BASIC THEORY OF COMPUTERS

AN/FSQ-7
COMBAT DIRECTION CENTRAL

1 April 1957

MILITARY PRODUCTS DIVISION
INTERNATIONAL BUSINESS MACHINES CORPORATION
KINGSTON, NEW YORK

COPY NO. 874

LIST OF REVISED PAGES

INSERT LATEST REVISED PAGES. DESTROY SUPERSEDED PAGES.

NOTE: The portion of the text affected by the current revision is indicated by a vertical rule in the left margin of a left-hand page and in the right margin of a right-hand page.

*The asterisk indicates pages revised, added or deleted by the current revision.

CONTENTS

CONTENTS (cont'd)

CONTENTS (cont'd)

CONTENTS (cont'd)

CONTENTS (cont'd)

CONTENTS (cont'd)

CONTENTS (cont'd)

CONTENTS (cont'd)

CONTENTS (cont'd)

CONTENTS (cont'd)

CONTENTS (cont'd)

CONTENTS (cont'd)

LIST OF ILLUSTRATIONS

LIST OF ILLUSTRATIONS (cont'd)

LIST OF ILLUSTRATIONS (cont'd)

LIST OF ILLUSTRATIONS (cont'd)

LIST OF ILLUSTRATIONS (cont'd)

LIST OF ILLUSTRATIONS (cont'd)

LIST OF TABLES

LIST OF RELATED MANUALS

Manual	*Title*
	THEORY OF OPERATION
T.O. 31P2-2FSQ7-12	Introduction to AN/FSQ-7, Combat Direction Central
T.O. 31P2-2FSQ7-22	Basic Circuits for AN/FSQ-7 Combat Direction Central
T.O. 31P2-2FSQ7-32	Theory of Operation of Central Computer for AN/FSQ-7 Combat Direction Central
T.O. 31P2-2FSQ7-42	Theory of Operation of Drum System for AN/FSQ-7 Combat Direction Central
T.O. 31P2-2FSQ7-52	Theory of Operation of Input System for AN/FSQ-7 Combat Direction Central
T.O. 31P2-2FSQ7-62	Theory of Operation of Display System for AN/FSQ-7 Combat Direction Central
T.O. 31P2-2FSQ7-72	Theory of Operation of Output System for AN/FSQ-7 Combat Direction Central
T.O. 31P2-2FSQ7-82	Theory of Operation of Power Supply System for AN/FSQ-7 Combat Direction Central
T.O. 31P2-2FSQ7-92	Theory of Operation of Marginal Checking for AN/FSQ-7 Combat Direction Central
T.O. 31P2-2FSQ7-102	Theory of Operation of Warning Light System for AN/FSQ-7, Combat Direction Central
T.O. 31P2-2FSQ7-112	Theory of Programming for AN/FSQ-7, Combat Direction Central
	INSTALLATION
T.O. 31P2-2FSQ7-5	Installation of AN/FSQ-7 Combat Direction Central
	OPERATING PROCEDURE
T.O. 31P2-2FSQ7-21 – T.O. 31P2-2FSQ7-122	Operating Procedure and Operating Procedure for Maintenance for AN/FSQ-7 Combat Direction Central
	MAINTENANCE
T.O. 31P2-2FSQ7-132	Introduction and Philosophy of Maintenance for AN/FSQ-7 Combat Direction Central

LIST OF RELATED MANUALS (cont'd)

Manual	*Title*
T.O. 31P2-2FSQ7-142	Maintenance Techniques and Procedures of Central Computer for AN/FSQ-7 Combat Direction Central
T.O. 31P2-2FSQ7-152	Maintenance Techniques and Procedures of Drum System for AN/FSQ-7, Combat Direction Central
T.O. 31P2-2FSQ7-162	Maintenance Techniques and Procedures of Input System for AN/FSQ-7 Combat Direction Central
T.O. 31P2-2FSQ7-172	Maintenance Techniques and Procedures of Output System for AN/FSQ-7 Combat Direction Central
T.O. 31P2-2FSQ7-192	Maintenance Techniques and Procedures of Power Supply and Marginal Checking for AN/FSQ-7, Combat Direction Central
T.O. 31P2-2FSQ7-202	Maintenance Techniques and Procedures of Warning Light System for AN/FSQ-7 Combat Direction Central
	SCHEMATICS
T.O. 31P2-2FSQ7-212	Schematics for Central Computer of AN/FSQ-7 Combat Direction Central
T.O. 31P2-2FSQ7-222	Schematics for Drum System of AN/FSQ-7 Combat Direction Central
T.O. 31P2-2FSQ7-232	Schematics for Input System of AN/FSQ-7 Combat Direction Central
T.O. 31P2-2FSQ7-242	Schematics for Output System of AN/FSQ-7 Combat Direction Central
T.O. 31P2-2FSQ7-252	Schematics for Display System of AN/FSQ-7 Combat Direction Central
T.O. 31P2-2FSQ7-262	Schematics for Power Supply and Marginal Checking of AN/FSQ-7, Combat Direction Central
T.O. 31P2-2FSQ7-272	Schematics for Warning Lights of AN/FSQ-7, Combat Direction Central
	PLUGGABLE UNITS
T.O. 31P2-2FSQ7-282	Pluggable Units for AN/FSQ-7, Combat Direction Central
	PARTS CATALOG
T.O. 31P2-2FSQ7-4	Illustrated Parts Breakdown for AN/FSQ-7 Combat Direction Central

LIST OF RELATED MANUALS (cont'd)

Manual	*Title*
	SPECIAL TEST EQUIPMENT
T.O. 31P2–2FSQ7–31	Test Set Memory Driver Panel TS-986/FSQ
T.O. 31P2–2FSQ7–41	Test Set Plug In Units TS-985/FSQ
T.O. 31P2–2FSQ7–51	Power Supply PP-15819/FSQ
T.O. 31P2–2FSQ7–71	Test Set, Amplifier TS-988/FSQ
T.O. 31P2–2FSQ7–81	Dummy Load DA-153/FSQ
T.O. 31P2–2FSQ7–91	Test Set, Metallic Rectifier TS-989/FSQ
T.O. 31P2–2FSQ7–101	Test Set, Diode Semi-conductor Device TS-990/FSQ
T.O. 31P2–2FSQ7–111	Calibrator, Oscilloscope FR-112/FSQ
T.O. 31P2–2FSQ7–121	Marginal Check Control, Remote (C-2022/FSQ)
T.O. 31P2–2FSQ7–131	Test Set, Electron Tube TV-11/FSQ
T.O. 31P2–2FSQ7–141	Distribution Box (J-779/FSQ)
T.O. 31P2–2FSQ7–151	Dynamic Timer Power Pack

PART 1
INTRODUCTION

CHAPTER 1
PURPOSE AND PLAN OF MANUAL

1.1 GENERAL

High-speed digital computers such as those used in the SAGE System, are complex machines, each of which may have more than a million electrical and electronic parts. A maintenance man cannot service such a machine properly without understanding how it works. This understanding should not be confined to one specific digital computer model because computer designs are continually being refined. A digital computer maintenance man, therefore, needs a general knowledge of digital computer design and operation. He must know what digital computers are, what they do, and how they do it.

The subject of digital computers, however, covers a large number of different machines and to discuss all of them in detail would be impractical. The information in this manual, therefore, is developed along general lines. Since this manual is intended for personnel responsible for maintaining high-speed military digital computers such as the AN/FSQ-7 and AN/FSQ-8, the characteristics of this type of computer are emphasized.

1.2 DIVISION OF MANUAL INTO PARTS

The manual is divided into parts so that information may be presented in a logical sequence. This sequence begins with basic general ideas and continues toward specific details.

Part 1 of the manual gives a general introduction to digital computers, presenting background information that ties together the details discussed in later parts. The background material begins with a brief survey of computing machines in general, since digital computers comprise only one class of computing machines. This survey defines computing machines and gives a brief history of their development. The basic elements which make up a digital computer are described; then an example is given to show how a digital computer would be controlled and how it would operate when solving a simple problem.

A digital computer works by performing certain arithmetic operations on digits. Part 2 describes the number systems and arithmetic basic to digital computers.

High-speed digital computers are constructed from special electronic and magnetic components. Part 3 explains how these components perform the basic tasks required of a digital computer. The descriptions are followed by brief circuit analyses that detail the operation of the electronic and magnetic components.

The material in Parts 1 through 3 provides a basic knowledge of the "building-blocks" used in constructing digital computers. How these "building-blocks" are put together to form a typical computing system is shown in Part 4. The system selected for this explanation is typical of many digital computers, but it is pertinent to the AN/FSQ-7.

Part 5 explains programming—how computers are controlled so that they perform specific operations. Since the subject of programming is closely related to computer capacities and limitations, these also are discussed briefly in Part 5.

CHAPTER 2

THE NATURE AND FUNCTIONS OF COMPUTING MACHINES

2.1 WHAT COMPUTING MACHINES ARE

2.1.1 Definition

Man uses numerous tools to simplify or speed up his tasks. As the activities required in business or war become more complex and more dependent on speed in handling data, man's dependence on tools to help him in such activities becomes acute. An important tool which man uses to simplify and speed up his handling of data is the computing machine.

A very large computing machine is shown in figure 1–1. Such a computing machine is actually a data processing device, that is, a device that performs mathematical and logical operations on data in a prearranged and controlled manner. To perform these operations, computing machines must be able to: (1) accept the items of data that are presented to them, (2) manipulate these items in a desired prearranged manner, and, (3) make the manipulated data available in useful form.

2.1.2 Example of Machine Data Processing

A simple example of data processing is selecting the largest of three numbers. If a man were asked to select the largest of three numbers, he could do so by comparing any two of the three, noting which of these two is larger, and then comparing it with the third. The second comparison would show which number was

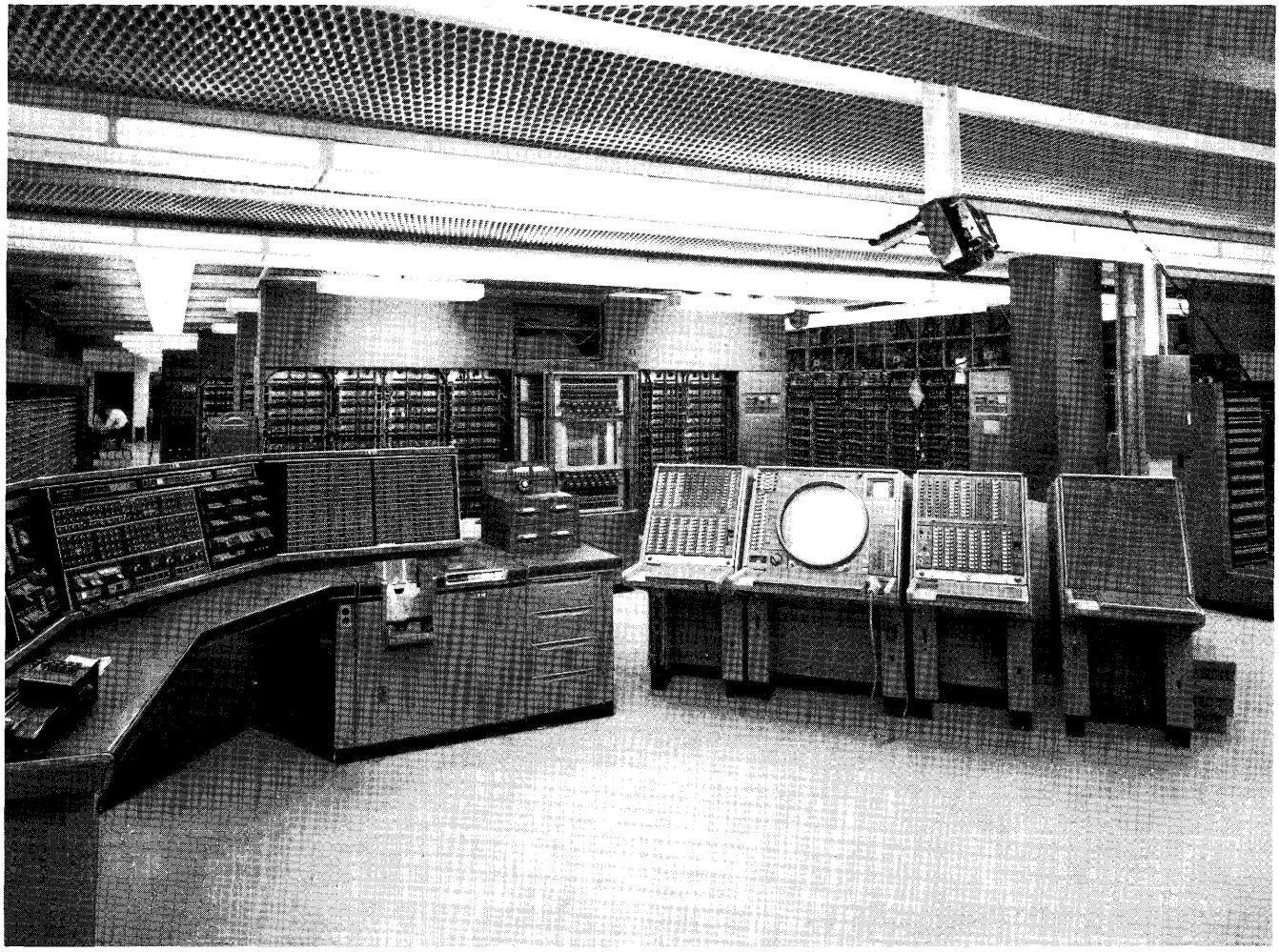


Figure 1–1. The AN/FSQ-7 (XD-1)

the largest of the three. In this problem, then, the data to be processed are the three numbers, and the manipulations of the data are the operations of comparing the numbers and selecting the largest.

To perform the same data-processing task, a computer would follow the same general procedure of (1) comparing two of the quantities, (2) selecting the larger, and then (3) comparing it with the third. In some ways, however, the procedure of the computer would differ from that of a man.

Unlike a man, a computer must perform a computation in order to select the larger of two quantities. In general, a man can tell by direct inspection which of two quantities is the larger; a computer cannot. A computer, however, can distinguish the difference between a plus (positive) and a minus (negative) quantity. Therefore, when comparing two quantities, a computer subtracts one from the other and notes whether the result is plus or minus. If the result is plus, the number subtracted is the smaller; if the result is minus, the number subtracted is the larger. Thus, a computer compares quantities by performing an arithmetic operation on them. The process is shown in figure 1–2 as it would be executed by a computer.

2.1.3 Arithmetic and Control Operations

To accomplish the typical data-processing task just discussed, a computer would perform two subtractions in sequence. In order for these two subtractions to lead to the desired result—selection of the largest of three quantities—the computer must follow a procedure based on certain control operations. The control operations are as follows:

a. Selecting any two of the quantities for the first subtraction.

b. Determining from the result of the first subtraction which of the two quantities is the larger.

c. Selecting the correct quantities for the second subtraction.

d. Determining from the result of the second subtraction which quantity is the largest of the three.

All such control operations are based on specific rules. Thus, in the preceding example, the computer selects the larger of two numbers, A and B, according to this rule: If subtracting A from B results in a plus quantity, select B; if the result is minus, select A. After selecting the larger number according to this rule, the computer follows a second rule to decide what quantities it should operate on for the second subtraction: If A is larger than B, subtract A from C; if B is larger than A, subtract B from C. From the plus or minus result of this operation, the final answer is obtained.

In the example, the control operations determine which quantities the computer operates on in performing the two subtractions and, also, what the computer does with the results of each subtraction. The example shows a typical combination of control and arithmetic operations in the processing of data. In general, control operations in computer data processing determine four fundamental variables:

a. What quantities the computer manipulates by arithmetic operations

b. What arithmetic operations the computer performs

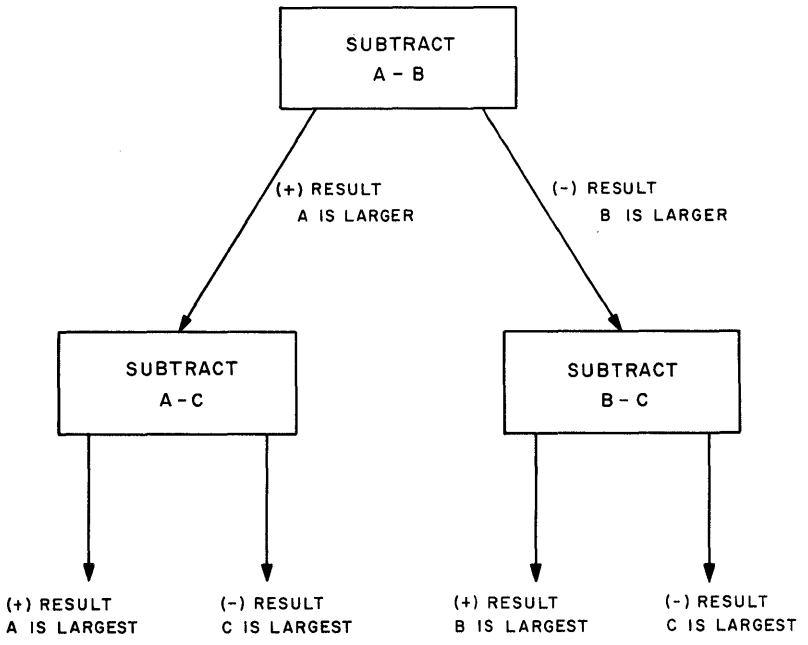


Figure 1–2. Data Processing — Finding the Largest of Three Numbers, A, B, and C

c. What sequence the computer follows in performing the arithmetic operations

d. What the computer does with the results of each arithmetic operation

Control operations are sometimes called logical operations because they select and arrange the steps in a given task of data processing in strict accord with logical rules for the task.

2.2 THE NEED FOR HIGH-SPEED COMPUTERS

2.2.1 General

If data-processing tasks were confined to simple problems such as the one discussed in the preceding section, there would be no need for complex, high-speed computers. In a small office, for example, where the only data-processing tasks are simple bookkeeping procedures, small, hand-operated computing machines (e.g., adding machines) are sufficient. The case is different when enormous quantities of data must be continually processed with extreme speed and accuracy. In such a case, it becomes practical to use a data-processing machine in place of a group of men working with pencil and paper or with small, hand-operated calculating machines.

2.2.2 Air Defense Needs

In the present-day air-defense system of the United States, huge quantities of data must be continually searched out and accurately processed at high speed. For example, when a strange aircraft is detected by radar, it must be identified as friendly or hostile. To accomplish this, the flight plans of all commercial airlines and friendly aircraft must be searched out and compared with the movement of the detected aircraft. At the same time, data must be calculated to determine precisely when and how the aircraft (or missile) is to be intercepted if it is found to be hostile. Naturally, if all these calculations are to be useful, they must be completed before the aircraft disappears. As the speed of aircraft increases far beyond 1000 miles per hour (mph), it becomes impractical for aircraft identification and interception to depend on a group of men who thumb through flight plans, ballistic tables, etc. It is absolutely necessary, therefore, to have a high-speed device that can accept, store, and process very large quantities of data rapidly and accurately and deliver the correct output information. Only such a high-speed device can provide correlated data quickly enough for use in making the correct tactical decisions necessary for air defense.

The large high-speed, electronic computing machines of today are well suited to this task. Such machines can multiply 6-digit numbers at speeds as high as 60,000 multiplications per second, with only one error in every 10 billion operations. Furthermore, digital computers can store large quantities of data and can locate and process needed items in a fraction of a second.

2.3 BASIC CLASSES OF COMPUTING MACHINES

2.3.1 Basis of Classification

Computers are classified, according to their basic principles of operation, as digital or analog.

2.3.2 Digital Computers

A digital computer is a computing machine that processes data expressed as digits or numbers, and manipulates the data by means of arithmetic or logical control operations in a predetermined manner, and generally delivers the resulting information in the form of digits. For instance, the number 34 might be represented in a digital computer as 3+4 pulses on a line as in A, figure 1–3.

A digital computer operates on data in much the same way that a man would manipulate the data in carrying out arithmetic computations with pencil and paper. Similar to a man making an arithmetic computation, a digital computer manipulates digits in a sequence of distinct steps determined by certain mathematical rules.

The digits used to represent items of data or specific instructions for processing the data must belong to a particular number system (such as the familiar decimal system) chosen for the computer model being used. Similarly, the results of operations by a digital computer are usually delivered in the form of numbers.

Since a digital computer operates on data in a series of distinct steps, there is necessarily some delay between the start and the completion of each operation. Furthermore, if a series of arithmetic operations are to be performed, each operation must be completed in turn before the next is begun. Nevertheless, the use of electronic and magnetic elements in digital computers permits these machines to perform thousands of operations in an extremely short time—some operations require only a few microseconds (usec).

In summary then, digital computers have the following basic characteristics:

a. All data handled by the computer must be in the form of digits of a particular number system.

b. The computer processes data by performing predetermined arithmetic and logical control operations on the digits. These operations are performed in discrete steps, much as arithmetic operations are performed with pencil and paper.

2.3.3 Analog Computers

An analog computer, unlike a digital computer, is a computing machine in which data are converted, for purposes of computation, not into digits but into physically measurable quantities such as lengths, voltages, or

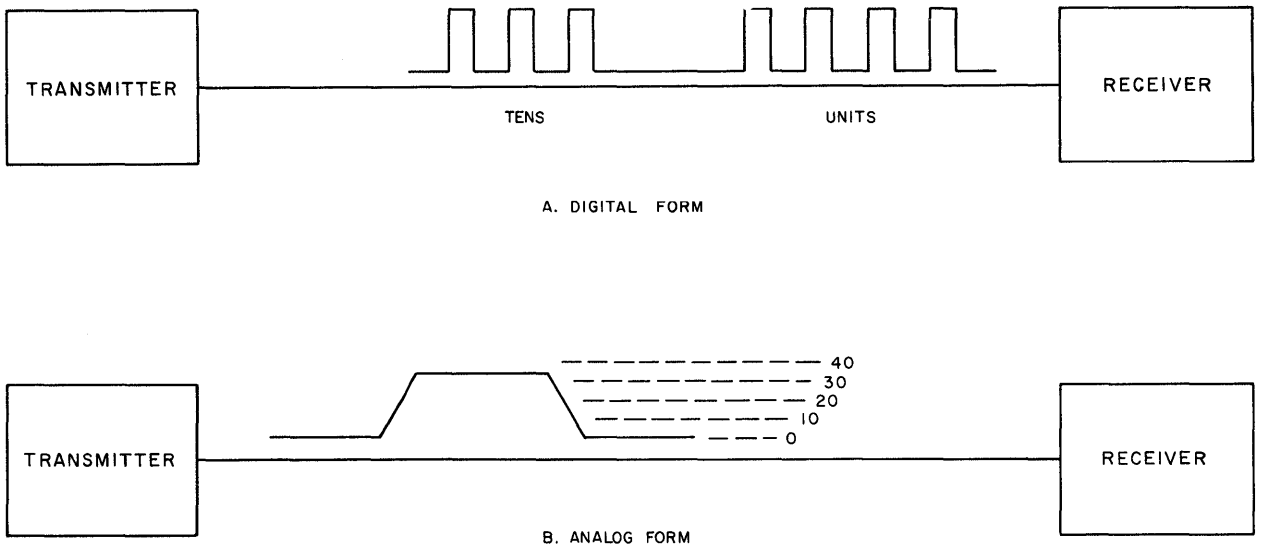


Figure 1–3. Digital and Analog Representations of the Number 34

angles (as shown in B, fig. 1–3). Computed results are obtained by the action of moving parts or electrical signals. These actions or signals do not represent digits. Rather, they are related to one another in such a way as to represent the relationships among the terms of a mathematical equation. They also interact with one another in such a way as to represent the mathematical operations indicated in the equation.

In other words, an analog computer solves problems by causing physical quantities to vary in a manner analogous to the way in which the variables in a problem change. For example, if distance equals velocity multiplied by time, a motor running at a speed proportional to velocity during a given time interval will turn a gear train through an angle proportional to distance. Thus, a continuous solution of distance in the equation: Distance = Velocity x Time, may be obtained. Action of this kind is typical of the manner in which analog computers solve problems. A fundamental characteristic of analog computers is that they provide continuous solutions to a given problem.

2.3.4 Physical Size of Components

The physical size of a computing machine is determined to a great degree by the job which it is to do. Hence, a simple calculator that is hand-operated and used only to add or subtract groups of numbers may be quite small. However, a data-processing machine such as the AN/FSQ-7, which must automatically store and process enormous quantities of data, is very large. An AN/FSQ-7 fills a building several stories high (see fig. 1–1).

2.4 HISTORY OF COMPUTERS

2.4.1 Early Computing Machines

The first counting aids used by man probably consisted of fingers, pebbles, or other similar items. One of the earliest "machines" is the abacus, which evolved from the use of pebbles. This device, shown in figure 1–4, is one of the simplest forms of an adding or counting machine. It consists of a series of rods on which the positioning of beads records the numbers 0 through 9. Addition or subtraction can be accomplished on each bar individually. However, the carrying of the 1 when a sum is greater than 9 cannot be done automatically.

The first machine that made provisions for automatic carrying of digits when the sum of a column is greater than 9 was the Pascal machine, invented in 1642. This has been termed the first authentic accounting machine and it was used to figure English currency. The machine was basically a hand-operated, gear-driven counter. Addition was accomplished by turning input wheels a distance equal to the money to be added. This is similar to the addition of mileage on an odometer.

The basic forerunner of the modern large-scale computers was the Babbage Analytical Engine, conceived by Charles Babbage in 1833. This machine, which operated somewhat similarly to a device called the Jaquard Loom, made use of cards and strips of metal with various holes punched in them to record numbers. A number was represented by an equivalent number of holes. After the Babbage machine, several improved types of computing machines were developed. Notable among them was the Hollerith machine, which used the Jacquard idea of holes punched in tape or cards. However, in the Hollerith machine these holes controlled electrical mechanisms.

2.4.2 Advent of Large, High-Speed Computers

The first of the large-scale, high-speed computing machines was the Mark I, completed in 1944 by Harvard University and International Business Machines. This

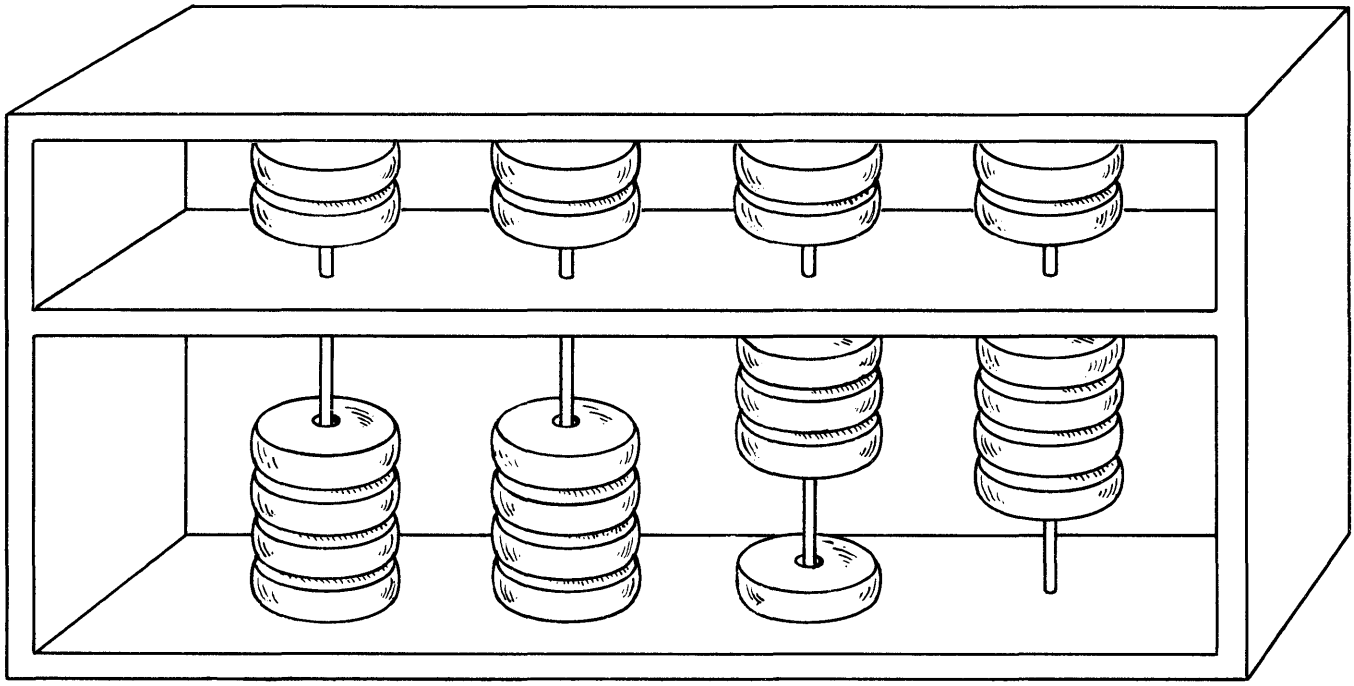


Figure 1–4. The Abacus – The Number Represented by the Position of the Beads is 34

machine uses the IBM punched-card method to insert the input data. Its output is typed out by an electric typewriter. The sequence of operations of the Mark I is controlled automatically. The machine can add, subtract, multiply, divide, or perform other related arithmetic operations. It is primarily a relay-operated device.

