notation for a memory area.

Data may be transmitted between two memory areas or between input-output units and memory. The block of data transmitted in a single operation will be assumed to consist of one or more records (see Chap. 4). A control word may be used for both indexing and data transmission. This generality makes it possible to associate a control word with a record and to use it to identify the record throughout an entire program, including reading, processing, and writing. The use of control words in transmitting data directly between input-output units and memory is further described in Chap. 12.

11.9. Data Ordering

A common procedure in data-ordering operations, such as sorting, merging, queuing, inserting, and deleting, is to move records from one memory area to another. With control words it is possible to replace the transmission of a record containing many data words by the transmission of a single control word specifying that record.

As an example, consider n records stored in random order. It is desired to write the records on tape in a sequence determined by comparing one or more *identifier* fields in successive records. After the comparison is made, the actual sequencing is accomplished by ordering the control words associated with the records. To make the comparison, the identifier of each record is located by specifying its address relative to the base address in the control word for that record. In the course of this procedure the control words may be placed in the correct order in successive memory locations. The sequence of the control words then specifies indirectly the sequence of the associated records. When the records are written on tape, the control words are used in the order of their addresses. Consequently the records appear on tape in the desired sequence. No record transmisson is required other than from memory to tape.

The preceding example illustrates the case of a group of records that are to be moved as one block. The records cannot be described by a single control word since they are not necessarily in successive memory locations if their sequence is to be changed. The block is then described by a series of control words. The transmission to or from input-output devices can, however, be mechanized as a single operation by defining a *chain* of control words.[1]

A control-word chain is started by the control word specified in the instruction. The chain may be continued by taking control words from successive memory locations. The chain is ended when some kind of end condition is sensed. A convenient end condition is the presence or absence of a bit in the control words. This bit will be called the *chain flag* or *index flag*. Thus, a single input or output instruction can, by means of a chain of control words, initiate the transmission of a group of records. Records that appear in memory in random order are said to be *scattered.*

Control words were introduced in the IBM 709 in order to permit grouped-record transmission to or from external devices. In the IBM 7070, control words can be used both for grouped-record transmission and for indexing. Both machines establish a chain of control words by placing the words in consecutive memory locations.

TABLE 11.3. SEQUENCE OF CONTROL WORDS

Old	*New*
A	*A*
B	*B*
C	*C*
← *D*	*E*
E	*F*
F	*G*
. ↑	.
.	.
.	.
X	*Y*
Y	*Z*
Z	

An example of data ordering is the deletion of one record from a group of records. Assume that the records *A* . . . *Z* are in consecutive memory locations. To delete record *D* from this series, the records *E* . . . *Z* may be moved to the locations previously occupied by *D* . . . *Y*. The use of control words greatly simplifies this procedure. The grouped

[1] The chaining concept has been developed independently by Newell, Shaw, and Simon, who have shown many interesting examples of its function on a simulated computer. A. Newell and J. C. Shaw, Programming the Logic Theory Machine, *Proc. Western Joint Computer Conf.*, February, 1957, pp. 230–240; A. Newell, J. C. Shaw, and H. A. Simon, Empirical Explorations of the Logic Theory Machine, *ibid.*, pp. 218–230; A. Newell and H. A. Simon, The Logic Theory Machine, *IRE Trans. on Inform. Theory*, IT-2, no. 3, pp. 61–79, September, 1956; J. C. Shaw, A. Newell, H. A. Simon, and T. O. Ellis, A Command Structure for Complex Information Processing, *Proc. Western Joint Computer Conf.*, May, 1958, pp. 119–128.

records can be in random order in memory with their order established by control words in consecutive memory locations. The deletion of record D is accomplished by removing its control word from the table of control words and moving all subsequent control words one space, so that they again form a continuous table. Table 11.3 illustrates this procedure. Each letter now represents a control word rather than the actual record. The insertion of a record in a group of records may be handled by reversing this process.

Some conclusions may be drawn concerning the use of control words in data transmission and data ordering.

1. Since record transmission is replaced by control-word transmission, an advantage in storage space and transmission time is achieved. The advantage of the procedure is dependent upon the size of the record. When a record is only one word long, it is, of course, more advantageous to transmit records directly.
2. The location of a record and its control word are independent, which facilitates data ordering by control-word manipulation.
3. The use of identical control words for both indexing and data transmission simplifies data-ordering operations.
4. The records can be scattered in memory. The control words, however, have their sequence indicated by the sequence of their memory addresses. As a result of this restriction, activity on one record may require relocation of several control words for subsequent records.

11.10. Refilling

The advantage of using control words in data handling is increased when control words as well as records can be scattered. If control words may be located at random addresses, a means for specifying their sequence in a chain must be provided. A straightforward solution has been found: into the control word is introduced a *refill* field, which specifies the memory address of its successor. The control (or index) word then contains three major fields: the value field, the count field, and the refill field, as shown in Fig. 11.2.

This solution is particularly attractive since it also completes the indexing requirements stated in Table 11.1. It was shown at that point that an indexing operation required specification of the following: index value, increment, count, next initial index value, and next initial count. All these quantities except the last two have been specified so far, either in instructions or in the control word. The last two quantities can now be specified by the *refill* address. This address can refer to a second control word, whose value and count field specify the next initial setting. In fact, the second control word is the next initial control word.

The refill field then serves the general purpose of linking a control word with the next control word to be used.

The operations that use the quantities mentioned above were listed in Table 11.1 as follows: address modification, incrementing, counting and zero testing, replacement of index value and count. All these operations, except for the last, have been specified as machine functions. The last operation can now be restated as: *Replace the index word by the word at its refill address location.* The operation as stated makes use of an implied address. Therefore, the operation can be part of an ADD TO VALUE, COUNT, AND REFILL instruction. This combination of operations is meaningful only when the refill operation is conditional. An obvious condition is that the count reach zero. Refilling is shown at the bottom of Fig. 11.5. The instruction repertoire includes other related instructions, such as an unconditional operation REFILL.

The refill operation can also be incorporated in input-output data-transmission control. The control words comprising a data-transmission chain need no longer be in successive memory locations. One control word can refer to the next through its refill address. The chain flag indicates the termination of the chain and hence stops transmission (see also Chap. 12).

The refill function requires that the refill address be part of the index word. When a computer word is not large enough to contain all three fields, a partial solution can be found by using two adjacent words in memory. This procedure has been used in the input-output control of the IBM 709. In that machine, a set of consecutive control words may be joined to a set at another location by inserting a word having the character of the instruction: *Continue with the control word at the specified location.*

An alternative use of the refill address has been considered. The refill address could be used as a branch address rather than as a control-word address. With this procedure, whenever the test condition is satisfied, a branch is made to a subroutine that takes care of all termination and initialization procedures. As a minimum, the control word can be reloaded, but more elaborate programs can be performed. This procedure is more general than the refill operation defined above. The cost of this generality, however, is loss in efficiency in the minimum *reload* procedure: a *branch* as well as a *load* operation is performed, and each control word requires an associated *load* instruction. In other words, the use of an implied address in the main program is obtained at the expense of explicit addresses in a subroutine. The ability to permit more elaborate initialization procedures is often incompatible with the use of the control word in different parts of a program. For these and other reasons, the refill operation in the 7030 has been preferred to the branch procedure or to any of the many variations thereof.

11.11. Indirect Addressing and Indirect Indexing

Indirect addressing consists in substituting another address for the address part of an instruction before that instruction is executed, without changing the instruction as stored in memory. A simple and effective form of indirect addressing is found in the IBM 709 and several other machines, where, under the control of an instruction bit, the operand address A_1 refers to another word in memory where the actual address A_2 of the final operand is located. It is possible to extend indirect addressing to more than one level by having the address A_2 refer to yet another word containing address A_3, which in turn refers to A_4, etc., until the process is terminated either by an end mark of some kind or by previous specification of the number of levels desired.

So that it will not be necessary in the 7030 to tie up a bit in every instruction for indirect addressing, a separate instruction, LOAD VALUE EFFECTIVE, is provided, which serves, in effect, as a prefix to the main instruction. The operation is illustrated in Fig. 11.8. Basically this instruction fetches an address from memory and places it temporarily in an index register. If this address is to be used as an indirect address in a subsequent instruction, a zero address part is added to the contents of the same index register by the regular address-modification procedure.

More precisely, the effective address of the LOAD VALUE EFFECTIVE instruction is used to fetch a second instruction word from memory. If that instruction again has the operation code of LOAD VALUE EFFECTIVE, the process is repeated and another instruction word is fetched. The indirect-addressing process terminates when the operation code is anything other than LOAD VALUE EFFECTIVE. The final, indexed operand address is stored in an index register, specified by the initial LOAD VALUE EFFECTIVE instruction. This procedure permits any number of levels of indirect addressing.

If the address part of the using instruction is not zero, the process may be termed *indirect indexing,* which gives another degree of flexibility over indirect addressing.

LOAD VALUE EFFECTIVE plays a second role in the 7030. Its operand is assumed to be an instruction word, and the operation code of the instruction word is examined to determine whether its address part is 18, 19, or 24 bits long. The address is automatically transformed to a standard 24-bit length before it is placed in the value part of the index register. All other indexing instructions, such as LOAD VALUE, are assumed to refer to index words; they do not provide format conversion, and their operands cannot be indexed.

The particular implementation of indirect addressing in the 7030 suggests the strong relationship between indirect addressing and additive address modification. Both processes modify addresses "on the fly" and serve to reduce the number of places where the program must alter addresses. In smaller machines, where a separate index adder may not be economically justified, it is possible to use indirect addressing instead

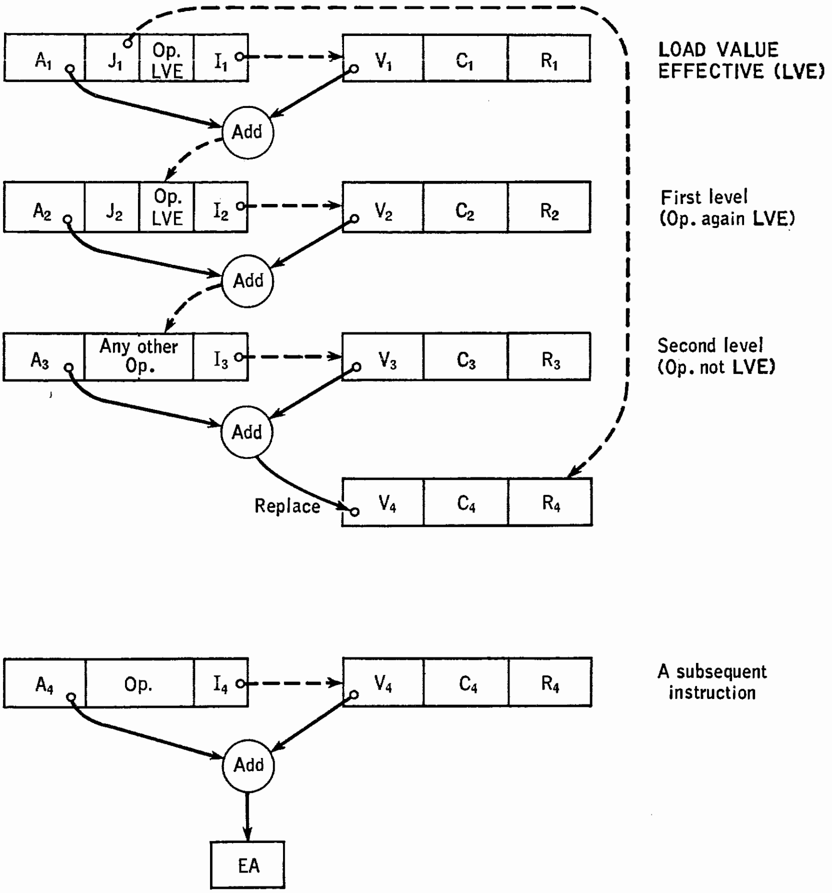


FIG. 11.8. Indirect addressing and indirect indexing. If $A_4 = 0$: indirect addressing. If $A_4 \neq 0$: indirect indexing. LOAD VALUE EFFECTIVE can be repeated automatically any number of times; two levels of indirect addressing are shown; last level is one where operation code encountered is something other than LVE.

of additive address modification and to form and increment the indirect addresses with ordinary arithmetical instructions. Fast substitution is simpler to implement than fast addition. The function of additive modification finds such frequent use, however, that extra equipment for fast indexing is fully justified in the larger machines.

11.12. Indexing Applications

The basic indexing formats and functions have been defined in the preceding sections. In the rest of this chapter the use of the indexing mechanism will be demonstrated; the examples used above to illustrate its evolution will be reexamined, and some more elaborate applications will be considered. Of the indexing applications, the simple example of vector multiplication described earlier will be discussed, and also its expansion to matrix multiplication.

The vector-multiplication program was listed in Table 11.2. The same program using the *refill* operation is shown in Table 11.4. Here the control words are automatically reset. When the program is executed repeatedly, it is sufficient to start at the initial setup instruction g. When, however, the execution of the program is stopped prematurely and must be restarted, the preparatory steps $g - 2$ and $g - 1$, which

TABLE 11.4. VECTOR MULTIPLICATION USING COUNT, BRANCH, AND REFILL

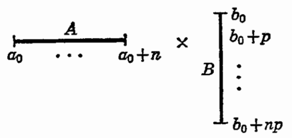
		Instructions
Preparation	$g - 2$	Load i from i_0
	$g - 1$	Load j from i_0
Initial setup	$\longrightarrow g$	Set accumulator to zero
Vector multiply, inner loop	$g + 1$	Load cumulative multiplicand from a_0 indexed by i ←
	$g + 2$	Multiply cumulatively by b_0 indexed by j
Housekeeping, inner loop	$g + 3$	Increment j by p, count, refill when count reaches zero
	$g + 4$	Advance i, count, refill when count reaches zero, branch to $g + 1$ when count does not reach zero
Vector multiply, outer loop	$g + 5$	Store cumulative product at c_0

Control words

Contents after executing the inner loop x times:

Address	*Index value*	*Count*	*Refill*
i	x	$n - x$	i_0
i_0	0	n	i_0
j	xp	$n - x$	i_0

Diagram of vector dimensions



load i and j, respectively, are required. Thus loading of i and j should always be part of the program-loading procedure. The control words i and j are specified by truncated addresses and are located in the index registers. The control word i_0 has a complete address and can be located anywhere in memory. The program illustrates the use of a COUNT, BRANCH, AND REFILL instruction. Because the base addresses a_0 and b_0 are part of the operand address, the control word i_0 can serve as a refill word for both i and j.

The program for matrix multiplication is outlined in Table 11.5 (it is also included in the Appendix as a programming example using actual instructions). Again the initial setup instruction h would be sufficient ordinarily, but preparatory instructions $h - 2$ and $h - 1$ are needed to permit restart after premature stoppage.

All three matrixes are assumed to be stored row by row. Index i progresses across the rows of matrix A, being advanced by 1 at the combination *index-and-branch* instruction $h + 7$. Index i repeats the same row p times, being refilled from i_0 each time at the end of the row. Index i_0 is then advanced by n to the next row (at $h + 11$); the process is repeated m times. Similarly, index j progresses down the columns of matrix B. j is incremented n times by p (at $h + 6$), after which j_0 is advanced by 1 to the next column (at $h + 10$) and used to reload j (at $h + 2$). The incrementation of j_0 is counted p times and used to determine the end of the product row. j_0 is then refilled from j_{00} to start again at the beginning of matrix B for the next product row. Index k is used to progress row by row through the product matrix C and to determine the end of the entire matrix multiplication.

The program shows that a reasonably complex indexing procedure can be described satisfactorily and compactly. The following observations may be made:

1. Only instructions $h + 6$ and $h + 11$ contain constants that describe the locations and dimensions of the matrixes. Both instructions could use a direct address instead of an immediate address, however. In that case, the program would be independent of the data. The use of a direct address slightly increases execution time.

2. The constants describing matrix locations and dimensions appear as single quantities in instruction and control-word fields. Note that only control words i_{00}, j_{00}, and k_0 are supplied by the programmer. All other control words are developed during program execution or preliminary setup.

3. The automatic refill is used in the inner loops. The refill operation is supplemented by load operations in the outer loops. The refill operation is no substitute for preparatory operations required for restart procedures.

TABLE 11.5. PROGRAM FOR MATRIX MULTIPLICATION

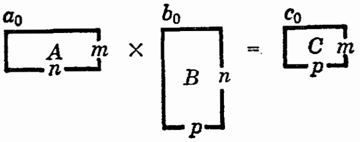
Instructions		
Preparation	$h - 2$	Load k from k_0
	$h - 1$	Load j_0 from j_{00}
Initial setup	→ h	Load i_0 from i_{00}
New product row procedure	$h + 1$	Load i from i_0
New vector product procedure	$h + 2$	Load j from j_0
	$h + 3$	Set accumulator to zero
Vector multiply, inner loop	$h + 4$	Load cumulative multiplicand from location specified by i
	$h + 5$	Multiply cumulatively by operand location specified by j
Housekeeping, inner loop	$h + 6$	Increment j by p
	$h + 7$	Advance i, count, refill when count reaches zero, branch to $h + 4$ when count does not reach zero
End of vector multiplication procedure	$h + 8$	Store cumulative product at location specified by k
	$h + 9$	Increment k by 1
	$h + 10$	Advance j_0, count, refill when count reaches zero, and branch to $h + 2$ when count does not reach zero
End of product row procedure	$h + 11$	Increment i_0 by n
	$h + 12$	Reduce count of k, refill when count reaches zero, and branch to $h + 1$ when count does not reach zero

Control words

Contents after executing the inner loop x times for the product matrix element c_{rs}:

Address	*Index value*	*Count*	*Refill*
i	$a_0 + rn + x$	$n - x$	i_0
i_0	$a_0 + rn$	n	i_0
i_{00}	a_0	n	i_0
j	$b_0 + s + xp$	$p - s$	j_{00}
j_0	$b_0 + s$	$p - s$	j_{00}
j_{00}	b_0	p	j_{00}
k	$c_0 + rp + s$	$m - r$	k_0
k_0	c_0	m	k_0

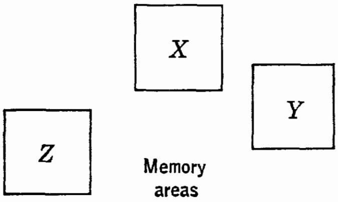
Diagram of matrix dimensions



11.13. Record-handling Applications

Record-handling techniques have application both in technical computation and in data processing. The examples to be discussed are a read-process-write cycle, ordering, and a file-maintenance procedure.

The use of control words for a simultaneous read-process-write cycle is illustrated in Fig. 11.9. Here X—x describes a control word, which, by its value and count fields, defines memory area X and which has the address x in its refill field. Location x contains the next control word in the chain, Y—y, defining record Y. Control word Z—z is placed at location y. Because control word X—x is stored at location z, a *ring* of three memory areas, X, Y, and Z, is set up in which X is followed by Y, Y by Z, and Z again by X. Both record areas and control words may be scattered throughout memory. Note that, in this notation, an upper case letter is used to denote the location of a record area and the corresponding lower-case letter is used to denote the location of the control word of the *next* area in sequence.



Location	Control word	Control words used
x	Y—y	Read X—x, Y—y, Z—z, X—x, ...
y	Z—z	Process X—x, Y—y, Z—z, ...
z	X—x	Write X—x, Y—y, ...

FIG. 11.9. *Read-process-write* chain.

The example of Fig. 11.9 shows the sequence of operations in a read-process-write cycle. While a record is being read into area Z, as controlled by control word Z—z, processing proceeds with control word Y—y using data in area Y, and data from area X are written under control of control word X—x. At the conclusion of each of these operations, the appropriate control word is refilled, and the areas are thereby cyclically permuted in function.

Instead of a single control word, a chain of n control words could be used in reading, while a second chain of n control words is used in processing, and a third chain of n control words is used in writing. To further elaborate the example, assume that processing consists of placing the n records in a preferred sequence. This sequencing operation was described above. Because of the refill field, however, the control words do not have to be in sequential locations. The advantage of this added degree of freedom will be shown in the following examples.

Assume that the records *A* . . . *Z* are scattered throughout memory. The associated control words *A*—*a* . . . *Z*—*z* establish their order. The correct order is here indicated by the alphabetic sequence. It is desired to delete record *H*, which is out of sequence, and to set its memory area aside. The control word *H*—*h* of this record is part of the chain *C*—*c* . . . *K*—*k* shown in the left half of Table 11.6. Interchanging the contents of locations *d* and *h* establishes a new order, as shown in the right part of Table 11.6, and *H* is no longer part of the sequence. A second interchange between *d* and *h* would reinsert *H*. Thus the complementary nature of insertion and deletion is reflected in the programming procedure.

TABLE 11.6. RECORD DELETION

Before		*After*			
Location	*Control word*	*Location*	*Control word*	*Location*	*Control word*
. . .	. . .	. . .	. . .		
b	*C*—*c*	*b*	*C*—*c*		
c	*D*—*d*	*c*	*D*—*d*		
d	**H—h**	*d*	**E—e**		
h	**E—e**			*h*	**H—h**
e	*F*—*f*	*e*	*F*—*f*		
f	*G*—*g*	*f*	*G*—*g*		
g	*I*—*i*	*g*	*I*—*i*		
i	*J*—*j*	*i*	*J*—*j*		
j	*K*—*k*	*j*	*K*—*k*		
. . .	. . .	. . .	. . .		

If it is desired to insert *H* in the sequence . . . *G*, *I*, *J*, . . . between *G* and *I*, the second interchange would be between *g* and *h*. Table 11.7 illustrates this case.

Because the sequence . . . *G*, *I*, *J*, . . . is part of the sequence *A* . . . *Z*, the example is equivalent to a sorting operation. The sequence . . . *G*, *I*, *J*, . . . may equally well be part of an independent sequence, as it is in file maintenance.

The interchange of two control words is performed conveniently by a SWAP instruction. This instruction interchanges the contents of two memory words. The insertion or deletion of a record involves only the SWAP of its control word with that of its successor. The insertion and deletion of a group of records is equally simple. Consider again the file *A* . . . *Z*. It is required to delete the group *P* . . . *R* from the file shown on the left in Table 11.8. A SWAP instruction is given for locations *c* and *r*, and so the new order becomes as shown on the right in Table 11.8.

TABLE 11.7 RECORD INSERTION

Before				*After*	
Location	*Control word*	*Location*	*Control word*	*Location*	*Control word*
. . .	. . .			. . .	. . .
b	*C—c*			*b*	*C—c*
c	*D—d*			*c*	*D—d*
d	*E—e*			*d*	*E—e*
e	*F—f*			*e*	*F—f*
f	*G—g*			*f*	*G—g*
g	**I—i**			*g*	**H—h**
		h	**H—h**	*h*	**I—i**
i	*J—j*			*i*	*J—j*
j	*K—k*			*j*	*K—k*
. . .	. . .			. . .	. . .

TABLE 11.8. GROUP DELETION

Before		*After*			
Location	*Control word*	*Location*	*Control word*	*Location*	*Control word*
. . .	. . .	. . .	. . .		
b	*C—c*	*b*	*C—c*		
c	**P—p**	*c*	**D—d**		
p	*Q—q*			*p*	*Q—q*
q	*R—r*			*q*	*R—r*
r	**D—d**			*r*	**P—p**
d	*E—e*	*d*	*E—e*		
e	*F—f*	*e*	*F—f*		
f	*G—g*	*f*	*G—g*		
g	*H—h*	*g*	*H—h*		
. . .	. . .	. . .	. . .		

One SWAP instruction deletes the group of records just as one SWAP instruction in the previous example deleted a single record. The only differences are the addresses of the instruction. The records *P* . . . *R* form a ring in sequence. (In the previous example, the deleted record *H* could be considered to form a ring in sequence, since its control word was stored at its own refill location.) The reinsertion of the records *P* . . . *R* can be performed by swapping again the contents of locations c and r.

In these examples the sequence of control words is changed by transmitting entire words. A different approach is to transmit refill fields only, leaving the rest of the control word unchanged in memory. This method can also be used in many applications.

11.14. File Maintenance

A simple case of the updating of a master file from a detail file will be discussed. Four tapes are used: the old master tape, the new master tape, the detail input tape, and the detail output tape. The detail records are processed in a simple input-process-output operation such as that described above. The master records are read from the old master tape, processed, and written on the new master tape. Reading, writing, and processing take place simultaneously. The processing of a master record may involve:

1. No activity
2. Updating
3. Deletion of obsolete records
4. Insertion of new records

Master records are read and written in blocks, each block containing a group of m records. Memory space is set aside for a total of $4m$ master records and their control words. Normally, m records are written on the new master file while m records are being read from the old master file. The remaining $2m$ record spaces are available for processing. These record spaces are divided into two groups: the current spaces and the spare spaces. The current record spaces contain records that either have been processed and are ready to be written on the new master tape or have been read from the old master tape and are available for processing. The spare record spaces contain no useful record information. The number of current and spare spaces varies throughout the processing, but their sum remains $2m$.

The control words used in reading and writing and the control words of the current records form a ring. The control words for the spare record areas also form a ring. Figure 11.10 shows the control words in diagram form and illustrates the cases discussed below for $m = 8$.

When a record is inactive or requires updating, the number of current and spare records remains unchanged. The record is addressed by means of its control word. After the processing is completed, the current control word is replaced by the next one in order by means of a REFILL instruction. The record is ready to be written on the new master tape. A count is kept of the records that are ready to be written. When the count equals m, a WRITE instruction is issued which is followed by a READ

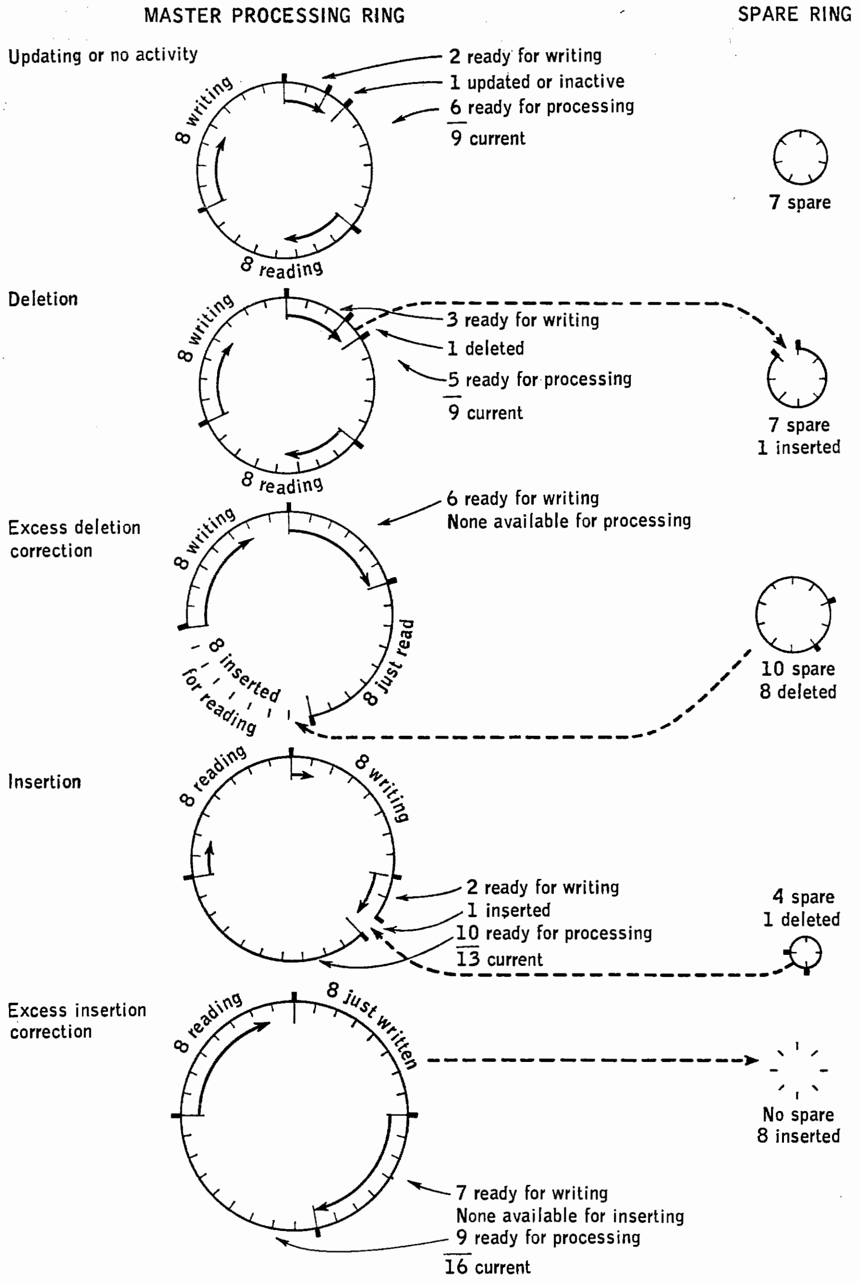


FIG. 11.10. Control-word diagram for file maintenance.

instruction. The record space of the records just written is used for the records to be read. The records just read are available for processing.

When a record is found to be obsolete and should be deleted, its control word is removed from the ring of current control words and inserted in the ring of spare control words. Because the control word is deleted, its record is not written on the new master tape. The count of records ready to be written is not changed. The control word of the next record is obtained, and processing continues.

When there is an excess of deletions, all current records may be processed before *m* records are ready to be written. In that case the number of spare record areas is always larger than *m*, and a corrective step can be taken. This step consists of deleting *m* control words from the spare ring and inserting them in the read-process-write ring. The control words are inserted as a block preceding the control words used in reading and following those used in writing. An extra READ instruction is given, and processing proceeds with the records that have just been read.

When a new record is to be inserted, a control word is removed from the ring of spare control words and inserted in the ring of current control words. The corresponding record area is then available for the new record. After the new record is processed, it is ready to be written.

When there is an excess of insertions, the spare control word ring may have been reduced to zero. A corrective step then should be taken: *m* control words are deleted from the read-process-write ring and used as a new spare ring. The *m* control words deleted are those last used in a WRITE operation. Writing is checked for completion. The next time that *m* records are ready to be written, the WRITE instruction is given, but the READ instruction is omitted.

The file-maintenance procedure outlined above illustrates the use of insertion and deletion of single records and groups of records. All the manipulations described are performed conveniently with control words and would require a great deal of housekeeping without the refill feature.

11.15. Subroutine Control

Another application of control words is in subroutine control. In the preceding discussion the control word specified a memory area that normally would contain data. The memory area might also contain instructions, however. A subroutine may be thought of as a record.

As an illustration, consider the use of exception subroutines, which are stored on tape, drum, or disk and are called in when the associated exceptions arise. The control word is used in the READ instruction; it can subsequently be used for address modification in the BRANCH instruction that refers to the subroutine and in the instruction that stores the instruction-counter contents. The subroutines, therefore, can be inserted conveniently in a main sequence of instructions.

11.16. Conclusion

The preceding discussion has shown the application of control words in address modification and in record handling. Both indexing and data-transmission techniques make it desirable to have an *index value, count,* and *refill* facility. The three fields in the control word and the associated machine functions satisfy these requirements. The control words provide substantial saving in program space and increase in machine speed. They simplify programming of housekeeping operations.

Control words do not introduce entirely new functions, since their operation can be simulated on any stored-program computer. Also, the introduction of *count* and *refill* is only a second-order improvement as compared with the first-order improvement of address modification through indexing. Control-word operation is, however, so much simpler than its simulation that several otherwise impractical methods of record control have now become feasible.

The indexing instructions have been described as they appear in the IBM 7030. Though elements of the system discussed here have been used in other machines, the 7030 control-word system as a whole is new, for the effectiveness of these techniques depends largely on the combination of all the features.

Chapter 12

INPUT-OUTPUT CONTROL

by W. Buchholz

12.1. A Generalized Approach to Connecting Input-Output and External Storage

One of the drawbacks of early computers was the primitive nature of their input and output equipment. A small amount of data might be entered on paper tape, computation would then proceed, and results would finally be printed on a typewriter. Subsequent development of input-output and external storage devices has given us not only faster equipment but also a greater variety. Magnetic tape, drums, and disks provide external storage to supplement the internal memory. Card readers, card punches, and mechanical line printers have become commonplace items in most installations. Fast cathode-ray-tube printers, displays, and plotters provide alternative output means. Phone lines and inquiry stations allow direct communication with computers. Analog-digital conversion equipment permits digital computers to be used to control continuously variable processes.

The list of input-output equipment may be expected to grow, and, as it grows, a computing installation will come to be characterized more by the array of external units than by the nature of the central computer. It is, therefore, desirable to avoid restrictions on the number and kinds of units that can be connected to a general-purpose computer. To achieve sufficient generality, the design of the 7030 input-output system followed these principles:

1. A large number of logically identical and independently operable input-output channels should be provided.

2. The input-output instructions should be independent of the nature of the units they control. They should identify a channel and the connected input-output unit only by addresses.

3. As a corollary, there should be no equipment in the computer that is

peculiar to any kind of input-output unit. All control circuits peculiar to a given unit are required to be part of the control box for that unit.

4. The operation of a channel should be independent of the speed of the input-output unit connected to it up to the maximum speed for which the channel is designed.

Because of the enormous range of speeds encountered (from 0 to about 10,000,000 bits per second), it was found desirable to provide more than one kind of channel, so as to cover the speed range economically. The differences lie mainly in the number of bits transmitted in parallel and the number of channels sharing common equipment. The discussion here will be concerned only with the basic channels, which can transmit 8 information bits in parallel at a rate of over 500,000 bits a second. More parallelism is needed, with present technology, to go to much higher speeds; serial-by-bit transmission may be desired to reduce the cost per channel when a large number of quite slow units are to be connected. It should be noted that the variations are associated only with speed ranges, not with functional differences.

The execution of input-output instructions takes place in a portion of the computer called the *exchange.* The exchange accepts input-output instructions from the instruction-preparation section of the computer and executes them independently of the rest of the computer. The exchange also contains common control equipment which is used in time-shared fashion by all channels operating simultaneously. The exchange is described in Chap. 16. The present chapter is concerned mostly with the instruction logic for operating any one channel.

12.2. Input-Output Instructions

Four basic instructions make up almost the entire repertoire for performing any kind of input-output operation: WRITE, READ, CONTROL, LOCATE.

WRITE causes a stream of data from the computer memory to be transmitted to an external unit, there to be written on a storage or recording medium. Conversely, READ initiates the flow to the computer memory of data that have been read on a storage or recording medium at an external unit. The medium may be a physical medium such as tape, cards, or paper; it may also be a phone line or the memory of another computer, which may be connected to this computer as if it were an input-output unit. (The terms *writing* and *reading* are used here in such a way as to describe the data flow with respect to the input-output unit. To avoid confusion, the different terms *storing* and *fetching* are used to describe the data flow with respect to internal memory.)

Each WRITE or READ instruction contains two addresses (Fig. 12.1): the *channel address* identifying the channel to which the desired unit is connected, and the *control-word address* specifying the memory location where additional information for executing the instruction is to be found in the form of a *control word.*

CONTROL and LOCATE resemble WRITE in that bits are transmitted to the external unit, but these bits are not data to be recorded. In CONTROL the bits, which are obtained from the second-address part of the instruction itself, are a code to direct the unit to perform specified functions other than writing or reading. In LOCATE the bits constitute a secondary selection address which is needed by some kinds of external units.

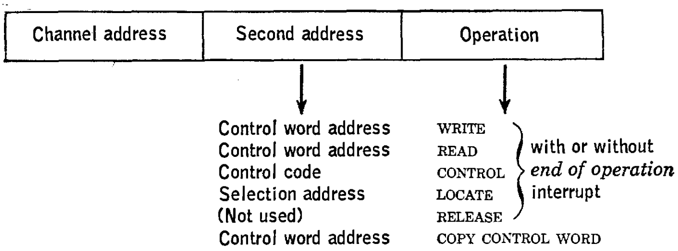


FIG. 12.1. Principal parts of input-output instructions.

There are two more instructions, COPY CONTROL WORD and RELEASE, which perform certain auxiliary functions in the exchange. Their use is infrequent, and they will not be considered further.

12.3. Defining the Memory Area

Basically, the control word (see Fig. 12.2, also Chap. 11) defines a continuous area in memory, which is the source of the data stream during writing or the sink for the data stream during reading. The location of the area is defined by the *data-word address,* which specifies the address of the first word, and the size of the area is defined by the *count,* which gives the total number of words in the area. (For simplicity, input-output operations can address only full memory words of 64 bits, and memory areas can be defined only as multiples of full words.) The first word is always the word at the lowest address. Writing or reading starts at that address and, unless otherwise specified, steps through progressively higher addresses to the end of the area, as defined by the count.

Each control word can define only one continuous memory area, but several control words can be chained together so that a single writing or reading operation can proceed through more than one continuous area. For this purpose, each control word contains a *refill address,* which gives the location of the next control word to be used, and a *chain flag,* which defines the extent of the chain. A chain flag of 1 permits writing or reading to continue automatically beyond the area of the current control word to the next area specified indirectly by the refill address; a chain flag of 0 stops the process at the end of the current area regardless of what is in the refill address.

Thus, a chain of control words defines a memory area in the larger sense in which successive words are not necessarily at consecutively numbered addresses. Because the same chain of words can be used as control words during reading, as index words during computing, and again as control words during writing, powerful procedures are available for complex record handling, as described in Chap. 11.

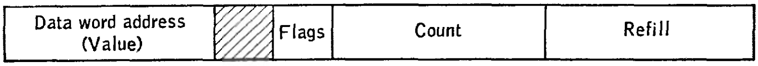


FIG. 12.2. Control word.

12.4. Writing and Reading

When a WRITE or READ instruction is given, the unit attached to the channel specified is started and data are transferred between that unit and the memory area defined by the specified control word (or chain of control words). Normally, a single block of data is transferred each time an instruction is given; a *block* of data is defined for each type of unit as the amount of information recorded in the interval between adjacent starting and stopping points of the recording medium.

The length of the block depends on the type of unit used. It may be the contents of one punched card, a line of printing, or the information between two consecutive gaps on magnetic tape. In some units the length of each block may be fixed by the design of the unit (card reader or line printer), but in other units the block length may be left variable (magnetic tape). Again, the length of a block may correspond to the natural size of one unit record; it may be set to correspond to a group of such records (e.g., for greater efficiency on magnetic tape); or it may occasionally be neither.

Thus, the size of the memory area is defined by the control word, and the length of the block is often, but not always, defined by the unit. One can distinguish three cases:

1. The block length is not defined, and the operation terminates when the end of the memory area is reached (including any chaining). For example, in writing on magnetic tape, the tape unit stops and creates an interblock gap whenever the last memory address of the last control word in the chain is reached. (Writing tape is different from reading tape, where the block size is defined by the previously written interblock gap.)

2. The block is shorter than or equal in length to the defined memory area, and the operation stops at the end of the block. The rest of the memory area, if any, is ignored.

3. The block is longer than the defined memory area, and the data transfer ceases when the end of the memory area is reached. Since the unit cannot stop in the middle of a block, it continues without transferring any data until the end of the block.

What has been described so far is *single-block* operation. For additional flexibility, *multiple-block* operation is also provided in the 7030 system. It is specified by setting a *multiple flag* in the control word to 1. In the multiple mode, when the end of a block is reached, the WRITE or READ operation is allowed to continue by starting a new block; the operation is finally terminated when the end of the defined memory area is reached. The multiple mode is generally equivalent to a sequence of WRITE or READ instructions in the single mode, except that the computer program is not interrupted until the sequence is finished. (This advantage is gained at the expense of more complex exception handling.)

It may be noted that the memory area defined by a chain of control words cannot be exceeded regardless of the mode. A properly defined set of control words thus provides protection against accidental erasures outside the memory area assigned to a specific input-output operation, such as might otherwise be caused by reading blocks longer than expected.

12.5. Controlling and Locating

Most input-output units require certain programmable control functions in addition to writing and reading, such as rewinding tape, selecting different operating modes, turning warning lights on and off, or sounding a gong. Instead of numerous highly specialized instructions for each of these functions, some of which might have to have different meanings for different units, a single CONTROL instruction is used here for the sake of generality. This instruction causes a code to be sent to the external unit, which interprets the code to perform the desired operation.

Thus CONTROL has only a single meaning in the computer. The specialized functions are defined separately for each external unit and form part of its design specifications. They may range from an elaborate set of control functions for some high-performance units to none at all for rather primitive devices. The input-output channels remain general in function and need not be altered as new devices are attached or as the mix of units is changed.

The control code is placed in the second address of the instruction, in the manner of an immediate address, and there is no reference to a memory location. The first address of the instruction specifies the channel, as before (Fig. 12.1).

The LOCATE instruction resembles CONTROL in all respects except that the bits sent to the unit are interpreted by the unit as a secondary selection address rather than a secondary operation code. Examples of the use of secondary addresses are: selecting the desired one of several tape units connected to a single channel; directing the access mechanism of a disk file to a desired position. The selection addresses are limited to a maximum of eighteen bits.

12.6. An Alternative Approach

The similarity between the above CONTROL and LOCATE operations suggests the possibility of combining them into a single *control* operation. The first 8-bit byte of the control data would become the secondary operation code; it could specify whether additional bytes containing address information were to follow. The number of bytes needed would be determined by the external unit.

The restriction on selection addresses, which are limited in length by the instruction format, can be removed by changing from immediate addressing to direct or even indirect addressing. With direct addressing, the second address of the new *control* instruction would specify a 64-bit memory word, part or all of which could be sent to the unit as desired. An even more general system is provided by indirect addressing, where the address specifies a control word which defines the address and the amount of information, as in WRITE. Although it would require an extra memory word and access, indirect addressing would have the advantage of simplicity, since this version of *control* would be executed in a manner identical with WRITE.

A second generalization would be to provide the inverse of this *control* operation, which we shall call *sense.* This *sense* operation would be a request to the external unit to send back various status indications, such as manual switch settings and reports of termination or error conditions. *Sense* would be treated like READ if indirect addressing were used, the status bits being stored in a memory area defined by a control word.

This alternative set of *control* and *sense* operations has the advantage of symmetry, simplicity, and flexibility. It was not incorporated in the 7030, but the scheme has since been adopted in other input-output systems.

12.7. Program Interruptions

One of the important functions of a *program-interrupt* system (Chap. 10) is to resynchronize the computer program with the external operations, which, having been initiated by the program, are completed independently. An equally important function is to request program attention to a process that is initiated externally, by an operator for example.

After giving an input-output instruction, the program is not allowed to proceed until the exchange has accepted or rejected the instruction. If the exchange finds that the desired unit is not ready to operate, or the channel is already in use from a previous instruction, or the instruction is incorrect, the exchange will reject the instruction by turning on an indicator and interrupting the program. Otherwise the instruction is accepted for execution by the exchange, and the program is released to proceed with the next instruction in sequence. The program may at any time initiate an input-output operation for another channel that is not busy. Any number of such operations may be accepted and processed by the exchange independently and simultaneously, up to the maximum traffic-handling ability of the exchange.

Thus, the exchange and the computer proceed independently once an input-output operation has been started. At the end of an input-output operation, the program is again interrupted. The channel address of the particular channel whose operation has been completed is supplied to the program; indicators show whether the operation was completed successfully or whether some unusual condition was encountered. Thus a program, which may have been waiting for the input-output operation to finish, can be resumed at the earliest opportunity without the need for repeated testing of the indicators. The unit that was stopped may be restarted by the program with a minimum of delay. The interrupt system, therefore, provides an effective method for bringing the program and the independently operable input-output units back into step at intervals.

An alternative mode exists whereby program interruption can be prevented when the operation ended normally, with interruption occurring only for the exceptions. Another mode suppresses all interruptions, so that a supervisory program, for example, may initiate a special input-output sequence before having to pay attention to a unit that has just completed its cycle. Additional flexibility is gained by writing the program to accept interruptions but storing the indications in a queue for later use if the interrupting unit is not to be given top priority (see Chap. 13 for a more extended discussion of these subjects).

A third type of interruption from an external source is a request to the program to issue an input-output instruction when no such operation has been in progress. This interruption is called *channel signal.* Frequently, the source is a human operator. A *channel signal* may be issued when the operator has loaded a unit with fresh material (tape, paper, cards) and pressed the *start* button. An operator may be ready to enter information from a keyboard; if a READ instruction is not already waiting for the information, the operator may, in effect, request such an instruction by pressing a button that causes a *channel signal* to be issued. The *channel* *signal* does not itself initiate any operation in the computer, and a suitable program must be available in the computer; so the programmer has full freedom to interpret such signals in any manner, including the option to ignore them when they are not appropriate.[1]

Another use of *channel signal* is as a second-level, end-of-operation interrupt signal. Some common control units permit two or more input-output units, attached to the same control unit and channel, to perform simultaneous operations, so long as only one operation involves data transmission over the channel. The secondary operations that do not require the channel (such as rewinding tape or locating a new position for an access arm on a disk file) are often of long duration compared with the primary operations that do occupy the channel (such as reading or writing). *Channel signal* then indicates the completion of the secondary operation. The two uses of *channel signal* are not unrelated. Even operator interventions can be considered to be under indirect program control, since instructions from the program to the operator are either implied or explicitly given, if human intervention is to result in meaningful actions by the program. The main difference lies in the less predictable waiting time and the surely erratic behavior of human beings.

In summary, *channel signal* is the computer's internal telephone bell. It summons the program to attend to the channel whose bell has rung. (To be quite fair, the computer, in turn, is allowed to sound a gong on the console to summon an operator.)

12.8. Buffering

Buffer storage external to the main memory is used in many computers to match data transmission rates and to minimize delays. The 7030 system, however, makes no use of external buffers when it is possible to transmit directly between the recording medium and the memory in sequential fashion. The card-reader- and card-punch-control units do contain buffers, because of a need to transpose bits; the reader and punch happen to feed cards row by row, whereas it is more desirable to transmit the data to and from memory in column-by-column form. Similarly, the chain printer used in the 7030 system, even though serial in operation, is designed so that the same bytes must be presented repeatedly and not in the sequence in which they appear on paper. Although programs could be written to make the necessary transformations inside the computer, it seemed that special buffer storage devices could do these highly repetitive

[1] One exception occurs when the computer is to be loaded with its initial program and a meaningful program cannot be assumed to exist in memory already. *Channel signal* is then used to start a built-in sequence to read one block from the unit that generated the signal. The initial program-loading sequence becomes inoperative once used.

chores more effectively. The buffers make the devices appear to the computer as if they were serial in operation.

Devices that are inherently serial, such as magnetic tape units, disk files, and typewriters, have no separate buffer storage. (We must distinguish buffer storage, which holds one or more complete blocks of data, from registers capable of temporarily holding one or more bytes in the control unit to smooth out the data flow and perform a small-scale buffering function "on the fly." As the term is used here, a buffer receives a complete block of data from one device and then transmits the block to another device, usually at a different speed. A buffer permits either device to stop between blocks and delay transmission of the next block indefinitely.) Since the introduction of buffer storage represented significant progress at one stage in the development of computers, its omission in the 7030, with the exceptions mentioned above, calls for a word of explanation.

The simplest technique, both in terms of equipment and programming, is unbuffered, single-channel operation. When an unbuffered computer issues an instruction to read or write, the input-output unit is started and data are then transmitted while the computer is waiting. The computer cannot continue until data transmission is ended. When input-output activity is at all high, long waiting periods greatly reduce over-all performance.

Buffered, single-channel operation was the next step. When transmission between a unit and its buffer is much slower than transmission between the buffer and main memory, it becomes possible to reduce the *waiting time of the computer* by the difference in transmission times and by omission of the start-up time of the unit. There is still an irreducible waiting time for the computer: the time for transmitting data between buffer and memory.

In applications where computing time is less than input-output time, the *waiting time of the input-output unit* becomes important. When only a single buffer is provided, the unit must wait until data transfer between buffer and memory has been completed. This wait includes the unavoidable delays after completion of the input-output cycle before the program can initiate the transfer, as well as the transfer time. By doubling the buffer storage area and alternating between the areas, one can avoid such delays in the operation of the input-output unit.

The 7030 (like several earlier computers) uses buffered, multiple-channel operation without requiring external buffers. Buffering is accomplished by transmitting data directly between the unit and memory over one of several input-output channels at whatever rate is required by the unit. Each channel operates independently of the computer and of the other channels. This may be termed *buffering in memory.* The internal memory acts as one large buffer for all channels. Its use is time-shared automatically among the computer and the input-output channels by allocating memory cycles to each device when needed. If more than one device demands attention, each is served in turn, with the highest priority assigned to the channel with the fastest input-output unit.

There are a number of advantages to this arrangement. An obvious advantage is a considerable reduction in equipment as compared with having a separate buffer on each channel, a saving which is partially offset by the prorated cost of the main memory areas serving as buffers. The size of the buffer area in main memory can be adjusted to the balance between cost and performance desired for each application, where the size of external buffers must be fixed in advance. Buffering in memory takes less real computer time. It is true that external buffers could be designed so that the number of memory cycles taken to transfer data between buffer and memory would be the same as would be required to transfer data directly between unit and memory; but, with buffering in memory, the memory cycles are sandwiched between memory cycles taken by the computer, and, since the computer does not normally use every cycle, a significant fraction of the data-transfer cycles is "free" and does not delay the computation.

Perhaps the most significant gain is the more direct control that the program can exercise. When double buffering is used externally for greater efficiency, the input-output unit runs ahead (on reading) or behind (on writing) by one block with respect to the program. As a result, if an error or other exception condition occurs, recovery is more difficult. With buffering in memory, data in memory always correspond to the block currently being read or written, and the pipeline effect is avoided. Operator intervention can be simplified. Moreover, the programmer has the option of any buffering scheme he would care to use, including no buffering at all. When speed of program execution is not important, the simplicity of programming an unbuffered operation without overlap is very appealing. This need not mean that the computer is used inefficiently. Since many channels are available, more than one such program can be run concurrently so that the overlap occurs between operations belonging to different programs, instead of between different operations in the same program.

12.9. Interface

Input-output units, regardless of type, must be connected to their exchange channels in the same manner, electrically and mechanically, if the channels are to be identical in design. This connection has been called the *interface*. If a common connection technique is used, any mixture of input-output units can be attached to a computer, the array of units being determined by the needs of the specific installation. This is shown schematically in Fig. 12.3.

A further requirement of the interface is that it permit the connecting together of any two units that can logically operate together (Fig. 12.4). A tape unit and its control unit may be connected to the computer, via an exchange channel, or they may be connected to a card reader and its control for off-line card-to-tape conversion, or to a printer and its control for off-line tape-operated printing. The same card reader or printer could, in turn, have been connected to exchange channels for on-line operation.[1] Figure 12.4 also indicates two computers connected together via exchange channels and a phone line. There is no inherent master-slave relationship; either unit can initiate data transfer.

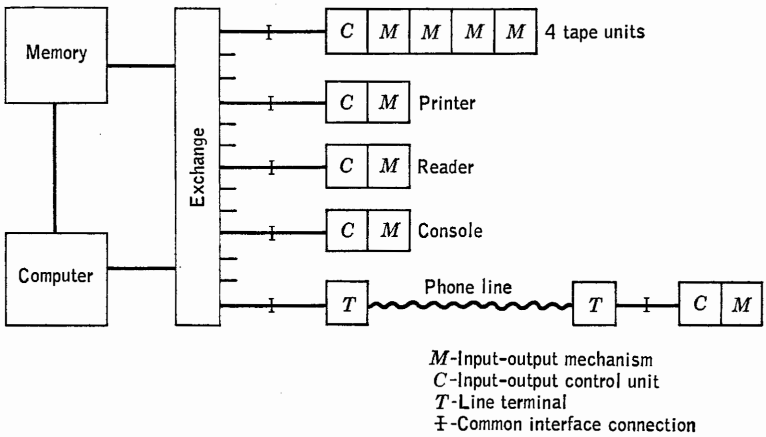


FIG. 12.3. Input-output connections to computer.

It is not possible to connect two tape units to copy data from one to the other; the absence of buffer storage in the tape-control unit prevents their synchronization. Nor does it make sense to connect a card punch to a printer. Also not shown is any direct connection between two exchange channels. Technical difficulties prevented this; it would have required an otherwise superfluous register in each channel. A junction box containing a register is needed to tie together the channels of physically adjacent computers.

[1] A somewhat similar technique was used in the IBM 702 and 705 systems to permit card readers, punches, and printers to be connected either on-line or off-line. This was done, however, by providing two different connections on the control unit, one for the computer and another for a tape unit. Also the approach was very much tape-oriented. The control units for the reader, punch, and printer each contained a complete set of tape control circuits. The present approach is based on a strict separation of functions.

The interface contains eighteen data lines (eight information-bit lines and a parity-bit line in each direction), a timing line, and several more lines corresponding to the instructions and indications referred to earlier in this chapter. As mentioned in connection with the CONTROL and LOCATE instructions, extensive use is made of addresses and codes transmitted over the data lines, instead of providing separate lines with more restricted meanings. Such generality provides assurance that improved or newly designed units can be connected to the same channels without changing the computer or its exchange.

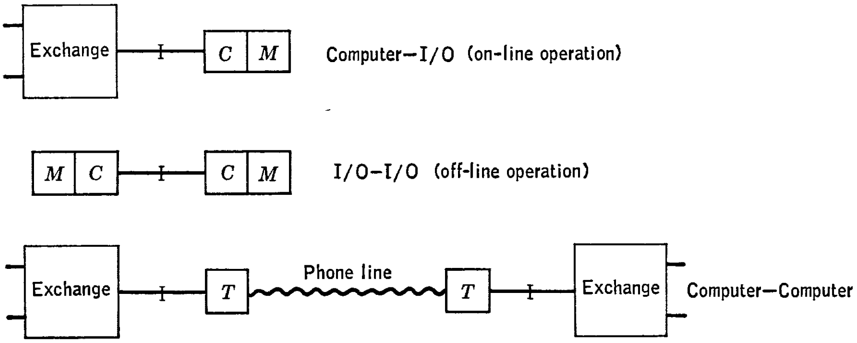

Exchange
C
M
Computer—I/O (on-line operation)
M
C
C
M
I/O—I/O (off-line operation)
Exchange
T
Phone line
T
Exchange
Computer—Computer

FIG. 12.4. Types of connections.

12.10. Operator Control of Input-Output Units

To achieve high performance, it is very desirable to require a minimum of operator intervention in the computer and in input-output units that are essentially automatic in operation. Operator intervention implies waits and errors, both of which serve to reduce system performance. Thus printers, card readers, and tape units have as few manual controls as possible; control is exercised entirely by the central stored program, with no plugboards or set-up switches on most of the external units. By contrast, typewriter keyboards and consoles, which have meaning only as they are manually operated, are equipped with a wealth of buttons and switches, but even those do not control computer functions, except as interpreted by a program.

Ignoring power on-off switches, all input-output units can be operated with just two buttons, labeled *start* and *stop* or with some equivalent names. *Start* places the previously loaded recording medium into operating position, checks all interlocks, turns on a *ready* condition, and sends a *channel signal* to the program. The unit is then under full computer control. *Stop* allows the operator to stop the device and turn off *ready;* the computer can then no longer operate the unit until *start* is pressed again. Thus *start* and *stop* provide an interlock between the operator and the computer by which the operator can exercise a minimum of necessary supervision. A separate *signal* button may be provided where an automatic *channel signal* after readying a unit is not desired.

Additional buttons are encouraged on individual units only when equivalent functions cannot be provided as well or better by the stored program. On some completely automatic units, such as disk files, even the *start-stop* pair of buttons is not needed.

Operating controls are to be clearly distinguished from the multitude of controls that may be needed by maintenance personnel. Maintenance controls are strictly separated from operating controls, and they are generally located where their use for normal operation can be discouraged.

Chapter 13

MULTIPROGRAMMING

by E. F. Codd, E. S. Lowry, E. McDonough, and C. A. Scalzi

13.1. Introduction

In recent years there has been a trend in computer design toward increased use of concurrent operation, with the prime aim of allowing more of the component units of a computer system to be kept in productive use more of the time. Two forms of concurrency have clearly emerged. The first, which we shall call *local concurrency*, consists in overlapping the execution of an instruction with that of one or more of its immediate neighbors in the instruction stream.

This form of concurrency was present in a very early machine, the IBM *Selective Sequence Electronic Calculator* (SSEC), which was capable of working on three neighboring instructions simultaneously. Such concurrency was later abandoned in the von Neumann–type machines, such as the IBM 701. Now that we have once again reached a stage in which the logical elements are much faster than the memories, the need for this type of concurrency has returned, and, in fact, the 7030 computer is capable of working on as many as eleven neighboring instructions simultaneously.

The second form, which we shall call *nonlocal concurrency*, provides for simultaneous execution of instructions which need not be neighbors in an instruction stream but which may belong to entirely separate and unrelated programs. It is this form of concurrency upon which we wish to focus attention in this chapter.

A computer system, in order to exhibit nonlocal concurrency, must possess a number of connected *facilities*, each capable of operating simultaneously (and, except for memory references, independently) on programs that need not be related to one another. A facility may be an

Note: The material in this chapter has previously been published by the same authors as: Multiprogramming Stretch: Feasibility Considerations, *Communs. ACM.*, vol. 2, no. 11, pp. 13–17, November, 1959.

input-output unit, an external storage unit, an arithmetic unit, a logic unit, or some assemblage of these units. In an extreme case each facility is a complete computer itself.

The 7030 is a multiple-facility system. The following facilities are capable of simultaneous operation on programs that need not be related:

1. One (or more) central processing units
2. Each input-output channel of the exchange
3. Each disk-storage access mechanism
4. The read-write channel of the high-speed disk synchronizer

(The several memory units in a 7030 system are not considered separate facilities, even though they may work momentarily on unrelated programs, because they behave, on the average, as a single larger unit of higher speed.)

The multiple-facility computing system bears a close resemblance to a job shop, although the analogy can be taken too far. Just as the jobs to be processed in a job shop are split up into tasks that can be handled concurrently by the facilities available, so programs can be subdivided into such tasks. At any instant the tasks being executed simultaneously may belong all to one program or to different programs. The procedure of running concurrently tasks that belong to different (perhaps totally unrelated) programs has a number of objectives: (1) to achieve a more balanced loading of the facilities than would be possible if all the tasks belonged to a single program; (2) to achieve a specified real-time response in a situation in which messages, transactions, etc., are to be processed on-line; (3) to expedite and simplify debugging and certain types of problem solving by making it economically feasible for the programmer to use a console for direct communication with and alteration of his program.

13.2. Multiprogramming Requirements

Several problems arise when concurrent execution is attempted of programs sharing a common memory. For example, it is almost certain that sooner or later, unless special measures are taken, one program will make an unwanted modification in another as a result of a programmer's blunder. Then again, when an unexpected event occurs, the handling of it is not merely a matter of deciding whether it was due to machine malfunction, programming blunder, or operator error. It becomes necessary to know which of the several programs may have been adversely affected and which (if any) was responsible.

Such problems make it desirable for a multiprogramming system, if it is to be generally accepted and used, to satisfy the following six conditions:

1. *Independence of preparation.* The multiprogramming scheme should permit programs to be independently written and compiled. This is particularly important if the programs are not related to one another. The question of which programs are to be coexecuted should not be prejudged even at the compiling stage.

2. *Minimum information from programmer.* The programmer should not be *required* to provide any additional information about his program for it to be run successfully in the multiprogrammed mode. On the other hand, he should be permitted to supply extra information (such as expected execution time if run alone) to enable the multiprogramming system to run the program more economically than would be possible without this information.

3. *Maximum control by programmer.* It may be necessary in a multiprogramming scheme to place certain of the machine's features beyond the programmer's direct influence (for example, the *time clock* and the *interval timer* in the 7030). This reduction in direct control by the problem programmer must not only be held to an absolute minimum but must also result in no reduction in the effective logical power available to the programmer.

4. *Noninterference.* No program should be allowed to introduce error or undue delay into any other program. Causes of undue delay include a program that gets stuck in a loop and the failure of an operator to complete a requested manual operation within a reasonable time.

5. *Automatic supervision.* The multiprogramming scheme must assume the burden of the added operating complexity. Thus instructions for handling cards, tapes, and forms for printing should be given by the multiprogramming system. Similarly, machine malfunctions, programming errors, and operator mistakes should be reported by the multiprogramming system in a standard manner to the person responsible. Again, all routine scheduling should be handled automatically by the system in such a way that the supervisory staff can make coarse or fine adjustments at will. Further responsibilities of the system include accounting for the machine time consumed by each job and making any time studies required for purposes of operation or maintenance.

6. *Flexible allocation of space and time.* Allocation of space in core memory and disk storage, assignment of input-output units, and control of time-sharing should be based upon the needs of the programs being executed (and not upon some rigid subdivision of the machine).

To implement by built-in equipment all the logic required to satisfy the above six conditions would be far too cumbersome and expensive. Further, the methods used to meet certain of these requirements (particularly the automatic-scheduling requirement) must be able to be varied from user to user because of varying objectives.

On the other hand, too extensive a use of programmed logic in a multiprogramming scheme can easily prove self-defeating, because the time taken by the machine to execute the multiprogramming program may offset the gain from concurrent execution of the problem programs. However, the raw speed and logical dexterity of the 7030 are such that it is practical to employ quite sophisticated programmed logic.

In the 7030, therefore, the conditions for effective multiprogramming are met by a carefully balanced combination of built-in and programmed logic.

13.3. 7030 Features That Assist Multiprogramming

First, let us consider four major features, built into the 7030 equipment, that facilitate multiprogramming: (1) the program-interrupt system, (2) the interpretive console, (3) the address-monitoring scheme, and (4) the clocks.

Program-interrupt System

This system is described in some detail in Chap. 10. Briefly, the system permits interruption of a sequence of instructions whenever the following four conditions are all satisfied:

1. The interrupt system is enabled.
2. No futher activity is to take place on the current instruction.
3. An indicator bit is on.
4. The corresponding mask bit is on.

The indicators reflect a wide variety of machine and program conditions, which may be classified into the following six types:

1. Attention requests from input-output units, the *interval timer*, or any other central processing units that may be attached to the system
2. Data exceptions, such as data flags, zero divisors, or negative operands in square-root operations
3. Result exceptions, such as lost carries, partial fields, or floating-point exponents beyond certain ranges
4. Instruction exceptions, such as instructions that should not or cannot be completed or should signal when they are completed
5. Entries to interpretive routines
6. Machine malfunctions

When several problem programs are being executed concurrently, certain of these conditions are of private concern to the particular program that caused their occurrence. Other conditions, particularly types 1 and 6, are of general concern. Each of the indicators for conditions of private concern has a variable mask bit that allows the current program to choose between suppressing and accepting interruption for the respective condition. On the other hand, each of the indicators for conditions of general concern has a fixed mask bit, permanently set in the *on* position. This feature, combined with appropriate measures for controlling the disabling of the entire interrupt system, makes it virtually impossible for an interruption of general concern to be suppressed and lost.

Another aspect of the interrupt system that is of importance in multiprogramming is the *interrupt table.* When an interruption is taken, control is passed (without any change in the contents of the instruction counter) to one of the instructions in an interrupt table. The base address of this table is variable; so several such tables may exist simultaneously in memory (for example, one table for each problem program), but only one is active at a time. The relative location within the active table that supplies the interjected instruction is determined by the indicator (and hence by the particular condition) causing interruption.

Exploitation of this interrupt system depends upon programmed interrupt procedures. This aspect will be taken up when we deal with programmed logic for multiprogramming.

Interpretive Console

It has been customary in general-purpose computers to provide a single console at which an operator can exercise sweeping powers over the whole machine. For example, by merely depressing the *stop* button the operator has been able to bring the entire activity of the machine to a halt. The normal requirement in multiprogramming, on the other hand, is to communicate with a particular program and at the same time allow all other programs to proceed. Pursuing the same example, we now desire to stop a program rather than stop the machine.

For this reason and also because it is required that several consoles with different functions be concurrently operable, the operator's console of the 7030 is not directly connected to the central processing unit. Instead, it is treated as an input-output device. Its switches represent so many bits of input and its lights so many bits of output. No fixed meaning is attached to either. By means of a console-defining routine one can attach whatever meaning one pleases to these switches and lights.

Address Monitoring

Each reference by the central processing unit to memory is checked to see whether the effective address falls either within a certain fixed area or within a second variable area. If the effective address falls within one of these two areas, which are to be *protected,* the reference is suppressed and an interruption occurs. The boundaries of the variable area are specified by two addresses (the *upper* and *lower boundaries*) stored within the fixed area. These address boundaries can be changed only if the interrupt system is disabled.

This monitoring scheme allows any number of programs sharing memory to be protected from one another effectively. At any instant, the central processing unit is servicing only one program, logically speaking. Suppose this is a problem program P. The address boundaries are set so that P cannot make reference outside its assigned area. Before any other problem program Q acquires the central processing unit, the address boundaries are changed to values that will prevent Q from making reference outside the area assigned to Q. The task of changing address boundaries is one of the programmed functions of the multiprogramming system.

Clocks

There are two clocks in the 7030 that can be used by programs. The first, referred to as the *time clock,* is a 36-bit binary counter which is automatically *incremented* by unity about once every millisecond. This clock can be read by a program under certain conditions but cannot be changed by a program. It is intended for measuring and identifying purposes, particularly in accounting for machine use, logging events of special interest, and identifying output.

The second clock, referred to as the *interval timer,* is a 19-bit binary counter which is automatically *decremented* by unity about once every millisecond. Under certain conditions the interval timer may not only be consulted but may also be set to any desired value by a program. Whenever the interval-timer reading reaches zero, an interruption occurs (if the interrupt system is enabled). The main purpose of this device is to provide a means for imposing time limits without requiring programmed clock-watching, that is, without frequent inspection of the time clock.

There are several other features in the 7030 that facilitate multiprogramming. For example, the autonomous operation of the exchange (Chap. 16) considerably reduces the frequency of input-output interruptions to the program.

13.4. Programmed Logic

Now we turn our attention to the programmed logic and discuss how the built-in logic may be exploited by programming techniques in order to meet the six requirements for acceptable multiprogramming. Three tools are at our disposal: (1) the supervisory program, (2) the compiler, and (3) the source language.

The supervisory program is assumed to be present in the machine whenever multiprogramming is being attempted. To the supervisory program is assigned the job of allocating space and time to problem programs.

Allocation of space includes determining which areas of memory and disk storage and which input-output units are to be assigned to each of the programs. The space requirements (including the required number of input-output units of each type) are produced by the compiler as a vector whose components are quantities dependent in a simple way upon one or more parameters which may change from run to run. Any space requirements depending on parameters are evaluated at loading time when the particular values of the run parameters are made available.

The supervisory program uses the precise knowledge it has of the space requirements of a problem program together with any information it may have regarding its expected execution time and activity pattern to determine the most opportune moment to bring that program into the execution phase. It is not until the decision to execute is made that specific assignments of memory space, disk space, and input-output units are put into effect. By postponing space allocation until the last minute, the supervisory program maintains a more flexible position and is thus able to cope more effectively with the many eventualities and emergencies that beset computing installations, no matter how well managed they are.

Allocation of time includes not only determining when a loaded program should be put into the execution phase but also handling queues of requests for facilities from the various programs being concurrently executed. The fact that both pre-execution queuing and in-execution queuing are handled by programming rather than by special hardware results in a high degree of flexibility. Thus, at any time, the supervisory program is able to change the queue discipline in use on any shared facility and so cope more effectively with the various types of space and time bottlenecks that may arise. On interruptible facilities, such as the central processing unit, which allow one program to be displaced by another, changes in queue discipline may be expected to have very considerable effect upon the individual and collective progress of the programs being coexecuted.

These allocating powers of the supervisory program have several implications. Most important of these is that the compiler must produce a fully relocatable program—relocatable in memory and in disk storage, and with no dependence on a specific assignment of input-output units. A further consequence is that the supervisory program is responsible for all loading, dumping, restoring, and unloading activities and will supply the operator with complete instructions regarding the handling of cards, tapes, and forms.

In order to meet the requirements (Sec. 13.2) of independent preparation of problem programs and noninterference with one another, it is necessary to assign the following functions to the supervisory program:

1. Direct control of the enabled-disabled status of the interrupt system
2. Complete control of the protection system and clocks
3. Transformation of input-output requests expressed in terms of symbolic file addresses into absolute input-output instructions (a one-to-many transformation), followed by issuing of these instructions in accordance with the queue disciplines currently in effect
4. Initial and, in some cases, complete handling of interruptions from input-output units and other central processing units

By convention, whenever a problem program is being serviced by the central processing unit, the interrupt system is enabled; when the supervisory program is being serviced, either the enabled or the disabled status may be invoked according to need. Adherence to this convention is assisted by the compiler, which does both of the following:

1. Refrains from generating in problem programs the instruction BRANCH DISABLED (an instruction which completely disables the interrupt system)
2. If it encounters this instructon in the source language itself, substitutes a partial *disable* (a pseudo instruction) in its place, flagging it as a possible error

So long as the interrupt system is enabled, the protection system is effective. Problem programs are therefore readily prevented from making reference to the areas occupied by other programs (including the supervisory program itself). They are further prevented from gaining direct access to the address boundaries, the interrupt-table base address, and the clocks, all of which are contained in the permanently protected area.

For the sake of efficient use of the machine, one further demand is made of the programmer or compiler. When a point is reached in a problem program beyond which activity on the central processing unit cannot proceed until one or more input-output operations belonging to this program (or some related program) are completed, then control must be passed to the supervisory program so that lower-priority programs may be serviced.

It is important to observe that the programmer is *not* required to designate points in his program at which control may be taken away if some higher-priority program should need servicing. This would be an intolerable requirement when unrelated programs are to be concurrently executed, especially if all arithmetic and status registers at such points have to contain information of no further value.

It is the interrupt system (particularly as it pertains to input-output) that makes this requirement unnecessary. The interrupt system allows control to be snatched away at virtually any program step, and the supervisory program is quite capable of preserving all information necessary to allow the displaced program to be resumed correctly at some later time.

In removing certain features of the machine from the direct control of the problem programmer, we may appear to have lost sight of the requirement that the programmer retain a maximum degree of control (Sec. 13.2). However, for every such feature removed, a corresponding pseudo feature is introduced. Take, for example, the pseudo *disable* and pseudo *enable* instructions. When a problem program P issues a pseudo *disable,* the supervisory program effectively suspends all interruptions pertaining to P (by actually taking them and logging them internally) until P issues a pseudo *enable.* Meanwhile, the interruptions pertaining to other programs not in the pseudo-disabled state are permitted to affect the state of the queue for the central processing unit.

Another example of a pseudo feature is the *pseudo interval timer;* one of these is provided for each problem program. The supervisory program coordinates the resulting multiple uses of the built-in interval timer.

The need to detect the fact that a program has become stuck in a loop or that an operator has not responded to an instruction from the supervisory program is met by allotting a reasonable time limit for the activity in question. When this interval expires without receipt by the supervisory program of a completion signal, an *overdue* signal is sent to an appropriate console. The interval timer is, of course, used for this purpose, and expiration of the interval is indicated by the *time signal* interruption.

13.5. Concluding Remarks

We have attempted to show that the design of a computer may be influenced quite strongly by the desire to facilitate multiprogramming. Developing a complete multiprogramming system is a major undertaking of its own, but both the computer and the programming system benefit from coordinating the initial planning for both.

Since the purpose here is to describe the structure of the computer rather than that of its programming systems, we have not discussed such other considerations as the optimizing and queuing problems that arise[1]

[1] E. F. Codd, Multiprogram Scheduling, *Communs. ACM,* vol. 3, no. 6, pp. 347–350, June, 1960, and no. 7, pp. 413–418, July, 1960; H. Freeman, On the Information-handling Efficiency of a Digital Computer Program, *Trans. AIEE,* paper no. 60-970.

or the detailed specifications of a supervisory program and operating system. They would go beyond the scope of this book. We merely note here that an experimental multiprogramming system has been developed for the 7030 along the lines discussed above. A comprehensive set of trial runs of this system has demonstrated successfully the feasibility of multiprogramming of the 7030.

13.6. References

Some earlier publications relating to multiprogramming are listed below.

S. Gill, Parallel Programming, *The Computer Journal,* vol. 1, no. 1, pp. 2–10, April, 1958.

C. Strachey, Time Sharing in Large Fast Computers, "Information Processing," UNESCO (Paris), R. Oldenbourg (Munich), and Butterworths (London), 1960, pp. 336–341.

W. F. Schmitt and A. B. Tonik, Sympathetically Programmed Computers, *ibid.*, pp. 344–348.

J. Bosset, Sur certains aspects de la conception logique du Gamma 60, *ibid.*, pp. 348–353.

A. L. Leiner, W. A. Notz, J. L. Smith, and R. B. Marimont, Concurrently Operating Computer Systems, *ibid.*, pp. 353–361.

J. W. Forgie, The Lincoln TX-2 Input-Output System, *Proc. Western Joint Computer Conf.*, February, 1957, pp. 156–160.

Chapter 14
THE CENTRAL PROCESSING UNIT

by E. Bloch

14.1. Concurrent System Operation

Early in the design of the 7030 system it appeared that a factor-of-6 improvement in memory speed and a factor-of-10 improvement in basic circuit speed over existing technology were the best one could look forward to during the time of the project. Since the performance level desired was much higher than could be obtained from faster components alone, the design had to provide for concurrent operation of various parts of the system wherever possible.

The need for concurrent operation affects all levels of the system, from the over-all organization to the details of specific instructions. Major parts of the system (Fig. 14.1) can operate simultaneously:

1. The core memory consists of several units of 16,384 words, operating on a 2.1-μsec read-write cycle. Each unit is self-contained and has its own clock, addressing circuits, data registers, and checking circuits. A typical 7030 system may have six memory units. To achieve a high degree of overlap, addresses are interleaved. The first four units share a block of 65,536 addresses, so that four consecutive word addresses lie in four different memory units. The next two units share a block of 32,768 addresses with two-way interleaving. If the four-unit block is assigned primarily to data and the two-unit block primarily to instructions, it is possible to achieve rates of up to one full data word and one half-word instruction every $\frac{1}{2}$ μsec. (Note that segregating data and instructions may help to increase speed, but it is not a necessary step, since the memory is logically homogeneous.)