The Harvard Mark I was highly successful, but relay operation was undesirably slow. The first all-electronic digital computer was the ENIAC, built by the University of Pennsylvania in 1946. This used 18,000 vacuum tubes and could add two 10-digit decimal numbers in 200 microseconds, or multiply them together in 2 to 3 milliseconds.

Most of the refinements in recent years, as exemplified by the IBM model 700 series (the first mass-produced digital computers) and the AN/FSQ-7, have been concerned with increasing the amount of information which can be stored in the machines and increasing the speed at which the machines operate. The relative slowness of relays, which are used in the Mark I (and Mark II), cannot be tolerated when processing data for the air-defense system. It is necessary to use special high-speed electronic circuits, which operate very much faster than relays. In addition, it is necessary to use special methods of data storage, that permit extremely high-speed insertion and extraction of data.

CHAPTER 3

ELEMENTS AND COMPONENTS REQUIRED BY DIGITAL COMPUTERS

3.1 THE LANGUAGE USED BY DIGITAL COMPUTERS

The task of a digital computer is to process data, expressed in the form of digits, by performing certain predetermined arithmetic and control operations on the data. These operations are predetermined by a set of instructions called the *program*, which has been previously made up by the operator.

In some computers, the instructions may be in the form of special wiring (i.e., of control circuits of the machine). Such a computer would use a control panel somewhat similar to a telephone switchboard to direct operations. This is called a control panel program. In stored program computers, however, the instructions are expressed in the form of digits. That is, each instruction is identified by a code number. The program, a list of code numbers, is fed to and stored in the computer in a manner similar to the input and storage of data. Then, by a process of decoding, the computer can direct itself in the solution of the problem for which the program was written.

A computer does not "understand" digits from a numerical point of view. It merely responds to specific physical conditions created in the components of the computer. These physical conditions—voltages, currents, etc., represent the digits. The physical conditions interact to produce a set of conditions that represent digits expressing the solution to a problem. Thus, the "language" used by computers consists ultimately of specific physical conditions in its components. Consequently, all data and instructions fed to a computer must be represented by specific physical conditions in the computer. For example, if the digits 0, 1, 2, 3, and 4 are to be presented to a digital computer, they must be presented as five separate physical conditions that can be set up in the components of the computer.

One type of physical condition that can represent a digit is a voltage on a voltage input line. To present five different digits to a computer, therefore, five different voltage input lines could be used. Similarly, voltages on output lines, obtained by processing the input data, could represent the digits in a desired manner. Then, if a voltage were applied to the line representing the digit 1 and another voltage were applied to the line representing the digit 2, along with the order to add the two digits, the response of the computer would be a voltage on the output line that corresponds to the digit obtained by adding 1 and 2; this would be the output line representing the digit 3.

The order for the computer to add the input digits —or any other order that the computer can carry out —would also be presented to it in the form of groups of digits. Digits representing instructions to the computer for operating on input data could be fed into a special set of control lines reserved for instructions only. With such an arrangement, the digits representing data and the digits representing orders would not be confused by the computer. If the order "add" were assigned the digit group 23, this order could be presented to the computer by voltages on the control lines representing digit 2 and the digit 3. Figure 1–5 illustrates how such an arrangement would lead to addition of the digits 1 and 2.

The preceding description is extremely simplified and not necessarily based on any system in common use. It does illustrate, however, the type of "language" that a digital computer "understands." Figure 1–5 gives an idea of the fundamental nature of information representation in a digital computer. To represent a 3, the 3-line must have a pulse on it; if no 3 is present, no pulse is present on the 3 line. In other words, an on or off condition represents the presence or absence of a digit, respectively.

The decimal system of numbers can be represented this way, but a far simpler number system, made up entirely of the digits 0 and 1, can be more easily used. This *binary* number system can represent any number by a series of 1's and 0's. For instance, a binary number equivalent to decimal 25 is 11001 (see Part 2). Because binary numbers can be used to represent any quantity, they can be used in digital computers. A 1 can be represented by the energized condition of a relay and 0 by the de-energized relay condition.

3.2 DIGITAL-COMPUTER ELEMENTS

3.2.1 Data Processing

3.2.1.1 General

Should we wish to know a single number that is equal to 24 times 512, we would multiply the two numbers together, thus:

$$24 \times 512 = 12{,}288$$

Both sides of the equation are equal—we did not gain any information by multiplying (processing)

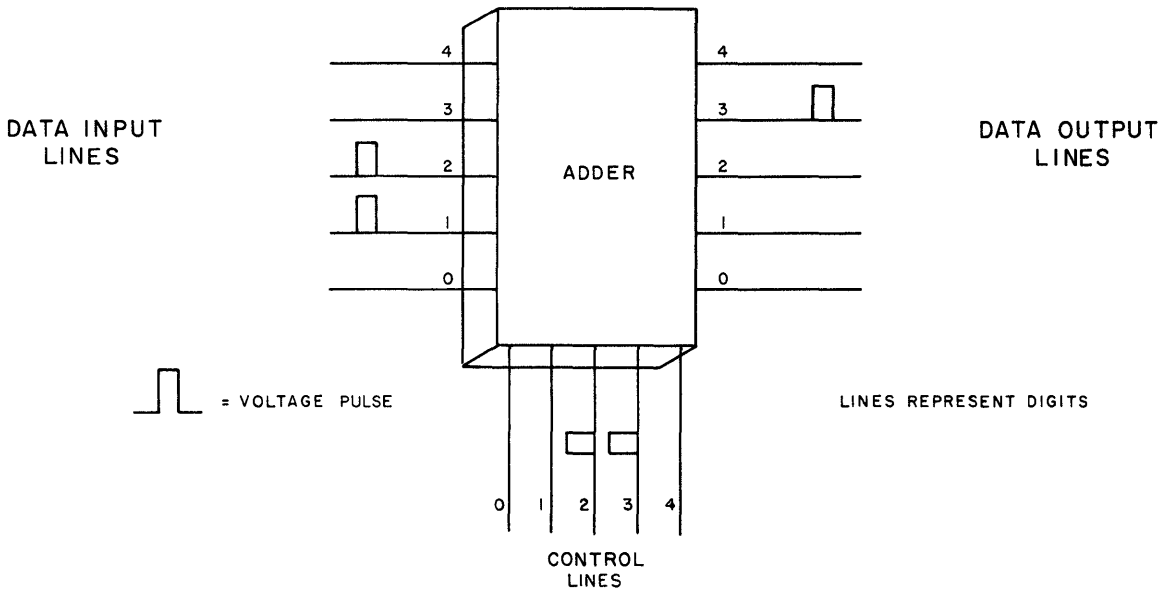


Figure 1–5. Numbers and Control Instructions Represented in the Form of Voltages

the data, although it is admittedly in a form more suitable for such purposes as comparison with other numbers, adding to other numbers, etc. It should always be kept in mind that data processing machines generate no new information even though their processing results in much greater usability of the existing data. It should also be remembered that the *operator* or *programmer* instructs the machine to perform *every* required step. *The machine does not think*—all of its operations and decisions must be built or programed into it by human effort.

3.2.1.2 Examples of Simple Data Processing

A man who processes data does so by following an exact set of rules, although he may not always be conscious of the fact. In making up a payroll, for example, a paymaster performs a series of predetermined operations which may be written down in a check-list which he must follow. The operations governed by the check-list and their sequence may be as follows:

1. Receive and store (write down) necessary data such as:
   a. Number of hours worked by each employee
   b. Pay rates for each employee
   c. Deductions from gross pay for each employee
2. On data stored for each employee, perform arithmetic operations such as:
   a. Multiply hours by hourly pay rate, write down partial products, and add partial products for complete product
   b. Multiply the sum obtained in operation a by a tax rate
   c. Subtract the product obtained in operation b from the product obtained in operation a
3. Make available, in a useful form, the results (output) of the preceding operations (1 and 2).

Throughout the task of making up a payroll, an efficient paymaster would perform only those operations required by the rules of the task. He would not, for example, add up the ages of all the employees, even if this data appeared on the original documents received. On the other hand, the paymaster would operate on all the data necessary for the task, and he would perform all necessary operations, as determined by the rules of the task.

Hence, to perform his work properly a paymaster must have some means of storing the required data and the appropriate instructions (such as paper, charts, tables, etc.). Morever, he must be able to extract data as needed and perform arithmetic operations in proper sequence. Finally, he must have means of making available in useful form the results of the operations—for example, a means of making out pay checks.

3.2.1.3 Machine Requirement for Data Processing

If the paymaster were making out the payroll on a computing machine, the machine would require facilities for:

a. Receiving necessary data
b. Storing the data
c. Controlling, by the rules of the task, the selection of data to be operated on and the proper sequence of operations
d. Performing required arithmetic operations
e. Making available in useful form the results of the operations

In other words, the computer would need an input element, a storage element, a control element, an arithmetic element, and an output element.

The operational elements listed above are the elements required by typical digital computers. The following paragraphs describe these elements and explain the tasks performed by each. Notice the similarity between corresponding computer operations and human operations, and remember that all steps originate with the human operator (programmer).

3.2.2 Input Element

The first operation that a digital computer must perform in a data-processing task is to accept the data pertinent to the task, and the instructions (program) for performing it. As explained in section 3.1, all data and instructions must be presented to a stored program computer in the form of digits, and these digits must be such that they can be represented by physical conditions in the computer components.

If each computer component has only two possible conditions—e.g., the open and the closed positions of a relay—all data must be presented as combinations of only two digits—e.g., 0 and 1.

Obviously, there must be an element that can accept the digits as they are presented by the operator, and this element should set up representative physical conditions in the computer. This input unit is called the *input element*: it provides one-way communication from the outside world to the computer. Data and instructions are fed to a computer through an input element; but an input element returns nothing to the outside world.

Items of data presented to the input element are not necessarily in the exact form that the rest of the computer elements can use. For example, in figure 1–6, a typewriter input might be used with which the operator would type out the data and instructions in a decimal code. The typewriter would have switches connected to each key which would convert the hitting of a key into an electrical impulse. The electrical impulse might then be converted to a binary code so that the computer could work with it. The input element usually performs this function of translating terms understandable to the outside world to those usable by the computer. Other input devices, such as card and tape readers, as well as various automatic input devices, are possible means of input. Nevertheless, the purpose of the input system is always the same—to translate the symbols of the outside world to those of the computer.

3.2.3 Output Element

The results of a digital computer's operations must be delivered to the user of the machine in an appropriate form. The element that accomplishes the transfer is the *output element*. The results of a computer's operations, however, are not necessarily in the form best suited for use outside the machine. Hence, an output element may include facilities for converting the results of the computer's operations into the form of output data best suited to the user of the machine. Thus, the answer to the problem might enter the output element in the form of binary electrical pulses. The output element may then convert these pulses to voltages that operate an electrically operated typewriter or a printing machine to print the final answer. (See figure 1–7.)

The output element, like the input element, is a one-way unit. It receives information from the other elements of the computer and transfers the information to the final user, but it does not return any information to the computer.

3.2.4 Arithmetic Element

Since the purpose of a digital computer requires that the machine perform arithmetic operations on the input data, a digital computer must obviously contain an element that can accomplish these operations. This is the *arithmetic element*. All data to be operated on arithmetically must enter this part of the computer. Likewise, most instructions determining what computations are to be performed must control the arithmetic element. (See fig. 1–8.)

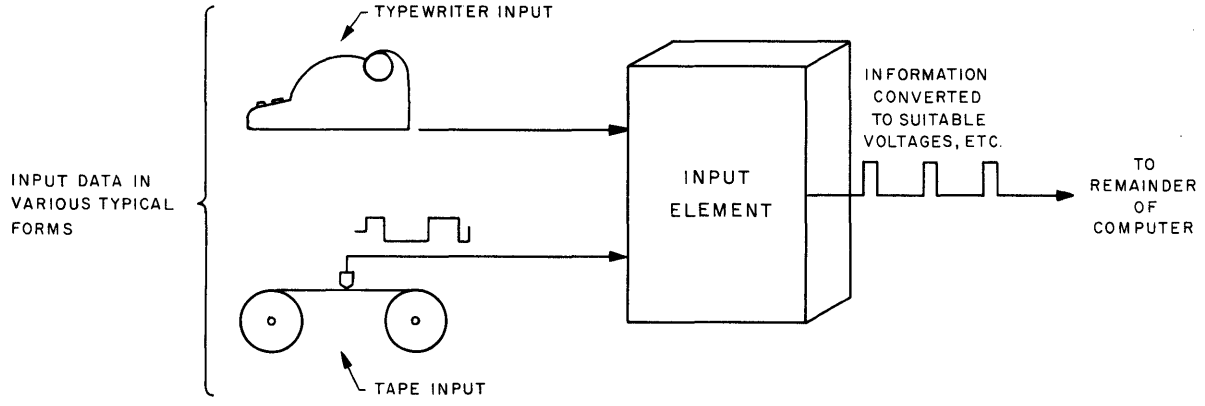


Figure 1–6. Input Element — This Element Receives Information and Converts it into Usable Form

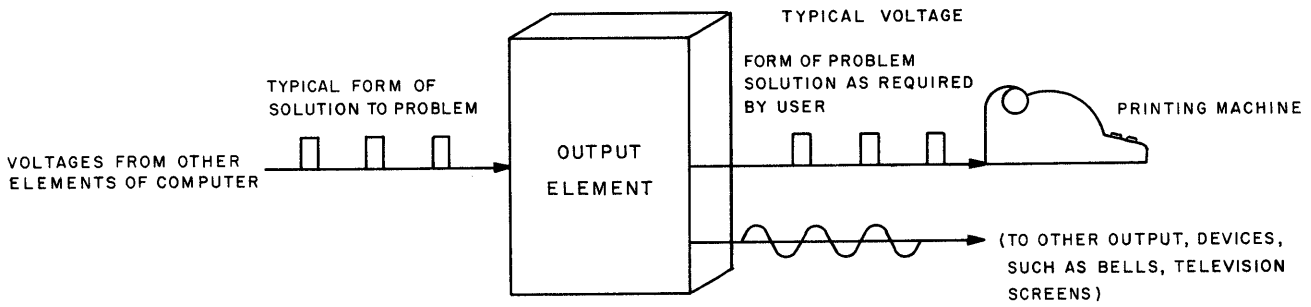


Figure 1–7. Output Element — This Element Converts Computer's Answer to the Problem into Form Usable by External Output Devices

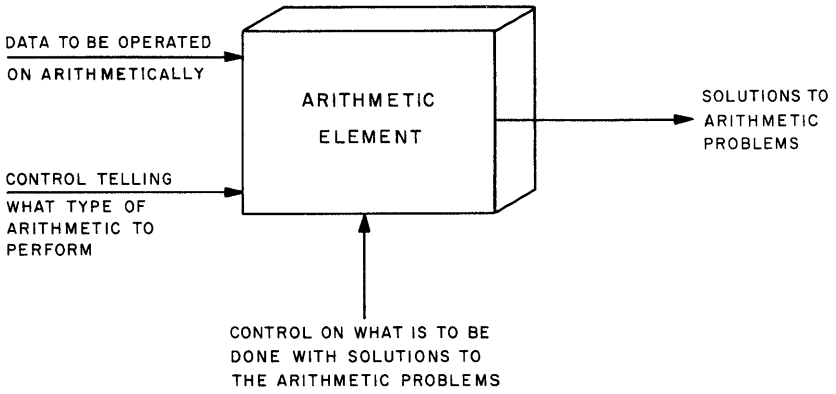


Figure 1–8. Arithmetic Element — Data Enters and Is Processed by this Element

Theoretically, it would be possible to build an arithmetic element which could perform most mathematical operations directly, just as a man performs them. This however, would require a very large and complicated arithmetic device and, consequently, is never done. Instead, the arithmetic element is usually designed to perform only a few basic operations such as addition, subtraction, multiplication, and division. (Three of these four operations are usually only an adaptation of either addition or subtraction.) If an arithmetic device can perform either addition or subtraction, and a few other simple operations, it can be made to perform almost any other mathematical operation by simply breaking the operation down into its fundamental operations. This is the way in which the arithmetic element is made to do the more complex mathematical operations that are often required.

The arithmetic operations in a high-speed machine such as AN/FSQ-7 and AN/FSQ-8 must be accomplished very quickly. Entire series of operations must be completed in a few microseconds (usec). This speed cannot be attained by mechanical or electro-mechanical devices (such as relays) but can be attained by electronic circuits. Electronic circuits for performing arithmetic operations are described in Part 3.

3.2.5 Storage Element

As stated in paragraph 2.3.2, operations in a digital computer are carried out in step-by-step fashion. For this reason, some of the information fed into a computer must be stored for indefinite periods prior to actual usage. The facilities required for storing information in a computer are included in the *storage element.*

Information fed into a computer is of three kinds:

a. Particular items of data to be processed

b. Instructions for performing the particular data-processing operations required (the program)

c. Reference data

Reference data—(c) above—must sometimes be stored for indefinite periods. For example, if the computer is used in the air-defense system, the reference data will include ballistic tables and flight plans of friendly aircraft. Such data are used over and over for successive problems. It would be impractical to feed this same information into the computer all over again for each new problem. If the storage element of the computer can retain such data indefinitely and quickly select individual items each time they are needed, problems can be solved much faster. Indeed, without fast, automatic, and reliable insertion and extraction of data

into and out of storage, working with a computer would be little more efficient than working with hand-operated calculators and reference tables.

Another type of information that can be stored is the *program*, or the set of instructions for performing a particular data-processing task. With a common program and common reference data stored in advance, the only additional input data required for a series of similar problems are those items that vary in value from problem to problem; that is, the actual numbers to be operated upon. Thus, the time required for insertion of input data for each new problem is accordingly reduced.

The basic characteristic of an information storage medium is that it has at least 2 stable states. For instance a light switch may be considered to be a memory element. It "remembers" whether the light is on or off, once it is set. In a computer the memory is capable of "remembering" a great many numbers which at any time must be almost instantaneously available to the rest of the computer. To accomplish this, many types of storage media can be used. One medium used is magnetic tape which uses recorders and readers very similar to home tape recorders. Another medium is magnetic drums. This device is similar to a tape recorder with very wide tape and a large number of parallel reading and recording heads. Actually the wide "tape" is a rotating drum coated with a magnetic material. There are also other storage devices, such as magnetic cores and cathode-ray tubes (see Part 3).

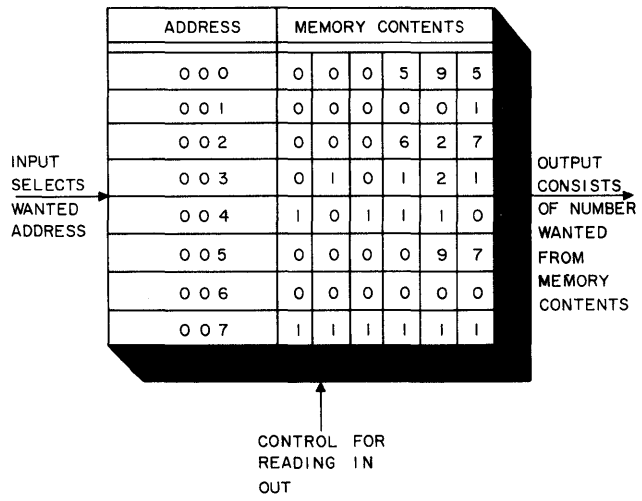


Figure 1–9. Storage Element — Possible Address and Contents Correlation

All storage elements have some basic characteristics that are similar. They always contain a number of storage locations, in each of which a single piece of data may be stored. Each of these locations is assigned a specific number called an address so that it may be selected by the computer either for insertion or extraction of data. For instance, in the diagram of the 8-location memory shown in figure 1–9, the addresses are shown on the left, opposite their actual contents. To refer to the number 10,121 the computer would refer to the location whose address is 003.

3.2.6 Control Element

There must be a definite sequence for the flow of data during processing by a digital computer. Data must be inserted into particular storage locations and then used in correct sequence at the appropriate times. The arithmetic element must also be "told" what operations to perform on the data and in what order to perform them. For all desired processing operations, moreover, the results of the arithmetic operations must be routed to the appropriate storage or output locations. Also, the transfer of all output data to the output element and, finally, to the user must be properly controlled to ensure the required sequence of information.

The entire sequence of operations by the computer is predetermined by the program (and the construction of the computer) for the data-processing task. The program, coded in the digital language used by the computer, is inserted through the input element, to be stored at specific addresses in the storage element. The element for interpreting and carrying out instructions contained in the program is the *control element* (see fig. 1–10).

By its interpretation of the program, the control element governs the flow of data and the sequence of operations performed by the computer. In a high-speed machine such as the AN/FSQ-7, special electrical circuits provide the required control. These circuits respond to electrical signals representing the digits that

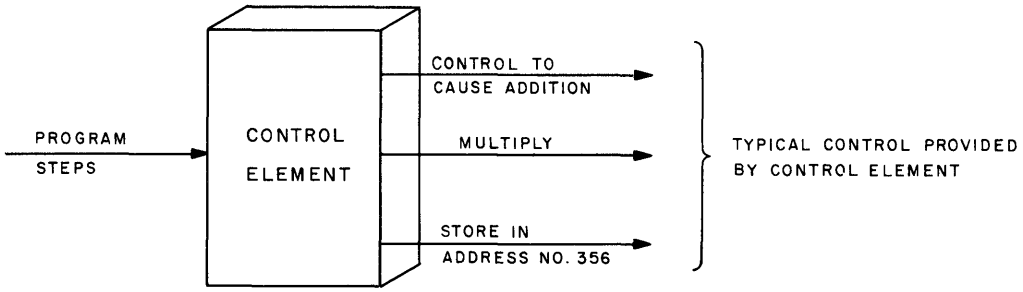


Figure 1–10. Typical Instructions from Control Element

make up the control instructions or program, producing appropriate control signals. The control signals cause arithmetic operations to take place and effect the transfer of data from one element of the machine to another. For instance, in figure 1—11 the action of the control element on the other element is shown. In this figure, information transfer is shown in heavy lines while control lines are light.

Control circuits are similar in some ways to arithmetic circuits. They operate in close association with all the circuits that carry out arithmetic or other processing operations (see Part 3).

3.2.7 General Organization

Figure 1—11 shows the general organization of the computer elements in a typical computer. Along with the various elements, information and control lines are shown which indicate the flow of information and control impulses through the computer. The heavy lines are information transfer lines; these lines will transfer both data and instructions between the input and the memory. Between the storage element and the output and between the storage element and the arithmetic element, data is transferred on these lines. From the storage to the control elements, only instructions are transferred on the heavy line. The light lines from the control element represent the control voltages which the control element sends to all elements.

3.2.8 General Operation

Both the computing section and the input-output elements operate under control of the program. A short computing operation will be explained; it will be assumed that the program and data are already in the memory when the computer starts.

The general operation of the computer in following a set of instructions can be illustrated by analogy. Assume that the paymaster (par. 3.2.1) is following a set of instructions in sequence, and he has just found the number of hours worked by a man. The next instruction he reads says: "Multiply the number of hours by the hourly pay rate." The paymaster would then obtain the hourly pay rate from wherever it was written down (stored) and multiply the hours by the pay rate. It can be seen that the instruction specified two things: the operation and the operand to be used in the operation.

The instructions in a computer program have somewhat the same form. They specify the operation to be performed and the address of the location in memory where the operand is kept. For instance a program may have an instruction in it which says:

> "Clear the arithmetic element and add the number which is in memory location 020."

To execute such an instruction, the control element would first cause the arithmetic element to be cleared of any numbers it might contain. Next the control element would cause the number in memory location 020 to be transferred from the memory, to be added into the arithmetic element.

Therefore, the execution of a series of instructions would be quite a simple matter. The instructions would be stored in memory in proper order. The data would be stored in the locations which correspond to the addresses of the pertinent instructions. After this was done, the control element would be made to pick out the first instruction and execute it, pick out the next instruction and execute it, and so on until an instruction told the computer to stop.

As an example:

Suppose that it is desired to add the numbers 512 and 608 together and to store the sum in memory location 012. Possible instructions are given in table 1—1 and it is assumed that the computer could execute the in-

TABLE 1—1. SOME POSSIBLE INSTRUCTIONS

ABBREVIATION	MEANING
CAD x:	Clear the arithmetic element and add in the number in memory location x. The completion of this instruction will cause the arithmetic element to contain the number which is in memory location x, where x is the address of the memory location specified.
ADD x:	Add to the number which is in the arithmetic element the number in memory location x. The completion of this instruction will cause the arithmetic element to contain the sum of the contents of the arithmetic element and the number in memory location x.
FST x:	Store the contents of the arithmetic element in memory location x. The completion of this instruction would cause the number in the arithmetic element to be transferred to memory location x.
HLT:	Stop the computer.

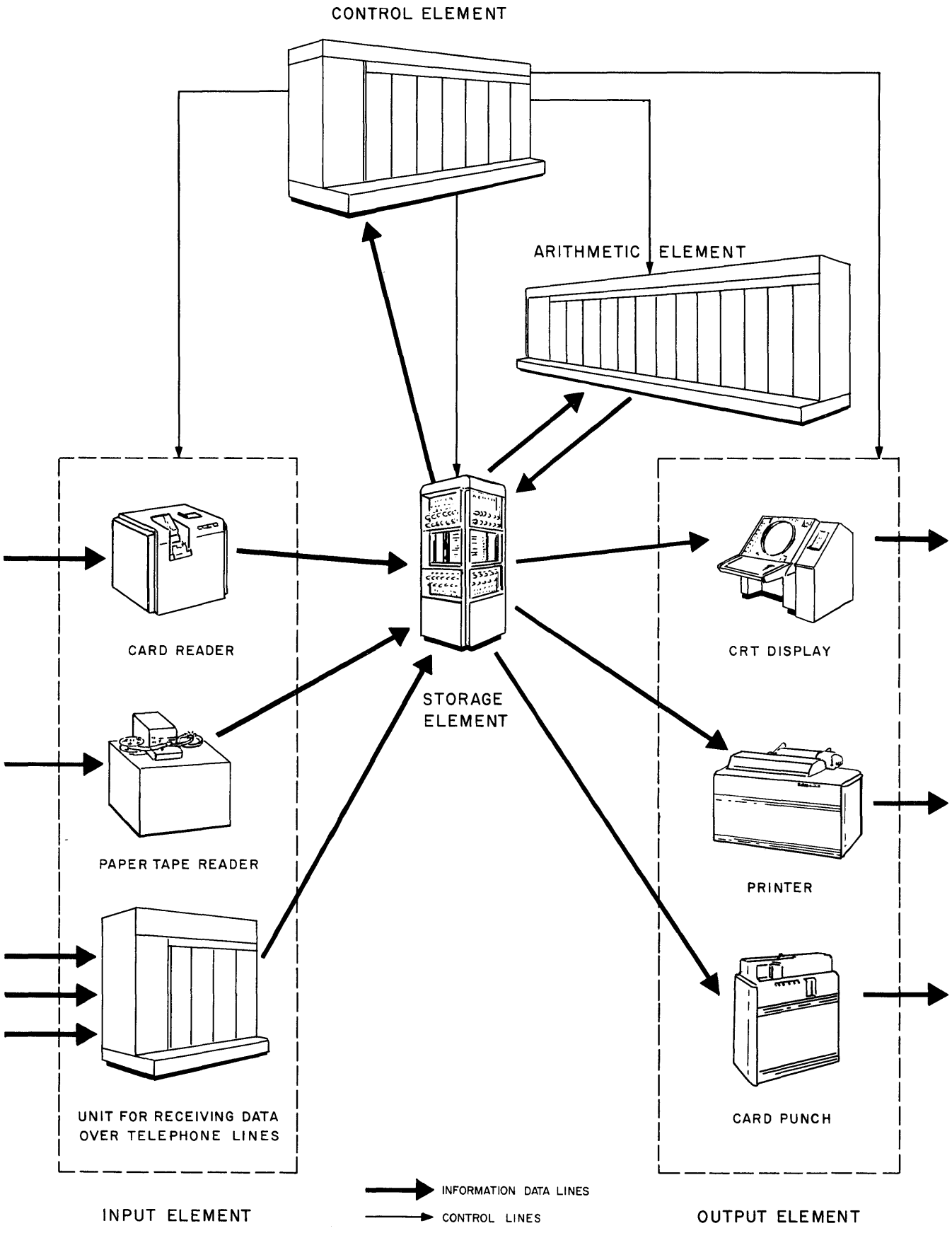


Figure 1–11. Organization of a Typical Digital Computer

structions in table 1–1. To solve the problem, a program such as that shown in the symbolic memory of figure 1–12 would be required.