2. The simultaneously operating input-output units are linked with the memories and the computer through the exchange, which, after receiv-

Note: The material in Chap. 14 has been adapted from E. Bloch, The Engineering Design of the Stretch Computer, *Proc. Eastern Joint Computer Conf.*, no. 16, pp. 48–58, December, 1959.

ing an instruction from the computer, coordinates the starting of the input-output equipment, the checking and error correction of the information, the arrangement of the information into memory words, and the fetching and storing of the information from and to memory. All these functions are executed independently of the computer. The high-speed disk units are controlled by the disk-synchronizer unit, which is similar in function to the exchange but is capable of much higher data rates. Memory cycles needed by the exchange and disk synchronizer are interleaved with those required by the computer.

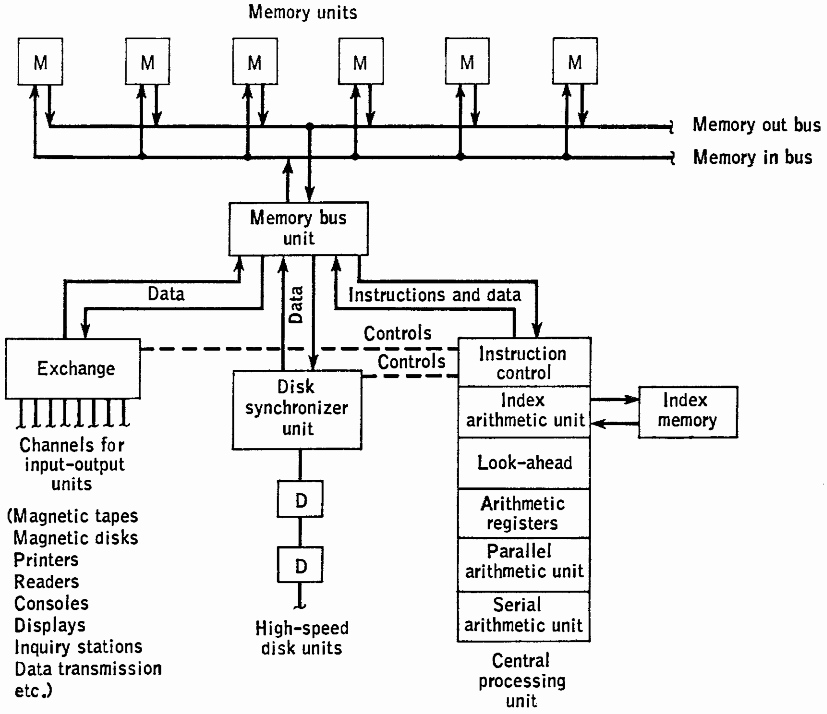


FIG. 14.1. 7030 system.

3. The central processing unit (CPU) processes and executes instructions with a high degree of overlap of internal functions.

The concurrent operation of various parts of the system was examined in Chap. 13 from the point of view of the user and programmer. In the present chapter we shall see how the need for effective concurrent operation has pervaded the design of the system and in particular of the central processing unit.

14.2. Concurrency within the Central Processing Unit

Most earlier computers have a sequential flow of instructions, as shown in Fig. 14.2. In turn, an instruction is fetched, the operand address is updated (by indexing or indirect addressing), the operand is fetched, and the instruction is executed. In some computers the instruction execution may be overlapped with fetching of the next instruction.

Compare this with the high degree of overlapping in the 7030 (Fig. 14.3). Two instruction words, which often represent four half-word instructions, and the operands for four more instructions can be fetched simultaneously. At the same time two more instructions can be updated and another executed. After completing its current stage of processing, each instruction advances to the next stage as soon as register space is available. Because the duration of each stage and the execution time of an instruction are variable, the process is not a cyclic one, and the actual number of instructions in process varies continually.

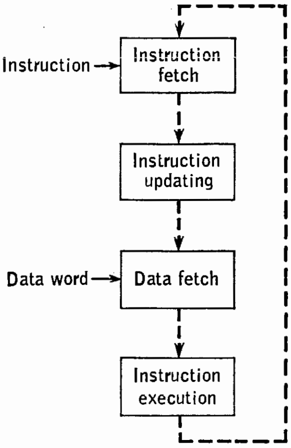


FIG. 14.2. Sequential operation.

All the units of the computer are loosely coupled together, each one controlled by its own clock system, which in turn is synchronized by a master oscillator. As may be expected, this multiplexing of the units of the computer results in a large number of registers and adders. In all, the computer has 3,000 register positions and about 450 adder positions.

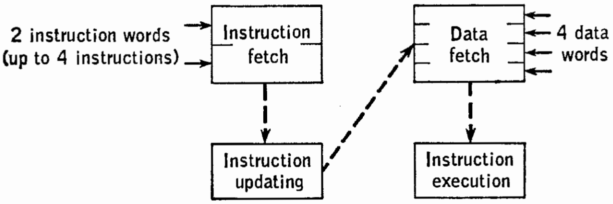


FIG. 14.3. Overlapped operation in 7030.

Despite the multiplexing and simultaneous operations of successive instructions, the result is always made to appear as if internal operation were sequential. This requires extensive interlock facilities.

14.3. Data Flow

The data flow through the computer is shown in Fig. 14.4. It is comparable to a pipeline which, once filled, has a large output rate no matter what its length. The same is true here. Once the flow is started, the execution rate of the instructions is high in spite of the large number of stages through which they progress.

The *memory bus unit* is the communication link between the memories on one side and the exchange, disk synchronizer, and CPU on the other.

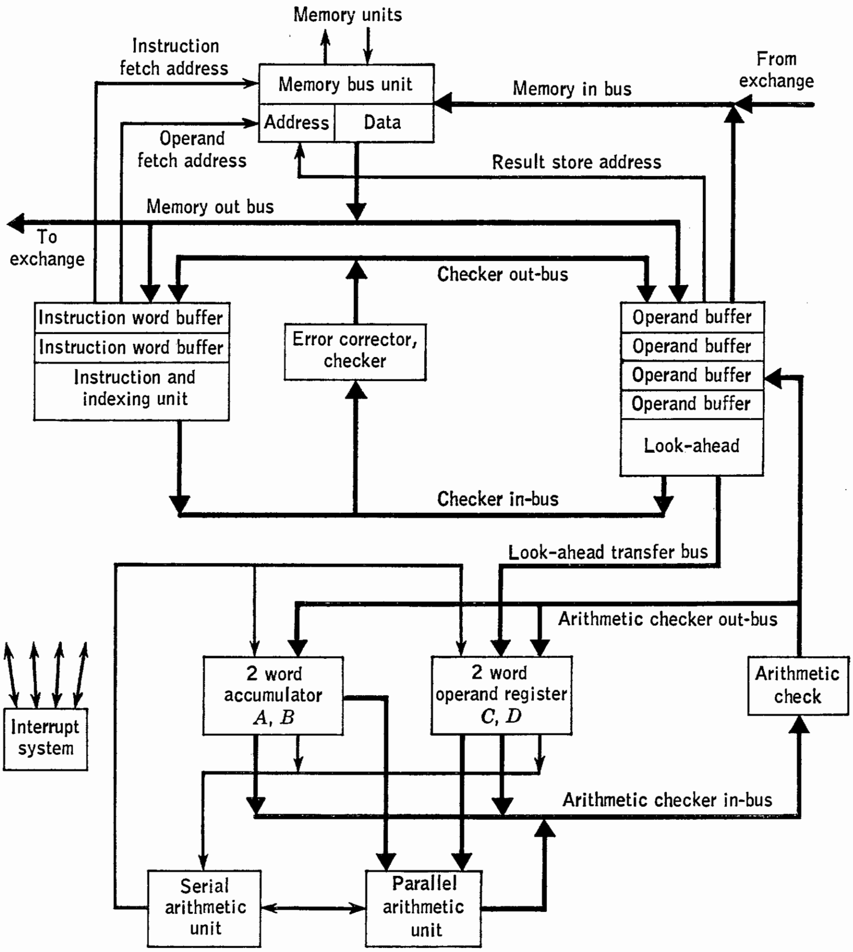


FIG. 14.4. Computer units and bus system.

The memory bus unit monitors the requests for storing in or fetching from memory and sets up a priority scheme. Since input-output units cannot hold up their requests, the exchange and disk synchronizer will get highest priority, followed by the CPU. In the CPU the operand-fetch mechanism, the *look-ahead unit,* has priority over the instruction-fetch mechanism. Altogether the memory bus unit receives requests from and assigns priority to eight different channels.

Since access to a memory unit can be independently initiated from several sources, a *busy* condition can exist. Here again, the memory bus tests for the busy conditions and delays the requesting unit until the desired memory unit is ready. The return address identifying the requesting unit is remembered, and the information is forwarded when it becomes available.

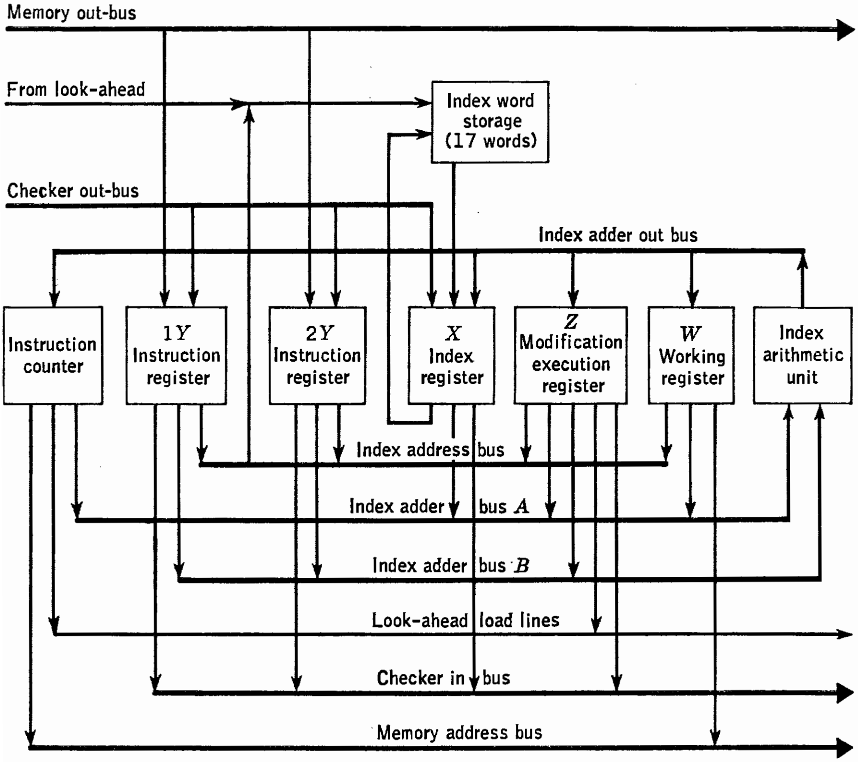


FIG. 14.5. Instruction unit.

Requests for stores and fetches can be processed at a rate of one every 0.3 μsec. If no busy or priority conditions exist, the time to return the word to the requesting unit is 1.6 μsec, a direct function of the memory read-out time.

The *instruction unit* (Fig. 14.5) is a computer all by itself.[1] It has its own instructions to execute, its own small memory for index-word storage,

[1] R. T. Blosk, The Instruction Unit of the Stretch Computer, *Proc. Eastern Joint Computer Conf.*, no. 18, pp. 299–324, December, 1960.

and its own arithmetic unit. As many as six instructions can be at various stages of progress in the instruction unit.

The instruction unit fetches the instruction words from memory, steps the instruction counter, and indexes all instructions. After a preliminary decoding of the instruction class, it recognizes and executes indexing and branching instructions; for other classes of instructions it initiates data fetches and passes the partially decoded instructions on to the look-ahead.

At the time the instruction unit encounters a conditional *branch* instruction, the condition may not be in its final state because other operations currently in progress may still affect it. To keep things moving, the assumption is made here that the *branch* condition will *not* be met, and the next instruction in sequence is fetched. This assumption and the availability of two full-word buffer registers keep the rate of flow of instructions to the computer high most of the time. When the assumption proves wrong, the instruction unit must backtrack to the branch point and follow the new sequence instead. This takes time, of course.

Two instruction words can be in the registers at any one time. As soon as the instruction unit starts processing an instruction, it is removed from the buffer, thus making room for the next instruction.

The index-arithmetic unit and the index registers complete the instruction unit. It should be noted that the index registers have been made an integral part of the instruction unit, so as to permit fast access to an index word without long transmission lines. There are sixteen index words available to the programmer, of which fifteen can be used for automatic address modification. The index registers are contained in a small memory unit made of multiaperture cores, which is operated in a nondestructive-read fashion where reading is much faster than writing. This permits fast operation most of the time, when an index word is referred to without modification. Additional time is needed only when modification is involved.

After it has been processed through the instruction unit, the updated (indexed) instruction enters one of four levels of the look-ahead unit (Fig. 14.4). Besides the necessary information from the instruction, the associated instruction-counter value and certain tag information are stored in the same level of the look-ahead. The operand, already requested by the instruction unit, will enter this level directly and will be checked and error-corrected while awaiting transfer to the arithmetic unit. The look-ahead unit also performs all storing operations.

The operating principles of the look-ahead unit, together with the sequencing of functions and the interlocking required to prevent out-of-sequence execution of instructions, are covered in Chap. 15.

The two-part arithmetic unit described below is a slave to the look-ahead, receiving from it not only operands and instruction codes but also the signal to start execution. The arithmetic unit signals to the look-ahead the termination of an operation and, in the case of *store* operations, places into the look-ahead the result word for transfer to the proper memory location.

14.4. Arithmetic Unit

The design of the main arithmetic unit was established along similar lines. Every attempt was made to speed up the execution of arithmetical operations by multiplexing techniques.

The arithmetic unit consists of a parallel unit for floating-point operations and a serial unit for variable-field-length operations. The two units use the same arithmetic registers, namely a double-length accumulator of 128 bits (the left part being called register A and the right part register B), and a double-length operand register of 128 bits (C and D). The same arithmetic registers are used because the program may at any time switch from floating-point to variable-field-length operation, or vice versa. The result that is obtained by a floating-point operation can serve as the starting operand for a variable-field-length operation, or vice versa.

Operations on the floating-point fraction and also variable-field-length binary *multiply* and *divide* operations are performed by the parallel unit. Floating-point exponent operations, variable-field-length (binary or decimal) *add* operations, and logical-connective operations are executed by the serial unit. The square-root operation and the binary-decimal conversion algorithm are executed in unison by both units. Decimal multiplication and division are not built in because they can be done faster —and quite conveniently—by a short subroutine using radix conversion and the fast binary arithmetic.

Salient features of the two units will now be described.

Serial Arithmetic Unit

The serial arithmetic unit (Fig. 14.6) contains two symmetrical portions, one for the accumulator (AB) registers and one for the operand (CD) registers, feeding into a common binary adder or logical-connective unit and branching out again into two similar circuits for returning the result to either pair of registers. A two-level switch matrix is used to extract 8 adjacent bits from any of 128 possible register positions, together with a by-pass for 8 bits which are to remain undisturbed. True-complement (inversion) circuits, both before and after the adder, take care of subtraction. A decimal-correction circuit is switched into the data path when decimal addition or subtraction is specified. The result is returned via another two-level switch matrix to the selected register positions. All other register positions remain undisturbed.

A single pair of bytes is extracted, arithmetic or logic performed, and the result returned to the registers in one clock cycle of 0.6 μsec. Longer fields are processed by repeatedly stepping the counters that control the switch matrixes. The operations are checked by parity checks on the switch matrixes and by use of duplicate arithmetic and logic units.

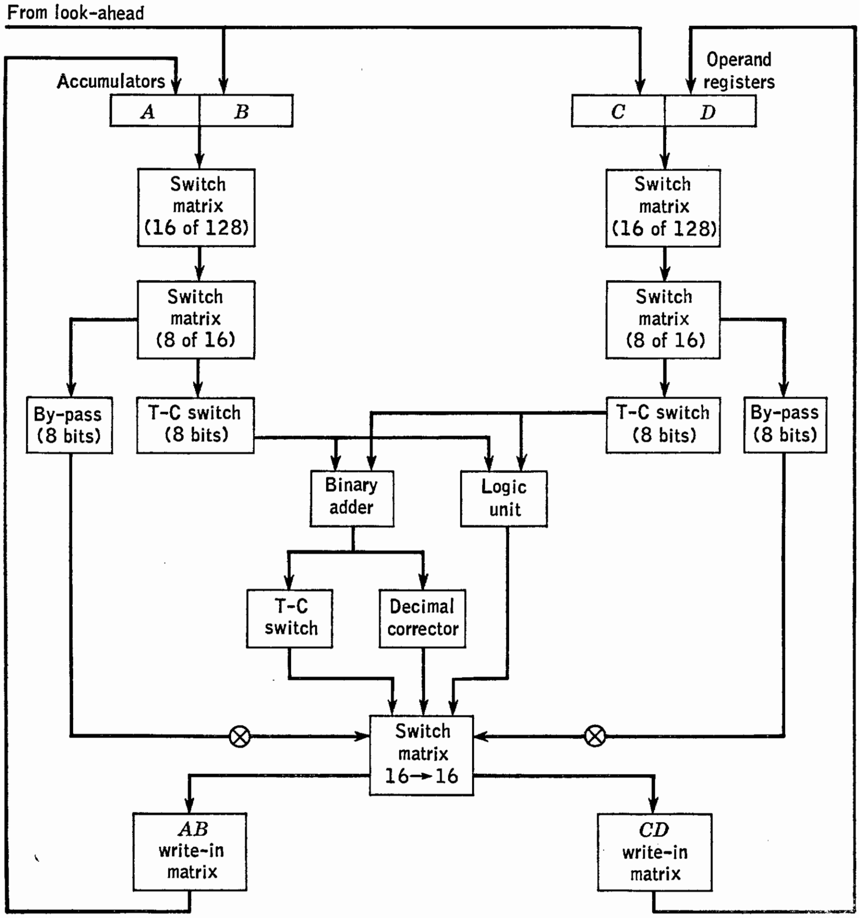


FIG. 14.6. Serial arithmetic unit. T-C: true-complement.

Parallel Arithmetic Unit

The parallel arithmetic unit (Fig. 14.7) is designed to execute binary floating-point operations at very high speed.[1] Since both single-length

[1] O. L. MacSorley, High Speed Arithmetic in Binary Computers, *Proc. IRE,* vol. 49, no. 1, pp. 67–91, January, 1961.

(48-bit fraction) and double-length (96-bit fraction) arithmetic are performed, the shifter and adder extend to 96 bits. This makes it possible to have almost the same speed for single- and double-length arithmetic. The adder is of a carry-propagation type with look-ahead over 4 bits at a time, to reduce the delay that normally results in a ripple-carry adder. This carry look-ahead results in a delay time of 0.15 μsec for 96-bit additions. Subtractions are carried out in 1s complement form with automatic end-around carry, but the result is always converted to absolute-value form with separate sign.

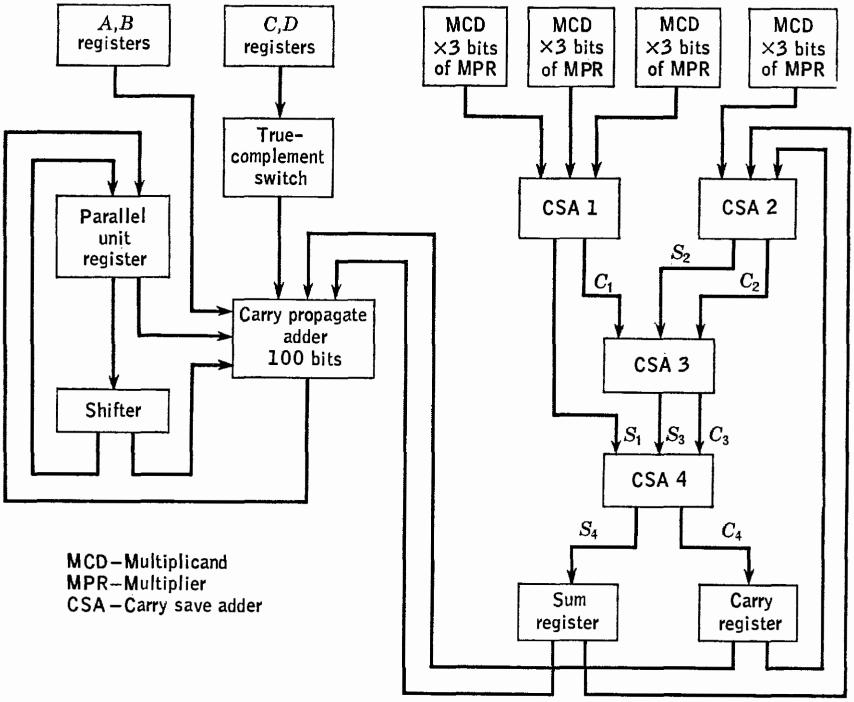


FIG. 14.7. Parallel arithmetic unit.

The shifter is capable of shifting up to four positions to the right and up to six positions to the left at one time. This shifter arrangement does the most common shifting operations in one step. For longer shifts the operation is repeated automatically.

To expedite the execution of the *multiply* instructions, 12 bits of the multiplier are handled in one cycle. This is accomplished by breaking the 12 bits into groups of 3 bits. The action is from right to left and consists in decoding each group of 3 bits. By observing the lowest-order bit of the next higher group, a decision is made as to what multiple of the multiplicand must be added to the partial product. Only even multiples of the multiplicand are available, and subtraction or addition of the multiples can result. The example of Fig. 14.8 will elaborate this point.

The four groups of multiplicand multiples and the partial product of the previous cycle are now fed into *carry save adders,* which operate according to the rules:

Sum $S = A \forall B \forall C$ where A, B, and C are either input or in-carry bits

Out-carry $C' = AB \vee AC \vee BC$

There are four of these adders, two in parallel followed by two more in series (Fig. 14.7). The output of carry-save adder 4 then results in a double-rank partial product, the product sum and the product carry. For each cycle this is fed into carry-save adder 2, and, during the last cycle, into the *carry propagate adder* for accumulation of the carries.

Groups	$n+4$	$n+3$	$n+2$	$n+1$	n
Multiplier bits	xx0	011	110	101	010
Multiplicand (C) additions		$4C-C$	$6C$	$6C-C$	$2C$
Equivalent to:		$4C$	$6C-8C$	$6C$	$2C-8C$
Final decoding		$4C$	$-2C$	$6C$	$-6C$

FIG. 14.8. *Multiply* example.

Since no propagation of carries is required in the four cycles when multiplicand multiples are added, this operation is fast, and it is the main contributor to the short *multiply* time of the 7030.

The *divide* scheme is similar to the *multiply* scheme in that several multiples of the divisor, namely 1, $\frac{3}{2}$, and $\frac{3}{4}$ times the divisor, are generated to reduce the number of *add* or *subtract* cycles needed to generate the quotient. It has a further similarity to another well-known *multiply* scheme, in that strings of consecutive 1s or 0s in the partial remainder are skipped, requiring only one *add* cycle for each string. The net effect is that the number of cycles is reduced, on the average, by a factor of 3.5 as compared with nonrestoring division.[1]

[1] C. V. Freiman, Statistical Analysis of Certain Binary Division Techniques, *Proc. IRE,* vol. 49, no. 1, pp. 91–103, January, 1961.

The power of this method may be illustrated by an example done three different ways. Let us assume the following normalized values:

Dividend		0.1 0 1 0 0 0 0 0	
Divisor	*DR*	0.1 1 0 0 0 1 1 0	(true form)
	DR′	1.0 0 1 1 1 0 1 0	(2s complement form)
	¾ *DR*	0.1 0 0 1 0 1 0 0 1	(obtained by shifting and adding *DR* to itself)

The leftmost bit represents the sign (0 for + and 1 for −). For division only, subtraction in the parallel arithmetic unit is more easily accomplished by adding the 2s complement of the number, whereas the 1s complement is used for other operations, as noted before.

Shift-add cycle	*Sign*	*Quotient*	*Comments*
	0.1 0 1 0 0 0 0 0		Dividend
(1)	1.0 0 1 1 1 0 1 0		Add *DR′*
	1.1 1 0 1 1 0 1 0	**0**.	Sign minus (1), hence q = **0**
(2)	1.1 0 1 1 0 1 0 0		Shift partial remainder left
	0.1 1 0 0 0 1 1 0		Add *DR*
	0.0 1 1 1 1 0 1 0	0 **1**.	Sign plus (0), hence q = **1**
(3)	0.1 1 1 1 0 1 0 0		Shift
	1.0 0 1 1 1 0 1 0		Add *DR′*
	0.0 0 1 0 1 1 1 0	0 1 **1**.	q = **1**
(4)	0.0 1 0 1 1 1 0 0		Shift
	1.0 0 1 1 1 0 1 0		Add *DR′*
	1.1 0 0 1 0 1 1 0	0 1 1 **0**.	q = **0**
(5)	1.0 0 1 0 1 1 0 0		Shift
	0.1 1 0 0 0 1 1 0		Add *DR*
	1.1 1 1 1 0 0 1 0	0 1 1 0 **0**.	q = **0**
(6)	1.1 1 1 0 0 1 0 0		Shift
	0.1 1 0 0 0 1 1 0		Add *DR*
	0.1 0 1 0 1 0 1 0	0 1 1 0 0 **1**. . . .	q = **1**
(7)	1.0 1 0 1 0 1 0 0		Shift
	1.0 0 1 1 1 0 1 0		Add *DR′*
	0.1 0 0 0 1 1 1 0	0 1 1 0 0 1 **1**. .	q = **1**
(8)	1.0 0 0 1 1 1 0 0		Shift
	1.0 0 1 1 1 0 1 0		Add *DR′*
	0.0 1 0 1 0 1 1 0	0 1 1 0 0 1 1 **1**	q = **1**
			etc.

FIG. 14.9. Example of nonrestoring division. *DR*: divisor; *DR′*: 2s complement of divisor; q: current quotient bit. One quotient bit is generated for every shift-and-add cycle.

Nonrestoring division is demonstrated in Fig. 14.9. One quotient bit is generated for each shift-and-add cycle, so that 48 cycles would be needed for the 48-bit quotient of the 7030. At each step, if the partial remainder has a sign bit 0, *DR′* is added; if the sign is 1, *DR* is added. The resultant partial remainder is shifted once to the left, and the inverse of its sign bit becomes the new quotient bit.

Figure 14.10 shows that the division can be shortened greatly by skipping over adjacent 1s or 0s in the partial remainder. Another way of saying this is that the partial remainder is normalized by shifting out those extra high-order bits which can be replaced directly by corresponding quotient bits. If the remainder is positive (in true form), these bits are 0s; if it is negative (in complement form), these bits are 1s. It may be shown that the quotient bit to be inserted for each shift is the same as the bit shifted out. This technique requires both the dividend and the divisor to be normalized at the start, as was already true in the numbers chosen for the example.

Shift-add cycle	*Sign*	*Quotient*	*Comments*
(1)	0.1 0 1 0 0 0 0 0		Dividend
	1.0 0 1 1 1 0 1 0		Add *DR′*
	1.1 1 0 1 1 0 1 0	**0**..............	$q = \mathbf{0}$
	←——┘		
(2)	1.0 1 1 0 1 0 0 0	0 **1**...........	Shift over 1s, $q = \mathbf{1}$
	0.1 1 0 0 0 1 1 0		Add *DR*
	0.0 0 1 0 1 1 1 0	0 1 **1**..........	$q = \mathbf{1}$
	←——┘		
(3)	0.1 0 1 1 1 0 0 0	0 1 1 **0**........	Shift over 0s, $q = \mathbf{0}$
	1.0 0 1 1 1 0 1 0		Add *DR′*
	1.1 1 1 1 0 0 1 0	0 1 1 0 **0**......	$q = \mathbf{0}$
	←————┘		
(4)	1.0 0 1 0 0 0 0 0	0 1 1 0 0 **1 1 1**	Shift over 1s, $q = \mathbf{111}$
	0.1 1 0 0 0 1 1 0		Add *DR*
	1.1 1 1 0 0 1 1 0		etc.

FIG. 14.10. Divide example with skipping over 1s and 0s. On the average, 2.6 quotient bits are generated for every shift-and-add cycle.

The skipping technique is based on reasoning that a positive partial remainder with leading zero bits must be smaller than any divisor that has been normalized. Hence, subtracting the divisor is certain to result in an overdraw, and the correct quotient bit is 0. Thus the cycle can be avoided by simply shifting and inserting the correct quotient bit. A negative, complemented remainder with leading 1s presents the converse situation. Its absolute value is certain to be less than the divisor, so that adding the divisor is bound to leave a positive remainder with a quotient bit of 1, and it is not necessary to do the actual addition. Once the partial remainder is normalized, inspection of the leading bit is not enough to tell whether adding or subtracting the divisor is necessary or not, and a full cycle is taken at that point.

The *divide* scheme actually used in the 7030 is an extension of the skipping technique, obtained by inspecting more than 1 bit of the remainder and divisor. One of three multiples of the divisor is selected for each *add* cycle by looking up a table (Fig. 14.11) on the basis of 3 high-order bits

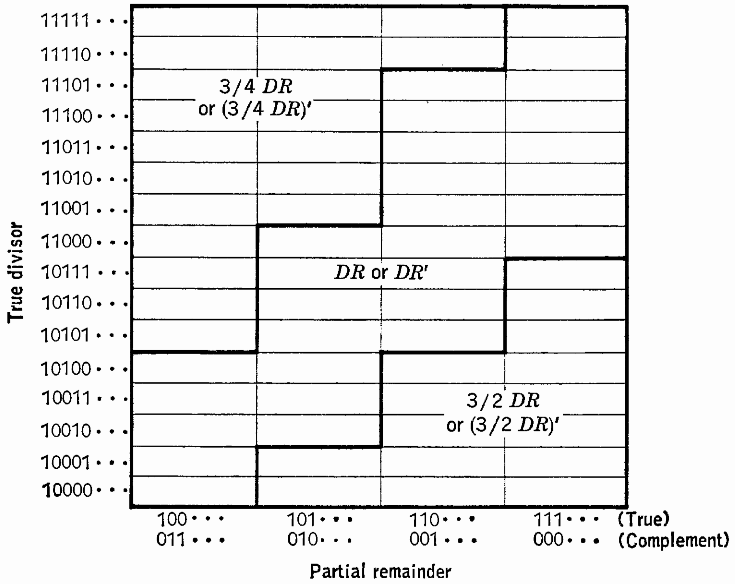


FIG. 14.11. Table for selecting divisor multiple. Select complement-divisor multiple if partial remainder is true. Select true-divisor multiple if partial remainder is a complement.

of the normalized previous remainder and 5 high-order bits of the normalized divisor. The addition is carried out, the new partial remainder is normalized, and the correct quotient bits are selected by the rules given in Fig. 14.12. The example with this technique in Fig. 14.13 shows a further reduction in the number of cycles.

The rules are considerably more complex than those in the previously cited techniques, but the reasoning is roughly as follows. After the partial remainder is normalized, the subsequent number of cycles can be further reduced by selecting a multiple of the divisor that is as close in magnitude to the remainder as possible, so that the magnitude of the new partial remainder—the difference of the two values—is as close to zero as

Selected divisor multiple	*Sign bit of* new *partial remainder*	*Quotient bits*
DR or *DR′*	0	1 0 0 0 0 0
	1	0 1 1 1 1 1
$\frac{3}{2}$ *DR*	0	0 1 0 0 0 0
	1	0 0 1 1 1 1
($\frac{3}{2}$ *DR*)′	0	1 1 0 0 0 0
	1	1 0 1 1 1 1
$\frac{3}{4}$ *DR*	0	1 0 1 0 0 0
	1	1 0 0 1 1 1
($\frac{3}{4}$ *DR*)′	0	0 1 1 0 0 0
	1	0 1 0 1 1 1

FIG. 14.12. Basic table for generating quotient bits. Additional rules: (1) Use only as many quotient bits, starting at the left, as the number of shifts needed to normalize the new partial remainder. (2) If only two shifts are needed for $\frac{3}{4}$*DR* or ($\frac{3}{4}$*DR*)′, invert the first quotient bit on the *next* cycle. (3) If more than six shifts are needed, take additional shift cycles and continue to generate 0 or 1 quotient bits, depending on remainder sign.

possible. As a result, there are more leading bits to be shifted out during normalization than before. Ideally, the divisor multiple is picked precisely so as to leave a remainder which, to single precision, is zero, so that the division is finished. For practical purposes, the selection was limited to a much cruder choice of one of three multiples: 1, $\frac{3}{2}$, and $\frac{3}{4}$ times the

Shift-add cycle	*Sign*	*Quotient*	*Comments*
(1)	0.1 0 1 0 0 0 0 0		Dividend
	1.0 0 1 1 1 0 1 0		Add *DR′*
	1.1 1 0 1 1 0 1 0 ←──┘	**0**.	q = **0**
(2)	1.0 1 1 0 1 0 0 0	0 **1**.	Shift over 1s, q = **1**
	0.1 0 0 1 0 1 0 0 1		Add $\frac{3}{4}$ *DR*
	1.1 1 1 1 1 1 1 0 0 1 ←──────────┘	0 1 **1 0 0 1 1 1**	q = **100111** etc.

FIG. 14.13. Example for divide method used in 7030. $\frac{3}{4}$ *DR* and $\frac{3}{2}$ *DR* (not shown) are used, as well as skipping over 1s and 0s. On the average, 3.5 quotient bits are generated each cycle.

divisor; 1 is used when the normalized remainder is close to the divisor in magnitude, $\frac{3}{2}$ when the remainder is much larger, and $\frac{3}{4}$ when it is much smaller.

The scheme always permits at least two shifts after each *add* cycle. As many as six shifts can be carried out in the same cycle as one addition; if more shifts are needed, extra cycles are used without addition. The limitation to a six-way shift is a matter of economy, but it only adds 5 per cent to the number of cycles that would be needed without this limitation.

The 7030 *divide* scheme is somewhat similar to a base-4 method described in the literature.[1] The base-4 method has a fixed shift of 2 bits per cycle, whereas the method described here allows from 2 to 6 bits of shift.

In floating-point *multiply* and *divide* operations, the arithmetic on the fractions is performed by the parallel arithmetic unit, as described above, while the serial arithmetic unit is executing the exponent arithmetic. Here, again, is a case where overlap and simultaneity of operation are used to special advantage.

14.5. Checking

The operation of the computer is thoroughly checked. An error-correction code is employed for transfers of data from memory. The code is attached to all words on the way into memory. Every time a word is fetched from memory, the code is checked. If a single error is indicated, a correction is made, the error is recorded on a maintenance output device, and computing continues.

Within the machine all arithmetical operations are checked either by parity, duplication, or a casting-out-three process. These checks are overlapped with the execution of the next instruction.

14.6. Component Count

Figure 14.14 gives the number of transistors used in the various sections of the machine. It becomes obvious that the floating-point unit and the instruction unit use the highest percentage of transistors. In the floating-point unit this is largely due to the extensive circuits for the speeded-up *multiply* and *divide* schemes. In the instruction unit most of the transistors are in the controls, because of the highly multiplexed operation.

[1] J. E. Robertson, A New Class of Digital Division Methods, *IRE Trans. on Electronic Computers*, vol. EC-7, no. 3, pp. 218–222, September, 1958.

Unit	*Number of transistors*	*Per cent of total*
Memory controls	10,500	6
Instruction unit:		
Data path	17,700	22
Controls	19,500	
Look-ahead unit:		
Data path	17,900	16
Controls	8,600	
Arithmetic registers	10,000	6
Serial arithmetic unit:		
Data path	10,000	11
Controls	8,700	
Parallel arithmetic unit:		
Data path	32,700	21
Controls	3,000	
Checking	24,500	14
Interrupt system	6,000	4
Total	169,100	100

Double Cards	4,025
Single Cards	18,747
Power	21 kw

FIG. 14.14. Component counts in the computer sections.

14.7. Performance

Figure 14.15 shows some examples of arithmetic speeds. Decimal *multiply* and *divide* instructions call for a subroutine; the times are not shown because they depend on the nature of the subroutine.

These figures give only a rough indication of the performance to be expected on a specific problem. Because of the large degree of overlap, one cannot correctly quote average times for individual operations that could be added together to give the total time for a sequence of such operations. It is possible for indexing and branching instructions, for example, to be completely overlapped with arithmetical operations, so that their effective time becomes zero. On the other hand, it is clear

	Time (μsec)		
	ADD	MULTIPLY	DIVIDE
Floating point	1.5	2.7	9.9
VFL binary (for 16-bit numbers)	3.6	10.5	16.2
VFL decimal (for 5-digit numbers)	5.4	Subroutine	Subroutine

FIG. 14.15. Examples of arithmetic speeds.

that a sequence consisting exclusively of indexing and branching instructions would take a significant amount of time.

The only valid way to time a specific program is either by measuring the time during actual execution or by simulating the intricate timing conditions of the 7030 dynamically on another computer.

14.8. Circuits

Having reviewed the CPU organization of the 7030, we shall briefly discuss the components, circuits, and packaging techniques used in the design.

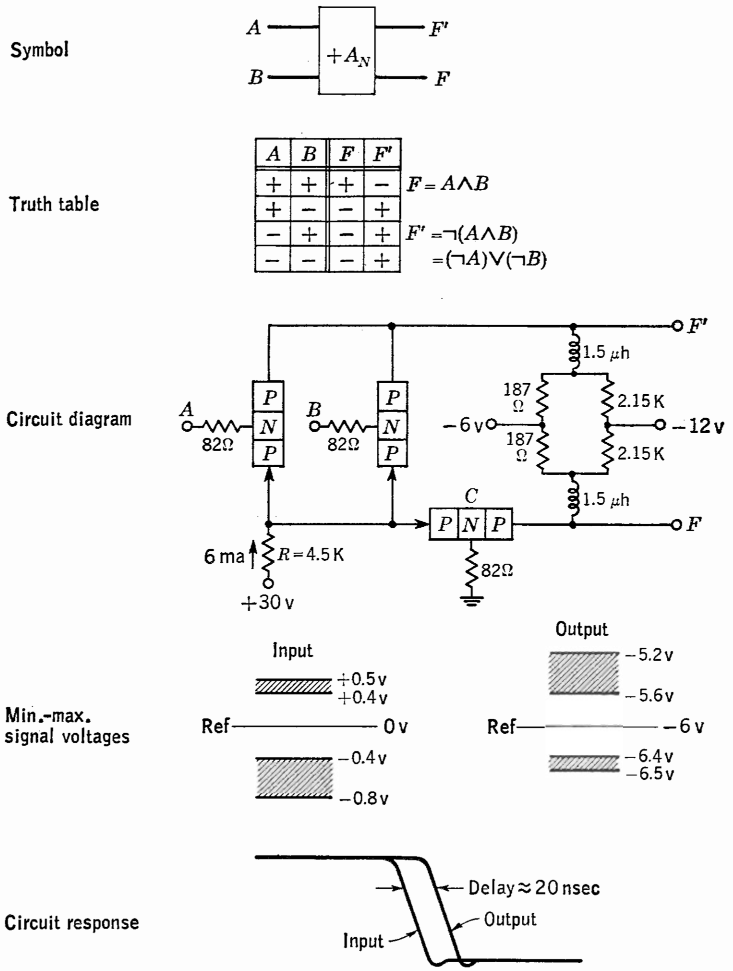
The basic circuit component is a high-speed drift transistor with a frequency cutoff of approximately 100 megacycles. To achieve high speed it is kept out of saturation at all times. The transistor exists in both a *PNP* and an *NPN* version. The main reason for using two versions is to avoid the problem of level translation caused by the 6-volt difference in potential between the base and the collector.

Figure 14.16 shows the *PNP* circuit. The inputs A and B operate at a reference voltage of 0 volt, which is established by the preceding circuit (not shown). If inputs A and B are both positive by 0.4 volt with respect to the reference voltage, their respective transistors cut off. This causes the emitter of transistor C to go positive with respect to its base and conduct a 6-ma current, flowing from the current source which is made up of the +30-volt supply and resistor R. As a result, output F goes positive by 0.4 volt with respect to its reference of −6 volts; at the same time output F' goes negative by 0.4 volt with respect to the reference.

When either of the inputs goes negative, its transistor becomes conducting. The emitter of transistor C goes negative and C is cut off. The result is that output F' goes positive and output F goes negative with respect to the reference.

The principle of this circuit is one of switching (or steering) a constant current either toward output F (C conducting) or toward output F' (A or B or both conducting). The *PNP* circuit provides both the logical function *and* and the function *not or*. Minimum and maximum signal swings are also shown in Fig. 14.16.

A dual circuit using the *NPN* transistor is shown in Fig. 14.17. The principle is the same, but the logical functions *or* and *not and* are obtained,



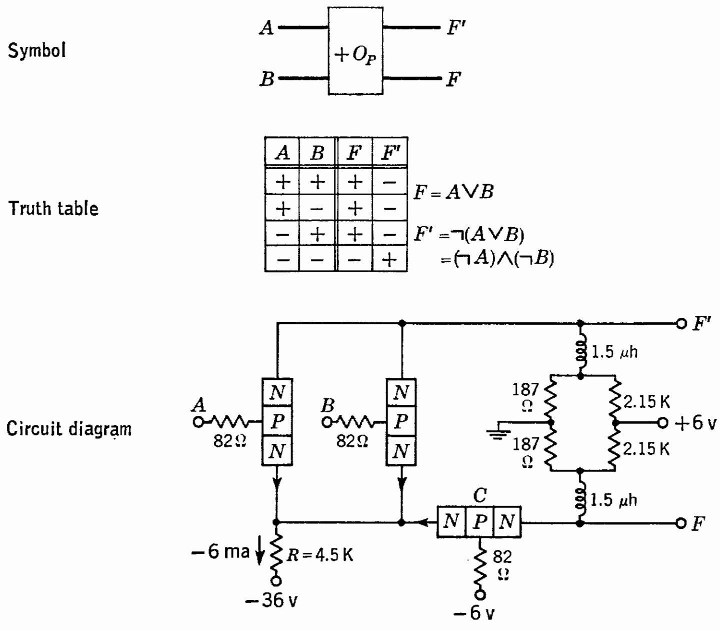
A	B	F	F'	
+	+	+	−	$F = A \wedge B$
+	−	−	+	
−	+	−	+	$F' = \neg(A \wedge B)$
−	−	−	+	$= (\neg A) \vee (\neg B)$

FIG. 14.16. Current-switching circuit, *PNP*. Symbols: ∧ *and*, ∨ *or*, ¬ *not*.

and the reference voltages are now −6 volts at the input and 0 volt at the output.

The circuits described so far are versatile enough so that they could be the only circuits used in the system. Because of the many data buses and registers, however, it was found useful to provide also a distributor function and an overriding function. This resulted in a circuit with a third voltage level which permitted great savings in space and transistors. Figure 14.18 shows the *PNP* version of the third-level circuit.

Without transistor X, transistors A and B in conjunction with the reference transistor C would work normally as a current-switching circuit, in this case an *and* circuit. When transistor X is added, with the stipulation that the down level of X be more negative than the lowest possible level of A or B, it becomes apparent that when X is negative the current will flow through that branch of the circuit in preference to branch F or F', regardless of input A and B. Therefore, the output of F and F' will be negative, provided input X is negative. Output G is the inverse of input X. If, however, X is positive, then the status of A and B will determine the function of F and F' implicitly. This demonstrates the overriding function of input X.



A	B	F	F'
+	+	+	−
+	−	+	−
−	+	+	−
−	−	−	+

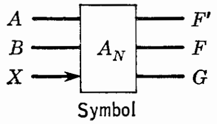
$F = A \vee B$

$F' = \neg(A \vee B)$
$= (\neg A) \wedge (\neg B)$

FIG. 14.17. Current-switching circuit, *NPN*.

Similarly, the *NPN* version, not shown, results in the *or* function at F if input X is negative and in a positive output at F and F', regardless of the status of A and B, if X is positive.

The speed of the circuits described so far depends on the number of inputs and the number of circuits driven from each output. The response of the circuits is anywhere between 12 and 25 nsec (nanoseconds, billionths of a second) per logical step, with 18 to 20 nsec average. The



A	B	X	F	F'	G
+	+	+	+	−	−
−	+	+	−	+	−
+	−	+	−	+	−
−	−	+	−	+	−
+	+	−	−	−	+
−	+	−	−	−	+
+	−	−	−	−	+
−	−	−	−	−	+

Truth table

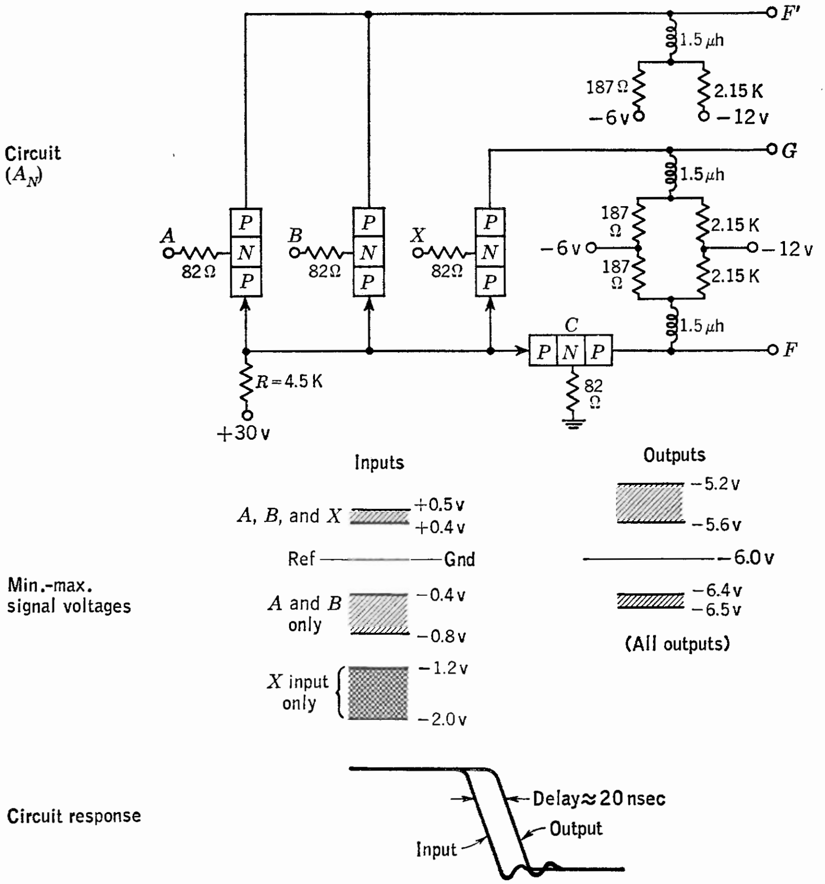
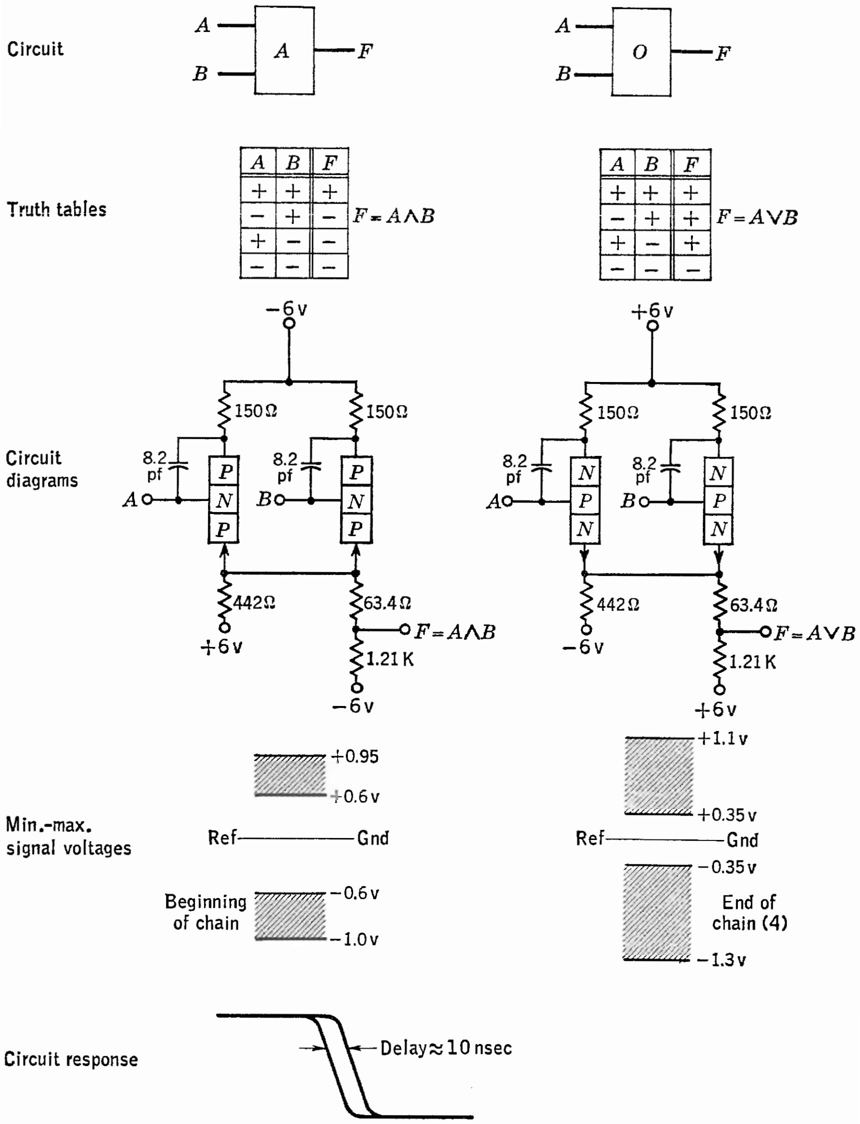


FIG. 14.18. Third-level circuit, *PNP*.



A	B	F
+	+	+
−	+	−
+	−	−
−	−	−

A	B	F
+	+	+
−	+	+
+	−	+
−	−	−

FIG. 14.19. Emitter-follower logic.

number of inputs allowable per circuit is eight. The maximum number of circuits driven is three. Additional circuits are needed to drive more than three bases, and, where current-switching circuits communicate over long lines, termination networks must be added to avoid reflections.

To improve the performance of the computer in certain critical places, emitter-follower logic is used, as shown in Fig. 14.19. These circuits have a gain less than 1, and they require, after a number of stages, the use of current-switching circuits as amplifiers and level setters. Both *and* and *or* circuits are available for both a ground-level (shown) and a −6-volt-level input (not shown). To change a ground-level circuit into a −6-volt-level circuit it is necessary to change the appropriate power supply levels. Because of variations in inputs and driven loads, the circuits must be designed to allow such variations over a wide range; this requires the feedback capacitor shown in the circuit, to maintain stability.

All functions needed in the computer can be implemented by the use of the aforementioned circuits, including the flip-flop function, which is obtained by connecting a *PNP* current-switch block and an *NPN* current-switch block together with proper feedback.

14.9. Packaging

The circuits described in the last section are packaged in two ways, as shown in Fig. 14.20. The smaller of the two printed circuit boards is

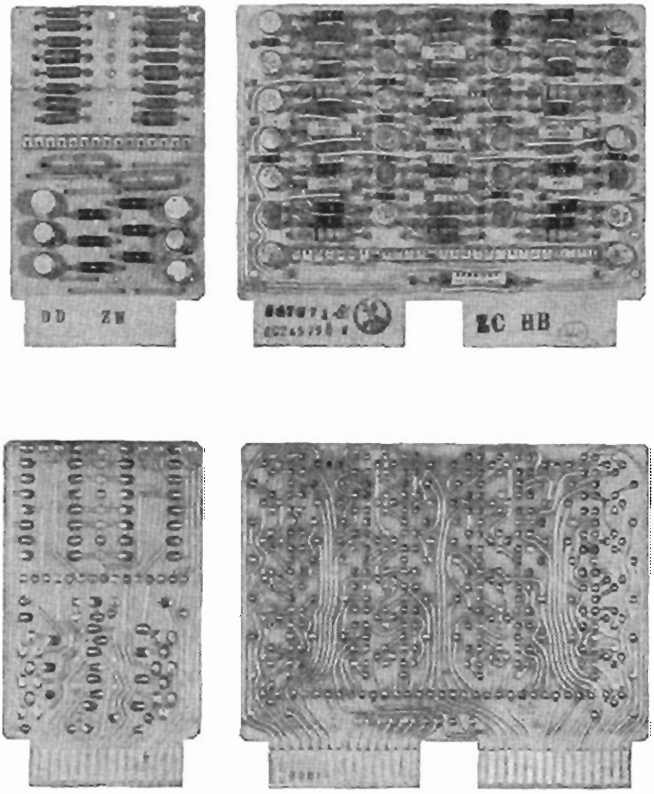


FIG. 14.20. Single and double circuit cards. Front and rear views.

called a *single card* and contains *and* or *or* circuits. The wiring is one-sided, and besides the components and transistors, a *rail* may be seen which permits the shorting or addition of certain loads depending on the use of the circuits. This rail has the effect of reducing the different types of circuit board needed in the machine. Twenty-four different boards are used, and, of these, two types reflect approximately 70 per cent of the total single-card population of the machine.

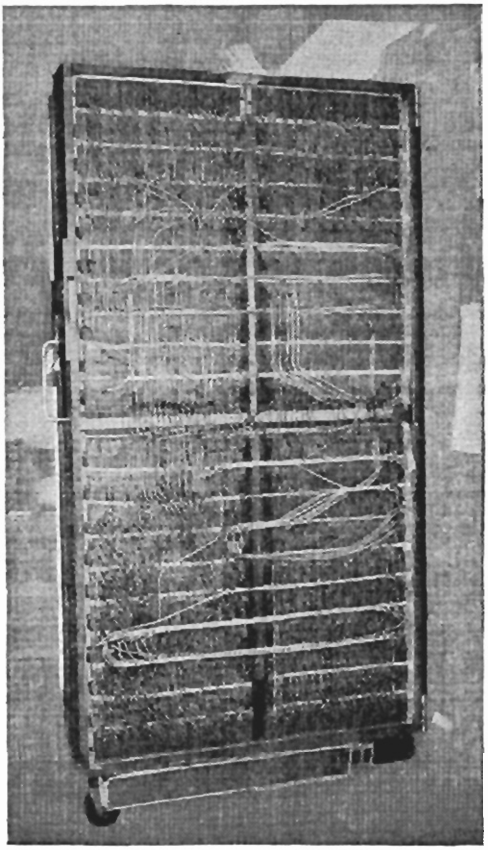


FIG. 14.21. The back panel.

Because of the large number of registers, adders, and shifters used in the computer, where the same functions are repeated many times, a second package was designed to be big enough to hold a complete function. This is the larger board shown in Fig. 14.20, called a *double card*. It has four times the capacity of a single card and has wiring on both sides of the board. Components are double-stacked. Again the rail is used to effect circuit variations for the different applications. Eighteen double-card types are used in the system. Approximately 4,000 double cards house 60 per cent of the transistors in the machine. The rest of the transistors are on approximately 18,000 single cards.

The cards, both single and double, are assembled in two gates, and two gates are assembled into a frame. Figure 14.21 shows the back-panel wiring of one gate, and Figs. 14.22 and 14.23 show the frame in closed and open position.

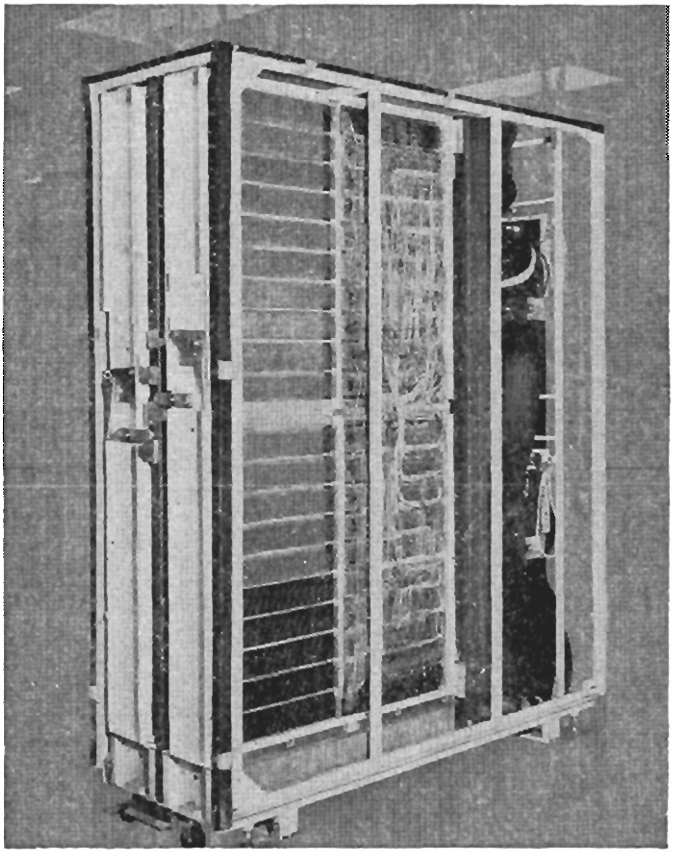


FIG. 14.22. The frame (closed).

To achieve high performance, special emphasis had to be placed on keeping the noise to a low level. This required the use of a ground plane which covers the whole back panel underneath the intercircuit wiring. In addition, the power-supply distribution system had to be of low impedance to avoid noise pick-up. For this reason a bus system consisting of laminated copper sheets is used to distribute the power to each row of card sockets. Wiring rules are that single-conductor wire is used to a maximum of 2 ft, twisted pair to a maximum of 3 ft, unterminated coaxial cable to a maximum of 5 ft, and terminated coaxial cable to a maximum of 100 ft. The whole back-panel construction, including the proper application of single wires, twisted pairs, or coaxial cable to minimize the noise on each circuit node, was laid out by means of a computer program.

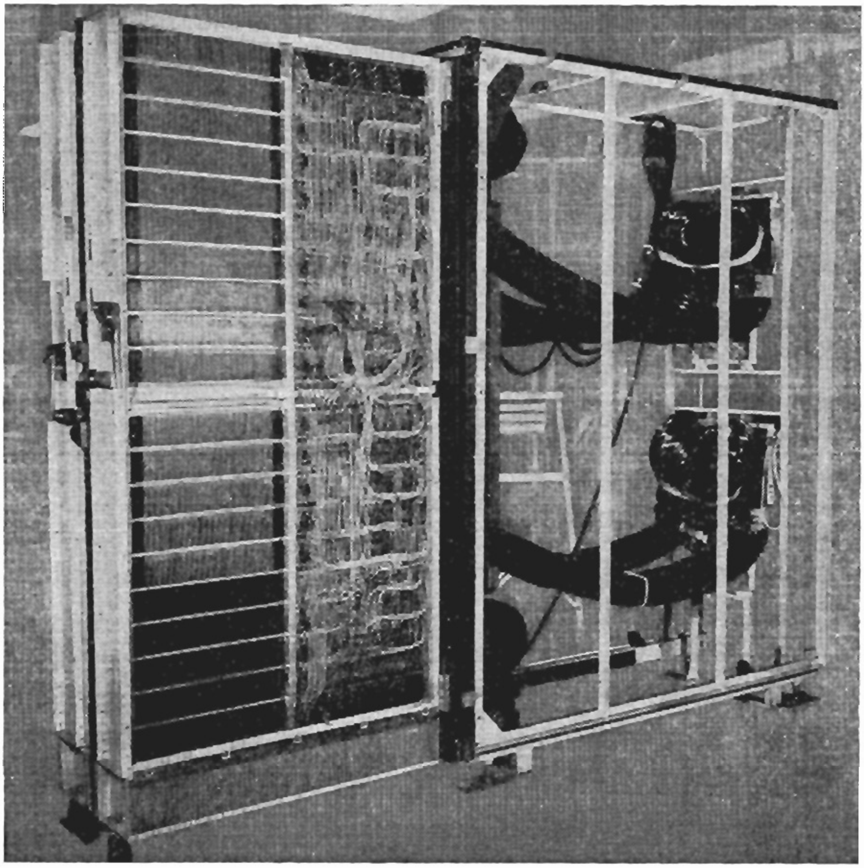


FIG. 14.23. The frame (extended).

With the high packing density made possible by the double cards, a single frame may consume as much power as 2 kw, the average consumption being around 1 kw. To reduce power distribution and regulation problems, a specially designed 2-kw power supply, using 400-cycle components for greater compactness, is mounted in each frame. The

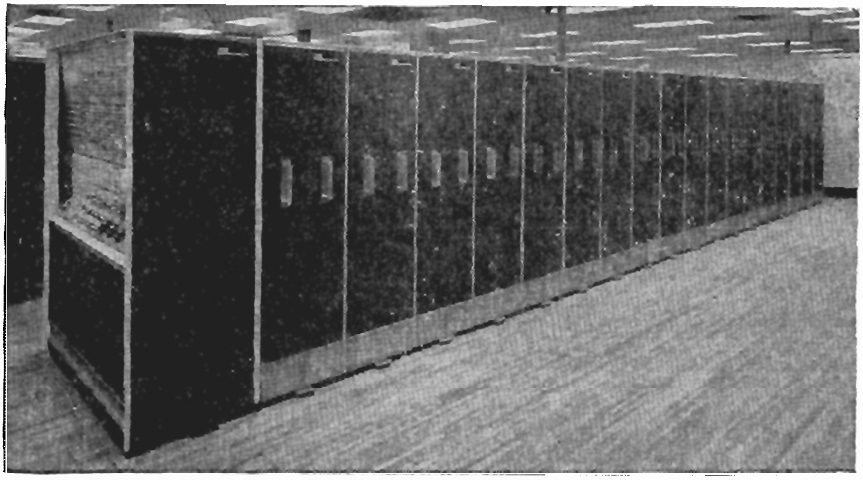


FIG. 14.24. The central processing unit.

supplies are fed from a regulated 400-cycle motor-generator set, which also serves the purpose of eliminating 60-cycle power-line variations.

The two gates of a frame are a sliding pair, with the power supply mounted on the sliding portion. All connecting wires between frames are coaxial cable arranged in layers to form a drape, which can follow the gate as it slides out of the frame.

Figure 14.24 shows eighteen of these frames tied together to form the entire central processing unit, as well as the CPU maintenance console.

Chapter 15
THE LOOK-AHEAD UNIT

by R. S. Ballance, J. Cocke, and H. G. Kolsky

15.1. General Description

The look-ahead unit is a speed-matching device interposed between the arithmetic unit and the memory. With multiple 2-μsec memory units—typically four units for data, each independently operable—it is possible to fetch or store a data word every 0.5 μsec. This rate would be high enough to keep up with the fast arithmetic unit, as well as a number of input-output units, were it not for unavoidable delays. The delay between the initiation of an operand transfer and the arrival of that operand at its destination is made up partly of the access time of the memory itself and partly of the time taken for the operand to pass through a series of switches and registers. More delay occurs if the desired memory unit happens to be busy finishing a previously initiated memory cycle or if it still needs to service a request with higher priority. The total waiting time may amount to several memory cycles.

The time spent by the arithmetic unit waiting for an operand may be greatly reduced by "looking" several instructions ahead of the one currently being executed. If the memory reference is initiated early enough, the operand will usually be available in a buffer register by the time the arithmetic unit is ready for it. Similarly, the arithmetic unit should be allowed to place a just-computed result into a buffer register for storing in memory while it proceeds with the next operation. By per-

Note: The discussion of the results of a timing simulator study, which governed the choice of design parameters of the look-ahead unit, is taken from an earlier paper by J. Cocke and H. G. Kolsky, The Virtual Memory in the Stretch Computer, *Proc. Eastern Joint Computer Conf.*, no. 16, pp. 82–93, December, 1959. That paper included a description of the simulator logic, which is omitted here. It also contained a description of the look-ahead concept on which the simulator was based; this chapter includes instead a simplified description by R. S. Ballance of the actual look-ahead unit as it exists in the Los Alamos system.

forming these collection, storage, and distribution functions, the look-ahead unit raises the effective speed of the arithmetic unit.

The look-ahead unit may also be considered as a buffer that helps to smooth the data flow through memory. With many parts of the system having independent access to memory, it is natural for peaks and valleys to occur in the demand for a given memory unit. Input-output units cannot be kept waiting long, and so they have a higher priority on memory than the central processing unit. If the CPU were simply stopped during a period of peak input-output activity, the waiting time would be lost completely. By having a reservoir for unexecuted instructions in the look-ahead registers, it is possible to make up some of the lost time by satisfying deferred CPU demand during a period of lower input-output activity. Thus the look-ahead helps to regulate the fluctuations in memory demand.

As has been described in Chap. 14, there are actually two such buffering devices in the central processing unit. One is the *instruction unit*, which fetches the instructions, indexes and partially executes them, and initiates memory references for operands. The other is the *look-ahead unit*, which consists of several *look-ahead levels*, each providing one stage of buffering. A level comprises a series of special registers, which receive a pre-decoded instruction from the instruction unit and wait for the operand to arrive from memory. The arithmetic unit (both the parallel and the serial parts, since they do not operate independently) receives the assembled operation and operand information as soon as everything is ready and proceeds with the operation. A *store* operation causes the result to be returned to an available level in the look-ahead unit and then to be sent to storage while the arithmetic unit proceeds to the next instruction.

The look-ahead unit may be described as a *virtual memory* for the arithmetic unit. The arithmetic unit communicates only with the look-ahead unit, not directly with the real memory; it receives instructions and operands from the look-ahead and returns its results there. The virtual memory, being small and fast, resembles in some respects the separate fast memory that was originally proposed for Project Stretch. It differs greatly, however, in that it takes care automatically of the housekeeping involved in the data and instruction transfers and thus avoids most of the extra time and all of the difficult storage-allocation problems associated with a hierarchy of memories of different sizes and speeds.

To make the housekeeping fully automatic and keep the task of "looking ahead" from being a burden on the programmer, it was necessary to solve several fundamental logical problems in the design of the look-ahead unit. One class of problems results from the ability of the machine to treat instructions as data. This ability is a basic property of stored-program computers, where instructions and data reside in the same alterable memory. As an example, consider the instruction sequence:

Location	*Instruction*
a	LOAD, b
$a + 1$	STORE ADDRESS, $a + 2$
$a + 2$	BRANCH, c

where a, b, and c are memory addresses. Unless precautions are taken, the instruction unit may be preparing the BRANCH instruction before it has been modified by STORE ADDRESS. (This ability to modify instructions was a major advance resulting from the invention of the stored-program concept. Its importance has diminished greatly with the advent of indexing, but it is still undesirable to prohibit instruction alteration or make it difficult to use.)

A similar problem may arise in the manipulation of data. Thus the expression $T_{i+1}^2 = (T_i + D)^2$ might be formed by the sequence:

LOAD, t
ADD, d
STORE, t
MULTIPLY, t

where t and d are the addresses of T and D. Here STORE changes the operand needed for the MULTIPLY instruction, which would already be under preparation.

A third example occurs in conditional branching, when the condition depends on the result of an operation yet to be completed by the arithmetic unit. To maintain efficient operation, the instruction unit must "guess" the outcome of the test and continue to prepare instructions. If the guess proves wrong, the already prepared instructions must be discarded, and any modifications of addressable registers must be rescinded before the instruction unit starts down the correct path.

Program interruption produces a similar situation. The instruction and look-ahead units may be working on instructions which may never recur after the interruption and which, therefore, should leave no traces.

These are logical pitfalls that would be very difficult to avoid by programming means. Hence the design of the look-ahead unit was required to make the CPU, despite its complex overlapped and nonsequential operation, appear to execute programs sequentially, one instruction at a time.

15.2. Timing-simulation Program

The detailed design of the look-ahead unit could not be completed until several system-design criteria were established. The complexity of the proposed system made it extremely difficult to analyze. Even the existence of the look-ahead unit could not be justified on the basis of simple calculations. At the same time, decisions were needed concerning such basic problems as the number of memory units, the interlacing and allocation of memory addresses, and the number of look-ahead levels required. Also of interest were trade-off factors for the speed of the instruction unit, the arithmetic unit, and the magnetic core memory units.

A timing-simulator program was written, for the IBM 704, to attempt a quantitative answer to such questions. This program simulated the timing of typical test problems on a computer system embodying the look-ahead concept. It should be stressed that the program was a *timing* simulator and did not execute instructions in an arithmetical sense. Also, the parameters for the study were chosen arbitrarily to cover ranges of interest and do not represent actual operation times used in the design. The simulator traced the progress in time of the instructions through the computer model, observing the interlocks necessary to make the look-ahead behave correctly.

Because of the concurrent, asynchronous operation of different parts of the computer, there are many logical steps being executed at any time, with each step proceeding at its own rate. This flow of many parallel continuous operations was simulated by breaking the time variable into finite time steps. The basic time step in the simulator was 0.1 microsecond.

Experience indicated that more information would be gained by making a large number of fast parameter studies, using different configurations and test programs, than could be obtained by a very slow, detailed simulation of a few runs with greater precision per run. Even so, the time scale was too fine for serious input-output application studies. These would have required a simpler simulator having a basic time interval at least ten times as coarse.

A series of studies were made, in which the main parameters describing the system were varied one or two at a time, in order to get a measure of the importance of various effects. After this the studies were specialized toward answering specific questions in the 7030 design.

Five test programs were selected as typical of different classes of problems.

1. *Mesh problem.* Part of a hydrodynamics problem containing a fairly "average" mixture of instructions for the kind of scientific problems found at the Los Alamos Scientific Laboratory: 85 per cent floating-point, 14 per cent index-modification, and 1 per cent variable-field-length instructions. The execution time of such problems is usually limited by the speed of the floating-point arithmetic unit.

2. *Monte Carlo branching problem.* Part of an actual Monte Carlo neutron-diffusion code. This represents a chain of logical decisions with

very little arithmetic. It contains 47 per cent floating-point instructions, 15 per cent index-modification instructions, and 36 per cent *branches* of the indicator and unconditional types. Its speed is largely instruction-access-limited.

3. *Reactor problem.* The inner loop of a neutron-diffusion problem. This consists of 90 per cent floating-point instructions (39 per cent of which are *multiply* instructions) and 10 per cent index-modification instructions. Its speed is almost entirely limited by the arithmetic unit.

4. *Computer test problem.* The evaluation of a polynomial using computed indices. It has 71 per cent floating-point, 10 per cent index-modification, 6 per cent variable-field-length, and 13 per cent *indicator branch* instructions. It is usually arithmetic-unit-limited, but not for all configurations.

5. *Simultaneous equations.* The inner loop of a matrix-inversion routine, having 67 per cent floating-point and 33 per cent index-modification instructions. Arithmetic and logic are about equally important. It is limited both by arithmetic and instruction-access speeds.

Some of the results of these studies are summarized below. For simplicity, only the first two problems, the mesh and Monte Carlo calculations, are illustrated; the other problems generally gave results intermediate between these two.

Number of Look-ahead Levels

Figure 15.1 shows the effect on speed of varying the number of levels of look-ahead. Curves for the Monte Carlo and mesh calculations with two sets of arithmetic- and instruction-unit speeds are shown. The arithmetic-unit times given are average for all operations. A number of interesting results are apparent from these curves.

1. The look-ahead organization provides a substantial gain in performance. The point for "0 levels" means that the arithmetic unit is tied directly to the instruction unit, although simple indexing-execution overlap is still possible.

2. The speed goes up very rapidly for the first two levels, then rises more slowly for the rest of the range.

3. A large number of levels does less good in the Monte Carlo problem than in the mesh problem, because constant branching spoils the flow of instructions. Notice that the curve for the Monte Carlo problem actually decreases slightly beyond six levels. This phenomenon is a result of memory conflicts caused by extraneous memory references started by the computer's running ahead on the wrong-way paths of branches.

4. The computer performance on a given problem is clearly lower for lower arithmetic speeds. It is important to note, however, that the sensitivity of the over-all speed to change in the number of levels is also less for lower arithmetic speeds. The look-ahead improves performance in either case, but it is not a substitute for a fast arithmetic unit.

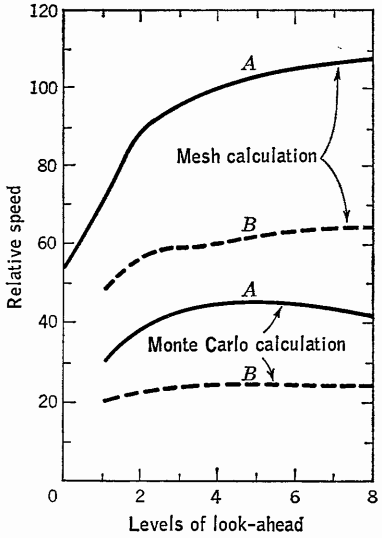


FIG. 15.1. Computer speed vs. number of levels of look-ahead. Four main memories, 2.0 μsec; two fast memories, 0.6 μsec; for two sets of arithmetic speeds:

	A	B
Arithmetic-unit time, μsec	0.64	1.28
Instruction-unit time, μsec	0.6	1.4

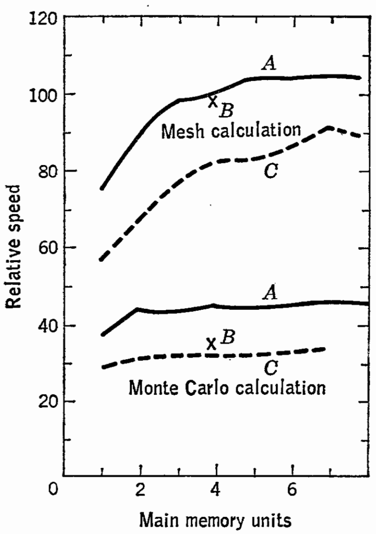


FIG. 15.2. Computer speed vs. number of main memory units. Four levels of look-ahead; arithmetic-unit time 0.64 μsec; instruction-unit time 0.6 μsec. *A:* instructions in separate 0.6-μsec memory; *B:* instructions in separate 2.0-μsec memory; *C:* instructions and data sharing same 2.0-μsec memory.