The control element executes the program by automatically picking out and executing instructions one after another, starting with location 000. The operation, therefore, would go somewhat as follows:

1. The operator presses the computer START pushbutton.
2. The control element obtains from memory location 000 the instruction *CAD 005.*
3. The control element first clears the arithmetic element and then transfers from memory location 005 to the arithmetic element the data word (number) 512.
4. The control element obtains from memory location 001 the instruction *ADD 007.*

Address	Contents of Location	
000	CAD	005
001	ADD	007
002	FST	011
003	HLT	
004		
005	512	
006		
007	608	
008		
009		
010		
011	1120	
012		

Figure 1–12. Contents of Memory for Solution of the Problem: 512 + 608 = ?

5. The control element causes the data word (number) 608 at memory location 007 to be added to the number in the arithmetic element to leave a sum, 1120, in the arithmetic element.
6. The control element obtains from memory location 002 the instruction *FST 011.*
7. The control element causes the data word 1120 in the arithmetic element to be stored at location 011 in the memory.
8. The control element obtains from memory location 003 the instruction *HLT.*
9. The control element stops the computer.

3.2.9 Summary

The sequence of events in carrying out a complete computation on a computer is as follows:

1. A program is prepared by a programmer.
2. Through the input devices, the computer operator loads the program and data into the computer memory (data storage section).
3. The computer is started and the control element automatically follows the program, transferring data from the memory to arithmetic element, (which performs the arithmetic operations called for in the program), and transferring intermediate results from the arithmetic element to the memory.
4. The final results are transferred from the memory to an output unit (still under control of the program).
5. The output unit translates the results to a form that can be read, heard, or seen by the operators.
6. The control element reaches a halt instruction, and the computer stops operation.

PART 2
COMPUTER ARITHMETIC

CHAPTER 1
INTRODUCTION

1.1 COMPUTERS AND INFORMATION

A digital computer processes information by the use of devices whose physical or electrical states correspond to definite, previously agreed upon meanings. These meanings may be represented in the form of numbers. Such a representation is a type of code in which the state of certain computer components is related to the numbers. These numbers, in turn, may be related to both numerical and non-numerical information, on the basis that certain combinations of numbers always represent a specific meaning. A number system is the medium by which information is prepared for computer processing.

1.2 POSSIBLE NUMBER SYSTEMS

Since a digital computer processes information which is expressed as numbers, a brief investigation of some available number systems is in order. The decimal system is the number system most universally known. This system derives its name from the total number of symbols used in notation—decem is the Latin word for ten. The decimal system of numbers uses the arabic notations 0, 1, 2, 3, 4, 5, 6, 7, 8, and 9, alone or in combination to express quantity or identity. Other number systems are possible and are available. The Mayan Indians used a vicenary system of very unusual-looking symbols. The Romans used the marks I, V, X, C, D, L, and M. Some mathematical tables have been computed in accordance with the duodecimal system; the eleventh and twelfth marks being X and E respectively.

Some of these number systems are the results of historical development peculiar to a culture—for example, the "twenty" system of the Mayans. Other systems are useful for certain functions and owe their existence to such specific usefulness. A number system is identified by the kind and the total number of symbols used to express the idea of quantity.

At present, the Arabic symbols are of major interest in the computer field. The variation in total number of symbols may be a new concept. The decimal system uses ten symbols; the binary system, two; and the octonary system, eight. All three systems use Arabic symbols to express numbers.

1.3 WHICH NUMBER SYSTEM IS BEST

The usefulness of the decimal system is evident from its universal acceptance as a mode of counting in science, in business, and in everyday life. The procedures for arithmetic operations in the decimal system are familiar and the symbols have been traditional for many centuries. No one would suggest a departure from the decimal system as far as noncomputer operations are concerned.

But in selecting a number system for use with a digital computer, primary consideration must be given to that system which is most advantageous for the computer. It is possible, and in some cases preferable to use the decimal system, but such a system is not usually convenient. A computer consists of physical devices which are bistable: a hole is either present or absent in a card; a relay is either opened or closed; a vacuum tube is either conducting or nonconducting; a magnetic material is magnetized in one direction or in the opposite; a crystal diode conducts very well in one direction but becomes a high resistance to current in the opposite direction. A computer, then, is inherently binary; therefore, the natural counting scheme for computer circuits is the binary system.

The binary system uses two symbols, 0 and 1, and bistable devices have two states; this is one advantage of this system. Another advantage of the binary system is the simplicity of arithmetic operations. Only two procedures are necessary: addition and subtraction. The rules to perform these procedures are very simple and lend themselves to automatic operation. But the binary system has the disadvantage of being difficult for a person to read or express quickly in words. This is because the expression of a number in binary requires a number

with approximately 3-½ times as many digits as the same number expressed in decimal notation. For instance decimal 9 expressed in binary is 1001.

It is evident that the decimal system is suitable for noncomputer operation and that the binary system is more useful in digital computers. The octal system is a convenient method for expressing the information that is being presented as inputs to the computer. The octal system uses the Arabic symbols 0, 1, 2, 3, 4, 5, 6, and 7. There is a natural relationship between binary and octal numbers and one octal number may represent three binary numbers. If input-output information is expressed in octal numbers, there is a reduction in notation and, therefore, easier reading for the operator.

CHAPTER 2
NUMBER SYSTEMS

2.1 DECIMAL NUMBERS

2.1.1 General

The 10 Arabic symbols, which are the familiar notations 0, 1, 2, 3, 4, 5, 6, 7, 8, and 9 used in the decimal system, owe their form to traditional acceptance. By definition, each symbol is termed a *digit* and represents quantity or identity—$990.00 represents a quantity of dollars whereas 900 Ninth Avenue may identify a building.

2.1.2 Positional Notation

The idea of number expressed by these symbols depends on another important characteristic of the decimal system: the value of a digit depends on its position; for instance, the digit 7 expresses different values in the expressions 70, 700, and 7,000. To determine the position of a digit, some point of reference must be known; in decimal notation, it is called the decimal point. The position of the digit to the right or left of this point indicates the value of the digit.

2.1.3 Radix

The *base* or *radix* of the decimal system is 10; that is, counting in the decimal system is accomplished by counting from 0 through 9 over and over again until the count is finished. The base (radix) of any numbering system always equals the total number of different symbols used for the digits of the system.

2.1.4 Counting

The decimal number system uses 10 symbols and an unlimited number of positions to express value. Each position to the left of the decimal point represents a positive power of 10, with the power increasing from each position to the next left position. Positions to the right of the decimal point represent negative powers of 10, with the power increasing in a negative direction from each position to the next right position. (The positive power of a number is the number of times the number is multiplied by itself, while the negative power of a number is the number of times the reciprocal of the number is multiplied by itself.)

A true number system consists of a collection of symbols with a systematic means of progression from one number to a higher or lower number. Counting is the progression from one number to a higher or lower number. Since numbers are expressed by symbols, a method of symbol combination is necessary once all the original symbols have been used. For example, a count to nine uses all the original symbols; in order to count past nine, combinations using two symbols are used up to 99. The combinations are selected in an orderly way as follows:

Progressing from 9 to 10:

1. Advance the symbol in the right position (9) back to the first symbol (0);
2. Move left to the next position and advance the symbol (an implied 0) to the next symbol in the system (1).

Progressing from 10 to 11 and on to 99.

1. Advance the symbol in the right position (0) to the next symbol in the system (1).
2. Continue advancing the symbol in this position until the last symbol (9) is reached.
3. Change the 9 to a 0, then advance to the left position and add a one to the symbol occupying that position.
4. Continue in this manner until the number 99 is expressed.

Progressing from 99 to 100 and on to 999.

1. After the last symbol is used in the right position, follow the procedures given in example 2, advancing to the left one position at a time.
2. This method of counting is used not only in the decimal system but also in the binary and octal systems.

2.1.5 Expression of a Decimal Number

The base of the decimal system is 10; any number may be expressed as a sum of the powers of 10, each power being multiplied by the coefficient of the term. For example, examine 2145.401 as a decimal number:

a. Note the reference point, the decimal point. Powers of 10 to left of the decimals point are positive, to the right are negative.

b. The expression is equivalent to:

$$2000 + 100 + 40 + 5 + 4/10 + 0/100 + 1/1000$$

$$\text{or } 2(1000) + 1(100) + 4(10) + 5(1) + 4(.1) + 0(0.1) + 1(.001)$$

$$\text{or } 2 \times 10^3 + 1 \times 10^2 + 4 \times 10^1 + 5 \times 10^0 + 4 \times 10^{-1} + 0 \times 10^{-2} + 1 \times 10^{-3}$$

Coefficient — Power of base

Usually, only the coefficients are expressed, and the part of the expression concerned with the power of the base is understood because of the notational position.

2.2 THE GENERAL EXPRESSION FOR A NUMBER

Any number may be expressed by the formula for the general expression for a number. The formula defines any number as follows:

$$N = C_nR^n + C_{n-1}R^{n-1} + \ldots C_0R^0 + C_{-1}R^{-1} + C_{-2}R^{-2} - \ldots$$

where:

C_n is the coefficient of the nth term;

R is the radix

The number to be expressed is N.

The highest power to which the radix is raised is n.

For example, the polynominal 212.75 can be expressed as:

$$212.75 = 2 \times 10^2 + 1 \times 10^1 + 2 \times 10^0 + 7 \times 10^{-1} + 5 \times 10^{-2}$$

or as;

$$N = C_nR^n + C_{n-1}R^{n-1} + C_0R^0 + C_{-1}R^{-1} + C_{-2}R^{-2}\ldots -.$$

The number 212.75 is a mixed number; i.e., it is an integer and a fraction. An integer is a whole number and the integral portion is 212; the fraction is .75. The decimal point is always the reference point, the numbers to the left are integers, and those to the right are fractions.

$R = 10$ because the base of the decimal system is 10.

$C_n = 2$ because the coefficient of the nth term is 2. The nth term is always the leftmost number.

$C_{n-1} = 1$ because the coefficient of the term one position to the right of the leftmost term is 1.

$C_0 = 2$ because the rightmost coefficient in the integral position of an expression is always C_0.

$C_{-1} = 7$, the first number after the decimal point.

$C_{-2} = 5$

Substituting the values in the equation,

$$212.75_{(10)} = 2 \times 10^2 + 1 \times 10^1 + 2 \times 10^0 + 7 \times 10^{-1} + 7 \times 10^{-2}$$

$$212.75_{(10)} = 2 \times 100 + 1 \times 10 + 2 \times 1 + 7 \times .1 + 5 \times .01 = 200 + 10 + 2 + .7 + .05.$$

$$212.75_{(10)} = 212.75$$

Thus it becomes obvious that a number expressed in the usual arabic symbols is merely a statement of the coefficients of the general formula. The expression for any number in any number system can be defined by the above formula. Take, for example, a number in the octal system—with a radix of 8. The octal number $245.32_{(8)}$ can be expressed in decimal as:

$$245.32_{(8)} = 2 \times 8^2_{(10)} + 4 \times 8^1_{(10)} + 5 \times 8^0_{(10)} + 3 \times 8^{-1}_{(10)} + 2 \times 8^{-2}_{(10)}$$

The subscript $(_8)$ means that the number is expressed in octal notation and subscript (10) means decimal notation.

To complete the operations indicated in the right-hand side of the above expression would result in the decimal equivalent for $245.32_{(8)}$. R is expressed as eight, which is a decimal number since seven is the largest symbol available to express a single digit in the octal system.

To express the number completely in octal terms, some octal expression whose value equals decimal 8 must be substituted for the radix. In the octal system, the symbol 10 (octal 10) is equal to decimal 8. The complete octal expression for the octal number $245.32_{(8)}$ would read:

$$245.32_{(8)} = 2 \times 10^2_{(8)} + 4 \times 10^1_{(8)} + 5 \times 10^0_{(8)} + 3 \times 10^{-1}_{(8)} + 2 \times 10^{-2}_{(8)}$$

2.3 BINARY NUMBERS

2.3.1 General

From actual experience, it is known that the amount of equipment required for a digital computer depends on the base of the number system utilized. Most modern computers are designed to use the binary system (base 2) because this system is the most efficient with present day components, which are binary in nature. The binary number system, with a base (radix) of 2, utilizes combinations of two digits to represent any number. A binary number will be made up of the binary digits 0 and 1 and will appear in a manner similar to this: 1101011. The term binary digit has been shortened so that in computer terminology a binary digit is called a bit. The binary numbers can be represented in electronic terms in a manner similar to the following examples:

(1) Relay	open = 0
	closed = 1
(2) Vacuum tube	conducting = 0
	nonconducting = 1

2.3.2 Binary Counting

Counting is started in the binary system in the same way as in the decimal system with the symbol 0 for the number zero in the right position and with 1

for one as the next progression. But at two in the binary system there are no more symbols available. To progress from one to two in the binary system, a carry operation similar to that used to progress from 9 to 10 in decimal is required. This move is to replace the 1 with a 0 and add a 1 to the next position to the left. The following is a listing of some numbers of equivalent value in the binary and decimal systems.

BINARY (BASE 2)	DECIMAL (BASE 10)
0	0
1	1
10	2
11	3
100	4
101	5
110	6
111	7
1000	8
1001	9
1010	10
1011	11

In the binary system, the value of the digit is determined by its position in relation to the binary point (similar to a decimal point).

2.3.3 General Meaning of a Binary Number

An examination of the above list will show that a binary number is also expressed according to the general expression for a number. In the case of binary numbers, however, there are only the two digits 0 and 1; hence, the radix of the system is two. The last number in the list, then, is actually an expression of the following equation with the radix expression dropped.

$$1011 = 1 \times 2^{3}_{(10)} + 0 \times 2^{2}_{(10)} + 1 \times 2^{1}_{(10)} + 1 \times 2^{0}_{(10)}$$

When the expression on the right is evaluated in decimal numbers, it will be found to equal decimal 11.

The meaning of a fractional binary number is also easily expressed in the general formula. Again, the powers of the radix to the right of the reference point (binary point) are negative powers. The expression .1101 in binary can be expressed as follows:

$$.1101 = 1 \times 2^{-1} + 1 \times 2^{-2} + 0 \times 2^{-3} + 1 \times 2^{-4}$$

In decimal this equals:

$$\frac{1}{2} + \frac{1}{4} + \frac{0}{8} + \frac{1}{16} = \frac{13}{16}$$

or

$$.5 + .250 + .0625 = .8125$$

2.3.4 Generating Binary Numbers

Binary numbers can be made up and checked for the equivalent decimal value without using a decimal-to-binary conversion table. The problem of representing a decimal number in binary form requires the calculation of the correct binary arrangement of 1's and 0's. There is a fast and simple method for solving such a problem and for checking the result. For example:

express $62.375_{(10)}$ in $N_{(2)}$

The first step in this procedure is to write in sequence the decimal values of the power of two:

64 32 16 8 4 2 1 1/2 1/4 1/8 1/16 1/32

Since 62 is less than 64, place a 1 under those values (highest) which when added together equal 62; place a 0 under the other values of the base 2. Place a 0 under the decimal values which are not included in the sum. Thus, the integral portion of the binary number has been generated. The fraction .375 equals 3/8 which is the sum of 1/4 plus 1/8. Place 1's under 1/4 and 1/8 and a 0 under 1/2.

Decimal	64	32	16	8	4	2	1	1/2	1/4	1/8
Binary	0	1	1	1	1	1	0.	0	1	1

To check the result, add all decimal values which have a binary weight of 1.

32	1/4
16	+1/8
8	3/8 or .375
4	
+ 2	
62	

The binary number for any decimal expression can be generated by this method.

2.4 OCTAL NUMBERS

2.4.1 General

The octal number system is similar to the decimal and binary systems. The only difference between the three systems is caused by the differences in radix. The radix of the octal system is 8. Therefore, the only symbols allowed in an octal number will be the digits 0, 1, 2, 3, 4, 5, 6, and 7.

2.4.2 Octal Counting

Octal counting proceeds from 0 to 7 just as in decimal. However, at 7 in the octal system there are no more symbols available. To progress from 7 to 8 in the octal system, a carry operation similar to that used

to progress from 9 to 10 in decimal is required. The following is a listing of some octal numbers and their decimal equivalents:

OCTAL (BASE 8)	DECIMAL (BASE 10)
0	0
1	1
2	2
3	3
4	4
5	5
6	6
7	7
10	8
11	9
12	10
13	11
14	12
15	13
16	14
17	15
20	16

In the octal system, the value of a digit is determined by its position relative to octal point.

2.4.3 General Meaning of an Octal Number

An examination of the above list will show that octal numbers are also expressed according to the general expression for a number.

$$210.8_{(8)} = 2 \times 8^2 + 1 \times 8^1 + 0 \times 8^0 + 2 \times 8^{-1}$$

The evaluation of the formula shows that the octal number $210.2_{(8)}$ is equal to the decimal number $136.25_{(10)}$.

2.4.4 Use of Octal Numbers

The octal number system is useful in connection with digital computers as a form of shorthand for binary notation. The relationship between the two systems can be stated as follows: since eight is the third power of two, three places in binary notation corresponds to one place in octonary notation. Each octal digit can be represented in binary by three digits (bits), and, conversely, every combination of three bits has a corresponding octal digit. The correspondence can be summarized as follows:

OCTAL	BINARY
0	000
1	001
2	010
3	011
4	100
5	101
6	110
7	111
247	010 100 111
526	101 010 110
5647	101 110 100 111

The use of eight symbols (octonary), rather than two (binary), provides a number language which is easier to read and write, and thus decreases the probability of error. The advantages of the octal number system, coupled with simplicity of the conversion between octal and binary notation, make the octal system a good working notation for operators who are preparing sets of numbers which eventually must be entered into a computer.

Programmers use the octonary system to express information because octal notation is shorter than binary and because conversion from the octal to the binary system can be performed by inspection.

CHAPTER 3
RADIX CONVERSION

3.1 INTRODUCTION

The preparation of information for digital computers and the processing of this information within the computer involves the use of the decimal, octal, and binary number systems. Conversion from one number system to another is sometimes necessary, and several methods are possible. In this chapter a general method based on the general expression for a number will be given. In some cases this general method involves more work than necessary, particularly when the method requires that arithmetic be performed in other than the decimal system. In such cases alternate methods have also been given.

3.2 GENERAL METHOD

The general method of radix conversion is based upon the formula:

$$N = C_nR^n + C_{n-1}R^{n-1} \ldots C_0R^0 + C_{-1}R^{-1} + C_{-2}R^{-2} \ldots -.$$

The rules to follow when converting by the general method are:

1. Write the number in the form of the general expression for a number. R is the radix of the number system from which the number is to be converted, but when R and C are written in the formula they are both expressed in the symbols of the number system to which the number is to be converted.
2. Add the various terms of the formula, using the the rules of arithmetic of the number system to which the number is being converted. The sum of terms is then the converted number.

Examples of the use of these rules are given in each of the types of conversion explained below.

3.3 BINARY TO DECIMAL CONVERSION

The conversion of an expression in binary form to its decimal equivalent is performed by using the general formula and rules that follow. For example, convert $10110.111_{(2)}$ to its decimal equivalent.

$10110.111_{(2)}$ = a mixed binary number; the integral portion is to the left of the binary point, the fraction to the right.

R = 2 because the conversion is being made from the binary system. R is the base of the system from which the conversion is being made; C and R are both expressed decimally; i.e., in the number system to which the number is being converted.

$C_n = 1$ because 1 is the coefficient of the leftmost position.

$R^n = 2^4$ because the base is raised to the fourth power; n equals the total number of integral positions less one.

C_{n-1}, C_{n-2}, C_{n-3} = the binary coefficients following in sequence from left to right.

$C_0 = 0$ because 0 is the rightmost binary bit in the integral position.

$C_{-1} = 1$ because 1 is the first binary cofficient after the binary point.

$C_{-2, -3}$ = the binary coefficients following in sequence to the right of C_{-1}.

$$N = C_nR^n + C_{n-1}R^{n-1} + C_{n-2}R^{n-2} + C_{n-3}R^{n-3} + C_0R^0 + C_{-1}R^{-1} + C_{-2}R^{-2} + C_{-3}R^{-3}$$

$$10110.111_{(2)} = (1 \times 2^4) + (0 \times 3^3) + (1 \times 2^2) + (1 \times 2^1) + (0 \times 2^0) + (1 \times 2^{-1}) + (1 \times 2^{-2}) + (1 \times 2^{-3})$$

$$10110.111_{(2)} = 1 \times 16 + 0 \times 8 + 1 \times 4 + 1 \times 2 + 0 \times 1 + 1 \times 1/2 + 1 \times 1/4 + 1 \times 1/8$$

$$10110.111_{(2)} = 16 + 0 + 4 + 2 + 0 + 1/2 + 1/4 + 1/8$$

$$10110.111_{(2)} = 22.875_{(10)}$$

Notice that arithmetic was performed entirely in the decimal system, the system to which the number is being converted.

3.4 DECIMAL TO BINARY CONVERSIONS

Three methods of decimal to binary conversion are given below. The first is the general method already explained, and the other two are somewhat easier methods which avoid use of binary arithmetic.

3.4.1 General Method

The general rule may be used to convert a decimal number to a binary number. As an example, convert the decimal number $22.851_{(10)}$ to its binary equivalent.

$$22.875_{(10)} = 10 \times (1010)^1 + 10 \times (1010)^0 + 1000(1010)^{-1} + 111(1010)^{-2} + 101(1010)^{-3}$$

$$22.875_{(10)} = 10100 + 10 + \frac{1000}{1010} + \frac{111}{(1010)\ (1010)} + \frac{101}{(1010)\ (1010)\ (1010)}$$

$$22.875_{(10)} = 10110 + \frac{1100100000}{(1010)^3} + \frac{1000110}{(1010)^3} + \frac{101}{(1010)^3}$$

$$22.875_{(10)} = 10110 + \frac{1101101011}{(1010)^3}$$

$$22.875_{(10)} = 10110.111_{(2)}$$

In the terms of the formula:

a. R = 10, which is the base of the system from which the number is being converted.

b. R and C are both expressed in binary, which is the system to which the number is being converted.

c. The multiplication of the powers by their coefficients and summing of the terms to get the final binary number is performed in binary arithmetic.

3.4.2 Radix Subtraction Method

In general, the method is to determine what power of 2 is the largest that is smaller than the decimal; this power is subtracted from the decimal number. Then an attempt is made to subtract the next lower power of 2 from the remainder. If this is not possible (results in a negative number), the next lower power of 2 is subtracted, and so on. For each successful subtraction, a 1 is generated for the binary number. For each unsuccessful subtraction a 0 is generated. For example, convert the decimal number $123_{(10)}$ to its binary equivalent.

The first step is to determine from table 2–1 the highest power of 2 which is equal to or less than $123_{(10)}$; this is 2^6. The next step is to express in decreasing sequence the powers of 2, starting with 2^6.

$$2^6 \quad 2^5 \quad 2^4 \quad 2^3 \quad 2^2 \quad 2^1 \quad 2^0$$

The formula for the general expression for a number states that the number is the sum of the coefficients multiplied by the base raised to the required powers. Therefore, the third step involves the determination of the coefficients of the above sequence of powers of 2. These coefficients will be expressed as the binary 1 or 0. This third step is accomplished by subtracting 2^6 from 123 and examining the remainder.

123
− 64
59 Remainder

TABLE 2–1. POSITIVE AND NEGATIVE POWERS OF 2

POSITIVE POWERS			NEGATIVE POWERS		
2^0	= 1				
2^1	= 2	2^{-1}	= 1/2	=	0.5
2^2	= 4	2^{-2}	= 1/4	=	0.25
2^3	= 8	2^{-3}	= 1/8	=	0.125
2^4	= 16	2^{-4}	= 1/16	=	0.0625
2^5	= 32	2^{-5}	= 1/32	=	0.03125
2^6	= 64	2^{-6}	= 1/64	=	0.015625
2^7	= 128	2^{-7}	= 1/128	=	0.007813
2^8	= 256	2^{-8}	= 1/256	=	0.003906
2^9	= 512	2^{-9}	= 1/512	=	0.001953
2^{10}	= 1,024	2^{-10}	= 1/1024	=	0.000977
2^{11}	= 2,048	2^{-11}	= 1/2048	=	0.000488
2^{12}	= 4,096	2^{-12}	= 1/4096	=	0.000244
2^{13}	= 8,192	2^{-13}	= 1/8192	=	0.000122
2^{14}	= 16,384	2^{-14}	= 1/16,384	=	0.000061
2^{15}	= 32,768	2^{-15}	= 1/32,768	=	0.000031

Note: Decimal fraction values have been rounded off to the nearest millionth place.

If there is a power of 2 that is equal to or less than 59, another subtraction is performed using that value. Table 2-1 shows that 2_5 is less than 59. Thus,

59
−32
27 Remainder

Continue subtracting powers of 2 until a remainder of 0 is obtained. Thus:
Thus:

27
−16
11 Remainder
− 8
3 Remainder
− 2
1 Remainder
− 1
0 Remainder

It is now possible to write the coefficients in the formula; the coefficient of those powers of 2 which were subtracted is 1; the coefficient of those powers which were not subtracted from the remainders is 0. Thus:

$$N_{(2)} = 1 \times 2^6 + 1 \times 2^5 + 1 \times 2^4 + 1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 1 \times 2^0$$

Since only the coefficients are normally written, the binary equivalent is $123_{(10)}$ is 1111011.

Conversion of a mixed decimal expression to its binary equivalent is performed in the same manner. This is the basic rule for performing the operation on the conversion of a mixed number: (1) the integral part of the binary number is equal to the integral part of the decimal number, and (2) the fractional part of the binary number is equal to the fractional part of the decimal number.

Advantage is taken of this rule in converting a mixed number. The integer part of the conversion is performed independently of the fraction conversion. For example, convert 3028.359375 to N_2. First, find in table 2–1 the highest power of 2 which is equal to or less than 3028 – also find the highest negative power of 2 lower than .359375. Subtract these values 2^{11} and 2^{-2} from the given decimal expression. Continue to subtract from the remainders, values of the powers of 2 which are equal to or less than these remainders until a remainder of 0 is obtained.

INTEGRAL PORTION		FRACTIONAL PORTION	
	3028	0.359375	
2^{11}	−2048	−0.25	2^{-2}
	980	0.109375	
2^9	− 512	−0.0625	2^{-4}
	468	0.046875	
2^8	− 256	−0.03125	2^{-5}
	212	0.015625	
2^7	− 128	−0.015625	2^{-6}
	84	0	
2^6	− 64		
	20		
2^4	− 16		
	4		
2^2	− 4		
	0		

Write in decreasing sequence the powers of 2, starting at 2^{11}.

$2^{11}\ 2^{10}\ 2^9\ 2^8\ 2^7\ 2^6\ 2^5\ 2^4\ 2^3\ 2^2\ 2^1\ 2^0.\ 2^{-1}\ 2^{-2}\ 2^{-3}\ 2^{-4}\ 2^{-5}\ 2^{-6}$

For each power of 2 that was subtracted from the remainders, write 1 as a coefficient of the power; for each power which was too high for the subtraction operation, write a 0 as a coefficient of that power. Thus, for the integral portion the expression is:

$$1 \times 2^{11} + 0 \times 2^{10} + 1 \times 2^9 + 1 \times 2^8 + 1 \times 2^7 + 1 \times 2^6 + 0 \times 2^5 + 1 \times 2^4 + 0 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 0 \times 2^0$$

Note that the decreasing sequence of powers must be carried down to 2^0 and when the remainder is 0 at 2^2, the coefficient must be 0 for 2^1 and 2^0.

Write the coefficients as the binary equivalent,

$$3028._{10} = 101111010100._2$$

The binary fraction is expressed as the coefficient of the negative powers of 2. The negative powers of 2 continue in increasing sequence from 2^{-1} to the highest negative power that is subtracted from the remainder. In the example, 2^{-1} equals .5 which is larger than .3; thus, it is necessary to subtract 2^{-2}, which is the highest power less than .359375. A coefficient of 1 is written for the term whose power can be subtracted; a 0 is written as a coefficient of a power which cannot be subtracted. Thus, for the fractional portion of the number, the binary expression is the sum:

$$0 \times 2^{-1} + 1 \times 2^{-2} + 0 \times 2^{-3} + 1 \times 2^{-4} + 1 \times 2^{-5} + 1 \times 2^{-6}$$

The actual binary fraction is:

$$.359375_{10} = .010111_2.$$

The binary equivalents for the mixed decimal expression are then united as a single binary expression:

$$3028.359375_{10} = 101111010100.010111_2$$

3.4.3 Division – Multiplication Method

A third method of converting from decimal to binary is done in two parts. First, the integral portion of the decimal number is converted, and this will equal the integral portion of the binary number. Then, the fractional portion of the decimal number is converted to the fractional portion of the binary number.

To convert the integral portion of the number, the integral portion is divided by 2; then, the quotient generated is again divided by 2 and so on. The binary coefficients are indicated by a 1 each time the division results in a remainder and by a 0 each time no remainder is obtained. The first binary bit generated is the least significant bit (LSB) of the binary expression; the last bit generated is the most significant bit (MSB). For example, convert $123_{(10)}$ to $N_{(2)}$:

Since $123_{(10)}$ is an integral expression, the division procedure can be used.

DECIMAL NUMBER	QUOTIENT	REMAINDER
123 ÷ 2	61	1 LSB
61 ÷ 2	30	1
30 ÷ 2	15	0
15 ÷ 2	7	1
7 ÷ 2	3	1
3 ÷ 2	1	1
1 ÷ 2	0	1 MSB

$$123_{10} = 1111011$$

To convert a fraction expressed in decimal form to its binary equivalent, the following method may be used. The fraction is repeatedly multiplied by 2. Each time that the multiplication generates a product having an integral portion, a 1 is entered as a coefficient of the equivalent binary expression. (The integer is then dropped.) Each time the product of the multiplication is a fraction, a 0 is entered into the binary sequence of coefficients. This process is continued until the required number of binary bits has been generated or until a product which is entirely integral is generated, in which case an exact conversion has been obtained. The binary fractional numbers are expressed in order of generation in relation to the binary point. For example, convert $0.875_{(10)}$ to the equivalent binary number:

DECIMAL CALCULATION	BINARY REPRESENTATION
.875 x 2 = 1.75	.1
.75 x 2 = 1.5	.11
.5 x 2 = 1.0	.111
.0 x 2 = 0	.1110

$$0.875_{(10)} = .1110_{(2)}$$

Sometimes the decimal fractions do not have an exact binary equivalent. In such a case the number of bits to be generated is determined by the precision of conversion required. Each bit generated makes the conversion more nearly exact. The conversion of a decimal mixed number (i.e., has whole and fractional portions) to the equivalent binary expression is performed using both procedures illustrated above. Separate the expression into the integral portion and the fractional portion, and convert each in accordance with the outlined procedure.

For example convert 130.359375 to N_2:

INTEGRAL PORTION

DECIMAL	QUOTIENT	REMAINDER
130 ÷ 2	65	0 LSB
65 ÷ 2	32	1
32 ÷ 2	16	0
16 ÷ 2	8	0
8 ÷ 2	4	0
4 ÷ 2	2	0
2 ÷ 2	1	0
1 ÷ 2	0	1 MSB

$$130_{10} = 10000010_2$$

FRACTIONAL PORTION

MULTIPLY x 2	BINARY RESULT
.359375 x 2 = .718750	.0
.718750 x 2 = 1.437500	.01
.437500 x 2 = .875000	.010
.875000 x 2 = 1.750000	.0101
.750000 x 2 = 1.50000	.01011
.50000 x 2 = 1.0000	.010111

$$.359375_{10} = .010111_2$$

Combining both operations, the mixed binary equivalent is:

$$130.359375_{10} = 10000010.010111_2$$

3.5 OCTAL TO DECIMAL CONVERSION

Octal to decimal conversion is usually accomplished by use of the general rules given in paragraph 2.2. As an example of this type of conversion by use of the general rules, convert the octal number $227.42_{(8)}$ into a decimal number:

$$227.42_{(8)} = 2 \times 8^2 + 2 \times 8^1 + 7 \times 8^0 + 4 \times 8^{-1} + 2 \times 8^{-2}$$

$$= 2 \times 64 + 2 \times 8 + 7 \times 1 + 4 \times .125 + 2 \times .015625$$

$$= 128 + 16 + 7 + .5 + .031250$$

$$227.42_{(8)} = 151.53125_{(10)}$$

Notice that in the terms of the formula: (1) R= 8 is the base of the system from which the number is being converted; (2) R and C are expressed in the decimal system, the system to which the number is being converted; and (3) the addition of terms to get the final decimal number is done in decimal arithmetic.

3.6 DECIMAL TO OCTAL CONVERSION

Three methods of decimal to octal conversion are given below. The first is the general method of paragraph 2.2; the other two are somewhat easier methods which avoid use of octal arithmetic.

3.6.1 General Method

A conversion of a decimal number to its octal equivalent is possible using the general rules. As an example, convert the decimal number $151.53125_{(10)}$ to its octal equivalent:

$$151.53125_{(10)} = 1 \times (12)^2 + 5(12)^1 + 1(12)^0 + 5(12)^{-1} + 3(12)^{-2} + 1(12)^{-3} + 2(12)^{-4} + 5(12)^{-5}$$

$$151.53125_{(10)} = 1 \times 144 + 5 \times 12 + 1 \times 1 + \frac{5}{12} + \frac{3}{12^2} + \frac{1}{12^3} + \frac{2}{12^4} + \frac{5}{12^5}$$

$$151.53125_{(10)} = 144 + 62 + 1 + \frac{5(12)^4}{12^5} + \frac{3(12)^3}{12^5} + \frac{1(12)^2}{12^5} + \frac{2(12)}{12^5} + \frac{5}{12^5}$$

$$151.53125_{(10)} = 227.42_{(8)}$$

In this conversion operation: (1) R = 10 is the radix of the number system from which the number is being converted; (2) R and C are both expressed in octal – the number system to which the number is being converted; and (3) the multiplication of the powers by their coefficients and the final summing of the terms is accomplished in octal arithmetic (see Ch. 5).

3.6.2 Radix Subtraction Method

The same basic principles are used in converting by the radix subtraction method from decimal to octal as were used in the decimal to binary conversion. The subtraction operation is performed in order to produce the coefficient of the powers, and the powers are indicated by the sequential arrangement as in the binary conversion operation. For example, convert the decimal number $1597.25_{(10)}$ to its octal equivalent.

The first step in the conversion is to separate the integral portion of the decimal expression from the fractional portion and to operate on each separately. From table 2–2, find the highest power of 8 that can be subtracted. Continue the subtraction until the remainder is 0. Thus:

INTEGRAL PORTION		FRACTIONAL PORTION	
	1597	.250	
8^3	− 512	−.125	8^{-1}
	1085	.125	
8^3	− 512	−.125	8^{-1}
	573	0	
8^3	− 512		
	61		
8^1	− 8		
	53		
8^1	− 8		
	45		
8^1	− 8		
	37		
8^1	− 8		
	29		
8^1	− 8		
	21		
8^1	− 8		
	13		
8^1	− 8		
	5		
8^0	− 1		
	4		
8^0	− 1		
	3		
8^0	− 1		
	2		
8^0	− 1		
	1		
8^0	− 1		
	0		

TABLE 2–2. POSITIVE AND NEGATIVE POWERS OF 8

POSITIVE POWERS		NEGATIVE POWERS	
$8^0 =$	1	$8^0 = 1$	$= 1$
$8^1 =$	8	$8^{-1} = 1/8$	$= .125$
$8^2 =$	64	$8^{-2} = 1/64$	$= .015625$
$8^3 =$	512	$8^{-3} = 1/512$	$= .001953$
$8^4 =$	4,096	$8^{-4} = 1/4,096$	$= .000244$
$8^5 =$	32,768	$8^{-5} = 1/32,768$	$= .000031$

The next step in this conversion procedure is to write in sequence the powers of the octal base, starting at the highest power that was subtracted from the integral expression. Thus,

$$8^3\ 8^2\ 8^1\ 8^0.$$

The coefficients of the terms are expressed as the number of times the power was subtracted from the decimal expression or any remainder. In the case of 8^3, the coefficient is 3 because 512 was subtracted three times from the expression that was being converted to the octal equivalent. The coefficients are $3x8^3 + 0x8^2 + 7x8^1 + 5x8^0$. Writing the octal equivalent in terms of the integral coefficients, $N_8 = 3075_{(8)}$. The fractional portion of the octal expression is written as the negative power of the base times a coefficient. The coefficient is the number of times the power was subtracted from the expression to be converted. In the example, 8^{-1} was subtracted twice and the coefficient is 2. Writing the fractional equivalent of $.25_{10}$, $N_8 = 2\times8^{-1}$ or .2. Combining both operations, the result is expressed:

$$1597.25_{(10)} = 3075.2_{(8)}.$$

3.6.3 Multiplication – Division Method

This method of conversion is accomplished in two parts. The integral and fractional portion of the number are converted differently. The integral portion is converted by a dividing process; the fractional portion by a multiplying process. To convert the integral portion of the decimal number to octal, divide the integral portion by 8, then divide the quotient by 8 and so on until the quotient is 0. The remainders obtained in the division will be the coefficient of the octal number with the least significant digit (LSD) generated first and the most significant digit (MSD) generated last. As an example, convert the decimal number $3844_{(10)}$ to its octal equivalent:

DECIMAL NUMBER	QUOTIENT	REMAINDER
$3844 \div 8$	480	4 LSD
$480 \div 8$	60	0
$60 \div 8$	7	4
$7 \div 8$	0	7 MSD

$$3844_{(10)} = 7404_{(8)}$$

To convert the fractional portion of a decimal number to octal, multiply the fraction by 8. The integral portion of the product thus obtained is the first digit of the octal fraction. The fractional portion of the product should be multiplied by 8. This time, the integral part of the product obtained is the second digit of the octal fraction. This process continues until the fraction of the product is 0, or until enough octal digits have been generated. For example, convert the decimal number $0.384_{(10)}$ to its octal equipment.

DECIMAL NUMBER	PRODUCT	INTEGRAL PORTION
0.384×8	3.072	3 MSD
0.072×8	0.576	0
0.576×8	4.608	4
0.608×8	4.864	4 LSD

$$0.384_{(10)} = 0.3044\ldots_{(8)}$$

A mixed number is converted by means of these operations performed upon the integral and fractional portion of the number, separately. For example, convert the decimal number $204.53125_{(10)}$ to its octal equivalent. First convert the integral portion:

DECIMAL NO. ÷ RADIX	QUOTIENT	REMAINDER
$204 \div 8$	25	4 LSD
$25 \div 8$	3	1
$3 \div 8$	0	3 MSD

Therefore, the integral conversion is $204_{(10)} = 314_{(8)}$. Now convert the fractional portion of the decimal number.

DECIMAL NO. × RADIX	PRODUCT	OCTAL DIGIT
$.53125 \times 8$	4.25000	4 MSD
$.25 \times 8$	2.00	2 LSD

Therefore, the fractional conversion is $.53125_{(10)} = .42_{(8)}$ and the full conversion is $204.53125_{(10)} = 314.42_{(8)}$.

3.7 OCTAL TO BINARY CONVERSION

Two methods of octal to binary conversion are given below. One is by use of the general rules of 2.2. The other, which is the recommended method, is by inspection.

3.7.1 General Method

The general rules may be used to convert an octal number to a binary number. As an example, convert the octal number $26.7_{(8)}$ to its binary equivalent.

$$26.7_{(8)} = 10 \times (1000)^1 + 110(1000)^0 + 111(1000)^{-1}$$

$$26.7_{(8)} = 10000 + 110 + \frac{111}{1000}$$

$$26.7_{(8)} = 10110.111_{(2)}$$

Notice that in the operations:

a. R = 8 is the radix of the number system from which the number is being converted.

b. R and C are both expressed in binary, the number system to which the number is being converted.

c. The multiplication of the powers by their coefficient and the final summing of the terms are accomplished in binary arithmetic.

3.7.2 Inspection Method

The recommended method of octal to binary conversion is by inspection, as there is a natural relationship between octonary and binary numbers. The base of the octal system is 8; the base of the binary system is 2; and 2^3 equals 8. The simple relationship is that one octal digit may be expressed by three binary bits.

OCTAL – BINARY	OCTAL – BINARY
0 = 000	4 = 100
1 = 001	5 = 101
2 = 010	6 = 110
3 = 011	7 = 111

The conversion from the octonary number to its binary equivalent can be accomplished by direct substitution. To convert any octonary number to its binary equivalent, replace each octonary digit by the grouping of three binary bits having equivalent value. For example, rewrite $56473.246_{(8)}$ as follows:

101 110 100 111 011 010 100 110

Here the groupings are separated just to call attention to the equivalences; in actual practice, there is no reason why the binary number cannot be written directly, with no spacing between groupings unless it is desired to retain groupings to facilitate checking.

3.8 BINARY TO OCTAL CONVERSION

Two methods of binary to octal conversion are given below. One is by means of the general rules given in 2.2. The other, which is the recommended method is by inspection.

3.8.1 General Method

A conversion from a binary to an octal number may be accomplished using the general formula. As an example of the operation, convert 10110.111 to an octal number.

$$10110.111_{(2)} = 1 \times 2^4 + 0 \times 2^3 + 1 \times 2^2 + 1 \times 2^1 + 0 \times 2^0 + 1 \times 2^{-1} + 1 \times 2^{-2} + 1 \times 2^{-3}$$

$$10110.111_{(2)} = 20 + 0 + 4 + 2 + 0 + .4 + .2 + .1$$

$$10110.111_{(2)} = 26.7_{(8)}$$

Notice that in the terms of the formula:

a. R = 2 is the base of the system from which the number is being converted.

b. R and C are expressed in the octal system, the number system to which the number is being converted. (Note particularly the fractional equivalencies.)

c. The addition of terms to get the final octal number is done in octal arithmetic.

3.8.2 Inspection Method

The recommended method of conversion from binary to octal is by substitution of the octal equivalents for the binary groups. To make this conversion, arrange the binary bits in groups of three, beginning at the binary point proceeding to the left and to the right. Fill out the extreme left or right group with 0's if necessary. Then directly substitute for each binary group its octal digit equivalent.

For example, convert 11100.111_2 to N_8

$$11100.111_2 = 011 \quad 100 \cdot 111_2$$

$$\begin{array}{ccc} 001 & 100. & 111_2 \\ \downarrow & \downarrow & \downarrow \\ 3 & 4. & 7 \end{array}$$

$$11100.111_2 = 34.7_{10}$$

CHAPTER 4
BINARY ARITHMETIC

4.1 ADDITION

4.1.1 General Rules

Binary addition is simple. Its rules are as follows:

$0 + 0 = 0$

$1 + 0 = 1$

$0 + 1 = 1$

$1 + 1 = 0 + 1$ to carry

These rules operate in all cases of addition and apply to both addition of integers and of fractions. Binary numbers are added from right to left, and the carry is added to the adjacent bit on the left. The following examples illustrate the rules for binary addition. Note that the carry is placed in the column, to which it will be added, in parentheses.

			(11)		(1)
0	1	10	11	100	101
+1	+1	+1	+1	+1	+1
1	10	11	100	101	110

The technical terms in addition are defined as the augend, addend, and the sum. The augend is the term that is to be increased; the addend is the term to be added to the augend; the sum is the result of the operation. For example:

101	Augend
+011	Addend
1000	Sum

4.1.2 Addition of Binary Numbers

In adding more than one number, the addition of the first set of numbers is performed and, to the sum, is added the third number. To the sum of the succeeding additions, add the next number until all the numbers have been totaled. For example, add:

a.
011
111
+110

	BINARY	DECIMAL
Addition of the first set of numbers	011	3
	+111	+7
First sum	1010	10
Addition of the third number	+110	+6
Final sum	10000	16

b.
1101
1001
0010
+1111

	BINARY	DECIMAL
Addition of the first set of numbers	1101	13
	1001	9
First sum	10110	22
Addition of the third number to the first sum	0010	+2
Second sum	11000	24
Addition of the fourth number to the second sum	1111	+15
Final sum	100111	39

Binary fractions are added in accordance with the rule that governs whole numbers. The binary point is fixed as in the decimal system. The carry from the addition of the binary fractions in the first position to the right of the binary point is an integer. For example, in the addition of the following fractions:

a. DECIMAL	BINARY
1/8	.001
3/8	.011
4/8 or .5	.100

b.	4/8	.10
	+6/8	.11
	10/8 or 1.25	1.01
c.	5 3/8	101.011
	6 7/8	110.111
	12 2/8 or 12.25	1100.010

4.2 BINARY SUBTRACTION

4.2.1 General

The rules for binary subtraction are as follows:

$0 - 0 = 0$

$0 - 1 = 1$ (borrow 1 and make $0 = 10$)

$1 - 0 = 1$

$1 - 1 = 0$

The technical definitions of the terms used in subtraction are minuend, subtrahend, and difference. The minuend is the number to be decreased; the subtrahend is the quantity of the decrease; the difference is the result of the operation. Thus:

0110 Minuend
100 Subtrahend
010 Difference

The similarity which exists between decimal and binary arithmetic when a carry is involved is analogous to the similarity which exists when a borrow is invovled. When subtracting a 1 from a 0, a 1 must be borrowed from the next higher order, diminishing that order by 1.

4.2.2 Direct Subtraction

The following examples illustrate the rules for binary subtraction and the method of borrowing from the next higher order.

	1101	1110	1100
a.	−0100	b. −0101	c. −1001
	1001	1001	0011

In example a, above, the subtraction of 0 from 1, 0 from 0, and 1 from 1 produces the difference. In example b, a 1 must be borrowed from the second order when attempting to subtract the 1 of the first order from 0. The 1 in the second order then diminishes to 0. In example c, a slightly different borrow situation arises. The 1 to be borrowed must come from the third order of the minuend. That 1 then diminishes to 0. The 1 of the first order of the minuend can then be borrowed from the 10 which appears in the second order. Borrowing the 1 from 10 leaves a 1 in the second order of the minuend. Applying the rules of binary subtraction then produces the difference shown.

Fractions are subtracted according to the rules and procedures for integral expressions. The procedures are the same as the ones for decimal subtraction; the rules are the binary rules for subtraction. For example, subtract:

	DECIMAL	BINARY
a.	21/32	.10101
	−16/32	.10000
	5/32 or .156	.000101
b.	4 3/4	100.11
	−2 1/2	10.10
	2 1/4 or 2.25	10.01

4.2.3 Complementing Method in Binary Subtraction

4.2.3.1 General

The examples that have been used to illustrate subtraction are methods of direct subtraction. The complement method of subtraction is a means of subtraction by addition. Computer design requirements do not allow for borrowing, so the complement method of subtraction fits in with computer design and capabilities.

A disadvantage of direct binary subtraction is tnat the direct subtraction of a number from a smaller number yields an incorrect result unless the subtraction is done by subtracting the smaller from the larger and then changing the sign of the difference. Such a procedure would be difficult in a computer. For example:

5/16	0.0101
−9/16	−0.1001
−4/16	?

The difficulty encountered with negative results and the problem of providing for borrowing in computer design are eliminated by converting the subtraction to an addition of negative numbers by means of the complement process.

4.2.3.2 Modulus

The complement system of subtraction is possible because it is possible to limit the number of significant digits to be used in any one problem or machine. The problem is then said to have a modulus, which is the count of the maximum number of numbers it would be possible to represent in this problem. For instance, suppose that a binary machine has facilities for handling only 4 places — the machine could represent 16 different numbers from 0 to $1111_{(2)}$. Such a machine has a modulus of 16 and is said to perform modulo 16 arithmetic.

The significance of the modulus of the machine is that each time an addition results in a number equal to or greater than the modulus of the machine, an integral multiple of the modulus is lost. An example of this action in everyday life is furnished by the automobile odometer. When it reaches 100,000 miles, it resets to zero and starts over. The odometer has lost 100,000 by resetting to 0. This property of machine-counting methods is important in the use of complements for subtraction by addition.

4.2.3.3 Derivation of Complement Method of Subtraction

The complement method of subtraction may be derived from the following identity:

$$P - M + (M - N) = P - N$$

where:

P = Minuend

N = Subtrahend

M = Modulus of the machine

P − N = difference sought.

To derive the complement system of subtraction, let (M − N) equal a number called the complement of N. Let C stand for this complement so M − N = C. Now substitute C in the identity:

$$P - M + C = P - N$$

or

$$(P + C) - M = P - N.$$

If M is moved to the other side of the identity, it becomes:

$$P + C = M + (P - N)$$

It now is evident that the minuend plus the complement of the subtrahend is equal to the difference of the minuend and subtrahend plus the modulus. It should now be recalled that when two numbers are added to obtain a sum greater than the modulus, the modulus is lost. Therefore:

$$P + C = P - N$$

in any system with a fixed modulus, provided only that the sum P + C is greater than the modulus of the number system used.

The above is a derivation of what, in binary arithmetic, is called the 2's complement system. A similar derivation of a 1's complement system may be derived using (M − 1) in place of M. In this case, however, the final equation will be:

$$P + C_1 - 1 = P - N$$

which implies that the difference sought will be found by adding 1 to the P + C_1. Note that C_1 is equal, in this case, to (M − 1) − N.

4.2.3.4 Generation of 1's Complement

Every computer has a modulus which is one larger than the largest number the computer can register. For example, a 6-place binary counter could express all the numbers from 0 to $111111_{(2)}$. The modulus of such a computer is $1{,}000{,}000_{(2)}$.

To obtain the 1's complement of a number, it was shown in the derivation above that the number must be subtracted from (M − 1). Therefore, to obtain the 1's complement of a number in a 6-place machine, the number is subtracted from $(1000000 - 1)_{(2)}$; that is, from $111111_{(2)}$. As an example, find the 1's complement of the binary numbers $101001_{(2)}$ and $01101_{(2)}$:

a.

```
111111  Modulus — 1
101001  Number
------
010110  1's complement of number
```

b.

```
111111  Modulus — 1
001101  Number
------
110010  1's complement of number
```

A close examination of the numbers and their 1's complements will show that the 1's complement in binary arithmetic is nothing more than the original number with bits reversed. That is, the original number's 0's are 1's in the complement while the 1's are 0's. The way to get the 1's complement, then, is by inspection; just exchange 0's for 1's and 1's for 0's. For example:

```
100101  number
0110010 complement.
```

4.2.3.5 1's Complement Subtraction

To perform subtraction by the 1's complement method, proceed in the following manner:

1. Find the complement of the subtrahend with respect to 1's.
2. Add the complement to the minuend.
3. Perform "end-around carry" if there is a carry out of the highest position of the difference. (This is explained below.)

The result is the difference in complement form if it is negative and in true form if it is positive. (Zero is considered negative.)

There are four possibilities, as shown by the examples below. All except the last will be treated exactly the same. The last will require the extra step of end-around carry. This is a carry from the highest order around to the lowest order which is required because of

the cyclical nature of the number system. The only time it is required is when the minuend is larger than the subtrahend, that is, when the answer will come out a true positive answer. Fortunately, whenever it is required, there is a carry from the left-most position, which serves as a reminder.

	EXAMPLES	DIRECT SUBTRACT	COMPLEMENT SUBTRACT
a.	Minuend is	+011011 Minuend	011011 Minuend
	smaller than	—101010 Subtrahend	010101 Subtrahend 1's complement
	subtrahend	—001111 Difference	110000 Complement of difference
b.	Minuend is	+011011 Minuend	011011 Minuend
	equal to	—011011 Subtrahend	100100 Subtrahend 1's complement
	subtrahend	000000 Difference	111111 Complement of difference
c.	Minuend is	—011011 Minuend	100100 Complement of minuend
	more negative	—(—)010011 Subtrahend	010011 Subtrahend
	than subtrahend	—001000 Difference	110111 Complement of difference
	and both are negative		
d.	Minuend	011011 Minuend	011011 Minuend
	is larger	—010101 Subtrahend	101010 Subtrahend 1's complement
	than subtrahend	+000110 Difference (1)	000101 True difference less 1
		↓	
		End carry ——→ (1)	
			000110 True difference

4.2.3.6 Generation of 2's Complements

In the derivation of the complement system, it was shown that a 2's complement of a number is equal to the modulus minus the number, (M — N). Therefore, to obtain a 2's complement in a 6-place machine, the number is subtracted from the modulus, 1,000,000. As an example, find the 2's complement of the binary numbers $101001_{(2)}$ and $001101_{(2)}$:

a. 1000 000 Modulus
101 001 Number
010 111 2's complement of number

b. 1000 000 Modulus
001 101 Number
110 011 2's complement of number

A close examination of the numbers and their complements will show that the 2's complement of a number is the same as the 1's complement with a 1 added to it. The 2's complement, therefore, may be formed by forming the 1's complement and adding a 1 to it. As an example, form the 2's complement of $001101_{(2)}$:

001101 Number
110010 1's complement of number
110011 2's complement of number

4.2.3.7 2's Complement Subtraction

To perform subtraction by the 2's complement method:

1. Find the 2's complement of the subtrahend.
2. Add this complement to the minuend.

The result is the difference in complement form if it is negative and in true form if it is positive. (Zero is considered positive.)

In the 2's complement system, there is no need to end-around carry. The results will always be correct without it. For example, solve examples b and d of 4.2.3.5 by the 2's complement system:

EXAMPLE	DIRECT SUBTRACT	COMPLEMENT SUBTRACT
b. Minuend is equal to subtrahend	011011 Minuend —011011 Subtrahend 000000 Difference (1) ↑ Discard Carry	011011 Minuend 101011 2's complement of subtrahend 000000 Difference
d. Minuend is larger than subtrahend	011011 Minuend —010101 Subtrahend +000110 Difference (1) ↑ Discard Carry	011011 Minuend 101011 2's complement of subtrahend 000110 True difference

4.2.3.8 Binary Sign Conventions

At this point, it is natural to raise the question of how negative numbers in complement form can be distinguished from positive numbers in true form. It turns out that in this regard, also, binary numbers offer an advantage with regard to representation. The sign of a number is binary in nature, that is, a number is either positive or negative with the exception of 0, which can be arbitrarily assigned a sign. Thus, a bit representing the sign can be used in addition to the bits representing magnitude. A 0 in the sign bit position can be interpreted to mean that the number is positive and in true form; a 1 in the sign bit position can be interpreted to mean that the number is negative and in complement form. If the sign bits are assigned to the most significant bit position and are treated as a part of the number in the addition operation, the resultant sign bit will be a true indication of the sign of the result. This sign operation is legitimate in both the 1's and 2's complement systems. In order to see how this works, four cases may be considered. The examples of subtractions listed in 4.2.3.5 are repeated below, with the sign bits added to illustrate the results of operating on sign bits in this manner. The sign bits are on the left, separated from the magnitude bits by a point (.). This is the usual practice.

Example:

a. Complement Subtraction
0.011011 Minuend
1.010101 1's complement of subtrahend
1.110000 1's complement of difference

b. 0.011011 Minuend
1.100100 1's complement of subtrahend
1.111111 1's complement of difference (this is often called negative zero)

c. 1.100100 1's complement of minuend
0.010011 Subtrahend
1.110111 1's complenment of difference

d. 0.011011 Minuend
1.101010 1's complement of subtrahend
(1) 0.000101
↓
End Carry ——→ 1
0.000110

From the examples, it can be seen that a sign bit may be used in subtraction as though it were a magnitude bit. It will then always indicate the sign. As a matter of fact, the sign bit increases the capacity of the machine twofold. This is because it allows the same number of magnitude bits to signify both positive and negative numbers.