Number of Memory Units

Figure 15.2 shows how internal computer speed varies with the number of memory units and with two different memory speeds. The entire calculation is assumed to be contained in memory. The speed gain from overlapping memories is quite apparent from the curves.

The original computer design assumed the use of two kinds of memory units, a large "main" memory unit (2.0-μsec cycle) and a pair of fast but smaller memory units (0.6-μsec cycle). The intent was to place the instructions for the inner loops in the fast memory and the necessarily large volume of data, as well as the outer-loop instructions, in the main memory. The graph shows the effects of changing some of the assumptions.

The speed differential between having and not having instructions separated from data arises from delays in instruction fetches when memory units are busy with data. This effect varies from problem to problem, being less pronounced for problems that are arithmetic-limited and more pronounced for logical problems.

The crosses in Fig. 15.2 are isolated points that show the effect of replacing the 0.6-μsec instruction memories by a pair of the 2.0-μsec memory units used as instruction memories only. The resulting performance change is small for the mesh problem, which is arithmetic-limited, but larger for the instruction-access-limited Monte Carlo problem.

Arithmetic- and Instruction-unit Speeds

Although everyone realized the effect of arithmetic speed on over-all computer performance, it was not until the simulator results became available that the true importance of the instruction-unit speeds was recognized. Figures 15.3 and 15.4 show a two-parameter family of curves giving the computer speed as a function of average arithmetic-unit and instruction-unit times.

Figure 15.3, in which the arithmetic time is the abscissa, shows an interesting saturation effect, where the computer performance is independent of arithmetic time below some critical value. Thus it makes no sense to strain execution speeds if the instruction unit is not improved correspondingly. The curves in Fig. 15.4 show a similar saturation effect as the instruction-unit times decrease. Thus each unit places a performance ceiling on the other unit.

Arithmetic-unit Efficiency

A frequently quoted fallacy is that the goal of improved computer organization is to increase the efficiency of the arithmetic unit. Actually this is not the goal itself. Arithmetical efficiency depends strongly on the mixture of arithmetic and logic in a given problem, and a general-purpose computer cannot be equally efficient on all problems. Moreover, the simplest way to increase arithmetic-unit efficiency in an asynchronous computer is to *slow down* the arithmetic unit.

The real goal of improved organization is to obtain maximum over-all computer performance for minimum cost. As long as efficiency remains reasonably high for a variety of problems, one tries to increase arithmetic speed, stopping this process when the over-all performance gain no longer matches the increase in equipment and complexity. Arithmetic-unit efficiency is a by-product of this design process, not the prime variable.

Concurrent Input-Output Activity

Because of the widely different time scales for input-output activity and internal instruction execution, the simulator cannot take into account the availability or nonavailability of specific data from input-output units. We can, however, observe the combined effect of the input-output devices operating at different rates simultaneously with computing.

The input-output exchange is designed for an over-all peak rate of one word every 10 microseconds. The high-speed disk synchronizer has a design limit of one word every 4 microseconds. Since the mechanical devices must take priority over the central processing unit in addressing memory, the computation slows down in memory-busy conflicts.

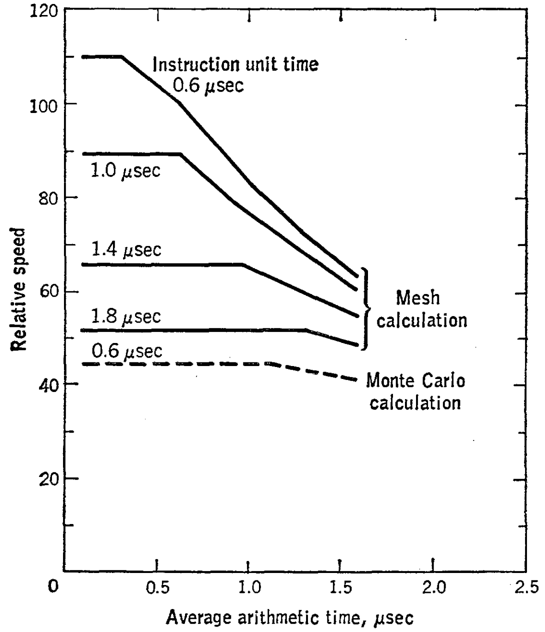


FIG. 15.3. Computer speed vs. arithmetic times for various instruction-unit times. Four levels of look-ahead; four units of 2.0-μsec memory; two units of 0.6-μsec memory.

Figure 15.5 shows an example of how internal computing speed is affected by input-output rates. At the theoretical *choke-off* point the input-output devices take all the memory cycles available and stop the calculation. It may be seen that this condition can never arise for any input-output rates presently attainable.

A 7030 system with only one or two memory units has lower performance than a system with more units, for three reasons: (1) the internal speed of the system is reduced by the loss of memory overlap; (2) the input-output penalty is higher when a given amount of input-output is run concurrently with the computation; and (3) the amount of data that can be held in the memory at one time is smaller, requiring more input-output activity to do the job. Note that increasing the memory size on a conventional computer effects improvement only with respect to the third of these factors.

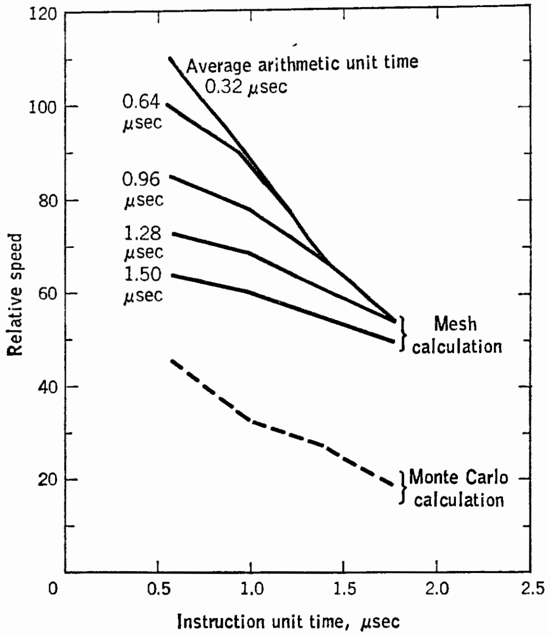


FIG. 15.4. Computer speed vs. instruction-unit times for various arithmetic-unit times. Same assumptions as in Fig. 15.3.

Branching on Arithmetic Results

Since a *branch* instruction spoils the smooth flow of instructions to the instruction unit, any *branch* in a program will cause some delay, but the most serious delays occur when branching is conditional on results produced by the arithmetic unit, which cannot be determined by the instruction unit in advance.

There are two basic ways in which branches conditional on arithmetic results can be handled by the computer:

1. The computer can stop the flow of instructions until the arithmetic unit has completed the preceding operation and the result is known, before fetching the next instruction. This procedure causes a delay at every such branch, whether taken or not.

2. As has been mentioned, the computer can "guess" which way the branch is going to go before it is taken, and proceed with fetching and preparing the instruction along the most likely path; but, if the guess was wrong, these instructions must be discarded and the correct path taken instead.

A detailed series of simulator runs were made to determine which was the better approach. Some general observations were:

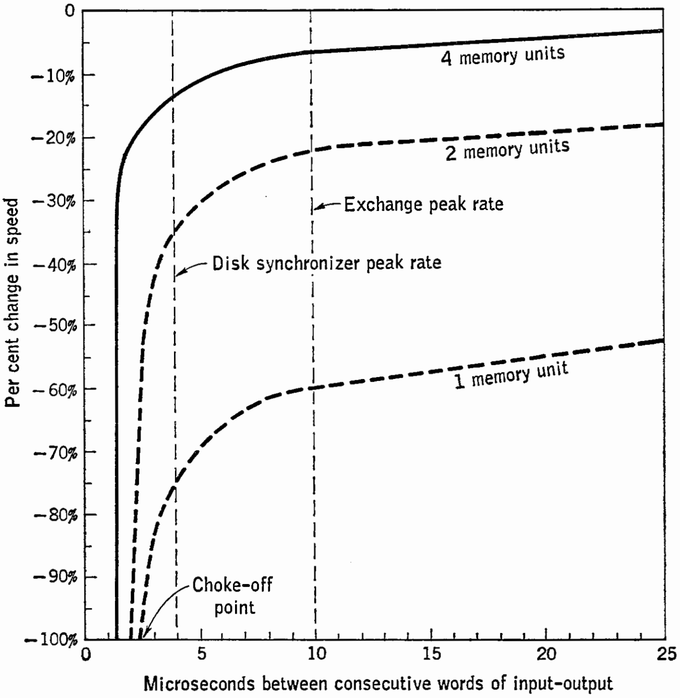


FIG. 15.5. Effect of input-output rate on internal computing speed. For Monte Carlo calculation.

1. For a problem with considerable arithmetic-data branching, the performance can vary by ±15 per cent depending on the way in which branching is handled.

2. Holding up at every branch point seems less desirable than any of five guessing procedures: condition will be *on*, condition will be *off*, condition remains unchanged, branch will be taken, branch will not be taken.

3. Unless there is an unusual situation in a problem with a very high probability that the branch will always be taken, the least time will be lost if one assumes that branching will *not* occur.

4. The theoretically highest performance would be obtained if each *branch* instruction had a *guess bit,* which would permit the programmer to specify his own guess of the most probable path. This would place a considerable extra burden on the programmer :or the gains promised. (It would also use up many valuable operation codes.)

5. There is a feedback in such design decisions. Knowing the way in which the machine "guesses" the branches, many programmers will write their codes so as to gain speed. The result is that the statistics of actual experience will be biased in favor of the system chosen, thus "proving" that it was the right decision.

Outcome of Simulator Studies

The results of the simulator studies led to these design choices for the 7030 system:

1. Four levels of look-ahead are provided.
2. The standard memory complement is two *instruction memories* and four *data memories,* all with 2-μsec cycle time; fewer or more memory units are optional. (The increase in performance possible with the faster 0.6-μsec instruction memories was felt not large enough to offset the reduction in storage capacity—1,024 words for each fast unit as compared with 16,384 words for each slower unit.)
3. The addresses of the four data memories are interlaced (i.e., four consecutive addresses refer to different memory units); likewise the addresses of the two instruction memories are interlaced separately.
4. For a *branch* instruction conditional on the result of arithmetic-unit operations, the instruction unit proceeds as if the *branch* will fail.

It should be noted here that these simulation studies were carried out before the detailed design of the computer and so the simulated model did not reflect accurately all subsequent design decisions. The actual computer performance should not be expected to follow the patterns of Figs. 15.1 to 15.5 exactly.

15.3. Description of the Look-ahead Unit

For expository reasons this description of the look-ahead unit and its operation is much simplified. Many of the checking procedures and the special treatment of internal data registers have not been included. At several places processing is described as if it were sequential, when it actually is overlapped.

Each of the four levels of the look-ahead unit is composed of a number of registers, tag bits, and counters (Fig. 15.6). The registers contain the following information, which will be required at the time the instruction in this look-ahead level is to be executed:

Operation code. Contains the partially decoded operation.

Operand. Contains, in general, the data on which the operation is to be performed.

Indicators. Contains a record of any of fifteen indicators that are to be set at execution time. Their setting is a result of instruction preparation in the instruction unit or of errors detected during look-ahead operation.[1]

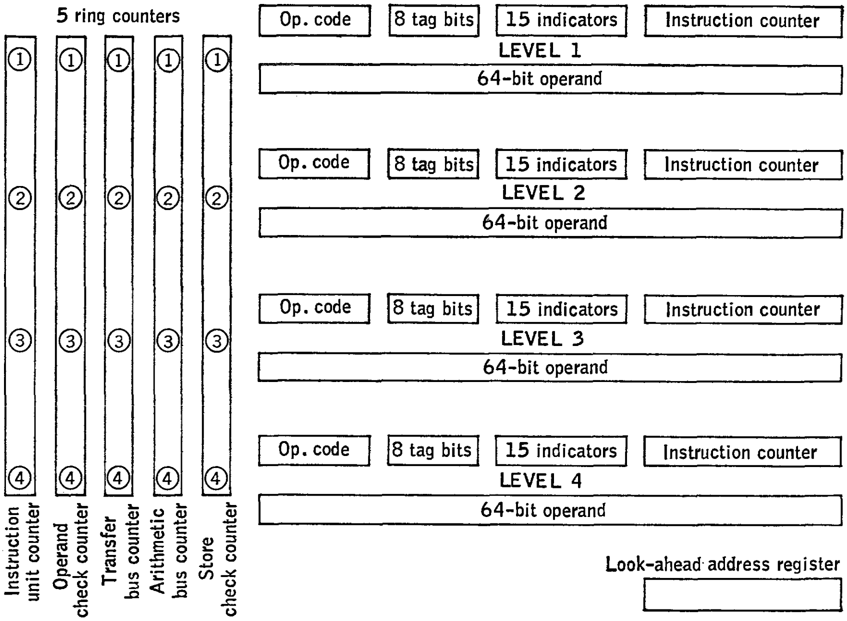


FIG. 15.6. Look-ahead registers.

Instruction counter. Contains the location of the instruction immediately following the instruction held in this level.

The following tag bits are used for control purposes:

Level filled bit. Indicates that the operand field has been filled.

Level checked bit. Indicates that the data have been checked and that their check bits have been converted to the form required by the particular operation to be performed.

[1] The fifteen indicators are: *machine check, instruction reject, operation code invalid, address invalid, data store, data fetch, instruction fetch, index flag, index count zero, index value less than zero, index value zero, index value greater than zero, index low, index equal, index high.*

Internal operand bit. Indicates that the operand is to come from an internal register rather than memory.

Instruction counter bit. Indicates that the instruction-counter field of this level is valid for use during an interrupt.

Look-ahead operation code bit. Indicates that the information in the operation-code field is to be used only by the look-ahead control.

Word-boundary crossover bit. Indicates that the VFL operand crosses a word boundary.

No operation bit. Indicates that the instruction is to be suppressed and treated as though it were a NO OPERATION instruction.

From bit. Designates the level *from* which forwarding can take place.

Five ring counters control the operation of the look-ahead unit, as they advance from one level to the next:

Instruction unit counter. Designates the next level of look-ahead to receive an instruction from the instruction unit.

Operand check counter. Designates the next level at which an operand is to be checked.

Transfer bus counter. Designates the next level to be transferred to the working registers of the arithmetic unit.

Arithmetic bus counter. Controls the functions necessary for proper operation of the interrupt system. Also designates the next level to have its indicator field entered into the indicator register and to receive any result from the arithmetic unit for later storing.

Store check counter. Designates the next level from which an operand is to be sent to storage. For non-*store*-type operations, this counter generally advances with the arithmetic-bus counter.

The counters advance from level to level under their own control. For example, after the instruction-unit counter has completed the loading of an instruction into level 1, it will advance to level 2, ready to receive an instruction for that level. After an operand has arrived, the operand-check counter can cause the operand in level 1 to be checked; the counter then advances to level 2 to check the operand there. Except for interlocks to keep the counters in proper sequence, the counters are free to advance as soon as their work is completed.

15.4. Forwarding

Each time a *store* operation is loaded into a look-ahead level, the operand address is placed in the common *look-ahead address register* (Fig. 15.6), and this level is tagged by turning *on* the *from* bit. The address of each subsequent data fetch is compared with the contents of the look-ahead address register, and, if they are equal, the data fetch is canceled and the operand field is *forwarded* from the tagged level. This forwarding process saves memory references and prevents the use of obsolete data.

When the look-ahead address register is not busy with a *store* instruction, it contains the address of the most recently loaded operand. Thus, if several successive references are made to the same address, only one memory fetch is required, the other levels receiving their operands by forwarding. Consider these instructions for forming A^3:

```
LOAD, a
MULTIPLY, a
MULTIPLY, a
```

The operand A is fetched from address a once for LOAD and then supplied to the two MULTIPLY instructions by forwarding.

Since only one look-ahead address register is provided, the look-ahead unit can handle only one *store*-type operation at a time.

15.5. Counter Sequences

Instruction-unit Counter

Figure 15.7 shows, in simplified form, the sequence of operations initiated by the instruction-unit counter at a given look-ahead level. Three types of instructions must be distinguished:

1. Instructions for which no data are to be fetched, such as *branch* instructions, which require no operand at all, or instructions with *immediate addressing*, where the operand is obtained from the instruction unit as part of the instruction
2. Instructions requiring an operand fetch from memory and
3. *Store*-type instructions

As soon as the instruction is loaded, a test for the type of instruction is made. If no more data are needed, the level is immediately tagged as having been filled and checked, and the sequence is ended. If the instruction is of the fetch type, a comparison is made with the look-ahead address register to see whether the data should be forwarded from another level, and the operand request (which had already been initiated by the instruction unit to save time) is canceled; otherwise the look-ahead address register, if available, is set to permit forwarding of this operand to another level.

A *store*-type instruction sequence must wait until the look-ahead address register is free of any earlier *store* operation, and the register is then set up for possible forwarding to another level.

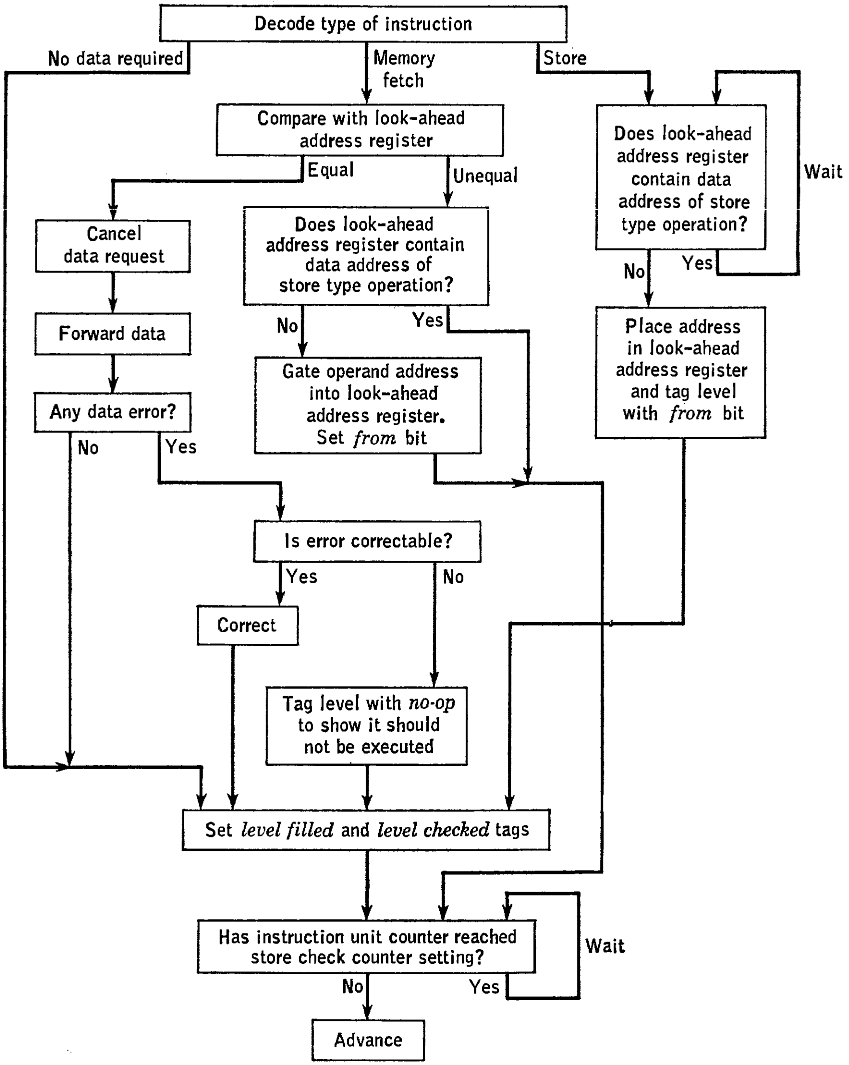

Decode type of instruction
No data required
Memory fetch
Store
Compare with look-ahead address register
Equal
Unequal
Does look-ahead address register contain data address of store type operation?
Wait
Cancel data request
Does look-ahead address register contain data address of store type operation?
No
Yes
Forward data
No
Yes
Place address in look-ahead address register and tag level with *from* bit
Gate operand address into look-ahead address register. Set *from* bit
Any data error?
No
Yes
Is error correctable?
Yes
No
Correct
Tag level with *no-op* to show it should not be executed
Set *level filled* and *level checked* tags
Has instruction unit counter reached store check counter setting?
Wait
No
Yes
Advance

FIG. 15.7. Instruction-unit counter advance sequence.

The instruction-unit counter is interlocked to prevent it from advancing to a level still occupied by the store-check counter. This prevents new information from destroying data yet to be stored.

Operand-check Counter

Operand-check counter action (Fig. 15.8) is not required after forwarding, since the operand will already have been checked by the instruction unit and the level-checked bit will be *on*. If the bit is *off*, the

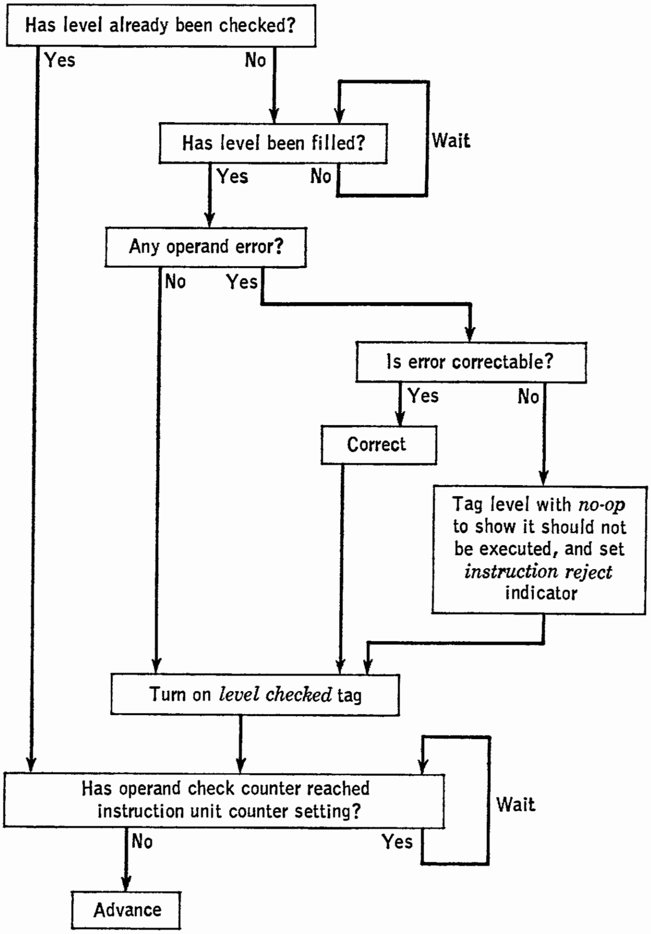

Has level already been checked?
Yes
No
Has level been filled?
Wait
Yes
No
Any operand error?
No
Yes
Is error correctable?
Yes
No
Correct
Tag level with *no-op* to show it should not be executed, and set *instruction reject* indicator
Turn on *level checked* tag
Has operand check counter reached instruction unit counter setting?
Wait
No
Yes
Advance

FIG. 15.8. Operand-check counter advance sequence.

counter will wait until the operand has arrived before proceeding with checking and error correction.

This counter is interlocked so that it will not pass the instruction-unit counter.

Transfer-bus, Arithmetic-bus, and Store-check Counters

Figures 15.9 and 15.10 illustrate some simple sequences for these three counters as applied to floating-point instructions. Each counter is appropriately interlocked with its predecessor.

The transfer-bus counter sends the completely assembled and checked information held in the current look-ahead level to the arithmetic unit

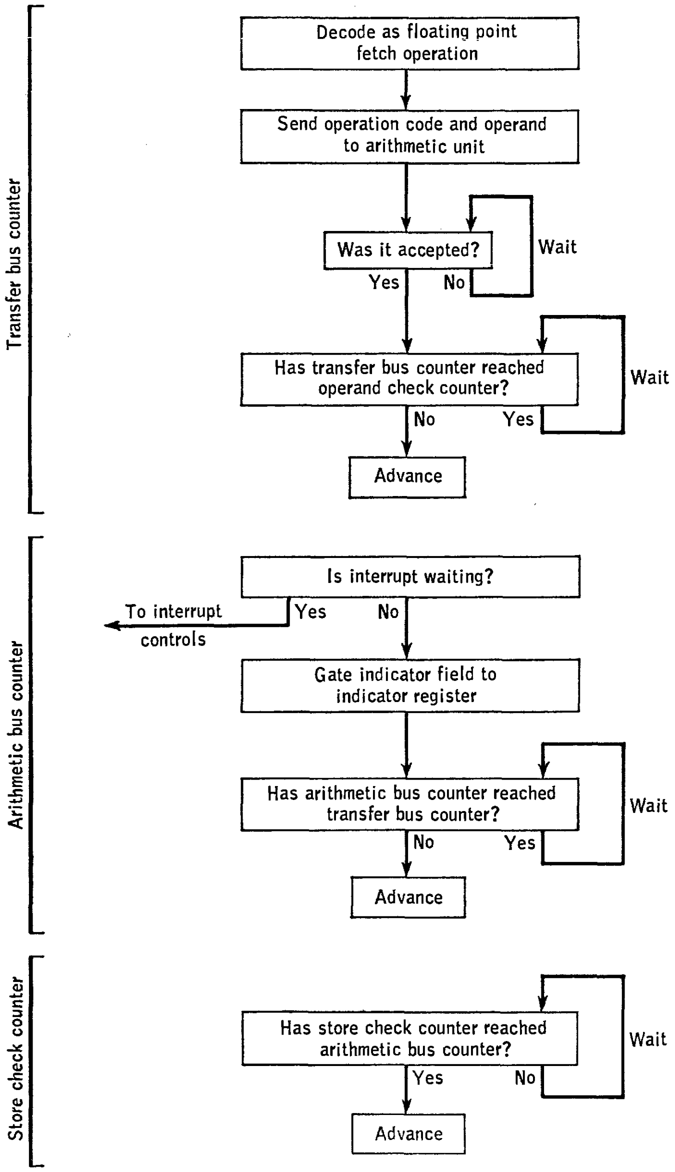

Transfer bus counter
Decode as floating point fetch operation
Send operation code and operand to arithmetic unit
Was it accepted?
Yes
No
Wait
Has transfer bus counter reached operand check counter?
No
Yes
Wait
Advance
Arithmetic bus counter
Is interrupt waiting?
To interrupt controls
Yes
No
Gate indicator field to indicator register
Has arithmetic bus counter reached transfer bus counter?
No
Yes
Wait
Advance
Store check counter
Has store check counter reached arithmetic bus counter?
Yes
No
Wait
Advance

FIG. 15.9. Transfer-bus, arithmetic-bus, and store-check counter advance sequences for floating-point fetch-type operations.

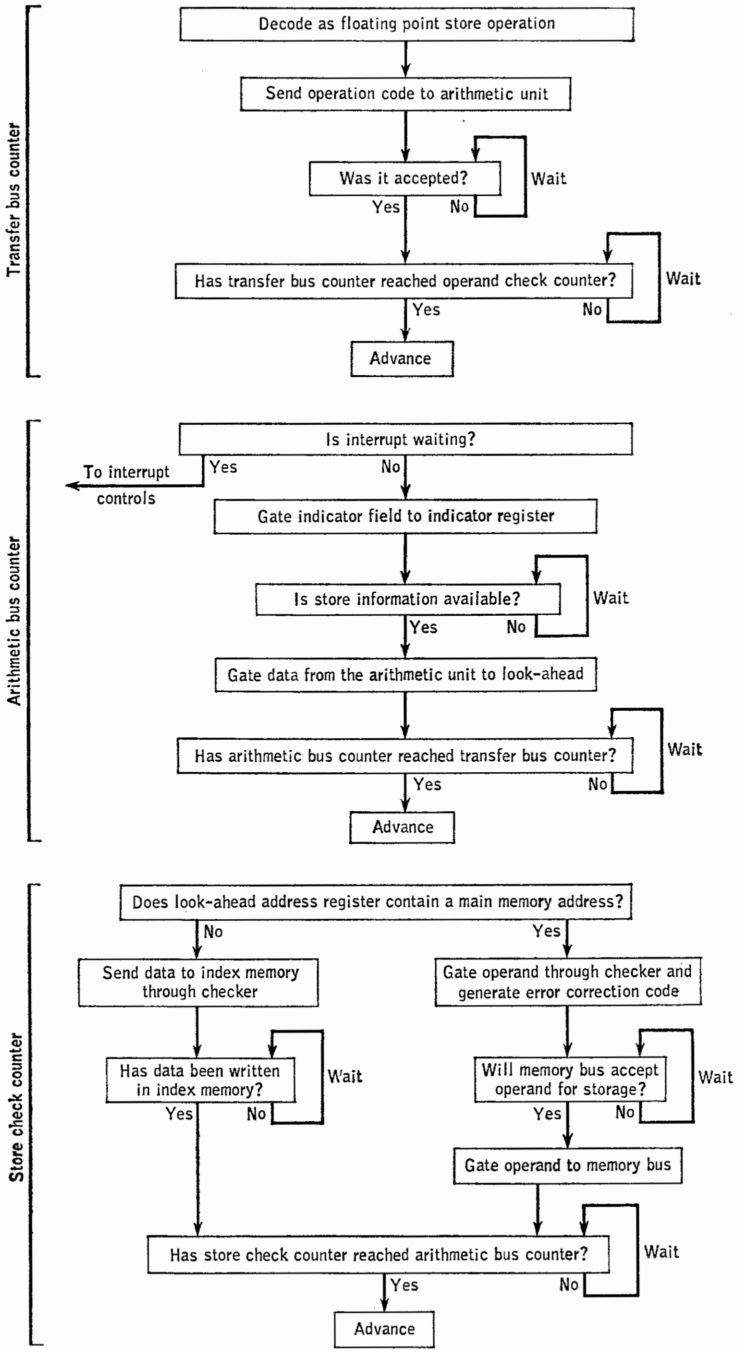

Transfer bus counter
Decode as floating point store operation
Send operation code to arithmetic unit
Was it accepted?
Yes
No
Wait
Has transfer bus counter reached operand check counter?
Yes
No
Wait
Advance
Arithmetic bus counter
Is interrupt waiting?
Yes
No
To interrupt controls
Gate indicator field to indicator register
Is store information available?
Yes
No
Wait
Gate data from the arithmetic unit to look-ahead
Has arithmetic bus counter reached transfer bus counter?
Yes
No
Wait
Advance
Store check counter
Does look-ahead address register contain a main memory address?
No
Yes
Send data to index memory through checker
Gate operand through checker and generate error correction code
Has data been written in index memory?
Yes
No
Wait
Will memory bus accept operand for storage?
Yes
No
Wait
Gate operand to memory bus
Has store check counter reached arithmetic bus counter?
Yes
No
Wait
Advance

FIG. 15.10. Transfer-bus, arithmetic-bus, and store-check counter advance sequences for floating-point store-type operations.

and waits until the information is accepted. This counter must test the no-operation bit, which, if *on*, would indicate that an error had occurred and require that the operation be suppressed.

The arithmetic-bus counter first tests whether an interrupt is waiting, which would cause the present sequence to be abandoned and control to be turned over to the interrupt system. If there is no interrupt, the fifteen indicator settings, previously accumulated during the preparatory steps in the instruction unit, now become valid and are set into the indicator register for test and possible interrupt after execution of the instruction at this level. If the instruction is of the *store* type, the arithmetic-bus counter is responsible also for transmitting the operand from the arithmetic unit to the look-ahead level (or directly to the destination when the address refers to an internal CPU register).

The store-check counter has little to do when no storing is required. For a *store*-type instruction this counter handles the transfer of the operand via appropriate checking equipment either to its destination if the address is in the index memory or to the memory bus if the address is in main memory.

There are numerous and more complex variations of these counter sequences, many of which involve more than one level of look-ahead. A variable-field-length instruction may use one level to hold the various items of operation-code information. The operand will appear in the next level, or in the next two levels if a memory-word boundary must be crossed. When a result is to be returned to memory, one or two additional levels are needed. Any progressive indexing requires a level to control the return of information to index storage. At each extra level the look-ahead unit inserts a pseudo operation code to control the action required. An extreme case is a VFL ADD TO MEMORY instruction with progressive indexing, which may require six successive levels (two levels being used twice).

15.6. Recovery After Interrupt

Whenever there is a change in instruction sequence, either by an interrupt signal or by a (successful) *branch* operation, the look-ahead unit must start recovery action. We shall describe the interrupt procedure as an illustration.

As soon as the arithmetic-bus counter senses an interrupt, the instruction unit and arithmetic unit are signaled to stop preparing and executing more instructions. The interrupt system is disabled temporarily. The look-ahead *housecleaning mode* is turned on.

The instruction-unit counter stops where it is. The operand-check and transfer-bus counters are allowed to advance until they reach the same level as the instruction-unit counter. The arithmetic-bus counter identifies each level, for which the instruction unit has previously modified an index word in the index memory, by tagging it as a pseudo *store* level. The old contents of the index word are placed in the pseudo *store* level, and the store-check counter is responsible for storing this word in the index memory.

Eventually all counters will be at the same level, and the look-ahead unit will then be empty. The proper instruction-counter setting is sent to the instruction unit to return that unit to the point in the program at which interruption occurred. The housecleaning mode in the look-ahead is turned off, and the instruction and arithmetic units are allowed to resume operation.

At this point the instruction unit has to turn *off* the indicator that caused the interrupt and fetch the extra instruction from the proper location in the interrupt table (see Chap. 10). This extra instruction is prepared and executed, after which the interrupt system is again enabled (unless the extra instruction specified that the system remain disabled). The temporary disabling of the interrupt system prevents secondary interrupts, which might cause the extra instruction to be suppressed and would leave no trace of the current interruption. The instruction unit is then ready to continue with normal loading of the look-ahead unit.

15.7. A Look-back at the Look-ahead

The 7030 look-ahead unit is a complex device, in theory as well as in practice. It contains many high-speed register positions to allow the system to race down the road and extensive controls for recovery if it has missed a turn. Even so, cost and other practical engineering considerations cause the look-ahead unit in the 7030 to fall far short of the ideal envisaged: a virtual memory with unlimited capacity and instantaneous recovery. Nevertheless, the unit does substantially raise performance by overlapping waiting periods and housekeeping operations with the execution of instructions.

As mentioned at the start, the basic reason for a look-ahead unit in a high-speed computer is the large discrepancy between the memory-cycle times and the instruction-execution times. If a much faster memory unit of equal size could be designed, the look-ahead unit could be greatly simplified or even eliminated. Improvements in memory technology are to be expected, but such improvements are again likely to be equaled or surpassed by corresponding improvements in arithmetical circuits. Thus the mismatch may be expected to continue in the future, indicating that many more refinements of the look-ahead principle will be applied in future high-performance computers, perhaps to a hierarchy of memories.

Chapter 16

THE EXCHANGE

by W. Buchholz

16.1. General Description

The function of the exchange is to direct the information flow between input-output or external storage units and internal memory. It transfers data between external units and any part of the main memory independently of the computer, and so it permits a number of external units to function simultaneously with the processing of data in the computer. Furthermore, the exchange provides a buffering action, for it transfers data on demand, as required by the unit, using main memory as buffer storage.

The exchange contains the common control facilities that are to be time-shared among the external units, thus keeping these units as simple as possible yet maintaining fully overlapped operation. The exchange also does the necessary bookkeeping of addresses and the assembly or disassembly of information without taking time away from the computer or from the internal memory. The only computer time involved is that needed to start and restart the operations. The only main memory cycles required during external operations are those needed to transfer the data to or from the final locations in main memory; these cycles are sandwiched between computing operations without interfering with the computer program except for the slight delays that may occur when the exchange requires a memory cycle at the same time as the computer.

When it encounters instructions that apply to external units, the computer executes all address modification. It sends the addresses and the decoded operation to the exchange, which determines from status bits available for each channel whether the unit required is ready. The exchange then releases the computer to continue with the program. Whenever time periods are available from other work, the exchange proceeds to obtain the operand (the control word, for instance) from memory and start the external unit. Thereafter, it carries out the data-transmission functions whenever the unit gives a request for service. Service requests are infrequent enough so that the exchange can handle the data flow for many units in an interleaved fashion.

There are eight input-output channels in the basic exchange, with provisions for expanding to 32 such channels by adding identical groups of circuits. The design also provides for the addition of a large number of low-speed channels by further multiplexing of one of the regular channels.

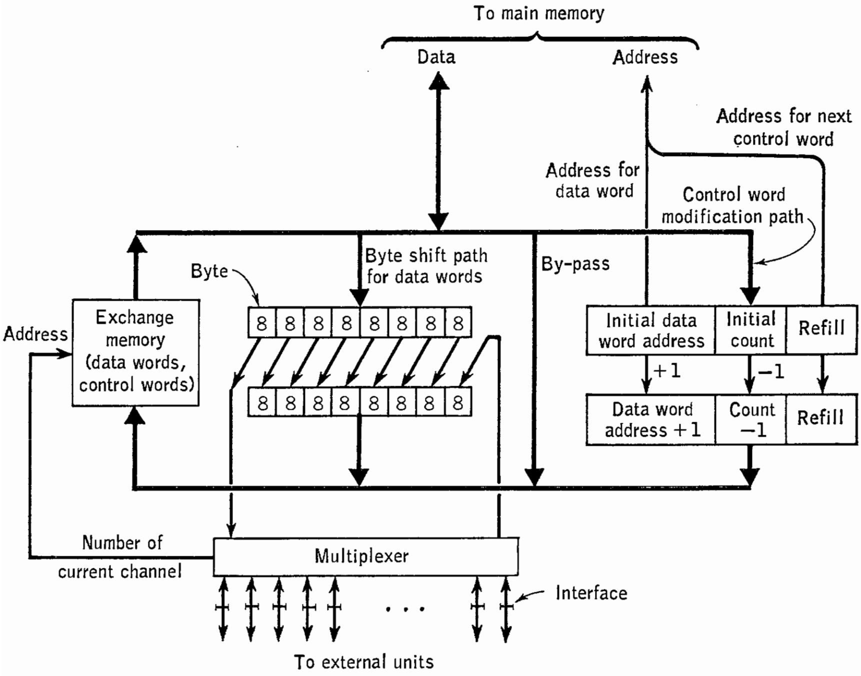


FIG. 16.1. Data-flow paths of exchange.

Regardless of speed, all channels are logically independent. Each channel can transmit data simultaneously with other channels, up to a maximum determined by the data-transmission rates. For simultaneous operation, only one input-output unit is connected to each channel. Where sequential operation is adequate, it may be desirable to share input-output control circuits among more than one input-output unit and operate the units on a single channel; magnetic tape units, for example, are provided with this equipment-sharing facility.

Each channel has an address, which becomes the address of the particular unit physically and electrically connected to that channel. When there is switching among multiple units connected to one channel, a second address must be given to select the unit to be connected to the channel before the operation to be performed by that channel is specified.

In a sense, the exchange is a separate special-purpose, fixed-program computer. It receives directions from the main program in the form of predigested instructions and control words. In general, the exchange performs those functions that remain unchanged from one job to the next, and it does such limited jobs more efficiently than the main computer could do them. Functions that vary from one job to the next, such as editing the data, are left to the program in the main computer. Editing, in fact, requires some of the most sophisticated features of the computer.

A simplified diagram of the data-flow paths of the exchange is shown in Fig. 16.1. This diagram is the basis for the brief discussion to follow.

16.2. Starting a WRITE or READ Operation

The heart of the exchange is a small, 1-μsec core memory which has space for a limited amount of data and control information for each channel. In a single 1-μsec cycle, a word can be read from this memory, modified, and returned to its location.

When the exchange receives a WRITE or READ instruction from the computer, it tests certain *status* bits before accepting the instruction. Status bits for each channel are stored in appropriate locations of the exchange memory. The exchange then obtains the control word specified by the instruction from main memory and stores it in the exchange memory. Each channel has a location for the current control word assigned to it. These control words are modified during data transfer to keep track of addresses and counts.

16.3. Data Transfer during Writing

The exchange also has a data-word location for each channel. This serves as a temporary buffer for data during transfer. (Actually, the faster channels have a pair of these locations for extra speed, but the logic is the same and will be explained as if only one such location existed.)

To start a WRITE operation, the exchange goes through a control-word modification cycle. It fetches the control word from the appropriate location in the exchange memory, increases the data word address by 1, decreases the count by 1, and returns the modified control word to its exchange memory location. The modification takes place in the control-word modification unit shown at the right in Fig. 16.1. The unmodified data-word address, extracted from the original control word, is used to fetch the first data word from main memory and store it in the exchange memory at the data-word location for this channel. The exchange then sends a signal to the input-output unit to start writing.

Writing takes place one byte at a time, where a byte consists always of

8 information bits and 1 parity-check bit (odd-count parity). When the unit is ready to write a byte, it sends a service request to the exchange. The exchange starts a 1-μsec memory cycle to pull the data word out of the appropriate location and pass it through the shift circuit shown in the center of Fig. 16.1. The leftmost byte is sent to the unit via a multiplexing circuit while the remaining bytes are shifted left by 8 bits. The shifted data word is returned to the exchange memory, still within the same memory cycle.

Each time a new byte is needed by the unit, the data-word cycle is repeated; the leftmost byte is extracted, and the remainder is shifted left. After the eighth such byte cycle, the data word is exhausted and a control-word cycle is started. The current data-word address is extracted to fetch a new data word while the control word is modified, adding 1 to the address and subtracting 1 from the count. Data transfer then continues with the new data word.

If the count in the control word goes to 0 and if chaining is indicated (the chain flag in the control word is set to 1), the refill address is used to fetch the next control word from main memory, and data transfer proceeds.

Thus, data transfer consists principally of 1-μsec data-word-shift cycles with control-word modification and data-word-fetch cycles interspersed every eighth byte, and occasionally a control-word-refill cycle. Since a single channel requests service only at intervals of many microseconds, other channels can have similar service during any 1-μsec period. The purpose of the multiplexer is to determine which channel has requested service, to send the channel number to the exchange memory as an address for selecting the appropriate data and control-word locations, and to gate the lines of this channel to the common data-handling circuits. If more than one channel requests service at the same time, the requests are handled in turn during different cycles, and no conflict arises. The worst-case condition occurs when all channels that are in operation happen to request service at the same time. The traffic-handling ability of the exchange is determined by how many channels it can service in the time between successive bytes or words of the fastest unit operating.

16.4. Data Transfer during Reading

Reading works much the same as writing. When a unit requests service, the incoming byte is gated through the multiplexer into the rightmost byte position of the current data word while the remaining bytes are shifted 8 bits to the left. Thus bytes are assembled during eight successive cycles into a word, which is then sent to main memory according to the current data-word address in the control word. Control-word modification and refill cycles are exactly the same as before.

The same data-word shifting and control-word modification equipment is used for both reading and writing. Read and write cycles from different channels may be freely intermixed; the direction of flow during a given 1-μsec cycle is determined by bits in the data-word location for each channel; these bits are set up by the instruction.

16.5. Terminating a WRITE or READ Operation

The end of a writing or reading operation may be sensed by the unit and signaled to the exchange; or it may be sensed by the exchange when the count in the control word goes to 0 and the chain flag in the current control word is 0, so that there is no control word to follow. In either case the exchange instructs the unit to stop.

The exchange then attempts to interrupt the computer program, to report (1) that the operation has ended and (2) whether it ended normally or any unusual conditions arose, such as a programming error, data error, machine malfunctioning, or the end of tape or paper. The address of the interrupting channel is also sent to the computer. Usually the program interrupt occurs at the end of the instruction currently being executed in the computer.

Occasionally the interrupt must be delayed. The program may have disabled the interrupt mechanism, perhaps to complete the processing of a previous input-output interrupt. The exchange then stores the appropriate status indications in the control-word location of the exchange memory and tries again later. When the interrupt finally succeeds, it is handled in the same way as if it had just happened.

There can be no confusion caused by simultaneous interruptions from more than one input-output unit. The exchange automatically presents them to the computer one at a time.

Interruptions due to a *channel signal* (see Chap. 12) are handled in the same way as end-of-operation interrupts, even if the *channel signal* is not the direct result of a previous operation.

16.6. Multiple Operations

Multiple-block WRITE and READ operations (see Chap. 12) are indicated by a *multiple flag* bit in the control word. When the unit signals that the operation ended normally, the exchange immediately restarts the unit just as if a new instruction had been given, and the program is not interrupted at this time.

16.7. CONTROL and LOCATE Operations

The operations CONTROL and LOCATE are set up in the same manner as WRITE, except that a different instruction line is activated. The control or address information is then transmitted to the unit as if it were data. Termination is also handled the same way.

16.8. Interrogating the Control Word

As writing or reading proceeds, the exchange continually modifies the appropriate control words stored in the exchange memory. The program may interrogate the current control-word contents during the operation by giving a COPY CONTROL WORD instruction, which transfers the current control word to a specified location in main memory. This operation finds use mostly in specialized supervisory programming; ordinary programs seldom require it because it is more convenient to wait for an automatic interrupt at the end of the operation.

It should be noted that the original control word, which is located at the main memory address specified by the instruction, is not modified in any way by the exchange. It retains the initial settings for use in subsequent operations.

16.9. Forced Termination

Occasionally it may be desirable to force an input-output operation to come to a halt; for example, a programming error may give rise to an endless control-word chain. To initiate the termination sequence immediately, a RELEASE instruction may be given even while an operation is in progress; RELEASE may also be used sometimes to reset the channel status to normal.

The RELEASE instruction functions in the same manner as the usual end-of-operation sequence, except that any exceptions (error conditions, etc.) are not reported because they are presumably no longer of interest.

Chapter 17

A NONARITHMETICAL SYSTEM EXTENSION

by S. G. Campbell, P. S. Herwitz, and J. H. Pomerene

17.1. Nonarithmetical Processing

One of the most interesting current trends in the computer field is the development of nonarithmetical techniques. Nonarithmetical problems are being attacked with increasing success, particularly in the area of the "soft sciences." Efforts in the fields of artificial learning, character recognition, information retrieval, gaming, and language translation account for a rapidly growing percentage of total computational activity. During the next few years it may be expected that work in such areas will materially enhance our understanding of the nature of learning, thinking, theorem proving, and problem solving.

Even problems considered to belong to the "hard sciences," which are usually associated with complex mathematical computations, may involve an enormous amount of nonarithmetical data processing. Weather forecasting is an excellent example. The scientist tends to view the weather as a tremendous hydrodynamics problem on a rotating sphere, in which the boundary conditions are very complex and the equations very difficult to manage. From another standpoint, however, the weather system represents a problem in information collection, transmission, storage, and processing—with all the characteristics to be expected of any large-scale file-maintenance activity. Much of the data, such as cloud type, are not really numerical, and the processing that such data usually undergo is not primarily arithmetical. Moreover, the data are highly perishable for most purposes—yesterday's weather is of interest only to the statistician. The weather system is in fact a very large real-time information-handling system, in which the value of the data begins to decrease the minute it is

Note: Section 17.1 is an introduction by S. G. Campbell, and the rest of the chapter is taken from a paper by P. S. Herwitz and J. H. Pomerene presented at the 1960 Western Joint Computer Conference.

taken and has diminished markedly by the time it can be transmitted to a potential user.

It seems characteristic of the conventional application of computers to the hard sciences that the resulting computation is relatively regular and that the operations are likely to consist mostly of specialized, complex operations. This is why scientific computers have acquired very powerful floating-point-arithmetic and indexing facilities. By contrast, conditions are chaotic in the nonarithmetical area: activities are likely to be irregular and to consist of relatively rudimentary operations, such as basic logical operations, counting, table look-up, and the simple process of hunting for some particular piece of information—looking for the proverbial needle of useful information in a haystack of noise.

To characterize the weather of the world in reasonable detail requires a rather staggering amount of data: perhaps 10^9 numbers. The problem of too much data and too little information is not limited to the weather system; as much, or more, information is required to characterize the operation of a large business, a large government organization, or a large social activity. No one person could look at all this information in a lifetime, much less during the useful life of the information itself (for although such information dies much more slowly than meteorological information, it perishes none the less). What the user often requires is some sort of characterization of some subset of the information in his system. Usually this characterization is something statistical: What is the net operating profit or loss from Flight 123 on Tuesdays over the past year? Since the user cannot look at all the data, he attempts to obtain its essential meaning from a weighted statistical average or to determine cause-and-effect relationships by correlating events that look as though they might be related.

Another difficulty is that it is frequently impossible to tell at the time the data are taken whether they are significant or not. This is particularly true of a system that collects data automatically; it may be more economical to let the system function at a constant data-gathering rate, rather than to try to speed it up when the information appears to be more important (for example, when the weather is bad) and slow it down when the information seems less pertinent. A data-processing system usually contains a great deal more data than it really needs. The main purposes of most data-processing installations are to reduce the amount of data stored, to make the significant data more accessible, and to provide effective statistical characterizations. Reduction in the amount of data stored may result from more efficient formats and encoding of information, from storing only primary data from which the system can generate other data, and from reducing the time lag in processing the data, so that the system does not need to store so much of it at any given time. Making data more accessible is sometimes the most significant function performed by the data-processing system, particularly in the routine operation of an organization. Provision of statistical summaries is frequently most important in providing information for management decisions and indicating general trends, although statistical information may also be used in the daily operation of a business.

Thus the primary problem in almost any field of knowledge is to map a large quantity of relatively disorganized information into a much smaller, more highly structured and organized set of information. Frequently it is not even the information that is important but its pattern. The most rudimentary attempts to find such patterns in information involve classification. Perhaps the first step, once the information has been acquired, is to arrange it in such a way that we can locate any particular subset required without having to look at all the information. (The information forms a set, the nature of the set being determined by whatever it was that made us take and keep the information.) The simplest way of accessing a subset would be to look at each piece of information to see whether it belonged to the subset or not. If there are properties of particular value, we may order the information in terms of these properties. For example, if the information consists of words to be put into a dictionary, we order it in terms of the first letter of each word; this is of great help in locating any specific known word, although it does not help at all if the object is to find all the words that end in x.

Sorting, that is, ordering data in terms of some property, is characteristic of this type of activity. If the amount of information is large, the expense of storage dictates that sorting be with respect to the most important characteristic. It would be too wasteful of expensive storage to store information sorted on very many different characteristics. As new information is needed, it must be merged with the old.

Sorting, merging, matching, etc., are, of course, the basic operations of file maintenance. In fact, the activity of business data-processing installations is quite typical of the nonarithmetical information processing we are discussing here. For that matter, so is much of the activity of scientific computing installations (if they would only admit it), for we must include the assembling, compiling, and editing functions that are peculiar to the programming and operating of the computer system itself.

File maintenance consists essentially in processing sets of operand data from *two data sources* to form a set of result data going to *one data sink.* The data sources may be visualized concretely as two input tapes, consisting of a file and transactions against that file, and the data sink may be visualized as an output tape, the updated file; but the same concept holds if the data are in core memory or stored in some other medium. The common case of multiple outputs may be represented by a single sequence of results which are switched to one of several destinations as required.

The concept of operating on two large sets of operand data to form a set of result data appears to be fundamental to nonarithmetical processing. It leads naturally to the idea that a processor, with built-in facilities for creating sources and sinks to generate and operate on long data sequences, would be a much more effective tool for large nonarithmetical applications than a conventional computer, which operates one field at a time. In such a processor the objective is to fetch two sets of data independently accessed from memory, to combine them in terms of certain processes, and to produce a third set which is put back independently into memory. The common processes of most interest are the elementary arithmetical operations, the logical operations, control operations, and comparison operations ($<$, $\leqq$, $>$, $\geqq$, $=$, $\neq$). *Table look-up* is required to define those operations which cannot readily be described in more elementary terms. (For example, the inputs might represent a pair of cities, and the output the airline fare between these cities as found in a table of fares.) One of the two sources or the sink may be missing.

Another concept is suggested by observation of the operation of a punched-card machine, where the same relatively simple process may be repeated many times for successive cards as they pass through the machine. There the process is usually defined by means of a plugboard which opens or closes paths for the data flowing through the machine. One is thus led to think of an electronic version of the plugboard, which is *set up* before starting and remains set until a change is indicated. Hence we speak of operating our processor in the *set-up mode.* Because control data are placed in high-speed registers, there is essentially no access time for instructions. The speed of the process is determined entirely by the data flow rate into or out of memory, the data being fetched or stored according to preset, but possibly very complex, indexing patterns.

Among the things we may wish to do, while passing data through the processor, are: (1) examine any of the three sets of data to look for a particular piece of information; (2) count the frequency of occurrence of various events in each set, including the occurrence of relationships between subsets of the data as well as the occurrence of the distinguished subsets themselves; (3) react to these occurrences by altering the process; and (4) perform a sequence of table look-ups, with some mechanism for determining when the look-up operation is to terminate. Having set up and started a process, we need, of course, a mechanism for breaking out of the set-up mode as necessary and for determining the state of affairs at that time.

The IBM 7951 Processing Unit, to be described in this chapter, was designed around these concepts to achieve maximum performance in a broad area of nonarithmetical information processing. The 7951, itself a machine of substantial size, is not a complete data processor; it is attached to a regular 7030 computer which performs the more conventional operations at high speed (Fig. 17.1). The extended system, which is referred to as the IBM 7950 Data Processing System, includes also two fast 1,024-word memory units, with a read-write cycle time of 0.7 μsec, and a very fast magnetic tape system capable of simultaneously reading and writing at a rate of 140,000 words of 64 bits per second. The memory and tape units are important contributors to the over-all performance of the system on nonarithmetical problems, but we shall be concerned here only with the logic of the 7951.

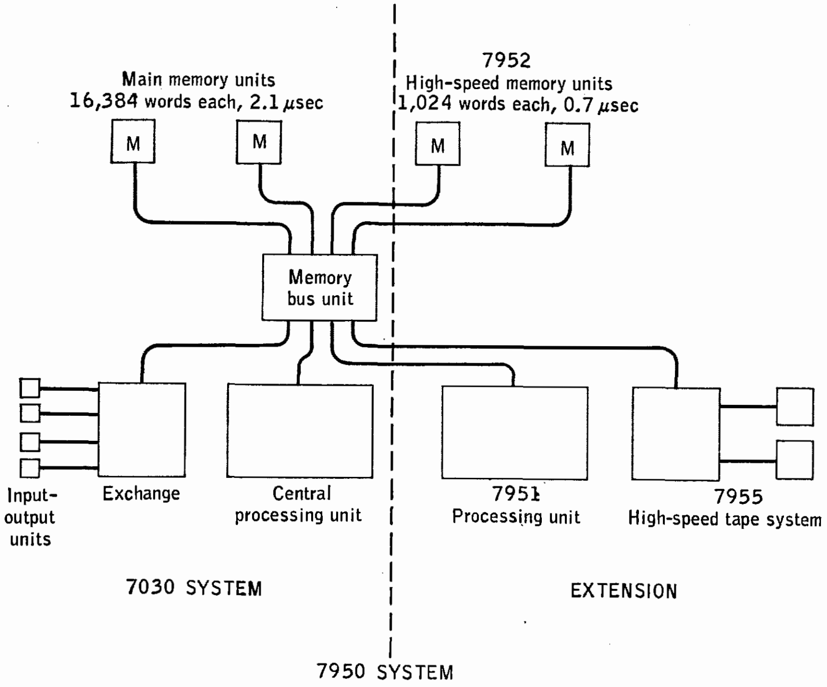


FIG. 17.1. Nonarithmetical extension of the 7030.

17.2. The Set-up Mode

Data pass through the 7951 Processing Unit serially, byte by byte. The byte, a quantity of 8 bits or less in parallel, is the basic information unit of the system. The *set-up mode* is primarily a design approach whose aim is (1) to select bytes from memory according to some pattern set up in advance and (2) to keep a steady stream of such selected bytes flowing through a designated process or transformation and thence back to memory (Fig. 17.2). Emphasis is on maximum data flow rate, so that the typicaly large volumes of information can be processed in minimum time. Processing time per byte is held to a minimum by specifying, in advance, byte selection rules, processing paths, and even methods for handling exceptional cases; hence, decision delays are suffered only once for a long sequence of bytes instead of being compounded for each byte.

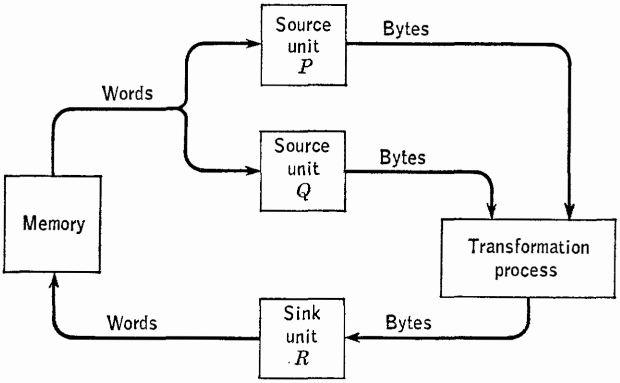


FIG. 17.2. Simplified data flow diagram.

17.3. Byte-sequence Formation

The selected bytes are taken from words stored in memory according to either simple or complicated patterns as chosen by the programmer. For technical reasons memory is organized into 64-bit words, but this artificial grouping is suppressed in the 7951, so that memory is treated as if it consisted of a long string of bits, and any one of these can be addressed for selection. As in the 7030, up to 2^{18} words of memory can be directly addressed, and, since the word size is exactly 2^6 bits, an address consists of 24 bits: 18 to select the word and 6 to select the bit within the word.

Data are transferred to and from memory 64 bits in parallel; selection to the bit level is accomplished by generalized operand registers called *source* or *sink units* (Fig. 17.3). There are two source units P and Q, which feed operands to the processing area of the 7951, and one sink unit R, which accepts results from the processing area. Each source or sink unit contains a *switch matrix*, which allows a byte to be selected with minimum delay, starting at any bit position within the register. To handle cases where a byte overlaps two memory words and to minimize waiting time for the next needed word from memory, each source or sink unit is actually two words (that is, 128 bits) long. The selection of these bytes is controlled by the low-order 7 bits of a sequence of 24-bit addresses, which are generated by the pattern-selection units. The byte output of a source unit is fed into the processing area through a bit-for-bit mask, which enables the programmer to select any subset of the 8 bits, including nonconsecutive combinations.

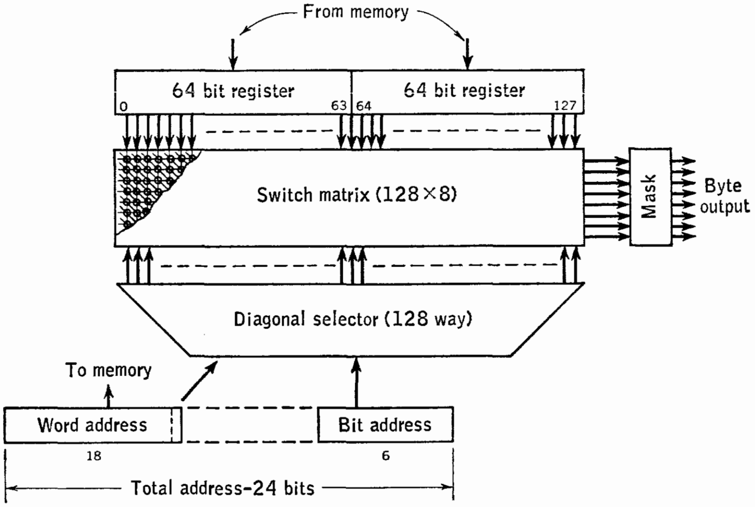


FIG. 17.3. Source unit. Sink unit is similar, except for data flow reversal.

17.4. Pattern Selection

The data input to the system may be highly redundant to any particular problem, and so a powerful mechanism is provided for imposing selection patterns on the data in memory. It is assumed that the very effective input-output control in the basic 7030 system will have grossly organized the contents of memory. For example, various characteristics may have been obtained for a population and recorded in uniform subdivisions of a file. A particular problem may be concerned with only a certain characteristic drawn from each record in the file. Thus data may be stored in memory in matrix form, and the problem may be to transpose the matrix.

Pattern selection in the 7951 resembles indexing in other computers, except that here the programmer determines the algorithm that generates the pattern, instead of listing the pattern itself. Each source or sink unit has its independent pattern-generating mechanism, which is actually an arithmetic unit capable of performing addition, subtraction, and counting operations on the 24-bit addresses. The programmer specifies patterns in terms of indexing levels, each level consisting of an address-incrementing value I, which is successively added to the starting-address value S, until N such increments have been applied, after which the next indexing level is consulted to apply a different increment. The programmer may then choose either that incrementing continue on this level or that the previous level be resumed for another cycle of incrementing.

Many other indexing modes are provided to permit almost any pattern of data selection. Particular attention has been given to direct implementation of triangular matrix selection and to the iterative chains of any formal inductive process, however complex.

In general the pattern-selection facilities completely divorce the function of operand designation from that of operand processsing, except that predesignated special characteristics of the operands may be permitted to change the selection pattern in some fashion.

The pattern-selection units determine the movement of data between the source or sink unit and memory, and, together with the source and sink units, they determine the byte flow in the processing area. The processing facilities and the selection facilities have been designed to give a flow rate of approximately 3.3 million bytes per second.

17.5. Transformation Facilities

Two facilities are provided for the transformation of data (Fig. 17.4). Extremely general operations on one or two input variables can be

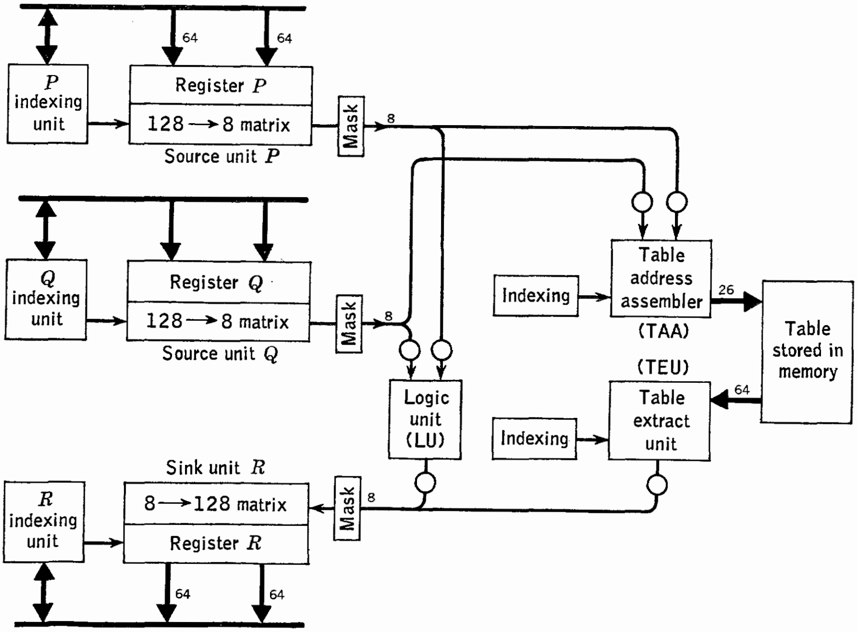


FIG. 17.4. Transformation facilities.

accomplished with the on-line table-look-up facility. Simpler operations can be done directly by the logic unit without involving memory look-up. The logic unit also provides a choice of several 1-bit characterizations of the input bytes (such as *byte from P > byte from Q*). These 1-bit signals can be used to alter the process through an *adjustment mechanism.*

The *table look-up* facility consists of two units. The more important logically is the *table address assembler* (TAA), which accepts bytes from one or two sources to form the look-up addresses that are sent to memory (Fig. 17.5). The other is the *table extract unit* (TEU), which permits selection of a particular field within the looked-up word. Both units have their own indexing mechanisms, and together they permit the programmer to address a table entry ranging in size from 1 bit to a full word and starting at any bit position in memory. This freedom is abridged only by considerations of the table structure chosen by the programmer.

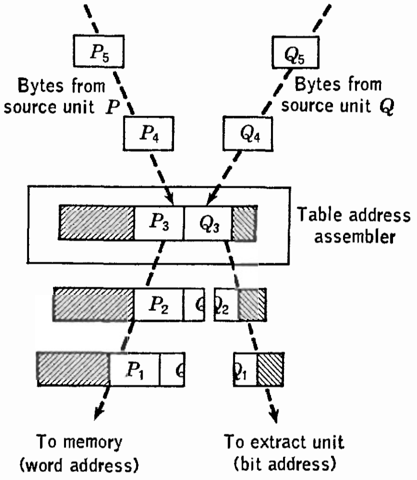


FIG. 17.5. Formation of look-up address.

The table look-up facility also provides access to the memory features of *existence* and *count.* Under instruction from the TAA, the main memory can use the assembled address to *or* a 1 into the referenced bit position; the referenced word, as it was just before the *or*ing, can be sent to the TEU. This feature may be used to indicate by a single bit the *existence* (1) or nonexistence (0) of an item in a set. In the *high-speed* memory a 1 may be either *or*ed (as in main memory) or *added* into the referenced bit position, with the same provision for sending the word before alteration to the TEU. The ability to add 1s into high-speed memory words permits use of these words as individual *counters.* Several counter sizes can be specified. (This counting feature is not provided in the main memory.)

17.6. Statistical Aids

The table look-up facility may be used to associate statistical weights with the occurrence of particular sets of bytes. For example, the occurrence of a byte P_i in the P sequence together with a byte Q_j in the Q sequence may be assigned a weight W_{ij}, which would be stored in a table and referenced by an address formed from both P_i and Q_j. Alternatively, a memory counter may be associated with each pair P_i,Q_j and stepped up whenever the pair occurs.

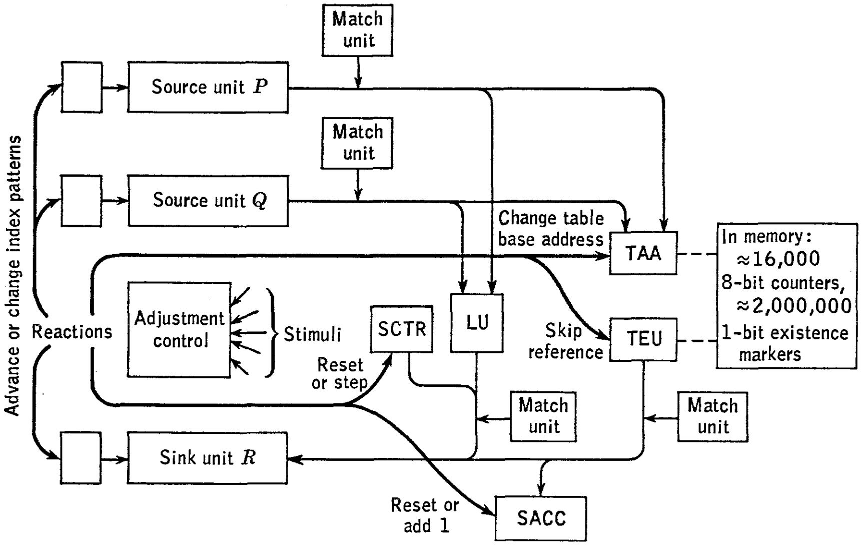


FIG. 17.6. Monitoring and statistical features with typical adjustment reactions.

A *statistical accumulator* (SACC) is provided (Fig. 17.6), either to sum the weights W over a succession of sets of bytes or to provide a key statistical measure of the counting results. SACC can also be used for many other accumulating purposes.

A *statistical counter* (SCTR) provides a way of counting the occurrences of any of a large number of events during the process. In particular, SCTR can be designated to count the number of weights W that have been added into SACC.

17.7. The BYTE-BY-BYTE Instruction

The table look-up unit, the logic unit, and the statistical units can be connected into the process in various ways by the programmer. As in a class of analog computers, these connections reflect the structure of a problem and are the electronic equivalents of a plugboard. The connection chosen by the programmer then causes each byte or pair of bytes sent through it to be processed in the same way; this very general processing mode is set up by the BYTE-BY-BYTE instruction. The connections, indexing patterns, and special conditions described below all form part of a prespecified setup, which can be regarded as a macro-instruction putting the computer into a specific condition for a specific problem.

17.8. Monitoring for Special Conditions

The concept of a continuous process with preset specifications is most meaningful when applied to a large batch of data that are all to be treated the same way. Within the data entering any particular process there may arise special conditions that call for either momentary or permanent changes in the process. For example, the transformation being performed may be undefined for certain characters, and so these must be deleted at the input; or a special character may be reserved to mark the end of a related succession of bytes, after which the process or the pattern of data selection must be altered.

Special conditions can be monitored in several ways. Special characters can be detected by *match units* (Fig. 17.6), to each of which can be assigned a special 8-bit byte which is matched against all bytes passing by the unit. There are four match units: W, X, Y, and Z, which can be connected to monitor the data at several different points. When a match occurs, the match unit can perform directly one of several operations, and it can also emit a 1-bit signal indicating the match.

A large number of 1-bit signals are generated by the various facilities to mark key points in their respective processes. These 1-bit signals, collectively called *stimuli*, can be monitored to accomplish specific operations, such as stepping SCTR or marking the end of an indexing pattern. They can also be used to accomplish a much wider range of operations through the adjustment mechanism:

Up to 64 stimuli are generated by the various processing, indexing, and monitoring functions in the 7951. For any particular problem those stimuli can be chosen which represent the significant properties of the data passing through. With each stimulus or coincident combination of stimuli, the programmer may associate one or more of a large number of reactions on the data, the process, or the indexing. These stimulus-reaction pairs are called *adjustments*. The adjustment mechanism gives the programmer a direct way of picking out those elements of the data which are different from the general run. These exceptional elements may provide the key to the pattern being sought, either because they are particularly relevant or distinctly irrelevant.

17.9. Instruction Set

Conventional arithmetical and scientific computational processes and all input-output operations are performed in the 7030 part of the system. When 7030 instructions are used, the system is said to operate in the *arithmetic mode;* when the instructions unique to the 7951 Processing Unit are to be used, the system is placed in the *set-up mode.* The set-up-mode instructions add a variety of extremely powerful data-processing tools to the basic 7030 operations. The instruction formats vary in length: 7030 instructions are either 32 or 64 bits long, whereas set-up-mode instructions have an effective length of 192 bits.

Set-up-mode instructions are very much like built-in subroutines or macro-instructions. Just as it is necessary to initialize a programmed subroutine, it is also necessary to initialize, or set up, the processor. About 150 parameters and control bits may influence the process. The processor is set up by loading values of some of these parameters and setting the desired control bits in certain addressable set-up registers prior to the execution of a set-up-mode instruction. Certain changes in the parameter values or control-bit settings generate stimuli, which may be used to terminate the data sequence, to make automatic adjustments to it, or to switch to the arithmetic mode of operation. The adjustment operations essentially constitute a second level of stored program and are used most generally to handle exception cases.

Thus the programmer sets up the processor to execute a set-up-mode instruction. The process is then started and automatically modified as dictated by the setup or the data. Much routine bookkeeping is done automatically by the several independent pattern-generating (indexing) mechanisms. Changing parameter values are always available for programmed inspection, if automatic inspection is not sufficient for the particular operation being performed.

Although most of the programming in the set-up mode of operation is centered around the BYTE-BY-BYTE instruction, a number of other instructions derive from the unique organization of the processor. The arrangement of the data paths and processing units facilitates one-instruction operations for performing many of the routine *collating* functions, such as merging, sorting, and file searching and maintenance, that are so common to data processing. The table look-up unit is used extensively in these as well as in several other instructions designed primarily for the logical manipulation of data.

Since such extensive use is made of parameter tables, transformation tables, and other data arrays, all of which require large memory areas, a special CLEAR MEMORY instruction is provided for clearing large blocks of memory in minimum time and with minimum programming effort. A single execution of this instruction will clear as few as 64 consecutive words or as many as 2,048, as desired. Clearing 2,048 words, for example, takes less than 3½ μsec, with only one instruction access to memory. A full memory complement of 2^{18} words could be cleared in less than 1 millisecond. To reset each memory word separately by ordinary programming would take very much longer.

17.10. Collating Operations

In order to perform merging, file searching, and other such collating operations, it is generally necessary to specify a number of parameters, such as record length, file length, control-field length and position, etc. In programming for the 7951, the programmer need only tabulate these parameters in proper order. They will then be utilized by the indexing mechanisms to cause data to be fetched from and returned to memory according to the patterns that naturally occur in such data.

The MERGE instruction contains eight independent control sequences that may be used to merge files or completely sort blocks of records. Options to be chosen by the programmer are concerned with whether files are to be arranged in ascending or descending order; whether the record block can be contained in at most half the available memory; and whether the control field is conveniently located at the start of the record.

The SEARCH instruction has twelve control sequences, each of which facilitates the abstracting from a master file of all records whose control fields bear one of six possible relationships to the control field of each record of a detail file. The possible relationships are the six standard comparison conditions $<$, $\leqq$, $>$, $\geqq$, $=$, $\neq$. If it is not desired to remove the records that meet the search condition, it is possible to tabulate their addresses automatically.

The instruction SELECT is used to select from a file the record having the least or the greatest control field.

For the purpose of facilitating file-maintenance operations, there is a collating instruction complex called TAKE-INSERT-REPLACE. When the operation is executed under *instruction control*, then a match between control fields of master and detail record causes the master record either to be removed from the master file or to be replaced by the detail record. Under *data control*, the action taken, whenever control fields match, is indicated by the contents of a special control byte in the detail record. The masters can be deleted or replaced; or the detail record can be inserted in the master file; or, under certain circumstances, the maintenance procedure can be interrupted when master records with special characteristics are located and then resumed with a minimum of programming effort.

Instructions such as the collating operations described above lead to a considerable reduction in the length of the generalized report generators, file-maintenance routines, and sorting and merging programs that might be expected to be associated with such a computer system.