4.2.3.9 Comparison of 1's and 2's Complement Subtraction

A study of the examples of 4.2.3.5 and 4.2.3.7 will show some of the advantages and disadvantages of the two systems. The chief advantage of the 1's complement system is the ease with which the complement is formed. Its chief disadvantage are the end-around carry operations, sometimes necessary, and the fact that negtive zero must be provided for. It would appear that the 1's complement system would be most suited to operations where the complementation process itself is a major part of the operation. The 2's complement process, on the other hand, would be advantageous in operations where the addition process is most important.

4.3 BINARY MULTIPLICATION

4.3.1 General Method

The rules for binary multiplication are similar to those of decimal multiplication. The rules for multiplying two single digits are the same in both systems. These rules are:

$$0 \times 0 = 0 \qquad 1 \times 0 = 0$$

$$0 \times 1 = 0 \qquad 1 \times 1 = 1$$

The general procedure when multiplying two multiple digit binary numbers is the same as that in decimal arithmetic. That is, the multiplicand is multiplied by a digit of the multiplier, and the partial product obtained is placed so that the least significant digit is under the multiplier digit. When all the partial products have been found, they are added together to find the final product. The only difference between decimal and binary multiplication, therefore, is in the summing of the partial products. In binary, the binary addition table is used while in decimal the decimal table is used.

As can be seen from the following examples, the method of obtaining partial products and then adding these to obtain the product is identical to that of decimal arithmetic.

Multiplicand	1010	10.11	1111
Multiplier	1101	100.1	1111
First Partial Product	1010	101 1	1111
Second Partial Product	0000	0000	1111
Third Partial Product	1010	0000	0000
Fourth Partial Product	1010	1011	1111
Total Product	10000010	1100.011	11100001

Note the placement of the binary point in the second example. The same rules hold for its placement as hold for placement of the decimal point in arithmetic.

The third example also brings out an interesting point. This is the multiplication of the two largest possible 4-bit numbers. The product is 8 bits long. In other words, the largest product that can result from the multiplication of two numbers will be no longer than the sum of the number of bits in the multiplier and multiplicand.

4.3.2 Add and Shift Multiplication

If a number is multiplied by the radix of the number system, this multiplication has the effect of shifting the number one place to the left with respect to the radix point. This is true in any number system. For example, multiply $12.51_{(10)}$ by 10 (the radix of the decimal system) and multiply the number $10.11_{(2)}$ by $2_{(2)}$ (the radix of the binary system).

	DECIMAL	BINARY
Number	12.51	10.11
Number Times Radix	125.1	101.1

Note that binary multiplication is nothing more than a series of add and shift operations.

4.3.3 Multiplication (or Division) of Negative Numbers

Two ways of multiplying (or dividing) negative numbers are possible. One way, which is seldom used because of its complications, is the multiplication (or division) of complements. This is possible if new rules of arithmetic to take care of the sign are used. The second way of multiplying (and dividing) negative numbers is to change the signs of both multiplier and multiplicand (divisor and dividend) so that they are both positive. Then when the multiplication (division) is finished, the sign of the product (or quotient) is changed according to algebraic rules of multiplication (or division).

4.4 BINARY DIVISION

4.4.1 General

Binary division is the process of counting the number of times that the divisor will go into the dividend. The count of the number of times the divisor may be subtracted from the dividend before a negative remainder results is called the quotient.

4.4.2 Direct Division

Direct binary division is performed by a series of subtractions of the divisor (actually a multiple of the divisor), just as it is in the decimal system. For example, divide $100011100_{(2)}$ by $1110_{(2)}$:

```
                    bd                                  1
                  1010.001
a. 1110/10001110.000        b. 1110/10001110.000
     a  1110                      1110000 000
     c   1111
         1110
          10 000
           1 110
              10
```

In example a, above, the first step in the process is to place the divisor below the dividend in a position which is as far to the left as possible but which will still allow a positive difference to result when the divisor is subtracted from the dividend. Since the divisor will go into this many bits of the dividend once, a 1 is

placed in the quotient at b in the same column as the lowest order digit of the divisor. The divisor is then multiplied by the quotient digit, and the resulting product (which, in this case, is the divisor) is subtracted from the dividend to produce the positive difference (line c) called the current remainder. The next digit in the dividend is brought down to the difference obtained on line c. Now, two procedures are possible. On the one hand, if the new number on line c is larger than the divisor, a 1 is placed in the next empty quotient bit position. The divisor is multiplied by this digit, and the resulting product is then placed under the partial remainder. Then the subtraction process can be repeated to obtain a new current remainder. On the other hand, if the number on line c is less than the divisor, as in the example, a 0 is placed in the quotient bit position, column d. The next digit of the dividend is then brought down to the difference, and a subtraction is performed. The process is continued until a quotient of required length is completed. As shown in the example, the binary radix point is treated the same as the decimal point would be in decimal division.

In b, above, only the first digit of the quotient is shown. When this digit is used to multiply the divisor (to get the number below the dividend) actually, multiplication by $1000_{(2)}$, (1×2^3), is being performed. In other words, the actual subtraction from the dividend is of a number $1000_{(2)}$ times the divisor. From this, it can be seen that the amount subtracted from the dividend is the quotient bit times the power of the quotient bit times the divisor.

Since the quotient bit is always either 0 or 1, the division process could be reduced to a series of subtractions of the divisor, multiplied by the power of the quotient bit being sought from the dividend. Each time a subtraction resulted in a positive current remainder, a 1 would be placed in the corresponding quotient bit position, and the process could be immediately repeated for the next quotient bit. Each time the subtraction resulted in a negative remainder, a 0 would be placed in the corresponding quotient bit. In this case, the current remainder would have to be restored to a positive number by adding the divisor back to it. Following this, the next quotient bit could be obtained by the subtraction of the divisor multiplied by the power of the next quotient bit.

Since the quotient bits are generated from left to right, the power of each quotient bit is one smaller than that of the last bit generated. This means that as the divisor is successively subtracted from the dividend (or current remainder), the divisor is shifted to the right in relation to the binary point. In other words, the division process can be reduced to a process of successive subtract and shift steps.

4.4.3 Division by Subtraction and Shift Methods

4.4.3.1 General

When division is mechanized, either of two subtract and shift methods is generally used. The restoring subtract and shift method is one which, after each unsuccessful trial subtraction, adds the divisor back to the remainder. The nonrestoring technique does not do this; consequently, it is somewhat faster.

Division will always be built into a machine so the quotient bits will start to generate in the correct place. One method of doing this is to make the rule that the first quotient bit will be in the position corresponding to or just above the least significant figure of the divisor when the most significant digits of the dividend and the divisor are lined up for the first trial subtraction. (Note that this first quotient bit could be a 0.)

4.4.3.2 Restoring Method

The restoring division process is as follows:

a. The dividend and divisor are lined up at the left so that their most significant digits are in line. This is the equivalent of multiplying the divisor by the power of the first quotient bit and then lining up the radix points of the dividend and the divisor. The radix points of the divisor, dividend, and quotient are now properly oriented.

b. An attempt is made to subtract the divisor times the power of the first quotient bit from the dividend. If the attempt is successful (result is positive or zero), a 1 is entered in the quotient position corresponding to the power used to multiply the divisor in step a, above. If the subtraction results in a negative answer, a 0 is entered in the quotient, and the divisor is added back to the dividend (the dividend is restored).

c. The divisor is then shifted right one place and, again, a trial subtraction is attempted this time from the current remainder or restored dividend, as the case may be. A 1 is entered in the next most significant quotient bit if the subtraction is successful, a 0 if it is unsuccessful. This process goes on until the required number of quotient bits has been generated.

An example should clarify this operation. The example below illustrates the division of the number $10001100_{(2)}$ by $1110_{(2)}$.

OPERAND NUMBER	OPERAND NAME	OPERATION	QUOTIENT BIT	QUOTIENT ORDER
10001100.	Dividend			
11100000.	Divisor x 2^4	Subtract		
—01010100.	Negative remainder		0	2^4
11100000.	Divisor x 2^4	Restoring Addition		
10001100.	Dividend			
1110000.	Divisor x 2^3	Shift and Subtract		
+00011100.	Positive remainder		1	2^3
111000.	Divisor x 2^2	Shift and Subtract		
— 011100.	Negative remainder		0	2^2
111000.	Divisor x 2^2	Restoring Addition		
+ 11100.	Positive remainder			
11100.	Divisor x 2^1	Shift and Subtract		
+ 00000.	Zero remainder (+)		1	2^1
1110.	Divisor	Shift and Subtract		
— 1110.	Negative remainder		0	2^0
1110.	Divisor	Restoring Addition		
0000.	Final remainder		Final Quotient 01010	

In the example, the dividend is written down and the divisor is written below it with the left-most digits in line. To accomplish this alignment while maintaining alignment of the binary points, it would be necessary to multiply the divisor by $10000_{(2)}$ or 2^4. This indicates that the first subtraction will determine the quotient bit whose power is 2^4. When the subtraction is carried out, it results in a negative current remainder. Therefore, the quotient bit in the 2^4 bit position will be 0. It is now necessary to restore the current remainder to a positive number. This is done by adding the divisor times 2^4 back to the current remainder. To determine the next digit of the quotient, the divisor is shifted one place to the right, and the resulting number is subtracted from the restored current remainder. This time, the subtraction results in a positive current remainder, indicating that the divisor times 2^3 will go into the dividend once. The bit in the 2^3 position of the quotient, therefore, will be 1. Since the current remainder is positive, no restoring add will be required. The next bit of the quotient will simply be determined by shifting the divisor one place to the right and subtracting. This process continues until a sufficient number of quotient bits have been generated or, in this example, until a remainder of 0 has been obtained.

4.4.3.3 Nonrestoring Method

The nonrestoring method of division is as follows:

a. The dividend and divisor are lined up left so that their most significant digits are in line. This is the equivalent of multiplying the divisor by the power of the first quotient digit and then lining up the radix points of the dividend and divisor. The radix points of the dividend, quotient, and divisor will then be in corresponding places.

b. A trial subtraction is made. If it is successful, a 1 is put into the quotient position corresponding to the power used in step a. If it is unsuccessful, a 0 is put in the quotient bit position.

c. Without restoring the dividend, a shift-right of the divisor is made. If the previous trial subtraction resulted in a negative remainder, the divisor will now be added to this remainder. If the previous trial subtraction resulted in a positive remainder, the divisor will be subtracted from the remainder. If the results of this addition or subtraction, as the case may be, are positive, a 1 is inserted in the quotient; if they are negative, a 0 is inserted.

d. The same routine is repeated until all required bits are generated.

e. The last current remainder may be a positive or a negative number. If it is a negative number, it

must be restored. This is done by adding the divisor back to the current remainder. As an example of this method, the division of 4.4.3.2 is repeated by this method.

OPERAND NUMBER	OPERAND NAME	OPERATION	QUOTIENT BIT	QUOTIENT ORDER
10001100.	Dividend			
11100000.	Divisor x 2^4	Subtract		
—01010100.	Negative remainder		0	2^4
1110000.	Divisor x 2^3	Shift and add		
+ 0011100.	Positive remainder		1	2^3
111000.	Divisor x 2^2	Shift and subtract		
— 011100.	Negative remainder		0	2^2
11100.	Divisor x 2^1	Shift and add		
+ 00000.	Positive remainder		1	2^1
1110.	Divisor x 2^0	Shift and subtract		
— 1110.	Negative remainder		0	2^0
1110.	Restoring add			
0000.	Final remainder		Final Quotient = 1010	

In the example, the dividend and the divisor are lined up on the left. To do this while maintaining orientation of the binary points of dividend and divisor, the divisor would have to be multiplied by 2^4. Therefore, the content of the 2^4 quotient bit position will be the first determined. Subtraction of the divisor from the dividend results in a negative current remainder. This indicates that the bit in the 2^4 position of the quotient will be 0. The next step is to shift the divisor to the right one place and *add* it to the current remainder. The combination of the first subtraction of 2^4 times the divisor from the dividend and the addition of 2^3 times the divisor is the equivalent of subtracting 2^3 times the divisor. Since this operation results in a positive current remainder, a 1 is placed in the 2^3 bit position. The next step is to shift the divisor to the right one place and, because the current remainder is positive, to subtract the divisor. This subtraction results in a negative number, so a 0 is placed in the 2^2 bit position of the quotient. Again, a shift-right of the divisor is made. This time, because the current remainder is a negative number, the divisor is added to the current remainder. Since the resulting new current remainder is a positive number, a 1 is placed in the 2^1 bit position of the quotient. Once more, a shift right of the divisor is made, and this time the divisor is subtracted from the current remainder. This last subtraction results in a negative number, so a 0 is placed in the 2^0 bit position of the quotient. Since the final current remainder is a negative number, a restoring addition is necessary. The divisor is added back to the current remainder to give the final remainder of 0.

4.4.4 Nonrestoring Division Using Complement Subtraction

As previously stated, subtraction is usually performed in a machine by means of addition of complements. When this is done in division, the actual division process does not change except that the negative current remainders will appear in complement form. The quotient bit generated will still be a 1 if the current remainder is a positive number and will be 0 if the current remainder is a negative (complement) number.

An example of the nonrestoring process is given below. The same numbers are used as were used in 4.4.3.3. Two things should be noted about the operation in the example. First, the 2's complement system is used. This is done primarily so that no provision for the end-carry operation will have to be made. The second point to notice is that when the current remainder is shifted to the right, the bit positions seemingly vacated on the left actually are having 0's shifted into them. If it is recalled that 1 is the complement of 0, it will become obvious that when a complemented number is shifted to the right the left-hand bits will be filled with 1's. This fits in with the convention of using a 1 in the sign bit to indicate a complement number and 0 to indicate a true number. When this convention is used and a shift-right occurs, the bits on the left of the current remainder will be filled with the same bit contained in the sign bit position. The sign bits themselves,

however, are not changed by the right-shift. The general procedure for the nonrestoring division by complement is as follows:

1. Write down the dividend and the 2's complement of the divisor so that their most significant bits are in line. This is the equivalent of multiplying the 2's complement of the divisor by the power of the first quotient bit.
2. Add the two numbers. If a true current remainder results, place a 1 in the quotient bit position corresponding to the power used in step 1. If a complement current remainder results, place a 0 in the quotient bit position.
3. The divisor is now shifted right. If the current remainder is a true number, complement the divisor and add it to the current remainder. If the current remainder is a negative number, add the true divisor to it.
4. If the results of this second addition are positive, insert a 1 in the next quotient position; if they are negative, insert a 0.
5. Repeat this process until all required bits of the quotient are generated.
6. The last current remainder may be a positive or a negative number. If it is a negative number, it must be restored. Do this by adding the true value of the divisor back to the current remainder.

The example of the process follows. The number $10001100_{(2)}$ is divided by $1110_{(2)}$.

SIGN BITS	MAGNITUDE BITS	NAME	OPERATION	QUOTIENT BIT-ORDER	
0	10001100	Dividend			
1	00100000	2's complement of divisor x2^4	Add		
1	10101100	Current remainder (negative)		0	2^4
0	01110000	True divisor shifted right	Add		
0	00011100	Current remainder (positive)		1	2^3
1	11001000	2's complement of divisor shifted right	Add		
1	11100100	Current remainder (negative)		0	2^2
0	00011100	True divisor shifted right	Add		
0	00000000	Current remainder (positive)		1	2^1
1	11110010	2's complement of divisor shifted right	Add		
1	11110010	Current remainder (negative)		0	2^0
0	00001110	True divisor	Add		
0	00000000	Final remainder		Final Quotient 1010	

In this example, the dividend and complement of the divisor are lined up on the left. Since this is the equivalent of multiplying the complemented divisor by 2^4, the first quotient bit generated will be inserted in the 2^4 order of the quotient. Addition of the two numbers results in a negative current remainder; so a 0 is entered in the 2^4 position of the quotient. Since the current remainder is a negative number, the true divisor will be shifted right and added to the current remainder. This addition results in a positive number; so a 1 will be inserted in the next quotient bit. Because the last current remainder was a positive number, the next operation will be to add the complement of the shifted divisor to the current remainder. This process continues until the four bits of the quotient have been generated. At this time, the remainder is negative, so a final restoring add is required.

CHAPTER 5
OCTAL ARITHMETIC OPERATIONS

5.1 GENERAL

The arithmetic operations, addition, subtraction, multiplication, and division, are performed in the octal system in a manner similar to the decimal operations, although octal counting is difficult. Table 2–3, for octal addition and subtraction, and table 2–4, for use in octal multiplication, should be used to check examples. Octal arithmetic may also be performed in a roundabout fashion by converting to decimal, performing the required operation, and then converting back to octal. In the following paragraphs, only the table methods will be shown; conversion methods are given in Chapter 3.

5.2 OCTAL ADDITION

Octal addition is performed in much the same way as decimal addition. A sum and carry technique is used in which the sum and carry are determined by reference to an addition table. Addition and subtraction are given in table 2–3. The sum of two digits is found where the column containing the addend digit and the row containing the augend digit intersect. For example, the sum of 7 and 6 is 15. The difference of two digits is found in the difference column. Find the minuend which is in the same column as the subtrahend digit. The row which contains this minuend also contains the difference. For instance, $12 - 6 = 4$. Examples in the use of this table for addition are given below. Other than the difference in addition tables used, the addition processes used in octal and decimal are exactly the same. The carries are shown in parentheses.

(1)	(1)	(112) carries
271.1	254.5	262.3
314.3	311.3	351.7
605.4	566.0	434.7
		1271.1

TABLE 2–3. OCTAL ADDITION – SUBTRACTION

(DIFFERENCE) AUGEND		(SUBTRAHEND) ADDEND	(SUBTRAHEND) ADDEND 0	1	2	3	4	5	6	7	10
0		0	0	1	2	3	4	5	6	7	10
1		1	1	2	3	4	5	6	7	10	11
2		2	2	3	4	5	6	7	10	11	12
3	(Difference)	3	3	4	5	6	7	10	11	12	13
4	Augend	4	4	5	6	7	10	11	12	13	14
5		5	5	6	7	10	11	12	13	14	15
6		6	6	7	10	11	12	13	14	15	16
7		7	7	10	11	12	13	14	15	16	17
10		10	10	11	12	13	14	15	16	17	20
						(Minuend) Sum					

5.3 OCTAL SUBTRACTION

Octal subtraction may be performed directly, as in decimal arithmetic, by a subtract and borrow routine. In this case, the subtract portion of table 2—3 is used. When the minuend is smaller than the subtrahend, a 1 must be borrowed from the next column to the left. This is indicated in the table by the two digit minuend numbers. In these numbers, the 1 stands for a carry whereas the other digit stands for the minuend. Examples of the use of this table and the general methods of subtraction are given below. The borrows are shown in parentheses above the column they were borrowed from.

	(11) borrows	(111) borrows
7254.3	7356.3	5432.3
6132.2	7266.6	4567.0
1122.1	0067.5	0643.3

5.4 OCTAL MULTIPLICATION

The operations used in octal multiplication are similar to the operations used in decimal arithmetic. The multiplicand is multiplied by one digit of the multiplier at a time to form a series of partial products that must be added to obtain the desired result. The digit-by-digit multiplications are performed using the products given in the octonary multiplication table, and the sums are obtained using the octal addition table. The position of the octonary point in the product, if either or both of the original numbers are fractional, is determined exactly as in decimal multiplication; that is, if there are two digits to the right of the octonary point in the multiplier and four digits to the right of the octonary point in the multiplicand, the point is positioned six places to the left of the least significant digit in the product.

Table 2—4 is a combination multiplication and division table. To use it for multiplication, read the numbers corresponding to the labels not in parentheses. To use it as a division table, read the numbers corresponding to labels in the parentheses.

In multiplication, the product is found at the intersection of the column containing the multiplicand digit and the row containing the multiplier digit. For instance, 6 x 7 = 52. In division, the quotient digit is found by searching the column which contains the division digit for the corresponding dividend digit (or digits). Then the row which contains this dividend digit intersects the quotient column where the proper quotient digit is located. For instance, 43 ÷ 7 = 5. As an example, using the table, multiply $462_{(8)}$ by $35_{(8)}$.

TABLE 2—4. OCTAL MULTIPLICATION — DIVISION

		MULTIPLICAND (DIVISION)										
		0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**10**	**11**	**12**
	0	0	0	0	0	0	0	0	0	0	0	0
	1	0	1	2	3	4	5	6	7	10	11	12
	2	0	2	4	6	10	12	14	16	20	22	24
Multiplier	**3**	0	3	6	11	14	17	22	25	30	33	36
(Quotient)	**4**	0	4	10	14	20	24	30	34	40	44	50
	5	0	6	12	17	24	31	36	43	50	55	62
	6	0	6	14	22	30	36	44	52	50	66	74
	7	0	7	16	25	34	43	52	61	70	77	106
	10	0	10	20	30	40	50	60	70	100	110	120
	11	0	11	22	33	44	55	66	77	110	121	132
	12	0	12	24	36	50	62	74	106	120	132	144
		Product (Dividend)										

$$
\begin{array}{r}
462 \\
\underline{\times\ 35} \\
2772 \\
\underline{1626\ \ } \\
21252
\end{array}
$$

The multiplicand is multiplied by each digit of the multiplicand, and the carry is added to the product of each individual multiplication. From the table, 5 x 2 = 12, write 2 and carry 1; from the table 5 x 6 = 36 + 1 (carry) = 37, write 7 and carry 3; from the table 5 x 4 = 24 + 3 = 27, write both digits. The same procedure is followed in the multiplication by 3. After the partial products are found, they are added according to the octal addition table.

5.5 OCTAL DIVISION

Octal division is performed like decimal division except that the octal division and subtraction tables are used instead of decimal. As an example of octal division, divide $21252_{(8)}$ by $35_{(8)}$.

$$
\begin{array}{r}
462_8 \\
35_8\overline{)21252_8} \\
\underline{164\ \ } \\
265 \\
\underline{256\ \ } \\
72 \\
\underline{72\ \ } \\
0
\end{array}
$$

$$
\begin{array}{r}
306_{10} \\
29_{10}\overline{)8874_{10}} \\
\underline{87\ \ } \\
174 \\
\underline{174\ \ } \\
0
\end{array}
$$

The most significant number in the quotient is generated by examining the division of 212 by 35 and deciding, on a trial basis, the largest number that 35 can be multiplied by, resulting in a product less than 212. The number selected is 4; this is because 5 x 35_8 = 221_8. The multiplication of 4 x 35 is performed using the octonary multiplication table. The subtraction of the product from 212 is performed by direct octal subtraction. The process is continued as in decimal long division until the required number of octal digits have been generated. Note that the divisors, dividends, and quotients agree in magnitude, but that the intermediate steps are different.

CHAPTER 6
NUMBER REPRESENTATION IN A COMPUTER

6.1 INTRODUCTION

In the discussion in Chapter 1, computer information was related to numbers in binary form. The question of how this information is arranged for presentation to the computer may be puzzling. It is known that information in the form of arranged patterns of 0's and 1's is coded for use in the computer. The unit of information arrangement and presentation is the computer word. This word may be data which is to be operated on or instructions which detail the operations and the order of operations to be performed; no matter what it means, however, it is numerical in form.

6.2 WORD SIZE

A computer word is of definite size; i.e., it consists of an exact number of binary symbols, each of which is termed a bit (in a binary machine). Each computer has its own word size which is of fixed length and arrangement. Some computers have been designed to handle computer words of 40 bits; others may use words of 30 bits or less. The number of bits in a computer word is expressed as its length. The length or size of the computer word makes available a definite number of positions for coding information in binary form. All positions are used whether or not all the bit positions are needed to represent the information. This is necessary because the computer handles information in a pulse-no-pulse code. A no-pulse has meaning, and unused positions in a word would be handled by the computer as a no-pulse indication.

The structure of a typical computer word is illustrated in figure 2–1. The word illustrated in figure 2–1 has 32 bit positions and is divided into left and right half-words. Each half word has 16 bit positions, 1 sign-bit position, and 15 positions for information.

The next question to be considered is what determines word size. The choice of word size is not arbitrary. There is an optimum word size for any digital computer which is related to the "average" problem to be solved by the computer, the number of digits in the instruction code, and by the degree of accuracy in computation. In general, the longer the word the more precise a computation will be (more places available).

6.3 FIXED AND FLOATING POINT COMPUTERS

Since arithmetic operations involving fractions are performed by digital computers, the position of the binary point is an important consideration. Two design techniques are used in digital computers for positioning the binary point. The fixed point system locates the binary point in the same position in any register. The floating point system produces the effect of indicating the binary point by expressing all numbers as products and in two parts. The first part of the expression is the coefficient, and the second part is the exponent to which the base has been raised. In the floating point system, 0.0008076 would be expressed as 0.8076×10^{-3}. Multiplication or division of two fractions involves the appropriate operation by the coefficients and the addition or subtraction of the exponents. Thus, to multiply .0012 by .0012

$.12 \times .12 = .0144$ since $.0012 = .12 \times 10^{-2}$

$10^{-2} \times 10^{-2} = 10^{-4}$

$.12 \times 10^{-2} \times .12 \times 10^{-2} = .0144 \times 10^{-6} = 0.00000144.$

The answer would be expressed as 0.144×10^{-5}.

It is possible to program (using special techniques) a fixed point computer so that computations are performed in a floating point manner. Otherwise, the

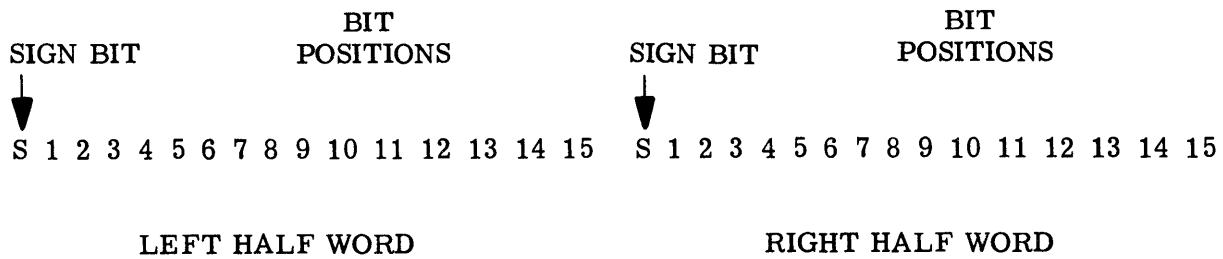


Figure 2–1. Computer Word

programmer must *scale* the problem as he programs it. (Scaling is briefly explained in Section 6.6.)

6.4 PRECISION AND ACCURACY

Precision and accuracy are terms that have distinct meanings and application in relation to numerical data. Accuracy refers to the correctness of the expression; precision is the degree of correctness. For example, the symbol π may be expressed as 3.14. This is a representation of the value of π accurate to three places, but it is not precise. Precision would require that the value of π be expressed to some greater number of decimal places, say 3.14159. Even with the value of π expressed to the fifth decimal position, the degree of precision is not absolute. The lower order digits have been rounded off. However, the 5-place precision may be the degree required for a given operation and would be suitable. Another example may be illustrated from everyday experience; a person is sometimes described as over 35 years of age. This may be accurate, but it is not precise, because that person may be precisely 35 years, 2 months, 1 day, 2 hours, 4 minutes, and 10 seconds in age. Even this detailed expression could be carried out to greater precision.

A quantity which is represented by four digits to the left of the decimal point is said to be specified to a precision of 1 part in 10,000 because 10,000 distinct numbers (0000 through 9999) can be represented by four such decimal numbers. If an accuracy to within $\pm 01\%$ is specified (in connection with a precision of 1 part in 10,000), it indicates that the unit digit of the number may be incorrect by 1 unit. The number $4047 \pm 0.01\%$ would imply a value somewhere between 4046.5 and 4047.5.

6.5 POSITIONAL AND ABSOLUTE SIGNIFICANCE

A quantity specified by a decimal number to a precision of 1 part in 10,000 can be said to be specified to four significant places. In numbers, it is important to understand the concept of positional significance. Throughout this part, the term "significance" has been used freely to designate the positional relation between individual digits of a number. In this sense, if digit x is to the left of digit y, it is more significant (has more value) than digit y. In addition to this concept of relative position significance, there is a concept of absolute significance. In terms of this second concept, a digit in a particular number is said to be either significant or not. In order to qualify as being significant in this absolute sense, a digit must contribute to the precision required. For example, in the following statement, "He is about 50 years old," the 0 is not significant. But a somewhat different case of significance arises in the multiplication of two 4-digit numbers, each of which is accurate to $\pm 0.01\%$. Here, the unit digits of both numbers may be off by as much as 0.9999 in either direction. For example, if 9001 is multiplied by 8001, the product is 72,017,001. However, the specified accuracy is such that the correct values of multiplier and multiplicand may be as small as 9000 and 8000 or may be as large as 9002 and 8002 (to the closet unit). Thus, the correct value of the product may lie anywhere in the range between 72,000,000, (i.e., 9000 x 8000) and 72,034,004, (i.e., 9002 x 8002). For this reason, the right-hand four digits of the product are not significant. On the other hand, the fifth digit from the right is significant, since it specifies the approximate center of the range of values (72,000,000 and 72,030,000) which bracket the correct product. After a calculation of this sort is completed, the result must be "rounded off;" i.e., those digits which seem to contribute something to the accuracy of the result, but actually do not, must be removed. For purpose of "round off" in the illustrative problem, the digit to the right of the least significant place does have some value; for if it is 5 or more the least significant digit should be increased by one unit. The above example should be rounded off to 72,020,000. (When binary notation is used, a 1 in the bit position to the right of the least significant place has a value which is equivalent to the value of a 5 to the right of the least significant place in decimal notation. Thus, in rounding off a binary number, a 1 is added into the least significant bit position if the position to the right contains a 1 but not if the position to the right contains a 0).

6.6 SCALING

Scaling the variables (the numerical values) which are to be operated upon in a digital computer solution is closely related to the concept of precision. Assume that a computer has a decimal modulus (4.2.3.2) of 10,000. Then, if a variable can be scaled so that its range falls exactly within the capacity of the machine, it can be represented to a precision of 1 part in 10,000. If, on the other hand, it is scaled so that its maximum value is represented (handled) by just three orders of the machine, it is only being represented to a precision of 1 part in 1000. In other words, there are two objectives to keep in mind when scaling variables for a computer solution. One is to scale the variables so that they do not exceed the capacity of the machine (if this happens, meangingless results will be obtained); the other is to scale the variables so as to use as much of the capacity of the machine as possible in order to obtain the maximum precision. The extent to which the full capacity of the machine can be used to represent a particular variable depends upon the exactitude with which the range of the variable is known.

Suppose that wind velocity is to be scaled for

representation in a computer and it is known that no velocity as large as 40 knots will be encountered in a particular problem. Then, 40 knots can be made equal to the largest number which the machine can handle or represent. On the other hand, if the machine handles only fractions (or decimals), 40 knots can be equaled to one machine unit, and a velocity of 20 knots will appear in the machine as 0.5.

PART 3
COMPUTER CIRCUITS AND DEVICES

CHAPTER 1
INTRODUCTION

The building blocks of a digital computer are its individual circuits, hundreds, often thousands of them, interconnected to accomplish the operations of transferring and processing data. Actually, however, there are not this many different circuits but a few basic types that are used again and again in different combinations. Before discussing in detail these basic computer circuits and other devices, certain fundamentals must be examined here, such as the types of electrical signals commonly used and the nature of the simple logic operations performed by the circuits.

1.1 INFORMATION SIGNALS

The transfer and processing of information in a digital computer is done by switching and storing information signals; that is, electrical signals representing numbers. Many or most of the common electronic parts (relays, vacuum tubes, crystal diodes, etc.) perform excellently in bistable (2-state or on-off) operation. Because of this, it is usually easiest to make computers work internally in the binary number system, and, in this case, the information signals must represent the binary digits, 1 and 0. (Some computers are built to work in decimal, using 2-state signals in a code to represent the 10 decimal digits, but this type will not be considered here.)

There are several possible ways of representing the binary 1's and 0's electrically. For example, two signal lines might be used, a voltage on one representing a 1, a voltage on the other indicating 0. This, however useful, would require more circuitry than is needed if used throughout the computer. When it is necessary to transfer numbers over a single signal line between circuits, as shown at (a) of figure 3–1, the easiest method is to place a d-c voltage on the line to represent a binary 1 but no voltage to represent a 0. The polarity of the voltage might be either positive or negative; the important point is that the presence of the voltage represents 1, the absence represents 0. (Circuits A and B are shown grounded to indicate the return circuit, or common; although the return is always necessary, it is usually taken for granted and omitted from block diagrams of computer circuitry.)

1.1.1 Voltage Level Representation

An alternative to this voltage-or-no-voltage method, perhaps somewhat better suited to vacuum-tube circuitry, uses a steady-state, positive d-c voltage

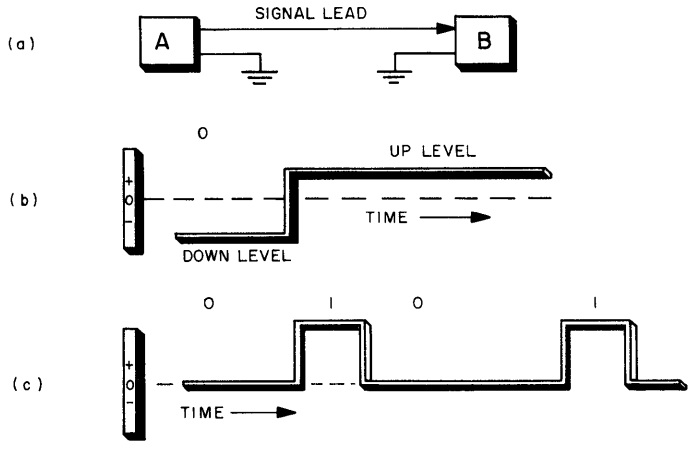


Figure 3–1. Common Number Signals

for a 1 and a negative voltage for a 0. Any reasonable voltage amplitudes may be used for the two d-c levels to meet circuit requirements. Actually, the negative voltage (often called the *down level*) may be negative only in relation to the more positive, or *up level,* voltage. This method of representing numbers with d-c levels is shown at (b) of figure 3–1. Circuit A indicates a 0 to circuit B by holding the signal line at the down level, a negative potential. When it is desired to signal a 1, the level is brought up to a positive potential and held there as long as necessary. As long as the level is up, a 1 is present at the input to circuit B; when it is down, the input is 0. It is possible to reverse this method of representation and call the down level a 1 and the up level a 0. In practice, this matter of polarity and number representation can become a headache to the designer, especially in vacuum-tube circuits where signal inversion takes place between grid and plate in each tube stage. To simplify circuitry, the method of representation chosen for the computer may be reversed in some groups of circuits. There is no objection to this, as long as the operation remains logical and consistent. The Central Computer of the AN/FSQ-7 and -8 uses this voltage level method of representing numbers (and another method, described below). The standard levels in this machine are —30 volts for the down level (or binary 0), and +10 volts for the up level (or binary 1).

1.1.2 Pulse Representation

One characteristic of the voltage level information signal is that it can be held up or down as long as necessary. In many cases, however, all that is needed is a 1-signal of very brief duration to trigger the following circuit, so a pulse can be used to represent a 1, as in (c) of figure 3–1. (This is a perfect, square pulse; in practical circuitry, it would tend to be round-shouldered.) If a pulse (positive or negative) represents a 1, it follow that the absence of a pulse logically represents a 0.

Using the method of pulse signals, the output line from circuit A remains at some reference level (usually ground potential) until a 1 must be transmitted to circuit B, whereupon a single pulse is generated by A and is placed on the line. The pulse appears at the input of circuit B, signalling at 1, and quickly dies out, but it must have sufficient amplitude and duration to produce the desired triggering of circuit B. When the quickly passing (transient) pulse has disappeared after triggering circuit B, the signal line returns to the zero reference level, again indicating 0 at the circuit B input. Not until another 1 must be indicated is another pulse sent. The AN/FSQ-7 and -8 computer uses pulse signals, as well as levels, to represent numbers. The pulse representing 1's are positive, 20 to 40 volts in amplitude and 0.1 microsecond (usec) in duration.

1.1.3 Transmission Methods

The levels and pulse signals are the two basic types (but not the only possible ones) used to represent numbers and, therefore, information, in digital computers. But a single level or pulse represents only one binary bit, yet the computer must work with long binary numbers (many bits), or *words.* How are these computer words transmitted from one part of the machine to another?

1.1.3.1 Parallel

If one signal line between two circuits can, at a given moment, transmit a 1 or a 0, it is reasonable to conclude that a complete, 5-bit word, for example, requires five lines in parallel (plus one common signal return). This is called *parallel* transmission. A parallel circuit capable of handling a sample 5-bit word, using relays with normally open contacts as combined switching and storage devices, appears at (a) of figure 3–2. Assuming that the five relays can be operated and held operated in any desired combination, any 5-bit binary number from 00000 through 11111 (decimal 31) can be transmitted to the five circuits labelled 2^4 through 2^0.

With all five relays unoperated, all contacts are open, and none of the lines are connected to the battery. Each signal line, therefore, is at the down level, and the input to each circuit is 0. Reading across all five inputs at (a), the result is 00000. If relay 0 alone is operated, only the input to circuit 2^0 is up (connected to battery). Now, reading across the five inputs at (b) of figure 3–2, the lines are:

2^4	2^3	2^2	2^1	2^0
down	down	down	down	up
representing				
0	0	0	0	1

or decimal 1.

As another example, simultaneously operating relays 4, 1, and 0 brings up the three corresponding lines, as shown at (c):

2^4	2^3	2^2	2^1	2^0
up	down	down	up	up
representing				
1	0	0	1	1

or decimal 19.

Changing the level on any signal line, in this case by operating or dropping the corresponding relay, changes the transmitted computer (binary) word. And, although it has been assumed that the operated relays are held operated to provide levels, pulse-type signals

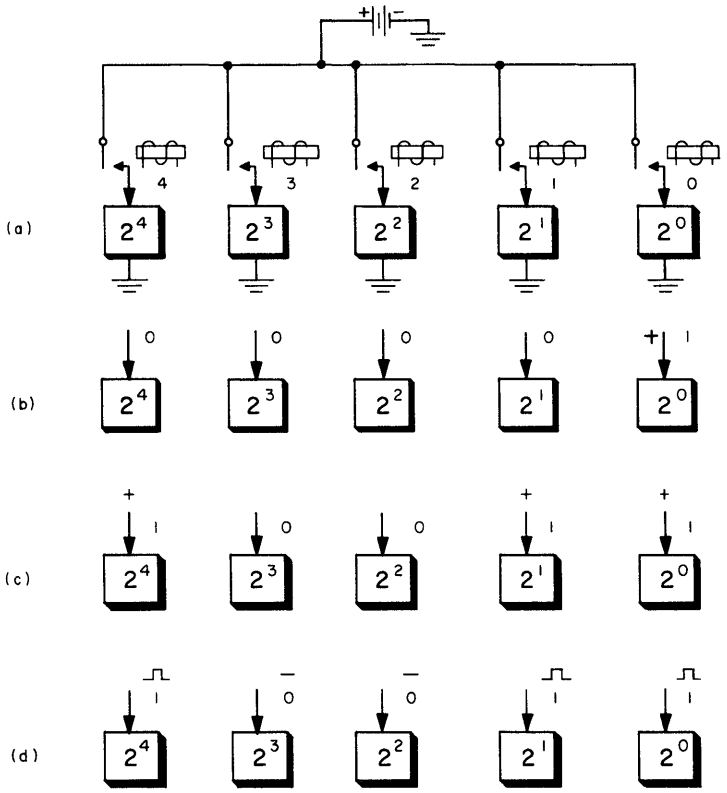


Figure 3–2. Parallel Transmission of Numbers

could be sent just as easily by simultaneously pulsing the desired combination of relays. To use the previous example, if the coils of relays 4, 1, and 0 are pulsed at the same time, their contacts close briefly and then open again. As shown at (d) of figure 3–2, this places a single pulse on each of the signal lines to circuits 2^4, 2^1, and 2^0, but there are no pulses on the inputs to circuits 2^3 and 2^2. So the number transmitted is again 10011, but this time it is sent by means of pulse signals rather than by voltage levels. Finally, remember that relays are shown in the circuit of figure 3–2 only as examples of devices providing the switching or storage functions. Each relay can be replaced by any type of circuit or device capable of performing the necessary (bistable) function and of delivering the proper information signals.

1.1.3.2 Serial

There is one other basic way of transmitting number signals, in addition to the parallel (or side-by-side) method just described. This second method also uses either levels or pulses, but sends the bits of the number, one after another, down a single line. Thus, the bits are sent in sequence, or serially, so this is called *serial* transmission. In the serial method, it is usual to send the least significant bit of the number first, followed by the other bits in order of increasing significance. This makes sense when it is remembered that addition, subtraction, etc., are performed bit by bit in this same order. The binary number previously used as an example, 10011, is shown in figure 3–3 as it would be sent in serial form, with pulse-type signals. The number is sent from circuit A to circuit B as a train of pulses and no-pulses. The first bit transmitted and, therefore, the first bit received by circuit B, is the least significant, or 2^0, bit, and the more significant bits follow in order. There must be spacing between successive pulses, otherwise they could not be distinguished by the receiving circuit.

1.1.3.3 Comparison of Methods

The principal differences between the parallel and serial methods of transmission show up in a comparison of figures 3–2 and 3–3. In the parallel method of figure 3–2, five sending and five receiving circuits are involved to handle a 5–bit number. In the serial method (fig. 3–3), only two circuits are needed: one sending, the other receiving. On the other hand, all

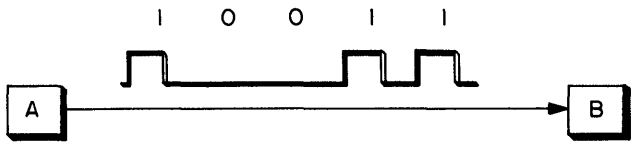


Figure 3–3. Serial Transmission of Numbers

bits are sent simultaneously in the parallel method, so the entire number is transmitted in the time it takes to send only one bit. In the serial method, the entire number is not known until all five bits have been sent, one after the other; so it takes five times as long to send the same complete number.

Thus, it becomes apparent that, in general, the parallel method offers much faster transmission of numbers than the serial, but requires more circuitry. In the AN/FSQ-7, -8 computer, in which each word is 32 bits long, use of the parallel method makes number transmission 32 times faster than if the serial method were employed. This is achieved at the cost of more equipment, justified, of course, by the importance of operating speed in this air defense computer. And while speed of number transmission is only one factor in determining the final operating speed, it is an important one.

1.1.4 Timing

Another matter of importance in transmitting numbers is timing.

1.1.4.1 Parallel Transmission

Consideration of the parallel method shown in figure 3–2 indicates that simultaneous operation of the relays signalling a given number is a must if circuits 2^0 through 2^4 are going to operate upon the number as soon as it is received. If the number signals were voltage levels, for example, and relays 4 and 0 were operated together, followed a moment later by relay 1, the receiving circuits would first get the number 10001, which would then change to 10011. With pulse-type signals of short duration, the same operation of the relays would send 10001 and then 00010. Either of these occurrences, resulting in incorrect numbers getting in, could cause errors in an arithmetic machine. It is important, therefore, in many or most uses of parallel transmission, to time all the bits of a number to arrive simultaneously at their destinations.

1.1.4.2 Serial Transmission

Timing is equally vital in serial transmission, as examination of figure 3–3 will indicate. Since the bit signals are sent down a single line, one after another, some rigid timing system is a necessity, especially with pulse-type signals. If the signals were sent at varying intervals, for example, the receiving circuit would have no way of telling whether a long space between pulses was a 0 or merely spacing. And if voltage level signals were used, the receiving circuit could not distinguish between a 1 and two consecutive 1's or between a 0 and two consecutive 0's. So the timing of serial transmission must be controlled, also. This is done by establishing the period of time necessary to send one bit, which is the smallest piece of information handled in the computer. If a pulse system is used, each pulse must last long enough to trigger the receiving circuit, and the space between pulses must allow sufficient recovery time to prepare the circuit for the receipt of another pulse. With serial voltage levels, each level must be held long enough to include both triggering and recovery times.

Once the time period for transmission of a single bit (called one *bit-time*) has been determined, the problem of serial timing is handled by rigidly controlling the length and spacing of the bit signals in every number (computer word) transmission. Figure 3–4 shows the timing of both the pulse-type and level-type signals making up the number 10011. The bit-times, measured from the start of the first signal in the number, are shown running from right to left across the top of the figure (the least significant bit, which appears first, is at the right). Notice that each bit-pulse in the pulse system appears at the start of the corresponding bit-time, lasts for a fixed period, and

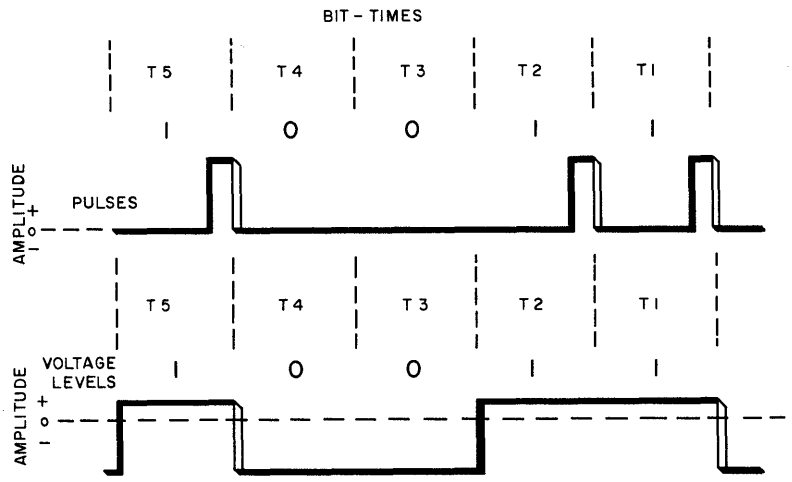


Figure 3–4. Timing of Serial Numbers

disappears, to be followed by a space four times the length of the pulse. When the bit to be represented is a 0, no pulse is sent during that bit-time, of course. The relative durations of pulses and spaces vary from one computer to the next. In some, for example, the space is the same width (in time duration) as the pulse. In the voltage level system, the line potential rises to the up level when a 1 is being transmitted and remains there during the entire bit-time. If the next bit is also a 1 (as it is in fig. 3–4), the level does not drop, even momentarily, but remains up for the next bit-time, also. Only when successive bits are different — changing from 1 to 0 or 0 to 1 — does the level change.

In many computers, the basic source of timing or synchronizing signals is the *clock,* usually a pulse generator which, controlled by an accurate oscillator, puts out a continuous string of rigidly timed pulses. The clock pulses are generated one per bit-time and are sent to all parts of the computer to control the transmission of numbers and the timing of operations. Thus, in a computer using a 1-usec bit-time, the clock must generate one million pulses per second, at exact 1-usec intervals, from the time the computer is turned on until it is shut down.

1.1.5 No-Signal Condition

One further point must be mentioned. Whenever quickly passing (short duration) signals are used (either pulse-type signals in serial or parallel number transmission or voltage levels in serial transmission) all signal lines are normally held at the 0 level when no numbers are being sent. This means, in effect, that 0's are kept in all circuits where no other numbers are being processed. (It is entirely possible to use a system of number signals in which this would not be true, such as positive pulses for 1's and negative pulses for 0's, with the line returning to ground potential between pulses, for example. However, such a system is complex and requires much equipment.)

On the other hand, voltage level signals used in parallel circuitry are not usually short-duration signals. In many types of parallel circuitry, the 2-line transmission method described in 1.1 is used, a signal (usually an up level) on one line representing 0, a signal on the other representing 1. As in all cases, 0's are kept in all circuits that are temporarily idle, but in this case the 0's are represented by up level voltages on the 0 lines. The levels, whether indicating 0's or 1's, are held on the lines as long as required. When certain numbers are no longer needed, they are removed from the circuitry holding them by means of control signals sent to return all circuits to the 0 state.

1.2 SWITCHING LOGIC

The preceding text has shown the basic types of signals used to represent information in digital computers and the basic methods of moving the information from one part of a computer to another. It has been mentioned that all the arithmetic and other operations performed in a digital computer are done by switching and storing information (in the form of numbers) in the proper combinations and sequences.

The operations carried out by the digital computer are operations of logic. Arithmetic — all mathematics, in fact — is rigidly based on logic; in other words, arithmetic is a systematic process of manipulating numbers involving simple operations carried out according to precise rules. If numbers are to be represented by voltage levels and pulses, as stated above, some system of manipulating these voltages according to the logical rules of arithmetic must be found. Circuits which accomplish this function in a computer are called logic circuits.

1.2.1 Logic Operations

How does switching enter into operations of logic? This can best be understood by looking first at the type of logic operation that can easily be performed by a switching circuit. The actual circuitry will be considered later.

1.2.1.1 OR Logic

One of the common logic operations is the alternative or choice, called the OR function. This comes into play whenever any one of two or more alternate possibilities can bring about a specified result. For example, "We'll go to the movies if George, Pete, or Joe shows up." In this case, the arrival of George OR Pete OR Joe leads to the result, movies. This can be written in shorthand form:

George OR Peter OR Joe = Movies

The situation can also be symbolized in diagram form, as shown in figure 3–5. The label in the block indicates

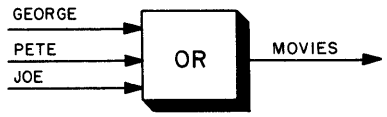


Figure 3–5. OR Situation Symbolized

that OR is the relationship between its "inputs," which are, of course, the arrival of George, Peter, or Joe. Another way of thinking of it — more accurate when dealing with equipment — is that the block applies the OR function to its inputs. The block produces an "output" — movies — only when the inputs meet the OR requirements: in other words, when at least one of the inputs appears. This diagram can be altered, as in figure 3–6, to illustrate the general case, any OR situation. Three inputs are shown, although any number

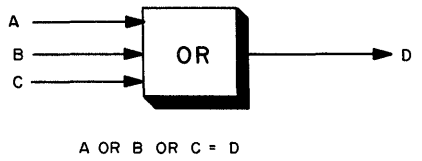


Figure 3–6. OR Function

except one is possible (one condition offers no alternative, hence no OR). The OR function produces a specified result, D, when any one of its input conditions, A OR B OR C, is satisfied. Notice that if any two, or even all three, of the inputs appear together, the output is still produced because no restriction is stated. The OR in this case includes all combinations, as well as one-at-a-time inputs, so it is called an *inclusive* OR. In digital computer logic circuits, the OR function is always inclusive unless otherwise specified.

The opposite OR situation is called the *exclusive* OR. It requires that any one, but no more than one, of two or more inputs produce a specified output. "We'll play golf if Harry or Jim comes, but not if they both arrive." Harry OR Jim result in golf, but it is definitely stated that Harry AND Jim does not.

1.2.1.2 AND Logic

That last statement indicates that the AND (combination) must be another logic function, which it is. The AND function requires that all of two or more possible conditions (inputs) be present at the same time to bring about a specified result (output). For instance, "You need inductance and capacitance and resistance to build a bandpass filter." All three are required – and all at the same time – to produce the result, a filter. If any one is missing, or if the three are present only at different times, the specified result is not produced. The logic can be written:

L AND C AND R = Filter

Figure 3–7 illustrates the AND function in diagram form. Again, any number of inputs except one is possible. The AND function produces a specified result, D, when all its input conditions, A AND B AND C, are fulfilled at the same time.

1.2.1.3 NOT Logic

Another logic operation of importance is the NOT function, called *inversion.* Inversion means a turning

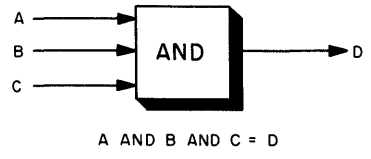


Figure 3–7. AND Function

upside down or a reversing of relationships. In working with 2-valued logic, this means changing every quantity to its opposite. Every "yes," when inverted, becomes a "NOT yes," which is the same as a "no." Similarly, a "no," inverted, becomes a "yes." Figure 3–8 shows

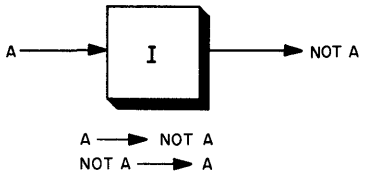


Figure 3–8. NOT Function (Inversion)

the symbol for the NOT function. The letter I, in the block, stands for "inverter," the commonly used name for the NOT block. The inverter can have only one input; if more than one quantity is to be inverted, a separate inverter is required for each. When input A is applied to the inverter, as shown in the figure, the output is NOT A. It is also possible to feed NOT A to the input and obtain A as the output.

The NOT function most often becomes necessary in conjunction with the OR or the AND. For example, someone might say "I'll go if Tom does, but not if it rains." Examination shows that this involves an AND function and a NOT.

Tom Goes AND (NOT Rain) = I Go

This can be diagrammed with an AND block and an inverter, as shown in figure 3–9; the combined func-

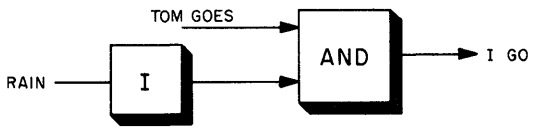
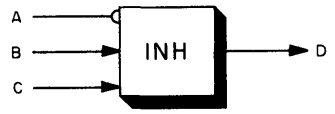


Figure 3–9. AND NOT Diagrammed

tions are often called the AND NOT. (An OR NOT arrangement can be put together in similar fashion from an OR and an inverter.) Notice that if the inverter input (rain, in this case) is present, it prevents the AND from producing an output. (The presence of an inverter input means no inverter output; hence, a missing input to the AND. The AND cannot operate unless all its inputs are present simultaneously.) This prevention of the AND operation is called *inhibiting* which, used this way, means about the same thing as prohibiting.

The inhibit function by itself is drawn as shown in figure 3–10. The semi-circular "button" indicates the inhibit input; the other inputs are usually arranged



AND INPUTS: B AND C AND (NOT A) = D
OR INPUTS: B OR C AND (NOT A) = D

Figure 3–10. Inhibit Function

to fulfill the AND function, although they can be made to operate as OR inputs, if desired. Operating in AND fashion, the inhibitor produces an output, D, when inputs B AND C AND (NOT A) are present. The presence of input A, however, inhibits (stops) the output, even if all other conditions are satisfied. There may be any number of AND or OR inputs (one or more), and the same is true of the inhibit inputs.

1.2.2 Circuit Logic

Now that the basic logic functions used in digital computer circuitry have been examined in terms of information only, it is time to see how physical circuits operate according to the rules of these functions. The inputs are now going to be electrical signals representing the facts or events that must be logically connected. The logic blocks previously used to diagram the functions are henceforth actual physical circuits. And, finally, each output is an electrical signal representing the specific result of applying the rules of a particular logic function to a particular set of inputs. To put it another way, each output is a logical conclusion.

To see how a switching circuit performs a logic function, consider the case of a home owner who wants to be warned when someone comes to either his front or back door. This involves the OR function, and the logic of this situation can be diagrammed as shown in figure 3–11, using the simple OR block. The ordinary

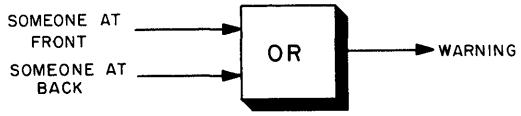


Figure 3–11. Logic of Doorbell Situation

manner of "solving" this, of course, is to install a doorbell circuit, with a pushbutton switch at the front door and another at the back and a bell inside the house. A battery can be used to power the bell, as shown in figure 3–12.

The pushbuttons are not parts of the logic circuit, but are simply devices to translate physical facts or events into electrical signals. They put the information into the circuit. The fact, "somebody at the front door,"

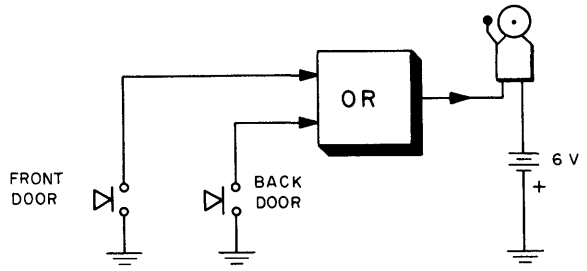


Figure 3–12. Doorbell Circuit, Showing Logic

is translated to a voltage of 6 volts when that "somebody" presses the front-door button. The voltage, which can also be considered as a binary 1, is applied to one input of the OR circuit. According to the OR function rule, an output is produced when one input OR the other is present. So a binary 1 at the front-door input results in an OR circuit output that rings the doorbell. Following this reasoning, a binary 1 at either OR circuit input represents "somebody present," so a binary 0 (no voltage, or 0 volts) must represent "somebody not present."