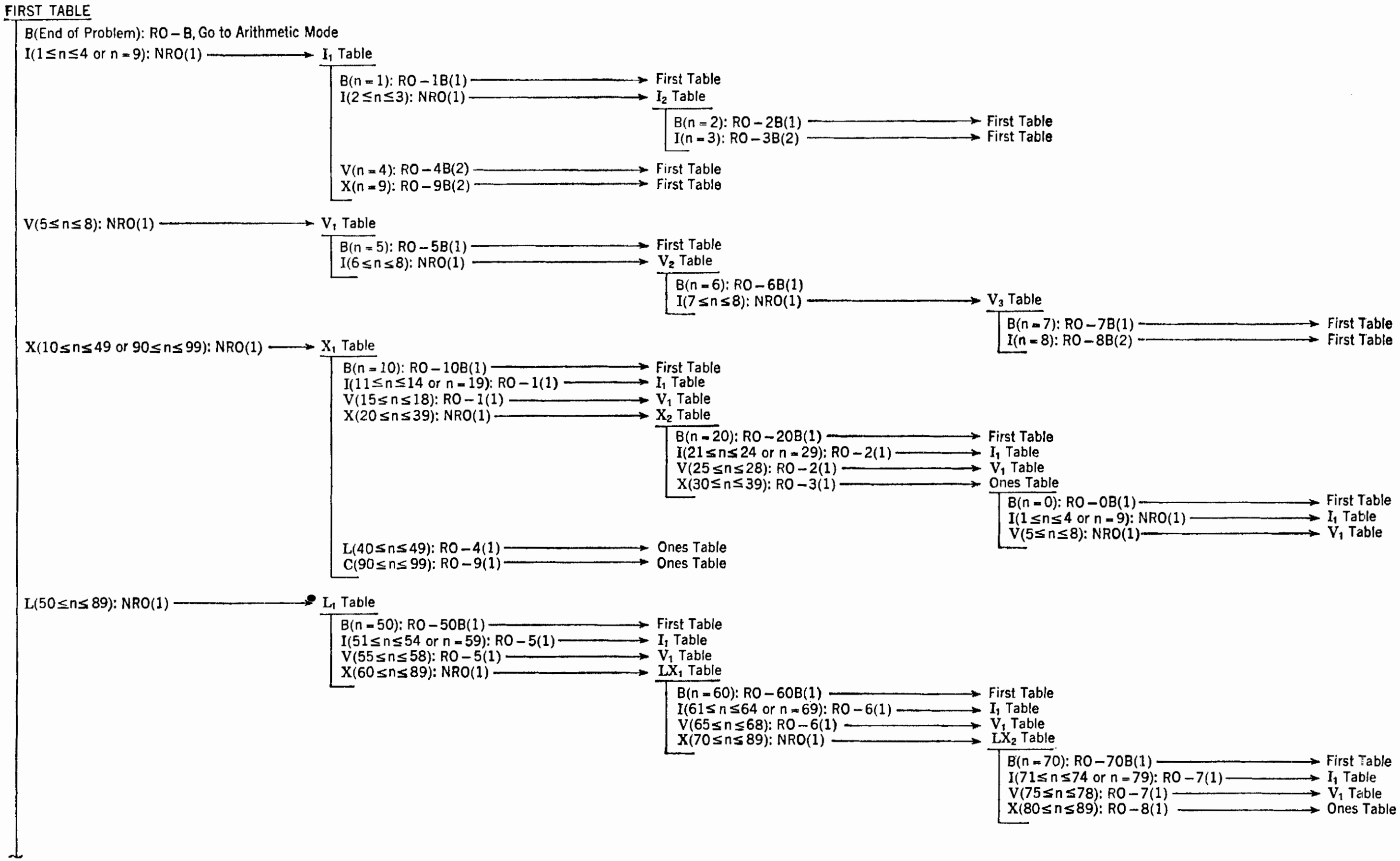
17.11. Table Look-up Operations

It is often desired to be able to obtain data from or store data at an address that depends indirectly on the data itself. The INDIRECT LOAD-STORE instruction permits wide latitude in the formation of such addresses and in the subsequent manipulation of the original data. In this operation parameters from one of the source units are used in the formation of an address in the table look-up unit. This primary address itself, or one of the two addresses found in the word at the memory location specified by the primary address, becomes either the origin of a field of data to be entered via the other source unit or the location at which the data field is to be stored by the sink unit. The data are moved from source to sink, and the entire cycle is repeated. The counting and *or*ing features of the table look-up unit are available to the programmer as modifications of the basic instruction-control sequence.

The second instruction complex built around the table look-up unit is SEQUENTIAL TABLE LOOK-UP, an extremely powerful but conceptually simple instruction for a class of data-dependent transformations. This instruction causes a series of table references to be made; each successive reference after the first is made to a table whose address is extracted automatically from the previously referenced table entry. Also, as each reference is completed, a variable amount of data may be extracted from the table entry. Moreover, the indexing of the input or output data may be adjusted according to the contents of the table entry (this is similar to the operation of a *Turing machine*). The applications of SEQUENTIAL TABLE LOOK-UP are manifold: editing for printing of numerical data, transliteration of symbols from one form to another, and scanning of computer instructions for assembly and compilation, to name a few.

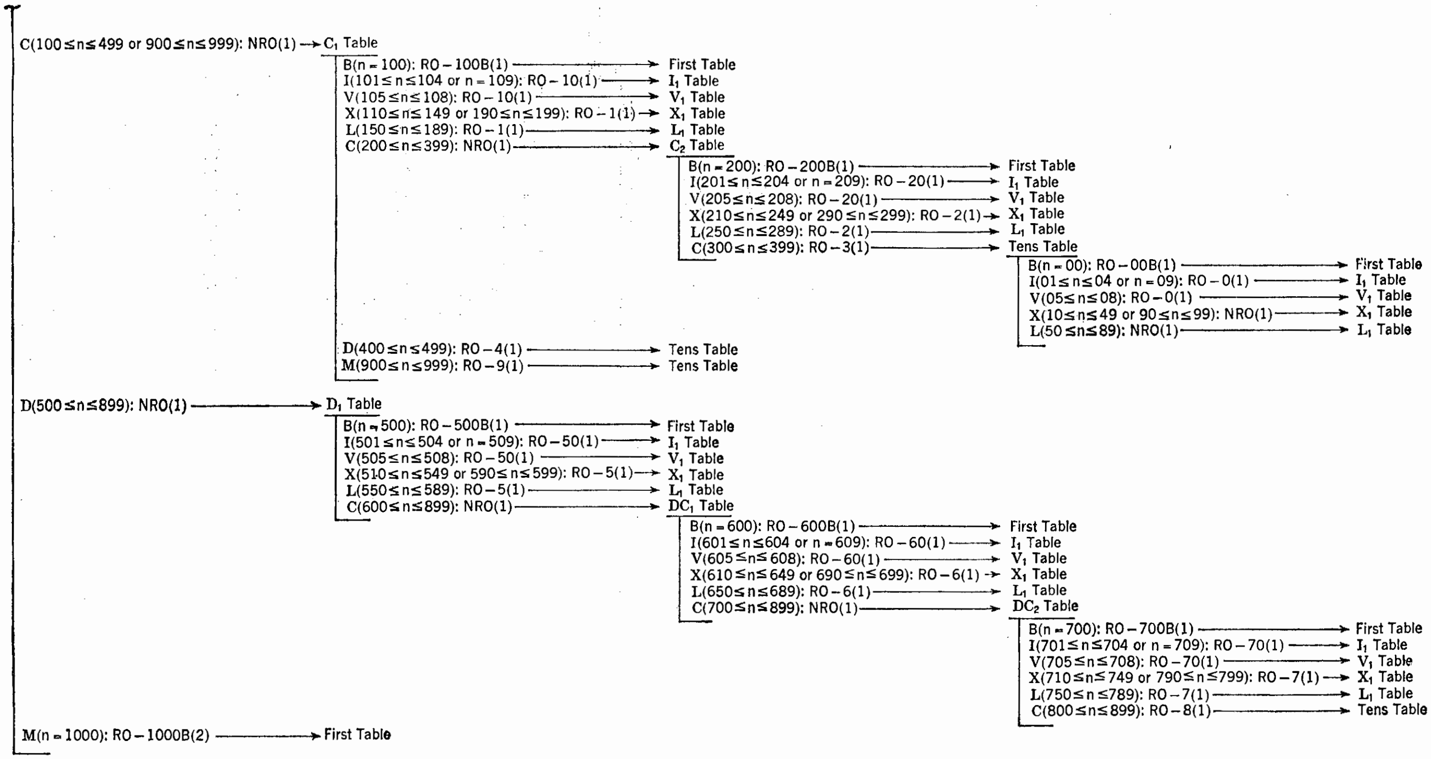
17.12. Example

The extensive use of tables in problem solution typifies the non-arithmetical processing approach, as will be illustrated by the transliteration of Roman numerals to Arabic. Several simplifying assumptions have been made so that the flow chart may be easier to follow: (1) The data—a set of numbers expressed in Roman numerals, each number separated from the next by a *blank* (*B*)—are assumed to be perfect, and only the characters *I*, *V*, *X*, *L*, *C*, *D*, and *M* are used; (2) the set of numbers is terminated by two *blanks;* (3) the use of four successive identical characters (like Roman IIII for Arabic 4) is forbidden. Finally, the



FIRST TABLE

- B(End of Problem): RO − B, Go to Arithmetic Mode
- I(1≤n≤4 or n=9): NRO(1) → I_1 Table
  - B(n=1): RO − 1B(1) → First Table
  - I(2≤n≤3): NRO(1) → I_2 Table
    - B(n=2): RO − 2B(1) → First Table
    - I(n=3): RO − 3B(2) → First Table
  - V(n=4): RO − 4B(2) → First Table
  - X(n=9): RO − 9B(2) → First Table
- V(5≤n≤8): NRO(1) → V_1 Table
  - B(n=5): RO − 5B(1) → First Table
  - I(6≤n≤8): NRO(1) → V_2 Table
    - B(n=6): RO − 6B(1)
    - I(7≤n≤8): NRO(1) → V_3 Table
      - B(n=7): RO − 7B(1) → First Table
      - I(n=8): RO − 8B(2) → First Table
- X(10≤n≤49 or 90≤n≤99): NRO(1) → X_1 Table
  - B(n=10): RO − 10B(1) → First Table
  - I(11≤n≤14 or n=19): RO − 1(1) → I_1 Table
  - V(15≤n≤18): RO − 1(1) → V_1 Table
  - X(20≤n≤39): NRO(1) → X_2 Table
    - B(n=20): RO − 20B(1) → First Table
    - I(21≤n≤24 or n=29): RO − 2(1) → I_1 Table
    - V(25≤n≤28): RO − 2(1) → V_1 Table
    - X(30≤n≤39): RO − 3(1) → Ones Table
      - B(n=0): RO − 0B(1) → First Table
      - I(1≤n≤4 or n=9): NRO(1) → I_1 Table
      - V(5≤n≤8): NRO(1) → V_1 Table
  - L(40≤n≤49): RO − 4(1) → Ones Table
  - C(90≤n≤99): RO − 9(1) → Ones Table
- L(50≤n≤89): NRO(1) → L_1 Table
  - B(n=50): RO − 50B(1) → First Table
  - I(51≤n≤54 or n=59): RO − 5(1) → I_1 Table
  - V(55≤n≤58): RO − 5(1) → V_1 Table
  - X(60≤n≤89): NRO(1) → LX_1 Table
    - B(n=60): RO − 60B(1) → First Table
    - I(61≤n≤64 or n=69): RO − 6(1) → I_1 Table
    - V(65≤n≤68): RO − 6(1) → V_1 Table
    - X(70≤n≤89): NRO(1) → LX_2 Table
      - B(n=70): RO − 70B(1) → First Table
      - I(71≤n≤74 or n=79): RO − 7(1) → I_1 Table
      - V(75≤n≤78): RO − 7(1) → V_1 Table
      - X(80≤n≤89): RO − 8(1) → Ones Table


$C(100 \leq n \leq 499$ or $900 \leq n \leq 999)$: NRO(1) → C_1 Table
$B(n = 100)$: RO − 100B(1) → First Table
$I(101 \leq n \leq 104$ or $n = 109)$: RO − 10(1) → I_1 Table
$V(105 \leq n \leq 108)$: RO − 10(1) → V_1 Table
$X(110 \leq n \leq 149$ or $190 \leq n \leq 199)$: RO − 1(1) → X_1 Table
$L(150 \leq n \leq 189)$: RO − 1(1) → L_1 Table
$C(200 \leq n \leq 399)$: NRO(1) → C_2 Table
$B(n = 200)$: RO − 200B(1) → First Table
$I(201 \leq n \leq 204$ or $n = 209)$: RO − 20(1) → I_1 Table
$V(205 \leq n \leq 208)$: RO − 20(1) → V_1 Table
$X(210 \leq n \leq 249$ or $290 \leq n \leq 299)$: RO − 2(1) → X_1 Table
$L(250 \leq n \leq 289)$: RO − 2(1) → L_1 Table
$C(300 \leq n \leq 399)$: RO − 3(1) → Tens Table
$B(n = 00)$: RO − 00B(1) → First Table
$I(01 \leq n \leq 04$ or $n = 09)$: RO − 0(1) → I_1 Table
$V(05 \leq n \leq 08)$: RO − 0(1) → V_1 Table
$X(10 \leq n \leq 49$ or $90 \leq n \leq 99)$: NRO(1) → X_1 Table
$L(50 \leq n \leq 89)$: NRO(1) → L_1 Table
$D(400 \leq n \leq 499)$: RO − 4(1) → Tens Table
$M(900 \leq n \leq 999)$: RO − 9(1) → Tens Table
$D(500 \leq n \leq 899)$: NRO(1) → D_1 Table
$B(n = 500)$: RO − 500B(1) → First Table
$I(501 \leq n \leq 504$ or $n = 509)$: RO − 50(1) → I_1 Table
$V(505 \leq n \leq 508)$: RO − 50(1) → V_1 Table
$X(510 \leq n \leq 549$ or $590 \leq n \leq 599)$: RO − 5(1) → X_1 Table
$L(550 \leq n \leq 589)$: RO − 5(1) → L_1 Table
$C(600 \leq n \leq 899)$: NRO(1) → DC_1 Table
$B(n = 600)$: RO − 600B(1) → First Table
$I(601 \leq n \leq 604$ or $n = 609)$: RO − 60(1) → I_1 Table
$V(605 \leq n \leq 608)$: RO − 60(1) → V_1 Table
$X(610 \leq n \leq 649$ or $690 \leq n \leq 699)$: RO − 6(1) → X_1 Table
$L(650 \leq n \leq 689)$: RO − 6(1) → L_1 Table
$C(700 \leq n \leq 899)$: NRO(1) → DC_2 Table
$B(n = 700)$: RO − 700B(1) → First Table
$I(701 \leq n \leq 704$ or $n = 709)$: RO − 70(1) → I_1 Table
$V(705 \leq n \leq 708)$: RO − 70(1) → V_1 Table
$X(710 \leq n \leq 749$ or $790 \leq n \leq 799)$: RO − 7(1) → X_1 Table
$L(750 \leq n \leq 789)$: RO − 7(1) → L_1 Table
$C(800 \leq n \leq 899)$: RO − 8(1) → Tens Table
$M(n = 1000)$: RO − 1000B(2) → First Table

[18 tables; 82 words (table entries)]

FIG. 17.7. Tables for conversion from Roman to Arabic numerals.

numbers to be transformed are all assumed to lie in the range from 1 to 1,000, inclusive.

The flow chart (Fig. 17.7) shows 18 tables consisting of a total of 82 memory words. Under each table heading a two-part entry is shown, the parts being separated by a colon. On the left of the colon is the argument being looked up, followed in parentheses by an indication of the range in which the final number or digit must lie. On the right of the colon the parameters of the table word corresponding to the argument are indicated symbolically; for example, RO-1*B* (meaning "read out the integer 1 followed by the character for a *blank*") or NRO (meaning "no readout"). This is followed by an integer in parentheses indicating what data byte is the next argument (0 means same byte, 1 means next byte, etc.). The arrow indicates the table in which the next argument is looked up.

As an illustration, consider the transliteration of DCLXXVIII:

1. D is looked up in the first table. The number must be in the range 500 to 899 inclusive. No digit is read out. The next argument is the next data byte.
2. C is looked up in the D_1 table. The range is 600 to 899. No readout. The next argument is the next data byte.
3. L is looked up in the DC_1 table. The range is 650 to 689. Read out 6. The next argument is the next data byte.
4. X is looked up in the L_1 table. The range of the unknown part of the number is 60 to 89. No readout. The next argument is the next data byte.
5. X is looked up in the LX_1 table. The range is reduced to 70 to 89. No readout. The next argument is the next data byte.
6. V is looked up in the LX_2 table. The range is now 75 to 78. Read out 7. The next argument is the next byte.
7. I is looked up in the V_1 table. The range of the next digit is 6 to 8. No readout. The next argument is the next data byte.
8. I is looked up in the V_2 table. The digit is 7 or 8. No readout. The next argument is the next byte.
9. I is looked up in the V_3 table. The final digit is 8. Read out 8*B*. The next argument is the *second* following byte (the next byte being a *B*). This would be the first byte of the next number to be transliterated and is looked up again in the first table.

The process just described yielded the number 678 for DCLXXVIII. Only one instruction, SEQUENTIAL TABLE LOOK-UP, was needed. In fact this single instruction serves to transform an entire set of numbers, continuing until the character *B* is looked up in the first table.

Clearly, the decision logic for the problem is incorporated in the structure of the tables. In constructing these tables the programmer concentrates on precisely this logic; most of the bookkeeping and other peripheral programming considerations are automatically taken care of. Wherever possible, this philosophy guided the systems planning of the 7951 Processing Unit.

Appendix A
SUMMARY DATA

A.1 List of the Larger IBM Stored-program Computers

The experience gained with earlier IBM computers played a major role in the development of the 7030. Because these earlier computers have been referred to in the text, it may be helpful to list them here. The computers are listed chronologically; the date of a computer is defined arbitrarily as the year of first public announcement. Only the larger computers that have been produced in multiples are shown. These include all 700 and 7000 series computers preceding the 7030, as well as the 650. The basic 650 is hardly a large computer in comparison with the others, but it deserves a place in the list because of its widespread use and because extended versions of it are used in much the same applications as many of the larger machines. The list excludes military computers and a series of smaller stored-program computers.

The listing distinguishes between the earlier computers constructed almost entirely with vacuum-tube circuits (V) and the 7000 series which is completely transistorized (T). Another common but not altogether satisfactory distinction is made between computers intended primarily for scientific applications (S) and those intended primarily for processing large files of alphanumeric data (D). In the 700–7000 series the chief technical characteristic distinguishing "scientific" computers is fast parallel binary arithmetic on numbers of fixed length, whereas the data-processing computers have serial decimal arithmetic and alphanumeric operations, for processing more readily fields of different lengths, as well as heavier emphasis on input-output. The smallest (650) computer on the list and the largest (7030) do not quite fit the classification. The 650, initially designed for numerical work, has found extensive application in data processing. The 7030, intended mainly for scientific applications, combines the characteristics of both classes and is thus also a very powerful data processor.

A general description of each current computer will be found in the corresponding General Information Manual published by IBM; detailed information is given in the Reference Manual for each machine. Some additional references to technical papers are given here.

Year	*Computer*	*Class*	*Comments*
1952	701	V, S	Parallel binary arithmetic, 2,048-word (36-bit) electrostatic memory[1]
1953	650	V	Serial decimal arithmetic, magnetic drum memory[2]
1953	702	V, D	Serial decimal arithmetic, variable-field-length, alphanumeric data handling, 10,000-character (6-bit) electrostatic memory[3]
1954	704	V, S	Redesigned 701 with new instruction set, 4,096-word magnetic core memory, built-in floating-point arithmetic, indexing, and higher speed
1954	705	V, D	Redesigned 702 with larger instruction set, 20,000 characters (Model I) or 40,000 characters (Model II) of core memory, higher speed, and simultaneous input and output
1957	709	V, S	Improved 704 with up to 32,384 words of core memory, multiple input-output channels buffered in memory, and faster multiplication[4]
1957	705 III	V, D	Improved 705 with an 80,000-character core memory, higher speed, more parallel operation, and multiple input-output channels buffered in memory
1958	7070	T, D	Serial decimal computer, partly patterned after the 650 but with major improvements; newer transistor and core memory technology place it in the 705 performance class at a lower cost[5]
1958	7090	T, S	Transistorized version of 709, about six times as fast
1960	7080	T, D	Transistorized version of 705 III, about six times as fast, with up to 160,000 characters of memory
1960	7030	T	Stretch computer described herein

[1] W. Buchholz, The System Design of the IBM Type 701 Computer, *Proc. IRE*, vol. 41, no. 10, pp. 1262–1275, October, 1953.

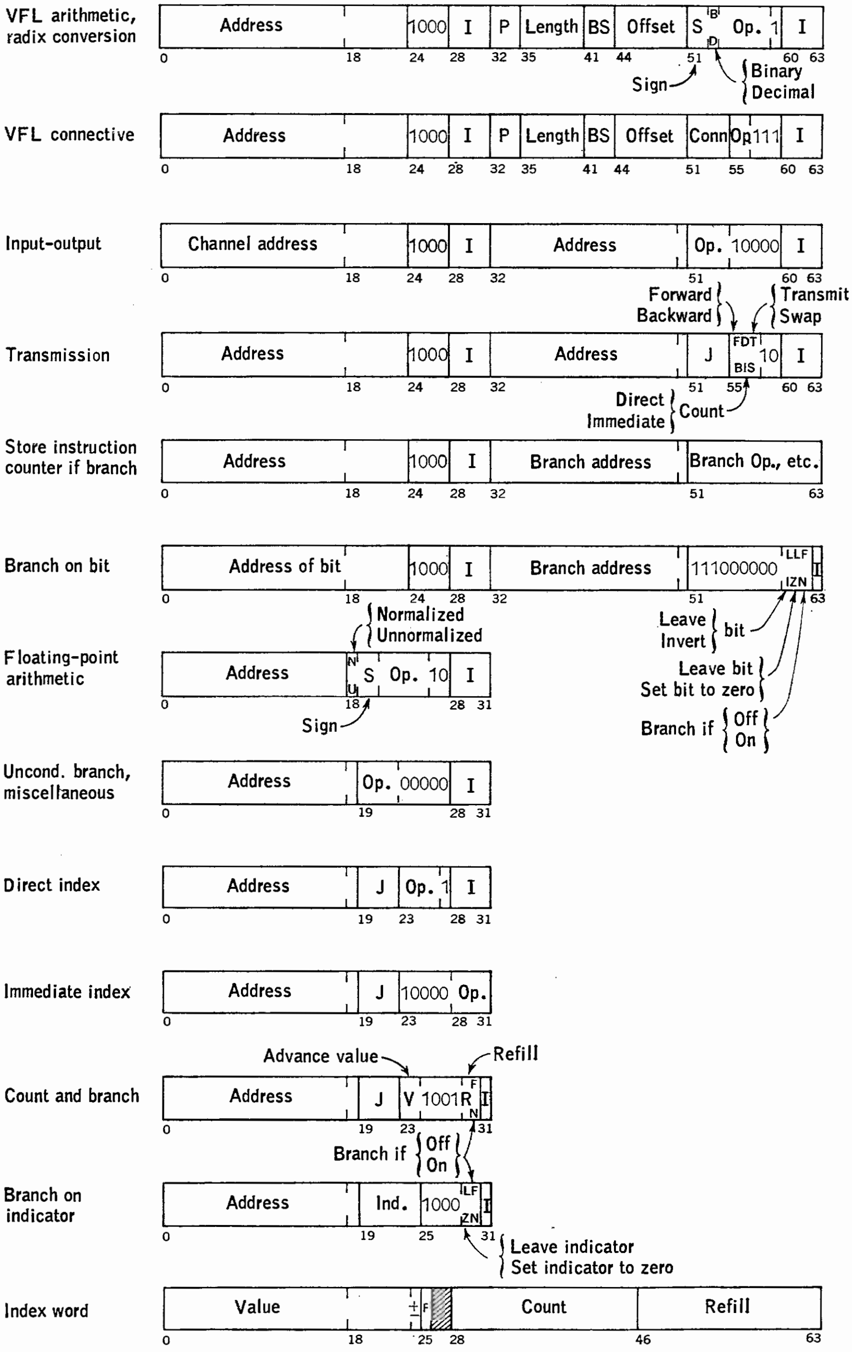
[2] F. E. Hamilton and E. C. Kubie, The IBM Magnetic Drum Calculator Type 650, *J. ACM*, vol. 1, no. 1, pp. 13–20, January, 1954.

[3] C. J. Bashe, W. Buchholz, and N. Rochester, The IBM Type 702, An Electronic Data Processing Machine for Business, *J. ACM*, vol. 1, no. 4, pp. 149–169, October, 1954.

[4] J. L. Greenstadt, The IBM 709 Computer, "Proceedings of the Sympoisum: New Computers, a Report from the Manufacturers," published by the ACM, March, 1957, pp. 92-96.

[5] J. Svigals, IBM 7070 Data Processing System, *Proc. Western Joint Computer Conf.*, March, 1959, pp. 222–231.

A.2 Instruction Formats


VFL arithmetic, radix conversion
Address | 1000 | I | P | Length | BS | Offset | S | B/D | Op. | 1 | I
0 18 24 28 32 35 41 44 51 60 63
Sign
Binary
Decimal
VFL connective
Address | 1000 | I | P | Length | BS | Offset | Conn | Op | 111 | I
0 18 24 28 32 35 41 44 51 55 60 63
Input-output
Channel address | 1000 | I | Address | Op. | 10000 | I
0 18 24 28 32 51 60 63
Forward
Backward
Transmit
Swap
Transmission
Address | 1000 | I | Address | J | FDT/BIS | 10 | I
0 18 24 28 32 51 55 60 63
Direct
Immediate
Count
Store instruction counter if branch
Address | 1000 | I | Branch address | Branch Op., etc.
0 18 24 28 32 51 63
Branch on bit
Address of bit | 1000 | I | Branch address | 111000000 | LLF/IZN | I
0 18 24 28 32 51 63
Leave
Invert
bit
Leave bit
Set bit to zero
Branch if
Off
On
Normalized
Unnormalized
Floating-point arithmetic
Address | N/U | S | Op. | 10 | I
0 18 28 31
Sign
Uncond. branch, miscellaneous
Address | Op. | 00000 | I
0 19 28 31
Direct index
Address | J | Op. | 1 | I
0 19 23 28 31
Immediate index
Address | J | 10000 | Op.
0 19 23 28 31
Advance value
Refill
Count and branch
Address | J | V | 1001 | R | F/N | I
0 19 23 31
Branch if
Off
On
Branch on indicator
Address | Ind. | 1000 | LF/ZN | I
0 19 25 31
Leave indicator
Set indicator to zero
Index word
Value | ± | F | Count | Refill
0 18 25 28 46 63

A.3 List of Registers and Special Addresses

Address	*Length (bits)*	*Mnemonic*	*Name*	*Notes*
0.0	64	Z	Zero	
1.0	19	IT	Interval timer	*p, a*
1.28	36	TC	Time clock	*p, b*
2.0	18	IA	Interrupt address register	*p*
3.0	18	UB	Upper boundary register	*p*
3.32	18	LB	Lower boundary register	*p*
3.57	1	BC	Boundary-control bit	*p*
4.0	64	MB	Maintenance bits	
5.12	7	CA	Channel address register	*b*
6.0	19	CPU	Other CPU	*c*
7.17	7	LZC	Left-zeros count	
7.44	7	AOC	All-ones count	
8.0	64	L	Left half of accumulator	*d*
9.0	64	R	Right half of accumulator	
10.0	8	SB	Accumulator sign byte	
11.0	64	IND	Indicator register	*e*
12.20	28	MASK	Mask register	*f*
13.0	64	RM	Remainder register	
14.0	64	FT	Factor register	
15.0	64	TR	Transit register	
16.0	64	x0	Index register 0	
to		to	to	
31.0	64	x15	Index register 15	

Notes: All unused bits in addresses 0.0 to 15.63 are permanently set to 0.

p Permanently protected area of memory

a Read-only, except for STORE VALUE, STORE COUNT, STORE REFILL, and STORE ADDRESS.

b Read-only. Address 1.28 means bit position 28 in word 1.

c In multiple-CPU systems, used to turn on *CPU signal* indicator in another CPU.

d In FLP operations only, the explicit operand address 8.0 is interpreted to mean the 64 bits 8.0 to 8.59 and 10.04 to 10.07, which combine to make up a proper single-length signed FLP number corresponding to the high-order part of the accumulator. In all other operations, a 64-bit operand at address 8.0 includes bits 8.0 to 8.63.

e Bits 11.0 to 11.19 are read-only.

f The rest of 12.0 are permanently set, read-only mask bits, 12.0 to 12.19 being 1s and 12.48 to 12.63 being 0s.

A.4 Summary of Operations and Modifiers

The mnemonic abbreviation is given in parentheses after the name.

Arithmetical Operations

a. *Operations Available in Both Variable-field-length and Floating-point Modes*

LOAD (L)

The accumulator contents are replaced by the memory operand, except for data flag bits.

LOAD WITH FLAG (LWF)

Same as LOAD, except that the data flag bits are included.

STORE (ST)

The memory operand is replaced by the accumulator operand, including the data flag bits.

STORE ROUNDED (SRD)

The operand is rounded before storing, but the accumulator is not changed.

ADD (+)

The memory operand is added algebraically to the accumulator operand, the sum replacing the accumulator contents.

ADD TO MAGNITUDE (+MG)

The memory operand is added algebraically to the magnitude of the accumulator operand, except that the accumulator is set to zero if the result attempts to change sign. The accumulator sign is ignored.

ADD TO MEMORY (M+)

The accumulator operand is added algebraically to the memory operand, the sum replacing the memory contents.

ADD MAGNITUDE TO MEMORY (M+MG)

The magnitude of the accumulator operand is added algebraically to the memory operand, except that the memory operand is set to zero if the result attempts to change sign.

COMPARE (K)

The accumulator operand is compared with the memory operand by algebraic subtraction; comparison indicators are set according to the result, but neither operand is changed.

COMPARE FIELD (KF) (VFL mode)
COMPARE MAGNITUDE (KMG) (FLP mode)

Same as COMPARE, except that the accumulator sign is ignored, and (in VFL) only a portion of the accumulator, equal in length to the memory, is compared.

COMPARE FOR RANGE (KR)

Used following COMPARE to determine whether the accumulator operand falls below (*accumulator low*), within (*equal*), or above (*high*) the range defined by the memory operands of the two instructions.

COMPARE FIELD FOR RANGE (KFR) (VFL mode)
COMPARE MAGNITUDE FOR RANGE (KMGR)(FLP mode)

Analogous to COMPARE FOR RANGE.

MULTIPLY (*)

The product of the memory and accumulator operands replaces the accumulator operand (see note).

LOAD FACTOR (LFT)

The memory operand is placed in the factor register, usually in preparation for MULTIPLY AND ADD.

MULTIPLY AND ADD (*+)

The product of the memory and factor-register operands is added algebraically to the accumulator operand (see note).

DIVIDE (/)

The accumulator operand (dividend) is divided by the memory operand (divisor), with the quotient replacing the accumulator operand and (in the VFL mode only) the remainder going to the remainder register (see note). (To obtain a remainder in floating-point division, use DIVIDE DOUBLE; see below.)

Note: In the decimal VFL mode, the operations MULTIPLY, MULTIPLY AND ADD, and DIVIDE are not executed directly, but operate like LOAD TRANSIT AND SET (see below) for execution by subroutine.

b. *Operations Available in Variable-field-length Mode Only*

ADD ONE TO MEMORY (M+1)

+1 or −1 is added algebraically to the memory operand, ignoring the accumulator.

COMPARE IF EQUAL (KE)
COMPARE FIELD IF EQUAL (KFE)

Same as COMPARE or COMPARE FIELD, respectively, except that the operation is performed only if the *accumulator equal* indicator is already on. It is used for multiple-field comparison.

LOAD TRANSIT AND SET (LTRS)

The memory operand is loaded into the transit register, and the offset field of the instruction is loaded into the all-ones counter for ready use as a pseudo operation code in interpretive fashion.

c. *Operations Available in Floating-point Mode Only*

RECIPROCAL DIVIDE (R/)

Same as DIVIDE, except that the operands are interchanged, the memory operand being the dividend and the accumulator operand the divisor.

STORE ROOT (SRT)

The square root of the accumulator operand is stored in memory.

LOAD DOUBLE (DL)

LOAD DOUBLE WITH FLAG (DLWF)

These are double-length operations similar to the single-length LOAD and LOAD WITH FLAG, except that an extra 48 bits to the right of the fraction being loaded are set to zero, whereas the single-length operations leave these bits unchanged.

ADD DOUBLE (D+)

ADD DOUBLE TO MAGNITUDE (D+MG)

MULTIPLY DOUBLE (D*)

Similar to ADD, ADD TO MAGNITUDE, and MULTIPLY, respectively, except that the fraction part of the accumulator operand is of double length (96 bits). (Note that floating-point MULTIPLY AND ADD is also a double-length operation.)

DIVIDE DOUBLE (D/)

Similar to DIVIDE, except that a 96-bit dividend is used and a remainder is produced and placed in the remainder register. Quotient and remainder are both of single length.

STORE LOW ORDER (SLO)

The low-order part of the double-length accumulator operand is stored in memory with the proper exponent.

ADD TO FRACTION (F+)

Same as ADD DOUBLE, except that the exponent of the accumulator operand is used as the exponent of both operands during addition.

SHIFT FRACTION (SHF)

The double-length fraction in the accumulator is shifted left or right by the amount specified in the address; the accumulator exponent is unchanged.

ADD TO EXPONENT (E+)

The exponent of the memory operand is added algebraically to the accumulator exponent.

ADD IMMEDIATE TO EXPONENT (E+I)

The address part of the instruction, interpreted as an exponent, is added algebraically to the accumulator exponent.

d. *VFL-arithmetic Modifiers and Addressing Modes*

Radix modifier (D, decimal; B, binary)
1: Arithmetic and data format are decimal.
0: Arithmetic and data format are binary.

Unsigned modifier (U)
1: The memory operand has no sign byte, and the operand is considered positive.
0: The memory operand has a sign byte.

Negative sign modifier (N)
1: The sign of the unreplaced operand is inverted.
0: The sign is used unchanged.

Immediate addressing (I)
The address part after indexing serves as the memory operand. This mode precludes progressive indexing.

Progressive indexing
The specified index value is used as the address of the memory operand of the VFL operation; this is followed by one of six immediate index-arithmetic operations (which see), as specified by a secondary operation code:

V + I   V − I
V + IC   V − IC
V + ICR   V − ICR

e. *Floating-point-arithmetic Modifiers*

Normalization modifier (N, normalized; U, unnormalized)
1: The result is left unnormalized.
0: The result is normalized automatically.

Absolute value modifier (A)
1: The sign of the memory operand is ignored, and the operand is considered positive.
0: The sign of the memory operand is used.
Note: This modifier is analogous to the VFL *unsigned* modifier.

Negative sign modifier (N)
Same as in VFL arithmetic.

Radix Conversion

a. *Operations*

LOAD CONVERTED (LCV)
The radix of the memory operand, considered as an integer, is converted and the result placed in the accumulator.

LOAD TRANSIT CONVERTED (LTRCV)

Same as LOAD CONVERTED, except that the result is placed in the transit register.

CONVERT (CV)

The accumulator operand, considered as an integer, is converted and the result returned to the accumulator. The binary operand corresponds in length and position to a single-length floating-point fraction.

CONVERT DOUBLE (DCV)

Same as CONVERT, except that the binary operand corresponds to a double-length fraction.

b. *Modifiers and Addressing Modes*

Same as in VFL arithmetic, except for

Radix modifier (D, decimal; B, binary)

Specifies the radix of the unconverted operand.
1: Conversion is from decimal to binary.
0: Conversion is from binary to decimal.

Connective Operations

a. *Operations*

CONNECT (C)

The memory operand is combined logically with the accumulator, according to the specified connective. The result replaces the accumulator operand. A left-zeros count and an all-ones count of the result are developed.

CONNECT TO MEMORY (CM)

Same as CONNECT except that the result replaces the memory operand.

CONNECT FOR TEST (CT)

Same as CONNECT except that the result is discarded after testing and both operands remain unchanged.

b. *Connective Code*

A 4-bit code $x_{00}\, x_{01}\, x_{10}\, x_{11}$ defines one of the sixteen connectives by listing the 4 result bits for each of the four states of a memory bit (m)

and the corresponding accumulator bit (*a*):

Operand bits		*Result bit*
m	*a*	
0	0	x_{00}
0	1	x_{01}
1	0	x_{10}
1	1	x_{11}

c. *Addressing Modes*

Immediate addressing
Progressive indexing
Same as in VFL arithmetic.

Index-arithmetic Operations

Note: Immediate index arithmetic, where the address serves as the (unsigned) operand, is distinguished from direct index arithmetic, where the (signed) operand is at the addressed location, by the operation code rather than by a modifier. Separate positive and negative immediate operations on the signed value field are provided because the operand is unsigned.

LOAD INDEX (LX)

The specified full word replaces the entire contents of the specified index register.

LOAD VALUE (LV)
LOAD VALUE IMMEDIATE (LVI)
LOAD VALUE NEGATIVE IMMEDIATE (LVNI)

The specified operand and sign replace the value field of the specified index register.

LOAD COUNT (IMMEDIATE) (LC or LCI)
LOAD REFILL (IMMEDIATE) (LR or LRI)

Replace the count or refill field, respectively.

STORE INDEX (SX)

The entire contents of the index register are stored at the specified location.

STORE VALUE (SV)
STORE COUNT (SC)
STORE REFILL (SR)

The value, count, or refill field, respectively, of the index register is stored in corresponding fields of the index word at the specified location.

ADD (IMMEDIATE) TO VALUE (V+ or V + I)

SUBTRACT IMMEDIATE FROM VALUE (V − I)

The specified operand is added to or subtracted from the value field.

ADD (IMMEDIATE) TO VALUE AND COUNT (V + C or V + IC)

SUBTRACT IMMEDIATE FROM VALUE AND COUNT (V − IC)

Same as above, and the count is reduced by 1.

ADD (IMMEDIATE) TO VALUE, COUNT, AND REFILL (V + CR or V + ICR)

SUBTRACT IMMEDIATE FROM VALUE, COUNT, AND REFILL (V − ICR)

Same as above and, if the count reaches zero, the word specified by the refill address replaces the contents of the index register.

ADD IMMEDIATE TO COUNT (C + I)

SUBTRACT IMMEDIATE FROM COUNT (C − I)

The address part is added to or subtracted from the count field.

COMPARE VALUE [(NEGATIVE) IMMEDIATE] (KV or KVI or KVNI)

The specified operand and sign are compared algebraically with the value field, and the index-comparison indicators are set.

COMPARE COUNT (IMMEDIATE) (KC or KCI)

The magnitude of the specified operand is compared with the count field, and the index-comparison indicators are set.

LOAD VALUE WITH SUM (LVS)

The value fields of all index registers, corresponding to 1 bits in the instruction address part, are added algebraically, the sum replacing the value field of a specified index register.

LOAD VALUE EFFECTIVE (LVE)

The effective address is used to fetch, eventually, a non-LVE instruction whose effective address replaces the value field of the specified index register.

STORE VALUE IN ADDRESS (SVA)

The value field of the index register is stored in the address part of the instruction at the specified location.

RENAME (RNX)

The contents of the specified index register are first stored at the address contained in the refill field of index register x0; the effective address of the RNX instruction is then loaded into the x0 refill field, and the specified index register is refilled from that address.

Branching Operations

a. *Unconditional Branching*

BRANCH (B)

The effective address of this instruction replaces the instruction-counter contents.

BRANCH RELATIVE (BR)

The effective address is added to the instruction-counter contents.

BRANCH ENABLED (BE)

Branch after enabling the interrupt mechanism.

BRANCH DISABLED (BD)

Branch after disabling the interrupt mechanism.

BRANCH ENABLED AND WAIT (BEW)

Same as BRANCH ENABLED, but no further instructions are executed until an interrupt occurs.

NO OPERATION (NOP)

Same as BRANCH to next instruction in sequence, regardless of the address part.

b. *Indicator Branching*

BRANCH ON INDICATOR (BIND)

Branch if specified indicator condition is satisfied.

On-Off modifier

1: Branch if indicator is *on* (1).

0: Branch if indicator is *off* (0).

Zero modifier

1: Set indicator to 0 after testing.

0: Leave indicator unchanged.

c. *Index Branching*

COUNT AND BRANCH (CB)

Reduce the count field of the specified index register by 1, and branch depending on whether the count has gone to zero or not; also increment the value field as specified.

COUNT, BRANCH, AND REFILL (CBR)

Same as COUNT AND BRANCH, but also refill the index register if the count has gone to zero.

On-Off modifier

1: Branch if count has gone to zero.

0: Branch if count has not gone to zero.

Advance modifiers

00: Leave value field unchanged.

01: Add ½ to value.

10: Add 1 to value.

11: Subtract 1 from value.

d. *Storing Instruction Counter*

STORE INSTRUCTION COUNTER IF (SIC)

If prefixed to any of the preceding *branch* instructions, store the instruction-counter contents at the specified location if the branch is successful.

e. *Bit Branching*

BRANCH ON BIT (BB)

Branch if the specified test bit meets the specified condition.

On-Off modifier

1: Branch if test bit is *on* (1).

0: Branch if test bit is *off* (0).

Zero modifier

1: Set test bit to 0 after testing.

0: Leave test bit unchanged.

Invert modifier

1: Invert test bit, after application of *zero* modifier.

0: Leave test bit unchanged.

Data-transmission Operations

TRANSMIT (T)

The contents of a first memory area are sent to and replace the contents of a second memory area.

SWAP (SWAP)

The contents of a first memory area are interchanged with the contents of a second memory area.

Immediate count modifier (I)

1: The number of words to be transmitted are specified in the instruction.

0: The number of words to be transmitted are specified in the count field of an index register.

Backward modifier (B)

1: Addresses are decreased by 1 for each word transmitted.

0: Addresses are increased by 1 for each word transmitted.

Input-Output Operations

WRITE (W)

Data are transmitted from memory to an input-output unit.

READ (RD)

Data are transmitted from an input-output unit to memory.

CONTROL (CTL)

Control information is sent from memory to an input-output unit.

LOCATE (LOC)

A selection address is sent to an input-output unit.

RELEASE (REL)

Any operation in progress for the specified channel is terminated immediately and status indications are reset.

Suppress end of operation modifier (SEOP)

Normal end-of-operation interrupt is suppressed after completion of any of the above five operations.

COPY CONTROL WORD (CCW)

The current control word for the specified channel is sent to memory.

Miscellaneous Operations

REFILL (R)

The index word at the specified memory address is replaced by the word located at the address contained in its refill field.

REFILL ON COUNT ZERO (RCZ)

A REFILL operation is performed only if the count field of the addressed index word is zero.

EXECUTE (EX)

At the specified address there is an operand which is executed as an instruction.

EXECUTE INDIRECT AND COUNT (EXIC)

At the specified address there is another address which is treated as a pseudo instruction counter: its operand is executed as an instruction, and the pseudo instruction counter is then advanced to the next instruction location.

STORE ZERO (Z)

Store an all-zero word at the specified full-word location.

A.5 Summary of Indicators

The indicator number is shown to the left of the name and the mnemonic abbreviation to the right in parentheses. The notation in brackets gives the class of indicator:

1 Interrupt mask bit always 1; always interrupts
m Interrupt mask bit set by programming
0 Interrupt mask bit always 0; never interrupts
P Permanent indicator; remains on until reset by interrupt or by programming
T Temporary indicator; corresponds to most recent result which affects it

Equipment Check

0. *Machine check* (MK) [1,P]
A general error has been detected by the CPU checking circuits.

1. *Instruction check* (IK) [1,P]
An error has been detected during the performance of the current instruction.

2. *Instruction reject* (IJ) [1,P]
The current instruction cannot be executed.

3. *Exchange control check* (EK) [1,P]
A general error has been detected by the exchange checking circuits.

Attention Request

4. *Time signal* (TS) [1,P]
The interval timer has gone to zero.

5. *CPU signal* (CPUS) [1,P]
A signal has been received from another, directly connected CPU.

Input-Output Reject

6. *Exchange check reject* (EKJ) [1,P]
An error was detected by the exchange while it was setting up the current input-output instruction.

7. *Unit not ready reject* (UNRJ) [1,P]
   The unit selected by the current input-output instruction was not ready to operate.
8. *Channel busy reject* (CBJ) [1,P]
   The channel selected by the current input-output instruction has not completed a previous instruction.

Input-Output Status

(Indicators 9 to 13 are used in conjunction with the *channel address* register, which contains the address of the input-output channel involved.)

9. *Exchange program check* (EPGK) [1,P]
   The exchange has terminated a previously initiated input-output operation because of a programming error.
10. *Unit check* (UK) [1,P]
    An error or malfunction has been detected by checking circuits at the unit or the channel.
11. *End exception* (EE) [1,P]
    The last operation for the channel encountered an exceptional condition.
12. *End of operation* (EOP) [1,P]
    The last operation for the channel was ended as specified by the instruction and its control words.
13. *Channel signal* (CS) [1,P]
    An attention-request signal has been received from the channel.
14. Reserved for future expansion.

Instruction Exception

15. *Operation code invalid* (OP) [1,P]
    An instruction was suppressed because the operation code or the modifiers were not valid.
16. *Address invalid* (AD) [1,P]
    An instruction was suppressed because the effective address was not valid.
17. *Unended sequence of addresses* (USA) [1,P]
    A one-instruction addressing or *execute* loop has been forcibly terminated after 1 millisecond (several hundred cycles).
18. *Execute exception* (EXE) [1,P]
    An *execute* operation was suppressed because it attempted to change the instruction counter.
19. *Data store* (DS) [1,P]
    An attempt to change the contents of a protected storage location, while the interrupt system was enabled, was suppressed.

20. *Data fetch* (DF) [m,P]
An attempt to fetch data from a protected storage location, while the interrupt system was enabled, is indicated, and, if the corresponding mask bit was 1, the data fetch was suppressed.

21. *Instruction fetch* (IF) [m,P]
An attempt to branch to an instruction at a protected location, while the interrupt system was enabled, is indicated, and, if the corresponding mask bit was 1, the operation was suppressed.

Result Exception

22. *Lost carry* (LC) [m,P]
A carry has been lost at the high end of the result.

23. *Partial field* (PF) [m,P]
An operation failed to use all of the significant operand bits.

24. *Zero divisor* (ZD) [m,P]
A *divide* operation with a zero divisor was suppressed.

Result Exception, Floating Point Only

25. *Imaginary root* (IR) [m,P]
The operand of a STORE ROOT operation was negative.

26. *Lost significance* (LS) [m,P]
An adding or shifting operation produced a result with a zero fraction and no overflow.

27. *Preparatory shift greater than* 48 (PSH) [m,P]
One operand in a FLP addition was shifted right, relative to the other operand, by more than 48 bits.

28. *Exponent flag positive* (XPFP) [m,P]
The result of a FLP operation had a positive exponent with an exponent flag of 1 propagated from an operand with an exponent flag of 1.

29. *Exponent overflow* (XPO) [m,P]
The positive result exponent has gone into the range $E \geqq +2^{10}$, generating an exponent flag of 1.

30. *Exponent range high* (XPH) [m,P]
The result exponent was in the range $+2^{10} > E \geqq +2^9$.

31. *Exponent range low* (XPL) [m,P]
The result exponent was in the range $+2^9 > E \geqq +2^6$.

32. *Exponent underflow* (XPU) [m,P]
The negative result exponent has gone into the range $E \leqq -2^{10}$, generating an exponent flag of 1.

33. *Zero multiply* (ZM) [m,T]
The result of a normalized FLP *multiply* operation was an order-of-

magnitude zero, with a zero fraction and no generated or propagated exponent underflow.

34. *Remainder underflow* (RU) [m,P]
The remainder after DIVIDE DOUBLE had a negative exponent $E \leqq -2^{10}$ and a generated exponent flag of 1.

Flagging

35. *Data flag T* (TF) [m,T]
36. *Data flag U* (UF) [m,T]
37. *Data flag V* (VF) [m,T]
Data flag *T*, *U*, or *V* of the current operand was *on*.
38. *Index flag* (XF) [m,T]
The index flag of the index word just modified was *on*.

Transit Operation

39. *Binary transit* (BTR) [m,P]
A binary VFL LOAD TRANSIT AND SET instruction was executed.
40. *Decimal transit* (DTR) [m,P]
A decimal VFL LOAD TRANSIT AND SET, MULTIPLY, MULTIPLY AND ADD, or DIVIDE instruction was executed.

Program

41 to 47. *Program indicator zero* to *six* (PG0 to PG6) [m,P]
These indicators are set by programming only.

Index Result

48. *Index count zero* (XCZ) [0,T]
The count field resulting from an index-word modification was zero.
49. *Index value less than zero* (XVLZ) [0,T]
50. *Index value zero* (XVZ) [0,T]
51. *Index value greater than zero* (XVGZ) [0,T]
The value field resulting from an index-word modification was less than zero, zero, or greater than zero, respectively.
52. *Index low* (XL) [0,T]
53. *Index equal* (XE) [0,T]
54. *Index high* (XH) [0,T]
An index *compare* operation showed the compared field in the specified index register to be lower than, equal to, or higher than the corresponding field at the effective address.

Arithmetic Result

55. *To-memory operation* (MOP) [0,T]
The operation just executed was of the *store* type.

56. *Result less than zero* (RLZ) [0,T]
The result of a data-arithmetic or radix-conversion operation was nonzero and negative.

57. *Result zero* (RZ) [0,T]
The result of a data-arithmetic, radix-conversion, or connective operation was zero.

58. *Result greater than zero* (RGZ) [0,T]
The result of a data-arithmetic, radix-conversion, or connective operation was nonzero and positive.

59. *Result negative* (RN) [0,T]
The result of a data-arithmetic or radix-conversion operation was negative, whether zero or not.

60. *Accumulator low* (AL) [0,T]
61. *Accumulator equal* (AE) [0,T]
62. *Accumulator high* (AH) [0,T]
A data-arithmetic *compare* operation showed the accumulator operand to be respectively lower than, equal to, or higher than the operand at the effective address.

Mode

63. *Noisy mode* (NM) [0,P]
When this indicator is *on,* all normalized FLP operations are performed in the noisy mode. (This indicator can be set only by programming.)

Appendix B
PROGRAMMING EXAMPLES

This appendix contains some short examples of programs essentially in machine language. The purpose here is not to teach programming, for a machine of this magnitude will always be programmed in symbolic form, nor is it claimed that these programs represent the best or the fastest method of solving each problem on the 7030. The purpose is merely to illuminate several features of the 7030 that are discussed in various chapters.

Notation

The following notation will be used in the examples. The notation is incomplete and does not cover some operations not used in the examples.

Integers

All integers are written in decimal form unless prefixed by a different radix in parentheses:

$$129 = (16)81 = (8)201 = (2)\ 1000\ 0001$$

Floating-point Numbers

An FLP number is written as a (signed) decimal fraction, followed by the letter E and a (signed) decimal integer representing the power of 2; + signs may be omitted:

0.5 E 0	(½)
0.8 E −4	(0.05)
−0.75 E 12	(−3,072)

The term XFNZERO denotes an *infinitesimal* (zero fraction, zero exponent, exponent sign negative, and exponent flag 1), which behaves arithmetically like a true zero. An alternative notation is 0.0 E −1024.

Addresses

Addresses are written as two decimal integers separated by a period. Thus 1257.48 is the address of bit 48 in word 1257 of memory. (These are *not* mixed decimal fractions.) Internal registers are referred to by the addresses listed in Appendix A.3. Index registers are referred to as x0 to x15 in the index addresses and as 16.0 to 31.0 in the operand address.

Short arithmetical expressions are to be evaluated with carries past the period being *modulo* 64:

$$\begin{aligned} 0.64 &= 1.0 \\ 1257.48 + 0.24 &= 1257.72 = 1258.08 \\ 3 * 7.48 &= 21.144 = 23.16 \end{aligned}$$

Half-length Instruction Format

$$\boxed{\text{O(M), J, A(I)}}$$

O Operation

M Mode symbols (M is replaced by one or more of the symbols listed below or omitted if there are none)

J *J*-index address (omitted in instructions that have none)

A Address

I *I*-index address (omitted if no address modification)

List of Mode Symbols:

FN FLP normalized

FU FLP unnormalized

F branch if indicator *off* (omitted for "branch if indicator *on*")

Z set indicator to 0 after test

$$\left.\begin{array}{ll} + & \text{add 1.0 to value} \\ \text{H} & \text{add 0.32 (half) to value} \\ - & \text{subtract 1.0 from value} \end{array}\right\}\text{for index branching only}$$

VFL Instruction Formats

$$\boxed{\text{O(M, L, BS), A(I), F(I)}}$$

O Operation

M Mode symbols (M is replaced by symbols listed below)

L Field length (1 to 64)

BS Byte size (1 to 8)

A Address

F Offset (0 to 127, may be omitted if 0)

I *I*-index addresses (there may be one for modifying the address and another for modifying the offset; either is omitted if not needed)

List of Mode Symbols:

VB	binary signed
VD	decimal signed
VBU	binary unsigned (the only one which applies to connectives)
VDU	decimal unsigned

Operation Codes and Suffixes

For greater clarity the operation codes are spelled out in the examples, although mnemonic symbols would ordinarily be used.

Operation modifiers are partly included in the mode symbols (above) and partly shown as suffixes to the operations. The suffixes may be one or more of the following:

NEGATIVE
ABSOLUTE
IMMEDIATE
COUNT
REFILL

Progressive indexing is shown by the addition of an *immediate indexing* code in parentheses after the operation:

(V + I)	add immediate to value
(V − I)	subtract immediate from value
(V + IC), (V − IC)	(same) and count
(V + ICR), (V − ICR)	(same), count, and refill

The connective-operation codes are followed by a 4-bit code to specify the connective (see Chap. 7); for example, CONNECT with the *and* connective is written

CONNECT 0001

Indicator branching operations will be written BRANCH IND where IND is replaced by the appropriate indicator abbreviation as listed in Appendix A.5. Thus BRANCH XCZ means *branch on index count zero.*

Data Formats

To distinguish program constants, etc., from instructions, one of these prefixes is used:

INDEX	index word consisting of *value, count, refill* separated by commas
VALUE	signed index value
DATA	any other data, such as a FLP number

B.1 Polynomial Evaluation (Table B.1)

The polynomial

$$p(x) = \sum_{k=0}^{m} a_k x^k$$

is best evaluated by the expression

$$p = (\ldots((a_m x + a_{m-1})x + a_{m-2})x + \ldots)x + a_0$$

using repeated floating-point multiplication and addition.

This example illustrates the universal-accumulator concept applied to floating-point arithmetic with simple indexing.

TABLE B.1. POLYNOMIAL EVALUATION

Location	*Statement*	*Notes*
Start		
100.0	LOAD INDEX, x1, 200.0	(1)
100.32	LOAD (FN), 301.0 (x1)	(2)
101.0	MULTIPLY (FN), 201.0	(3)
101.32	ADD (FN), 300.0 (x1)	(4)
102.0	COUNT AND BRANCH (−), x1, 101.0	(5)
102.32	STORE (FU), 202.0	(6)
103.0	BRANCH ENABLED AND WAIT, 103.0	Stop
200.0	INDEX, M.0 − 1.0, M, 200.0	
201.0	DATA, X	
202.0	DATA, P	
300.0	DATA, A0	
301.0	DATA, A1	
302.0	DATA, A2	
....	....	

Notes: (1) Set up index register 1.
(2) Load accumulator with initial $a_m = a_{1+m-1}$.
(3) Multiply accumulator contents (a_k) by x.
(4) Add a_{k-1}.
(5) Traverse loop m times, each time reducing index value by 1.0.
(6) Store result.

B.2 Cube-root Extraction (Table B.2)

The cube root

$$x = \sqrt[3]{N}$$

may be found by means of the recursion formula

$$x_{k+1} = x_k\left(\frac{1}{2} + \frac{3N/2}{2x_k{}^3 + N}\right)$$

Let N be a normalized FLP number with exponent P and fraction F. A suitable choice of a starting value x_0 will give a high accuracy in very few iterations. For example, a value of x_0 with exponent

$$p = \frac{P}{3} \text{ rounded to nearest integer in the positive direction}$$

and fraction

$$f_0 = 0.7109375 = 0.1011011 \text{ (binary)}$$

will give full 48-bit accuracy for any N in three iterations ($k = 3$), except for a possible rounding error in the last iteration. This value of p is the final exponent of the (normalized) result, and the f_0 value is selected to give about equal iteration errors at the extreme values of the final fraction.

A starting value with a fixed fraction was chosen for simplicity in the programming example. By a more elaborate formula[1] it is possible to choose a closer value of x_0 that will yield the desired accuracy by only one or two applications of the recursion formula. Such a program would be longer and somewhat faster.

This program shows an effective combination of VFL and FLP arithmetic.

[1] E. G. Kogbetliantz, "Computation of Sin N, Cos N and $\sqrt[m]{N}$ Using an Automatic Computer," *IBM J. Research and Development*, vol. 3, no. 2, pp. 147–152, April, 1959.

TABLE B.2. CUBE-ROOT EXTRACTION

Location	*Statement*	*Notes*
100.0	LOAD INDEX, x1, 200.0	Start
100.32	LOAD (FU), 204.0	
101.0	ADD IMMEDIATE TO EXPONENT (FU), −1	(1)
101.32	ADD (FN), 204.0	
102.0	STORE (FU), 203.0	(2)
102.32	LOAD (VB, 12, 1), 204.0, 117	(3)
103.32	ADD IMMEDIATE (VBU, 1, 8), 1, 117	(4)
104.32	DIVIDE IMMEDIATE (VBU, 2, 8), (2) 11, 116	(5)
105.32	STORE ROUNDED (VB, 12, 1), 8.0, 117	(6)
106.32	ADD TO FRACTION (FU), 202.0	(7)
107.0	STORE (FU), 205.0	(8)
107.32	MULTIPLY (FN), 205.0	
108.0	MULTIPLY (FN), 205.0	
108.32	ADD IMMEDIATE TO EXPONENT (FU), 1	(9)
109.0	ADD (FN), 204.0	
109.32	RECIPROCAL DIVIDE (FN), 203.0	(10)
110.0	ADD (FN), 201.0	(11)
110.32	MULTIPLY (FN), 205.0	
111.0	STORE (FU), 205.0	(12)
111.32	COUNT AND BRANCH, x1, 107.32	(13)
112.0	BRANCH ENABLED AND WAIT, 112.0	Stop
200.0	INDEX, 0.0, 3, 200.0	
201.0	DATA, 0.5 E 0	
202.0	DATA, 0.7109375 E 0	
203.0	DATA	
204.0	DATA, N	
205.0	DATA, X	

Notes: (1) Form $N/2$ by subtracting 1 from exponent.

(2) Place $3N/2$ in temporary storage.

(3) Treating exponent of N as a signed VFL number P, load magnitude into accumulator exponent position and exponent sign into sign register.

(4) Form $P + 1$ (to bias subsequent rounding operation in positive direction).

(5) Divide by 3 (binary 11). Offset is chosen to give signed quotient with one binary place to right of point for rounding. (Rest of accumulator, corresponding to FLP fraction magnitude, is cleared.)

(6) Form FLP number 0.0 E p, where $p = (P + 1)/3$ rounded to integer ($= P/3$ rounded with positive bias). Rounded, signed result is returned to accumulator exponent position (exponent sign replaces extra quotient bit). Fraction sign is immaterial.

(7) Form $x_0 = f_0$ E p.

(8) Store x_0 as first trial root.

(9) Form $2x_k^3$.

(10) Form $(3N/2)/(2x_k^3 + N)$.

(11) Add $\frac{1}{2}$.

(12) Store x_{k+1}.

(13) Traverse loop three times.

B.3 Matrix Multiplication (Table B.3)

An m-by-n matrix A and an n-by-p matrix B are multiplied to produce an m-by-p matrix C. Each element of A is a single FLP number, the elements being stored row by row at consecutive word locations starting with A.0. Similarly, matrixes B and C are stored row by row starting at B.0 and C.0, respectively. This program, which is essentially the matrix multiplication example of Table 11.5, illustrates indexing procedures.

TABLE B.3. MATRIX MULTIPLICATION

Location	*Statement*	*Notes*
100.0	LOAD INDEX, X1, 201.0	Start
100.32	LOAD INDEX, X2, 202.0	
101.0	LOAD INDEX, X3, 203.0	(1)
101.32	LOAD INDEX, X4, 17.0	
102.0	LOAD INDEX, X5, 18.0	(2)
102.32	LOAD DOUBLE (FU), 204.0	(3)
103.0	LOAD FACTOR (FN), 0.0 (X4)	
103.32	MULTIPLY AND ADD (FN), 0.0 (X5)	(4)
104.0	ADD IMMEDIATE TO VALUE, X5, P	(5)
104.32	COUNT BRANCH AND REFILL (+), X4, 103.0	(6)
105.0	STORE ROUNDED (FN), 0.0 (X3)	(7)
105.32	ADD IMMEDIATE TO VALUE, X3, 1.0	(8)
106.0	COUNT BRANCH AND REFILL (+), X2, 102.0	(9)
106.32	ADD IMMEDIATE TO VALUE, X1, N	(10)
107.0	COUNT BRANCH AND REFILL, X3, 101.32	(11)
107.32	BRANCH ENABLED AND WAIT, 107.32	Stop
201.0	INDEX, A.0, N, 17.0	i_{00}
202.0	INDEX, B.0, P, 202.0	j_{00}
203.0	INDEX, C.0, M, 203.0	k_0
204.0	DATA, XFNZERO	

Notes: (1) Load index register x1 (i_0) from i_{00}, x2 (j_0) from j_{00}, and x3 (k) from k_0.
(2) Load x4 (i) from x1 (i_0) and x5 (j) from x2 (j_0).
(3) Clear double-length accumulator before starting cumulative multiplication.
(4) Accumulate product element in accumulator.
(5) Increment j by p to advance to next column element of B.
(6) Increment i by 1 to advance to next row element of A. Traverse inner loop n times. At the end, reset i to i_0 to restart same row of A.
(7) Store product element.
(8) Increment k by 1 to advance to next product element.
(9) Increment j_0 by 1 to start next column of B. Traverse middle loop p times. At the end, reset j_0 to j_{00} to return to beginning of B.
(10) Increment i_0 by n to start next row of A.
(11) Traverse outer loop m times.

B.4 Conversion of Decimal Numbers to a Floating-point Normalized Vector (Table B.4)

A group of 25 decimal fixed-point numbers is to be converted to a normalized vector of 25 binary FLP numbers. The decimal numbers are positive, unsigned, and ten digits long. The decimal digits are expressed in a 6-bit code whose low-order 4 bits are the corresponding binary integers; thus the field length is 60 and the byte size 6. The decimal numbers are stored consecutively starting at address D. The vector is to be "normalized" by replacing each number F_k by the expression

$$\frac{F_k}{\sqrt{\sum_k F_k{}^2}}$$

(The meaning of the term *normalization* here differs from that used in describing FLP arithmetic.) The vector is to be stored in consecutive word locations starting at address F.0.

This example shows the use of radix conversion and progressive indexing combined with FLP operations.

TABLE B.4. CONVERSION OF DECIMAL NUMBERS TO A FLP NORMALIZED VECTOR

Location	*Statement*	*Notes*
100.0	LOAD INDEX, x1, 201.0	Start
100.32	LOAD INDEX, x2, 202.0	
101.0	LOAD (FU), 203.0	
101.32	STORE (FU), 204.0	(1)
102.0	LOAD CONVERTED (V + I) (VDU, 60, 6), 0.60 (x1), 68	(2)
103.0	STORE (FU), F.0 (x2)	(3)
103.32	MULTIPLY (FN), 8.0	(4)
104.0	ADD (FN), 204.0	
104.32	COUNT BRANCH AND REFILL (+), x2, 101.32	
105.0	STORE ROOT (FN), 204.0	(5)
105.32	LOAD (FN), F.0 (x2)	
106.0	DIVIDE (FN), 204.0	
106.32	STORE (FU), F.0 (x2)	(6)
107.0	COUNT AND BRANCH (+), x2, 105.32	
107.32	BRANCH ENABLED AND WAIT, 107.32	Stop
201.0	INDEX, D, 0, 201.0	
202.0	INDEX, 0.0, 25, 202.0	
203.0	DATA, XFNZERO	
204.0	DATA	

Notes: (1) Current ΣF_k^2 to temporary storage.

(2) Convert decimal integer to binary and place in FLP fraction position of accumulator, the exponent being zero. Progressive indexing is used to increment the index value by 0.60 after the present operand field is fetched.

(3) Store F_k temporarily in unnormalized form. (The exponent need not be adjusted to correspond to the actual decimal-point position of the original field, for the subsequent normalization process cancels out the exponent discrepancy.)

(4) Square F_k.

(5) Place $(\Sigma F_k^2)^{1/2}$ in temporary storage.

(6) Replace each F_k by normalized value.

B.5 Editing a Typed Message (Table B.5)

One of the chief uses for the logical-connective operations is in editing input and output data. Editing may cover a variety of different operations, and only a few of these will be illustrated in this skeletonized but useful example. It includes the various connective operations, the left-zeros count applied to indexing, zero tests, and a byte-size adjustment.

A message, which has been entered on a typewriter like the one on the 7030 console, is edited to delete control characters which appear in the coded message whenever a control key (such as carriage return) is struck. Deletion here means removing the control character and closing the gap (not just replacing the character with a *blank*). The number of control characters is not known in advance, and the length of the edited message must be determined by looking for an END code. The input message is stored in memory starting at address 300.0, and the block of edited output data is to be stored at address 400.0. For subsequent input-output operations it is necessary to fill any unused portion of the last word of the block with 0 bits. A control word for use with read-write instructions, containing in the count field the number of words in the output block, is to be made up and stored at 201.0.

In the 8-bit code used with the typewriter, all control bytes (other than *blank*) are distinguished from data by a 1 bit in the high-order position. The program shown tests this bit in eight characters at a time. The left-zeros count is used to locate the control byte (or the leftmost control byte if there are several among the eight). The control byte is then tested for the END code, which is 1111 1110. Advantage is taken of the coincidence that the complement of this code is a single 1 bit, which, by a suitable offset, serves as the mask to isolate the high-order bit in the previous test for all control characters. (Such a short cut is not necessarily sound programming, but it offers here an additional opportunity to demonstrate the flexibility of the VFL system in general and of the connective operations in particular.)

TABLE B.5. EDITING A TYPED MESSAGE

Location	*Statement*	*Notes*
100.0	LOAD INDEX, x1, 0.0	Start
100.32	LOAD INDEX, x2, 0.0	
101.0	CONNECT IMMEDIATE 0011 (VBU, 9, 1), (2) 1 1111 1111, 63	(1)
102.0	CONNECT 0011 (VBU, 64, 8), 300.0 (x1)	(2)
103.0	CONNECT TO MEMORY 0101 (VBU, 64, 8), 400.0 (x2)	(3)
104.0	CONNECT FOR TEST 0001 (VBU, 64, 8), 8.0	(4)
105.0	ADD TO VALUE, x1, 7.17	
105.32	ADD TO VALUE, x2, 7.17	(5)
106.0	BRANCH RZ, 102.0	(6)
106.32	CONNECT FOR TEST 1001 (VBU, 8, 8), 300.0 (x1), 71	(7)
107.32	ADD TO VALUE, x1, 200.0	(8)
108.0	BRANCH RZ (F), 102.0	(9)
108.32	CONNECT TO MEMORY 0000 (VBU, 56, 8), 400.0 (x2)	(10)
109.32	ADD TO VALUE, x2, 200.32	(11)
110.0	LOAD INDEX, x3, 201.0	
110.32	LOAD COUNT, x3, 18.0	(12)
111.0	STORE INDEX, x3, 201.0	
111.32	BRANCH XCZ, 112.32	(13)
112.0	BRANCH ENABLED AND WAIT, 112.0	Stop
112.32	BRANCH ENABLED AND WAIT, 112.32	Error
200.0	VALUE, 0.08	
200.32	VALUE, 0.56	
201.0	INDEX, 400.0, 0, 201.0	

Notes: (1) The operand, specified by immediate addressing, is the 9-bit field 1 1111 1111. Specifying a byte size of 1 causes each 1 to be expanded to an 8-bit byte 0000 0001. The full operand, therefore, consists of nine such bytes. The connective 0011 and the offset of 63 then cause the left half accumulator to be filled with the pattern 1000 0000 1000 0000 .... (The final 1, which spills into the right half, is not used.)

(2) 64 bits (eight bytes) of input data are placed into the right half accumulator.

(3) These 64 bits are tentatively stored in the output area.

(4) The data field is *and*ed with the test pattern in the left half accumulator. The left-zeros count register contains either 64 or the position of the first "control byte" with a high-order 1 bit.

(5) The left-zeros count is added to both the input and output indexes.

(6) Branch if the *result zero* indicator is *on*, i.e., if there is no control byte.

(7) A 0000 0001 byte from the test pattern is matched against the control byte; if the control byte is an END code, all result bits are 0.

(8) In any case, skip the control byte in the input data by adding 0.08 to the index value.

(9) Branch to the beginning of the loop if the *result zero* indicator is *off* after

the last test (*not* END), thus starting with the 64 bits following the control byte, which may include several bytes transferred previously but which now must be offset by one byte. (Multiple control bytes in a 64-bit field will be taken care of one at a time.)

(10) Enough 0s are inserted to fill the last word of the block.

(11) The output index value is rounded up to the next full-word address.

(12) The index value from x2 is transferred to the count field of the control word being made up in x3, dropping the bit-address portion and leaving only the number of full words in the block.

(13) Test for a zero index count, which could result from an END code in the first data byte and be interpreted as a word count of 2^{18} at the output.

B.6 Transposition of a Large Bit Matrix (Table B.6)

TABLE B.6. TRANSPOSITION OF A LARGE BIT MATRIX

Location	*Statement*	*Notes*
100.0	LOAD INDEX, x1, 201.0	Start
100.32	LOAD INDEX, x2, 202.0	
101.0	LOAD INDEX, x3, 203.0	
101.32	CONNECT 0011 (V + I) (VBU, 1, 1), N.0 (x1), 63 (x3)	(1)
102.32	SUBTRACT IMMEDIATE FROM VALUE COUNT AND REFILL, x3, 0.32	(2)
103.0	BRANCH XCZ (F), 101.32	(3)
103.32	STORE (V + IC) (VBU, 64, 8), 1.0 (x2)	(4)
104.32	BRANCH XCZ (F), 101.32	(5)
105.0	LOAD COUNT IMMEDIATE, x2, N	(6)
105.32	ADD TO VALUE AND COUNT, x1, 204.0	(7)
106.0	BRANCH XCZ (F), 101.32	
106.32	BRANCH ENABLED AND WAIT, 106.32	Stop
201.0	INDEX, A.0, 64*N, 201.0	
202.0	INDEX, B.0, N, 202.0	
203.0	INDEX, 0.0, 64, 203.0	
204.0	VALUE, −N*N*64.0 + 0.01	

Notes: (1) Assemble 64 successive column bits in the right half accumulator by indexing the offset from 63 to 0.

(2) 0.32 means a 1 in the low-order bit of the 19-bit address in this instruction, which matches the low-order position of the offset field; thus the effective offset will be reduced by 1.

(3) Branch if the index count is not zero.

(4) Store 64 row bits of the transposed matrix.

(5) Traverse the loop n times.

(6) Reset x2 count to n.

(7) Advance the column index to the start of the next column by subtracting the length of the column ($64n^2$ words) and adding 0.01.

In a computer with efficient bit-handling facilities, bit matrixes can be useful tools. For instance, computing time and storage space for sparse matrixes can be saved by storing only the nonzero elements in consecutive locations and using bit matrixes to indicate the position of successive zero and nonzero elements in the complete matrix. The present example of transposing such a bit matrix illustrates the use of bit address and offset indexing with VFL operations (see Table B.6).

A square $64n$-by-$64n$ bit matrix beginning at address A.0 is to be transposed and stored at nonoverlapping addresses starting at B.0. The technique chosen here is to assemble 64 successive bits of a column into a 64-bit word in the accumulator and then store that word in a row of the transposed matrix.

A more efficient, but longer, program can be written by making use of the bit-interleaving ability of the connective operations. The core of such a program is the transposition of a small 8-by-8 matrix within the accumulator; this is done in eight steps, 8 bits at a time (see Table B.7). By dividing the larger matrix into a group of 8-by-8 submatrixes, each submatrix may be transposed separately and stored at the mirror-image position with respect to the main diagonal of the full matrix. The full program, not shown here, would be almost four times as fast as that of Table B.6.

TABLE B.7. TRANSPOSITION OF AN 8-BY-8-BIT MATRIX

Location	*Statement*	*Notes*
100.0	LOAD INDEX, X1, 201.0	Start
100.32	LOAD INDEX, X2, 202.0	
101.0	STORE ZERO, 9.0	(1)
101.32	CONNECT 0111 (V + I) (VBU, 8, 1), 0.08 (X1), 7 (X2)	(2)
102.32	SUBTRACT IMMEDIATE FROM VALUE AND COUNT, X2, 0.32	
103.0	BRANCH XCZ (F), 101.32	
103.32	STORE (VBU, 64, 8), A.0	(3)
104.32	BRANCH ENABLED AND WAIT, 104.32	Stop
201.0	INDEX, A.0, 0, 201.0	
202.0	INDEX, 0.0, 8, 202.0	

Notes: (1) Clear right half accumulator.
(2) *Or* 8 bits from A.0 into the accumulator, 8 bits apart.
(3) Store transposed matrix back into A.0.

INDEX