1 = Somebody present

0 = Somebody not present

Thus, electrical signals can be made to represent the binary numbers which, in turn, are made to represent specific items of information. The 6 volts can be thought of as the up level voltage, in which case 0 volts is the down level voltage.

Now, the OR circuit itself, inside the block in figure 3–12, must be constructed to operate in accordance with the rules (logic) of the OR function; in other words, it must be built to produce an output when a binary 1 (up level voltage) appears at one input OR the other. What must the output be? Well, the bell must ring when somebody is present (at either door); binary 1 represents "somebody present," so the output must be a binary 1 or an up level of 6 volts. The current that flows as a result of applying this up level is capable of ringing the bell, so the choice of output is logically and electrically satisfactory. The bell can be considered as a device to transfer the information, "somebody present," to the homeowner.

When there is nobody present at either the front or back door, a binary 0 (down level) is present at each OR circuit input. In this case, the output must also be a binary 0, or down level, representing "somebody not present." The down level cannot cause the bell to ring. The conditions of this situation are so simple it is apparent that the OR circuit itself need be nothing more than a parallel connection of wires from

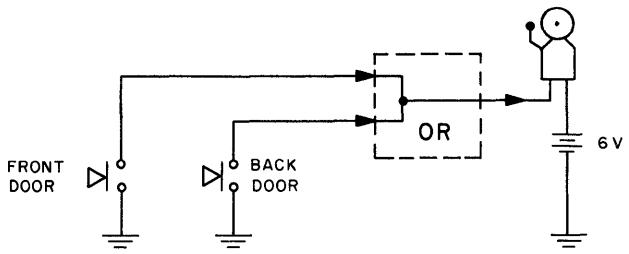


Figure 3—13. Complete Doorbell OR Circuit

the pushbutton switches, as shown in figure 3—13. Notice that, however simple it may be, this is the only part of the circuit that fulfills, by itself, the requirements of the OR function. It is the parallel method of connection that offers alternate input possibilities, making this an OR circuit. It is important to understand this distinction, although in practice it is common to speak of the entire parallel circuit, including the switches, as the OR circuit.

This doorbell OR circuit provides a simple illustration of the manner in which a logic operation is carried out by an electrical or electronic circuit. Regardless of the type of logic circuit, 2-valued information is represented by binary numbers 0 and 1, which, in turn, are represented by electrical signals. Means or devices are provided to get these signals into the circuit at the proper place and time. By building the circuit to operate upon the electrical signals according to the rules of the desired logic operation, the resulting output signals represent logical decisions or conclusions reached in accordance with the built-in rules. To be useful, these outputs are transmitted or transferred either to some other circuit or out of the computer.

So the computer logic circuits cannot "think" and do not know what information their inputs or outputs represent. There is nothing miraculous about them. They simply accept electrical input signals and operate with them in accordance with the circuit design, just as ordinary radio or TV circuits must do. All the thinking is done by the designers who build the rules of logic into the circuits and the programmers who direct the operation of the computer. The advantage of the complete computer is that it can perform a long sequence of these simple logic operations, at extremely high speed, by sending signals through a chain of logic circuits. By performing the proper sequence of operations, the computer does arithmetic.

CHAPTER 2

SWITCHING AND SMALL-SCALE STORAGE CIRCUITS

2.1 COMPUTER LOGIC CIRCUITS

What switching devices are commonly used in logic circuits? The answer, today, is relays, semiconductor (crystal) diodes, vacuum tubes, transistors, and magnetic cores. Tomorrow's answer may be different, for research is constantly seeking smaller, faster, more efficient, and more reliable switching devices. Today's logic circuits are shown, briefly, below. The actual circuit coverage is brief because, except in relay computers (those composed principally of relays), a good working knowledge of a digital computer rarely requires knowing exactly what is inside a logic circuit.

All the AND circuits in a given computer, for example, are usually identical and individually packaged. The machine is ordinarily serviced by locating and pulling out a defective package and replacing it with a spare, so the technician works down to the logic circuit level but not inside the circuit. His schematics are "logic block diagrams," showing each circuit as a block labelled for the type of logic operation it performs, like the blocks in figures 3–5 through 3–12. (Figure 3–9 is a miniature logic block diagram, showing logic circuits interconnected.) The technician must know the types of electrical signals used and the inputs, outputs, and "rules" of each type of logic circuit. In other words, to understand and troubleshoot the computer, it is necessary to know the logic but not the individual circuits. For a complete understanding of the computer, it is necessary to know how logic circuits are made up, using the various switching devices mentioned earlier.

2.1.1 Relay Logic Circuits

The doorbell OR circuit shown in figure 3–13 is hardly the sort of thing one would expect to find in a digital computer. Yet, consider how closely it resembles the OR circuit from a relay type of computer, shown in figure 3–14. The relay OR circuit must, of course, produce an output of 1 when a 1 appears at any of its inputs. The manner in which such a circuit can be put to practical use in making a built-in "logical decision" can be seen from this example.

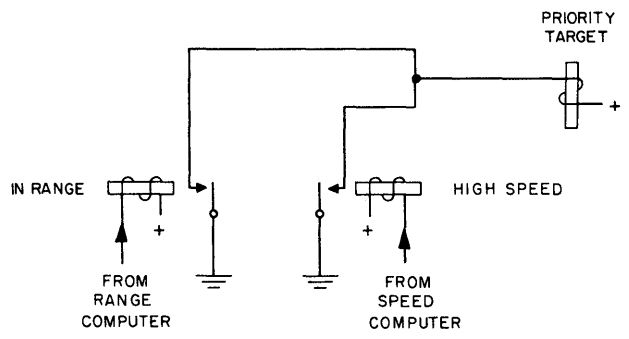


Figure 3–14. Relay OR Circuit

Figure 3–14 shows a priority circuit in an air defense computer. If an enemy aircraft is within fairly close range, or is traveling at very high speed, it must be dealt with before an aircraft that is further away or approaching more slowly. To remember that this aircraft, or target, has special priority, the relay computer uses the OR circuit, shown in figure 3–14, to operate the PRIORITY TARGET relay.

Binary 1 signals from small analog range and speed computers operate either the IN RANGE or the HIGH SPEED relays (or both) if the aircraft being tracked by radar is within a certain range or approaching faster than a certain speed. Operating either relay closes its normally-open contacts and applies an up level voltage, a binary 1, through the parallel connection to the relay coil. Thus, the PRIORITY TARGET relay is operated when the target is in range OR flying at high speed. The contacts of the PRIORITY TARGET relay are located in other circuits, so this relay is not only a memory but also a device to transfer the binary 1, produced by this OR circuit, into other circuits. An OR circuit like this can be given any reasonable number of inputs simply by placing more sets of normally-open contacts in parallel.

This relay computer OR circuit, then, is almost exactly similar to the simple doorbell OR circuit. Computers composed principally of relays are not often built today, due to the comparatively slow operate and release times of the relays, but the relay logic circuits are nevertheless still important. The automatic dial telephone exchanges, for example, the world's largest digital data-processing machines, or computers, use thousands of relays. And some relay circuitry is often used in electronic computers, especially in the input and output elements.

A relay circuit to handle the AND operation must produce an output of 1 only when 1's are on all its inputs. As an example, suppose the bridge of a battleship must be alerted whenever an aircraft identified as hos-

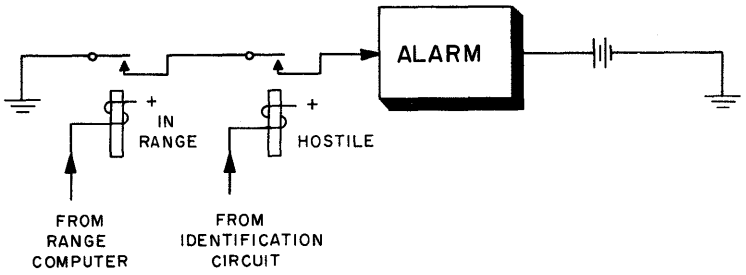


Figure 3—15. Relay AND Circuit

tile flies within 20,000 yards of the ship. The relay computer AND circuit shown in figure 3—15 automatically sounds the alert. The coil of the HOSTILE relay is connected to an identification circuit that operates this relay whenever a hostile aircraft is detected. Operating this relay, in effect, signals that there is a binary 1 at this input. The hostile aircraft is automatically tracked and a range computer operates the IN RANGE relay if the aircraft comes within 20,000 yards of the ship. This signals a binary 1 on the second input. The normally-open contacts of the two relays are connected in series in the alarm circuit; thus, when a hostile aircraft is detected AND its range is less than 20,000 yards, both sets of contacts are closed and a d-c level is applied to the alarm device to sound the alert. Putting it in general terms, when binary 1's are on both inputs, the circuit produces an output of 1. The logic is:

In Range AND Hostile = Alarm

The circuit is not limited to two inputs, of course. More sets of normally-open contacts can be added in the series path.

A very important requirement is that, regardless of the number of inputs, all must be signaling 1's *at the same time* to produce an output 1. In the AND circuit of figure 3—15, if only one set of contacts closed, or if one set closed and then opened again before the second set closed, the alarm device could not operate. In this example, it is likely that an aircraft would be identified as hostile before it came within the specified range. Therefore, the HOSTILE relay would be operated first. The AND circuit cannot produce an output, however, unless both input conditions are satisfied simultaneously, so no alert could be sounded (no output produced) until the IN RANGE relay also operated.

Inversion — the NOT operation — is easily accomplished with relays by using a normally-closed set of contacts. The above discussions have indicated that a binary 1 is inserted into a circuit by closing a pair of normally-open contacts. This is done by operating the relay on which the contacts are mounted. In other words, a binary 1 operates a relay, and the 1 is transferred into another circuit by the closing of the relay's normally-open contacts in that circuit. Now, consider the relay that has normally-closed contacts. When this relay is not operated, its contacts are placing a binary 1 in some other circuit, so the relay is receiving binary 0 (at its coil) and its contacts are indicating NOT 0, or 1. When a binary 1 operates the relay, its contacts open, indicating 0. Thus, using normally-closed contacts accomplishes the NOT operation. Either OR NOT or AND NOT circuits can be built. The AND NOT operation can be illustrated by making a change in the example of figure 3—15. The HOSTILE relay is simply replaced by a FRIENDLY relay with a set of normally-closed contacts connected in series with the normally-open contacts of the IN RANGE relay, as shown in figure 3—16.

It must be assumed that the identification circuit operates the FRIENDLY relay only if an aircraft is

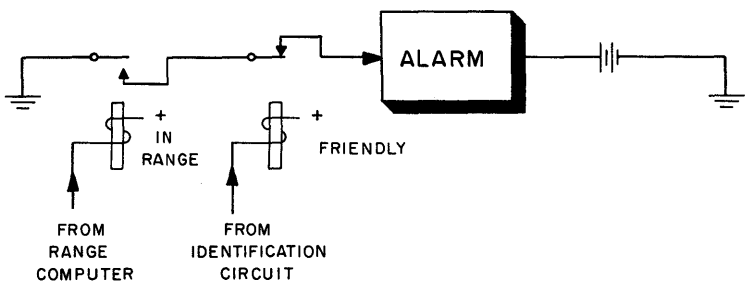


Figure 3—16. Relay AND NOT Circuit

identified as friendly. When the FRIENDLY relay is operated, its contacts open. If the aircraft comes within 20,000 yards, the contacts of the IN RANGE relay close, but no alarm is given because the alarm circuit is broken by the open contacts of the FRIENDLY relay. However, if the incoming aircraft is hostile, the FRIENDLY relay is not operated, and its contacts remain closed. When the RANGE relay now operates, the alarm circuit is completed, and the alert is given. Because of the series connection, the network is still an AND circuit, but the normally-closed contacts introduce inversion. The logic of the circuit is now:

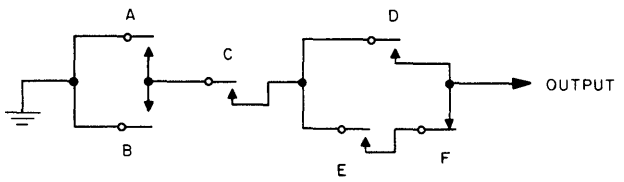
In Range AND (NOT Friendly) = Alarm

An OR NOT circuit is constructed like an OR, but using a normally-closed set of contacts in one (or more) of the parallel paths.

Many different combinations of logic operations are possible in relay contact networks, of course, using various arrangements of series-parallel paths. A relatively simple example is shown in figure 3–17, with the logic written below the circuit. The relay coils are not shown, which is the normal practice in relay work. In the logic of figure 3–17, note that there are two parallel OR connections and three parts to the principal series AND connection. For experience in recognizing and working with logic, the reader might try drawing the logic block diagram of this circuit, using logic blocks like those of figure 3–5 through 3–12. Use as few blocks as possible to do the job correctly, just as a designer would attempt to reduce the number of logic circuits in a computer.

2.1.2 Diode Logic Circuits

Many computers use what is called *diode logic*, performing most or all of the logic operations in circuits made up of semiconductor diodes, and using vacuum tube or transistor circuits primarily for building up attenuated pulses or levels, where necessary. The basic diode OR circuit appears in figure 3–18. The crystal diode, of course, is like the vacuum tube diode. It has an anode and a cathode, identified as shown, and offers



(A OR B) AND C AND [D OR (E AND NOT F)] = OUTPUT

Figure 3–17. Sample Relay Logic Combination

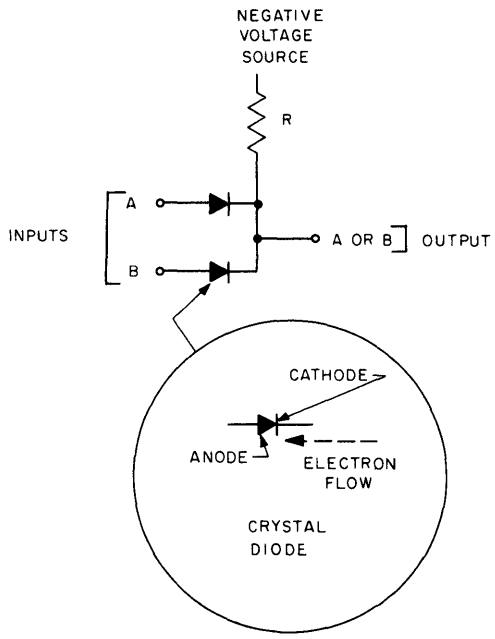


Figure 3–18. Diode OR Circuit

practically no *forward resistance* to the flow of electrons from cathode to anode; in other words, it conducts easily when the anode is made more positive than the cathode. However, when the cathode is more positive than the anode, the diode offers a very high *back resistance*, and practically no current can flow.

The input lines to the diode OR are connected in parallel, each through a separate diode, to the output. (Only two inputs are shown, but it is possible to add more.) As in the doorbell and relay circuits, it is the parallel connection itself that makes it an OR circuit. The diodes are required to isolate the inputs from each other to prevent interaction between the circuits supplying the input signals. The junction is tied through resistor R to a source of voltage more negative than the level used to represent binary 0. Therefore, when 0's are on both input lines, both diodes conduct because the anodes are more positive than the cathodes. Since the diodes offer practically no resistance to current flow in this direction, nearly all the voltage drop in the circuit is across the relatively large resistance of R. Thus, the output, tied to the more positive end of R, is at approximately the same voltage level as the input lines, indicating binary 0. When a positive-going voltage level, representing a binary 1, appears on either input line, there is a greater difference of potential between the negative source and that input. Again, nearly all the increased voltage drop appears across R, so the voltage at the output end of R rises to approximately the binary 1 level. The same effect occurs if 1's appear simultaneously on both inputs.

In this manner, the circuit carries out the OR operation by putting out a 1 when there is a 1 on either input (or both). The circuit shown in figure 3—18 is basic and could be made to work with either pulse or voltage level signals but, because of the d-c bias, operates somewhat better with levels. For pulse signals, load resistor R is often connected to ground instead of a negative voltage source, and a peaking inductance may be placed in series with the resistor. The circuit operation is the same as that described above, except that the diodes do not conduct when 0's are on the input lines and the output line is then at ground potential.

Note that this diode OR circuit of figure 3—18 can be substituted for the relay OR circuit, shown in figure 3—14, by connecting the range computer to one input and the speed computer to the other. The PRIORITY TARGET relay would most likely be replaced by some other type of memory and information-transfer device because of loading and other electrical problems. The important point is that the problem of registering a priority target can be solved just as well by the diode OR as by the relay OR circuit.

The diode AND circuit, shown in figure 3—19, also depends for its operation upon the voltage drop across load resistor R, which, in this case, is connected to a source of positive bias voltage. The diode connections are the reverse of those in the OR circuit, and positive-going binary 1's are applied to the cathodes. (More AND inputs can be added, although only two are shown in this figure.)

When relatively negative binary 0's are on both inputs, both diodes conduct because the anodes are more positive than the cathodes. The forward resistance of the diodes under these circumstances is only a few ohms, so practically all the voltage drop is across the load resistor, placing the output line at approximately the binary 0 level. As long as there is still a 0 at one input, the diode in that line continues conducting, and the voltage drop across resistor R keeps the output line at the 0 level. Therefore, the anode of the other diode is held at the 0 level, and a more positive 1 appearing at its cathode cuts it off but does not affect the output level.

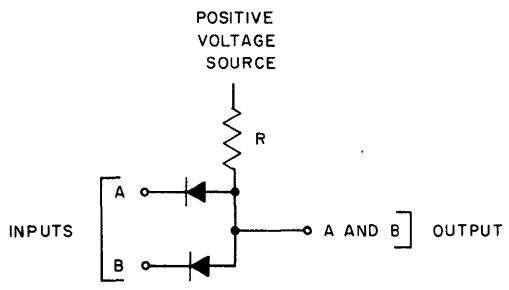


Figure 3—19. Diode AND Circuit

When 1's appear simultaneously at both input lines, the total voltage drop across the circuit is decreased by an amount equal to the amplitude of the 1 signal. Both diodes conduct, and nearly all the decreased voltage drop is across resistor R, so the voltage at the output end of R rises to approximately the level of the input lines, or binary 1. (When pulse signals are used, the output is a pulse; applying the simultaneous input pulses and getting the pulse output is sometimes called *firing* the AND.)

As soon as the signal at any input drops back to the binary 0 level, the total voltage drop across the circuit to that input increases. Since this drop is chiefly across R, the output line returns to the 0 level, and the diodes in the other lines are again cut off. Thus, the diode AND produces an output of 1 only when all its inputs are 1's. This diode AND, therefore, can easily substitute for the relay AND circuit of figure 3—15.

When the AND circuit (diode or any other type) is used with pulse signals, it immediately becomes apparent that timing is vitally important, since all the 1 inputs to the AND must appear at the same instant. It is possible that these pulses may come from different parts of the computer, some even from outside, and, therefore, they may become available at slightly different times. For this reason, it is often necessary to introduce devices that delay some of the pulse signals by differing amounts of time. For example, if the signal coming to input A of the AND circuit in figure 3—19 is available at bit-time T1, while the 1 signal for input B appears at T4, the A signal must be delayed three bit-times to make the two appear simultaneously at the AND circuit. As shown in figure 3—20, a three bit-time

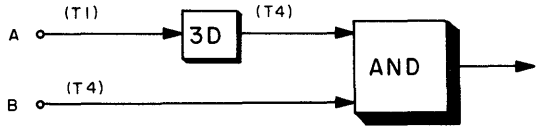


Figure 3—20. Use of Compensating Delay

delay circuit is placed in the lead to input A. The pulse enters this circuit at T1 and spends the next three bit-times getting through it. When the signal comes out, at T4, the second signal is just arriving on input B of the AND, so the two are made to arrive together by means of the compensating delay.

One other important matter must be mentioned. Notice that both the diode OR and AND circuits operate as described only when the binary 1 signals are more positive than the binary 0 signals. If the signal polarities are reversed — that is, if binary 1 is represented by a negative-going pulse or level — the OR circuit of figure 3—18 performs the AND operation, while the

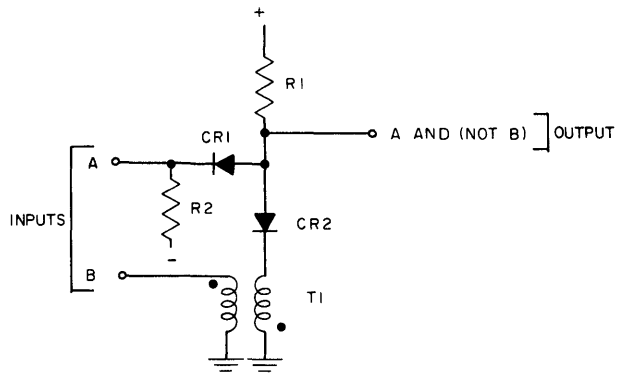


Figure 3–21. Diode Inhibit Circuit for Pulse Signals

AND circuit of figure 3–19 performs the OR operation. Thus, reversing the signal polarities interchanges the diode circuit functions. In practice, a circuit is named for the logic it performs (which does not change) at the spot where it is used. The NOT operation is more difficult to accomplish with diodes, although it can be done easily for pulse-type signals by using a polarity-inverting transformer and a source of continuous pulses (1's), either from the clock or from a pulse generator. The basic arrangement, shown in figure 3–21, is actually an inhibit (AND NOT) circuit.

The AND leg of the circuit includes resistor R2 and diode CR1, connected from a negative voltage source to the output line and common load resistor R1, which is tied to a positive source. The values and voltages are selected to place the output line at ground potential (binary 0) when there is a binary 0 on input A. The inhibiting branch of the circuit contains CR2, which conducts at all times, and the secondary of polarity-inverting transformer T1. There is a sizable inductance but practically no resistance in this branch.

When a positive-going pulse, representing a binary 1, appears at input A (with a 0 at input B), the difference in potential from the cathode of CR1 to the positive source is decreased, and the voltage on the output line rises to approximately the binary 1 value. The inductance of the transformer secondary opposes any sudden change in current, so there is little change in current flow and no clamping action by CR2 during the brief time that the pulse lasts. Thus, the pulse on input A is reproduced at the output. If, however, a positive-going pulse appears at input B at the same time as the pulse at input A, the secondary of the polarity-inverting transformer drives the cathode of CR2 more negative. This causes CR2 to conduct heavily and act as a clamp, preventing a pulse from rising at the output line. Therefore, the circuit produces an output of 1 only if a 1 is present at A AND (NOT B). (If a 1 appeared only at input B, a negative-going pulse would be formed at the output, but the output line can easily be clamped to prevent this.)

To use this inhibit circuit as an inverter, it is only necessary to connect a source of continuous positive pulses (one pulse each bit-time) to input A and connect the signal line on which the inversion is desired to input B. Now, the input to A is always 1, so if the signal on input B is 0, the output is 1; if the signal on B is 1, however, the output is 0. If this circuit is to be used for inhibition rather than straight inversion, more AND inputs and more inhibit inputs can be added in parallel with those shown in figure 3–21. The basic OR circuit can be modified in a fashion similar to this to develop an inhibiting OR.

All three types of diode logic are used in a digital computer by interconnecting them in various combinations and sequences to perform the proper logic operations on input signals at the proper times. (This is true of logic circuits using any kind of components, of course.) For example, consider a case with six inputs, which can be designated by the letters A through F. These are available as pulse-type signals on six lines. Not all are 1's at the same instant, or bit-time; instead, they appear as varying combinations of 1's and 0's. The designer knows this and must put together or arrange a group of logic circuits to produce an output pulse (binary 1) only when certain input combinations appear. One desired combination, for instance, is 1's on inputs A, C, and D and 0's or 1's (no matter which) on inputs B, E, and F. Putting together all the desired combinations, the designer finds that his circuit arrangement must produce an output of 1 when the following logical conditions are met:

$$(A \text{ OR } B) \text{ AND } C \text{ AND } [D \text{ OR } (E \text{ AND NOT } F)] = 1$$

where each letter represents the presence of a 1 at that input.

Once reduced to this form, the rest is easy. The designer is using diode logic circuits, each of which delays any pulse passing through it by an amount equal to one-quarter of a bit-time, usually written as $\frac{1}{4}$ D. (The delay is actually due to pulse-timing and reshaping circuits following the diodes.) Inputs A and B must be fed into an OR circuit, while E and F must go into an AND NOT or inhibit circuit, as shown at (a) of figure 3–22. Input D and the output of the inhibit go into another OR circuit. Now, the outputs of the two OR's, along with input C, are fed to an AND. The output of the AND circuit should be the desired result. But is it? Remember that each circuit causes a delay of $\frac{1}{4}$ D, and it immediately becomes obvious that this circuit arrangement at (a) of figure 3–22 will not work as desired for pulse signals. The logic is right, but the timing

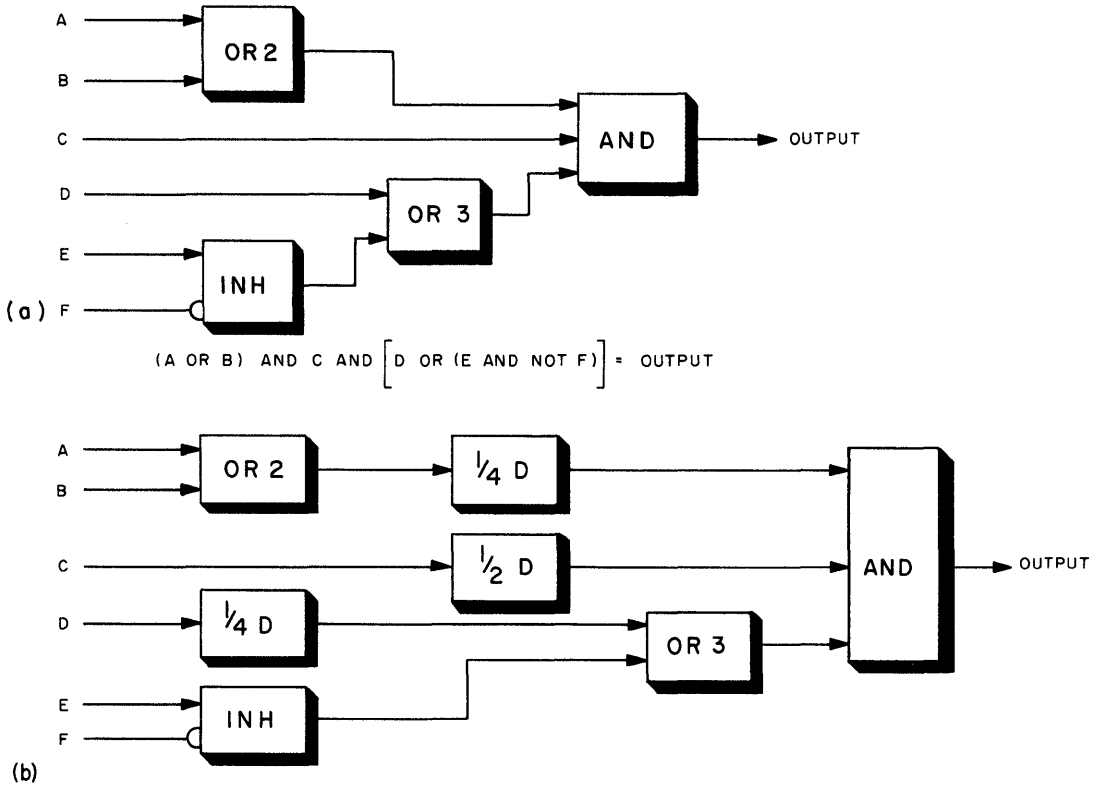


Figure 3–22. Sample Logic Circuit Combination

is not. (This arrangement at (a), incidentally, is the correct answer to the problem of drawing the logic of fig. 3–17.) A pulse appearing on input E, if it gets through the inhibit circuit, reaches the input to the AND one-half bit-time later than a pulse entering at the same instant on input C and encountering no delays. Pulses arriving on A, B, or D are each delayed one-quarter bit-time before reaching the AND. To make it possible for the desired input combinations to produce an output from the AND circuit, the designer must delay all the input signals by equal amounts. This is done by adding delay circuits in the proper places, as shown at (b) of figure 3–22. Now, the pulses of any of the desired combinations appearing at the input terminals reach the AND at the same instant and fire it, producing the binary 1 output pulse.

This illustrates the manner in which logic circuits are put together to perform the desired combinations or sequences of logic operations in a digital computer.

2.1.3 Vacuum Tube Logic Circuits

Vacuum tubes are excellent switching devices, offering the advantage of high speed and the possibility of amplifying signals as they are switched. They have comparatively large space, power, and cooling requirements, however, and these disadvantages indicate that the vacuum tube will see less and less use in the future, as newer devices are perfected.

One of the basic vacuum tube circuits is the NOT, or inverter, shown in figure 3–23. This is a simple triode amplifier in which driving the grid more positive makes the plate more negative. Thus, feeding the grid with a relatively positive signal representing binary 1 produces a less positive (relatively negative) plate voltage representing binary 0. In other terms, an input of A produces an output of NOT A. It is also true that an input of 0, or NOT A, yields an output of 1, or A. Although the NOT circuit is quite straightforward, there are many possible variations of the OR and AND circuits. One type of vacuum tube OR circuit appears in figure 3–24. A twin-triode tube is used, and the output is taken from the common cathodes which

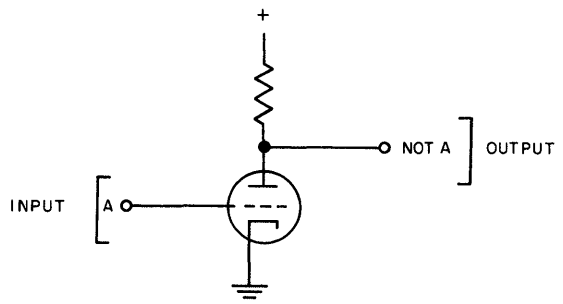


Figure 3–23. Vacuum Tube NOT Circuit

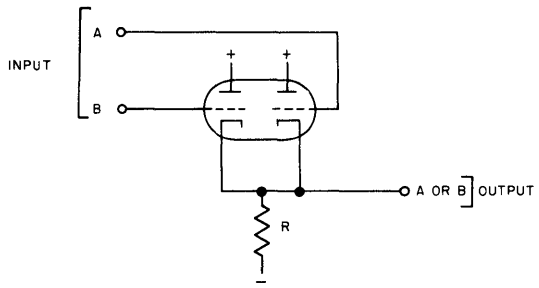


Figure 3–24. Vacuum Tube OR Circuit

are returned to a negative voltage source through a load resistor, R. (Signal inversion is avoided in this circuit by taking the output from the cathode circuit.)

When binary 0's (pulses or voltage levels) are on both inputs, A and B, both grids are held at a relatively negative potential, and the common cathodes are maintained at approximately this level, indicating binary 0. When positive-going 1 is placed on one of the input lines, the corresponding tube section conducts, and the increased current flow through resistor R causes a voltage rise on the output line, indicating a binary 1. If a 1 now appears on the second input line also, conduction increases in the other tube section, but the current flow through resistor R remains practically unchanged because the circuit values and arrangement cause the current to be divided between the two halves of the tube. In other words, when both tube sections are conducting at the same time, the two together draw approximately the same total current as either section conducting separetely. This arrangement is necessary because the output 1 must have the same amplitude whether one or both inputs are 1's. Thus, the circuit performs the OR operation. If more than two inputs must be taken care of, additional twin-triode stages must be used. With this type of circuit, two twin-triodes can handle four OR inputs; the cathodes of the second tube are simply tied to the output line shown in figure 3–24, and one cathode resistor serves both tubes.

In some types of circuitry, however, it is not possible to tie two circuits together in this fashion. For electronic reasons of circuit operation, loading, etc., the two stages must be kept separate, each feeding its own output line. In a case like this, a third OR stage becomes necessary to combine the output lines from the first two. Thus, three 2-input OR stages are required to combine four OR inputs, and either of the two arrangements shown in figure 3–25 is possible.

The circuit arrangement at (a) of the figure is undesirable for many uses because the input signals do not all pass through the same number of stages. Input C and D, for example, must each pass through three stages, or "levels of logic," while B passes through two and A through one. In some cases, this could cause timing

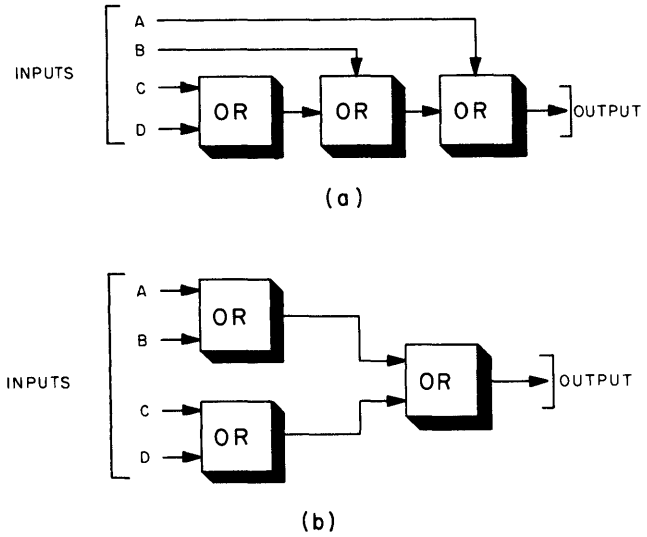


Figure 3–25. Multiple-Input OR Arrangements

difficulties of the type illustrated in figure 3–22. From the timing standpoint, the symmetrical arrangement at (b) of figure 3–25 is better. Here, the four input signals are delayed equally, and all pass through the same number of logic levels. (If the first two OR stages are vacuum tube circuits, it is entirely possible that the final OR stage might be nothing more than a parallel connection of their outputs, most likely through blocking diodes that prevent circuit interaction. This parallel connection is, nevertheless, a stage of diode OR logic.) This arrangement, at (b), is a basic "many-to-one" set up, sometimes called a converging switch because many inputs are brought together and switched down to one (or a few) outputs. If the inputs must have the AND relationship, AND circuits are used in place of OR's. The logic circuits themselves may be made up of switching devices of any practical type. Another common arrangement (not shown) is a "one-to-many," or diverging switch, which feeds the output of one logic circuit to several others in parallel, thus branching a single signal to several different paths.

One possible vacuum tube AND circuit appears in figure 3–26. The circuit consists of two inverters, V1 and V2, feedingan inverting AND, V3. This is the fundamental circuit, of course, like all circuits shown in this chapter. For satisfactory electrical operation, various refinements must be added.

The operation of vacuum tube inverters has been described; that of V3 can best be understood in terms of relatively positive and negative signals at its grids and plate, summarized in the small table included in the illustration. The electronic operation of V3 is identical to that of the twin-triode in the OR circuit in that one section alone draws approximately the same amount of current as both sections operating together.

AND OPERATION

INPUTS		I OUTPUTS		V3 OUTPUT
A	B	V1	V2	
−	−	+	+	−
+	−	−	+	−
−	+	+	−	−
+	+	−	−	+

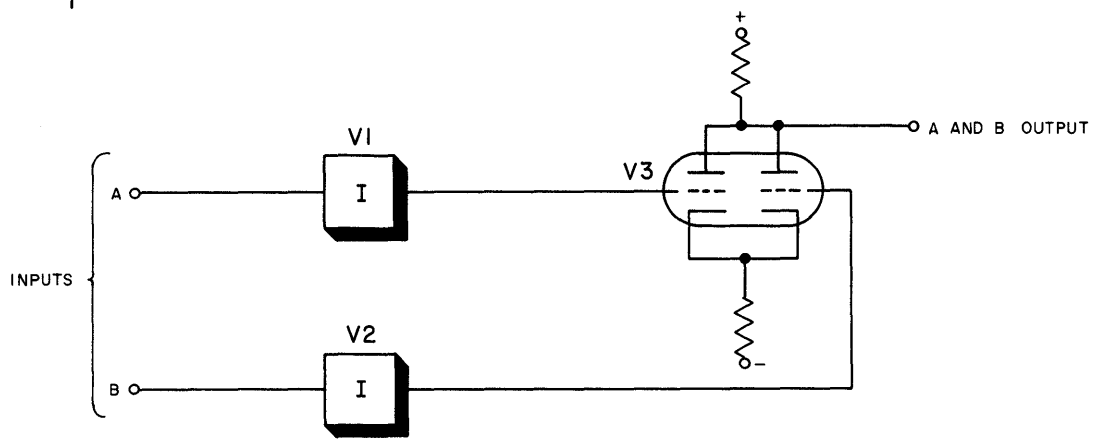


Figure 3–26. Vacuum Tube AND Circuit

When positive (1) signals are applied to both grids of V3, both tube sections conduct heavily, and the output, taken from the plate circuit, is relatively negative at the binary 0 level. To place 1's on both grids of V3, 0's must be applied to the inverter inputs (V1 and V2). In other words, with 0's at inputs A and B, the AND circuit output is 0. Since either tube section alone can handle the full current load, it follows that the circuit output is 0 as long as there is a 0 at either input because the corresponding inverter places a 1 at one grid or the other of V3. Thus, one tube section conducts heavily, and the output is a 0.

When 1's appear simultaneously at both inverter inputs, however, the inverters apply 0's to the grids of V3. These relatively negative voltages cut conduction through both halves of the twin-triode, and the plate voltage swings positive, to the binary 1 level. So the vacuum tube AND circuit provides an output of 1 only when both inputs are 1's. If pulse-type signals are used, V1 and V2 can be replaced by polarity-inverting transformers.

Notice that the logic of these vacuum tube OR and AND circuits is reversed, like that of the diode circuits, if 1's are represented by negative voltage levels or pulses instead of positive signals, as described here. That is, with negative signals, the OR of figure 3–24 becomes an AND and the AND of figure 3–26 becomes an OR.

At first glance, it seems easy to use a multigrid tube to perform the AND operation, applying different signals to different grids, so plate current can flow only when all signals are present at the same time. This seldom works out in practice, however, because different grids tend to have different characteristics and varying effects on the plate current. The design of special tubes brings in problems of expense and reliability, although a few with two control grids are in use. One special form of AND circuit using a multigrid tube is the *gate circuit*, or gate tube, abbreviated GT. A gate is a circuit that passes a signal only when another, controlling signal is present; actually, it is a 2-input AND. In this special form, however, the input to be gated is always a pulse, while the control signal may be either a voltage level or another pulse. Only when the GT is made ready, or *conditioned*, by the presence of the control signal can the input pulse be gated through.

Since the output is always a pulse (the tube can conduct only when positive signals are on both grids), a transformer can be placed in the plate circuit to correct the polarity inversion. Thus, the output pulse has the same polarity as the input pulse. The circuit is shown in figure 3–27, using an ordinary pentode. Usually, the signal to be gated is applied to the control grid and the control signal is fed to the suppressor. If both signals are pulses, the control signal is generally made to last longer and is applied a little ahead of the input pulse to prepare the tube to respond to the fast-rising input pulse.

2.1.4 Transistor Logic Circuits

Although transistors are frequently thought of as replacements for vacuum tubes and can often be used in similar logic circuits, some newer types are well-suited to straightforward use as switches. This type of circuitry, in fact, has been called *direct-coupled transistor logic*. Transistors offer several advantages for

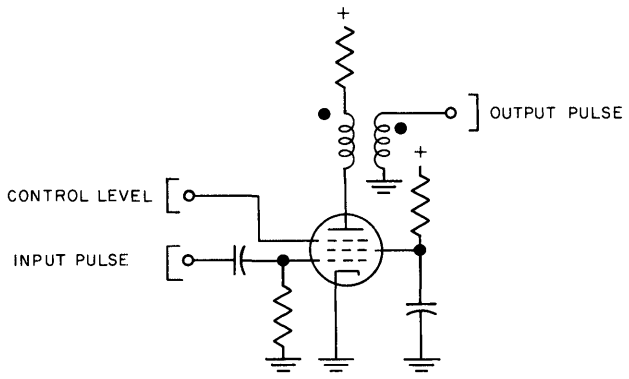


Figure 3–27. Gate Circuit

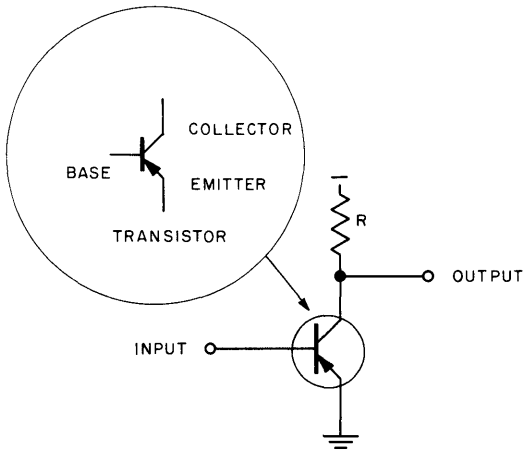


Figure 3–28. Basic Transistor Switch

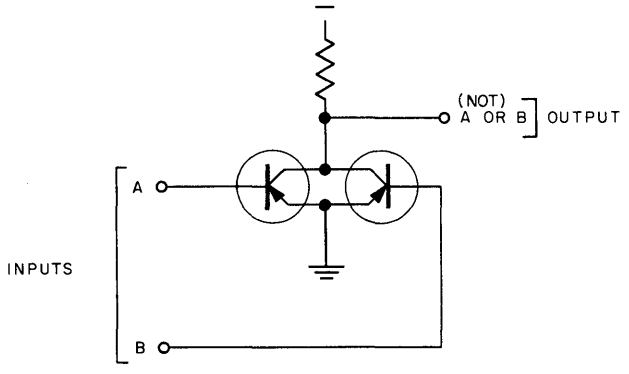


Figure 3–29. Transistor OR Circuit

digital computer use. They are small and well-suited to miniaturized circuits, require little power, and do not dissipate much heat. As switches, they are as fast as vacuum tubes; hence, they can be used in high-speed computers, and they have excellent reliability.

The basic transistor switch is shown in figure 3–28. The emitter is grounded, and the collector is normally biased at about —3 volts. With no signal on the base, that is, with the base at approximately ground potential the transistor is cut off. The input, which must be a negative-going signal, supplies current to the base, switching on the transistor and driving it immediately to saturation. When this happens, the collector swings from its normal bias of —3 volts almost to ground potential, placing a positive-going level or signal on the output line. Therefore, this is an inverter, or NOT switch, which means that if a negative signal at the input is considered to be a binary 1, the positive output signal must be a binary 0. This reversal of polarity through each circuit might, at first, seem to complicate the logic of the machine, but actually does not because it occurs in a regular and consistent fashion, just as it does in most vacuum-tube circuits.

The transistor OR circuit, shown in figure 3–29, uses two or more transistor switches in parallel. A given transistor is turned on when a negative signal appears at the corresponding input, thus producing a positive signal at the output line. (If two or all transistors are turned on, the output is still the positive signal.) Although this is actually an inverting OR (an OR NOT), it is rarely called this. In practical circuitry, it may be followed by an inverter, if necessary.

The transistor AND circuit appears in figure 3–30. This is nothing more than a pair of switches in series, and there can be no conduction until both are turned on at the same time. When this occurs, the output swings positive. (Again, this is really an AND NOT and may be followed by an inverter.) More AND inputs can be placed in the series arrangement, but there is a limit to the number because the voltage drops across the transistors, small as they are, are added together and reduce the voltage swing of the output.

The manner in which direct-coupled transistor circuits are tied together is extremely simple. A sample is

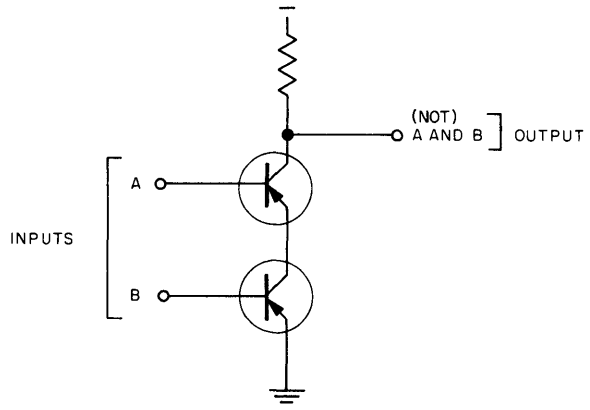


Figure 3–30. Transistor AND Circuit

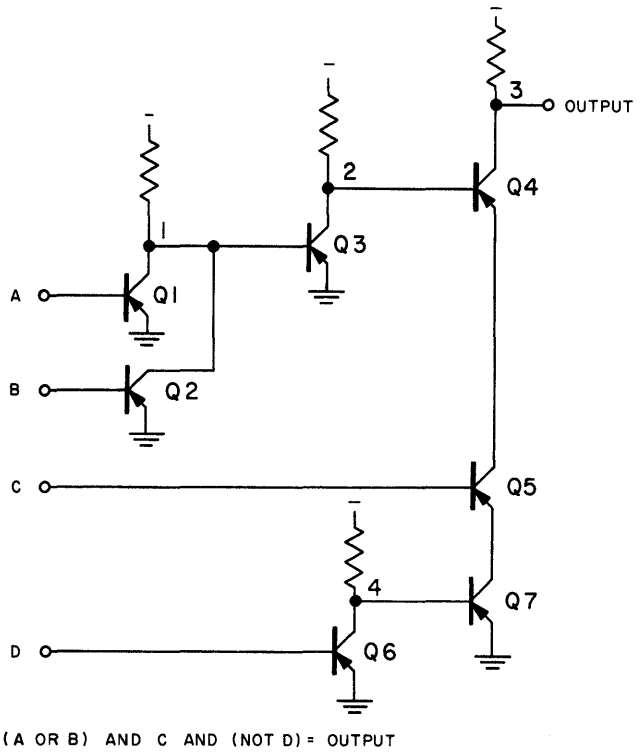


Figure 3–31. Sample Transistor Circuit Arrangement

shown in figure 3–31. This a straightforward connection of switches to perform the logic operations shown below the drawing, based on the assumptions that the input signals are negative and a positive output signal is wanted when the desired input requirements are satisfied. (Under certain conditions, the circuit could be further simplified.) All important voltage changes occur at the load resistors, one of which must be connected to each collector or collector-base junction. Accordingly, the point where each resistor is connected is called a *node*, and the operation is most easily followed by considering the voltage changes at these nodes.

In figure 3–31, transistors Q1 and Q2 form an OR circuit; thus, when either A or B appears (as a negative signal), the corresponding transistor is turned on and node 1 goes from −3 volts to approximately ground potential. Until this occurs, the base of Q3 is biased negative by node 1, and Q3 conducts, holding node 2 at ground and keeping Q4 cut off. When A or B appears, however, node 1 goes to ground and shuts off Q3, whereupon node 2 becomes negative and turns on Q4. Input C, also a negative signal, turns on Q5 directly. Input D is NOT input, so Q7 must be turned on only when D is not present. The inverter, Q6, accomplishes this. When D is absent, Q6 is shut off, and negative bias from node 4 turns on Q7. When D appears, however, Q6 turns on, and node 4 goes to ground, cutting off Q7.

Thus, 4, Q5, and Q7 in the AND circuit are on only when the desired logic conditions are satisfied. When these three transistors are conducting, node 3 is at ground potential, placing the desired positive (relatively positive) signal on the output line. In a manner similar to this, any desired combination of logic operations can be carried out with transistors.

2.1.5 Magnetic Core Logic Circuits

Magnetic cores, originally developed and now widely used as storage devices, appear to have definite possibilities for use as switching or logic devices as well. The theory of cores is emphasized here because, currently, they are used so extensively for storage and may find equally extensive use for logic in the near future. The cores, shaped like tiny doughnuts, are made of a material with magnetic properties giving it a hysteresis loop (the characteristic curve of its magnetic properties) that is nearly rectangular. Some metals and certain ceramic materials called *ferrites* possess the desired characteristics and are used to make cores that are good switching devices. The ferrite cores, small in size, are used almost exclusively for storage at the date of this writing. The cores made of metal—usually molybdenum permalloy, sometimes a 50-percent nickel-iron alloy—are called *tape cores* because they are formed by wrapping turns of a thin tape made of the metal around a small bobbin. Thus, the small core is built up of a number of turns or laminations, as indicated in figure 3–32. This laminated structure holds eddy currents to a minimum and thereby reduces power losses, increasing efficiency.

At least three small coils are ordinarily wound on each tape core. Two of these are obviously for input and output of information; the third is needed for sensing, to extract, or *read out*, the information (1 or 0) stored in the core. To use the core for logic operations, instead of or in addition to storage, it must have two or more input coils. The basic, three-winding core is

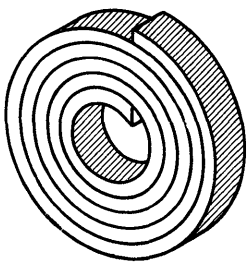


Figure 3–32. Tape Core Construction

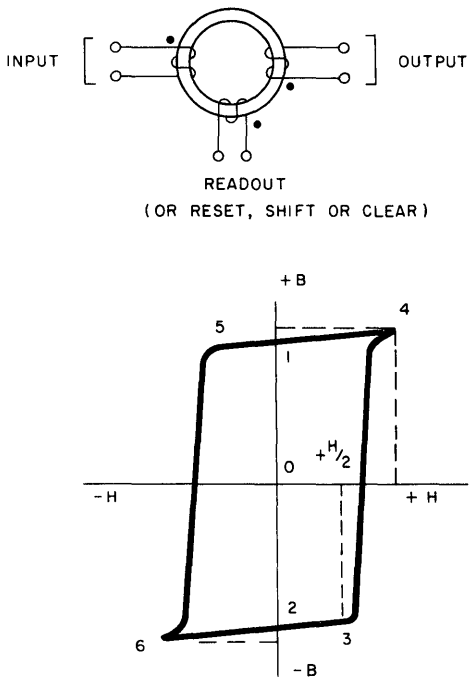


Figure 3–33. Magnetic Core and Hysteresis Loop

shown schematically in figure 3–33, along with the closed-curve hysteresis loop of its magnetic characteristics. Since the core is made of a magnetic material, it is really a small magnet with the direction of flux (magnetic lines of force) running either clockwise or counterclockwise inside the metal ring. This direction of magnetization can be quickly reversed by applying a current pulse of the proper polarity and amplitude to a coil wound on the core. Thus, by deciding that one direction of flux represents a 1, the other a 0, a pulse on the input coil can be made to insert a 1 which the core then stores because its direction of magnetization can be changed only from outside. Inserting information into a core can be called writing a 1 or a 0 into the core, or setting the core to 1 or 0. Taking out the stored binary bit is called *reading out*, or simply *readout*.

How are these operations performed? This can best be seen by use of the hysteresis loop of figure 3–33, which shows the effect of an external magnetizing force, H, on the density and direction of the magnetic flux, B, in the core. The signs indicate magnetic polarity, or flux direction. The magnetizing force is applied, of course, by sending a pulse through a coil wound on the core, making the dotted terminal (fig. 3–33) negative. (Dots at the terminals of all input coils and the readout coil indicate the terminals that must be made negative to apply a magnetizing force in the desired direction. The output coil dot indicates the terminal that becomes negative when a 1 is read out.)

When no pulse is applied and there is no current in the coil, the external magnetizing force (H) is at zero, and the density of the lines of force remaining in the magnetic core (called the *remanent flux density)* is shown by the point at which the loop intersects the B axis. Since the remanent flux may be aligned in either direction (clockwise or counterclockwise, looking at the core in figure 3–33), there are two possible points of intersection, labeled 1 and 2.

For the sake of illustration, it can be assumed that flux in the direction —B represents binary 0, the direction +B represents binary 1, and that the core has been left in the 0 state, with the remanent flux density at point 2. If a pulse is now applied to the input coil, a magnetic field begins rapidly building about the coil, and this external magnetizing force, acting in the direction +H, opposes the remanent flux in the core. As H increases, the total flux density decreases from point 2 toward point 3 on the curve. When the external field reaches half-strength (+H/2), the flux density is at point 3. As H continues to increase, a point is reached (the bend or knee of the curve, beyond point 3) where the external field overcomes the permanent field of the core. The flux in the core swiftly drops through zero and reverses direction. The flux density then increases very rapidly toward point 4 on the curve as H rises to its maximum value. At point 4, the core is saturated (cannot hold any more lines of force). As the pulse in the input coil dies away, H returns to zero, and the flux density decreases from point 4 to point 1, where it remains until some other external force is applied. Thus, the core has been switched and is now storing a 1. The switching takes place in a few microseconds at the most. If another 1 is applied to the input coil, the external field this time strengthens, rather than opposes, the flux in the core, so the flux density simply goes from point 1 to saturation at point 4, then back to point 1 again. Since input pulses are always the same polarity, the input coils are used only to set the core to 1.

To read information out of a core, a pulse is applied to the readout winding in a polarity (dotted terminal negative) that sets up a field in the —H direction. In other words, the readout pulse acts to set the core to 0. The readout winding may be called by various other names, such as reset, shift, or clear winding. If the core is at 0 already (has not been set to 1) the remanent flux density is at point 2 on the hysteresis loop, and the application of the magnetizing force —H merely moves the flux density to point 6 (saturation in the 0 direction). When the force is removed, the flux density returns to point 2. This small change in flux density induces only a very small voltage in the output coil.

However, if the core is in the 1 state when the readout pulse is applied, the remanent flux density is at point 1. The force —H is sufficient to pull the flux density past the knee of the curve at point 5 and down

to saturation in the 0 direction at point 6, thus switching the core back to 0. When —H disappears, the remanent flux density returns to point 2. The large change in flux density from +B to —B induces a large voltage in the output coil. (The dotted output terminal goes negative.) This voltage, of course, can be used to set other cores or drive other logic circuits. (Notice that there must also be an induced output voltage when the core is set to 1 because of the flux change from —B to +B. This voltage is opposite in polarity to the desired output voltage, however, and can be blocked by simply connecting the anode of a crystal diode to the undotted output terminal.)

Before leaving the hysteresis loop and the theory of magnetic core operation, one other important matter must be mentioned for future use. This is the fact that a magnetizing force of half-strength (H12, positive or negative) is not capable of switching the core from one state to the other. Therefore, a pulse of half the normal input or readout current amplitude cannot switch the core, since the strength of a magnetic field is directly proportional to the amplitude of the current used to set it up. This is not particularly important in dealing with cores used as logic devices, but becomes significant in using arrays of cores for large-scale storage, which is discussed in Chapter 4.

A magnetic core OR circuit, shown in figure 3—34,

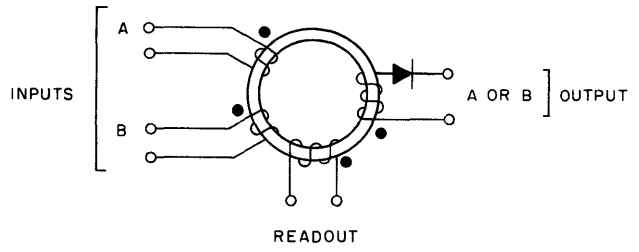


Figure 3—34. Magnetic Core OR Arrangement

is easily made by winding a core with two or more input coils, each of which can set the core to 1, independently of the others, when a pulse input signal is received. (It should by now be apparent that pulse-type signals are used in most core applications.) When one of the OR inputs is pulsed, the core is switched and stores the 1 until a readout pulse is applied. As explained above, the readout signal resets the core to 0 and, in the process, reads out the stored 1 as a pulse on the output line. The device is then ready to perform the OR operation again. Notice particularly that if more than one input pulse appears before the readout pulse, this arrangement does *not* produce an output for each of them. In the usual circuits, however, all readout windings are pulsed at regular intervals, ordinarily every bit-time. Thus, if a core is set to 1 by a pulse (or simultaneous pulses) appearing during a given bit-time, the core is almost immediately set back to 0 by the next readout pulse, so an output is produced for each input or combination of inputs, when desired.

An output pulse (if any) is produced almost at the instant the readout pulse is applied. This output may be sent to the input of another core which gets its readout pulse from the same readout line as the first core. This means that the second core could receive an input pulse and a readout pulse at the same time. Whether this core would be read and set properly or not would depend on the split-microsecond timing of the two pulses, and such chances cannot be taken in computer circuitry. The problem is easily solved by delaying the output pulse from each core long enough to be certain that the readout pulse has passed before the output pulse reaches the next input. The delay is introduced by inserting a resistance-capacitance, or resistance-capacitance-inductance, network in series with each output, as shown in figure 3—35.

The inhibit operation is easily performed by a magnetic core. As shown in figure 3—36, the core has one or more OR input windings and an inhibit winding that is larger than and opposite in polarity to the others. When a pulse is present on the inhibit input, the resulting magnetic field opposes or bucks the field set up by one or more OR inputs and thus prevents the core from being set to 1. The inhibit winding must, therefore, be large enough to cancel the effect of simultaneous pulses on all the OR inputs. Two or more

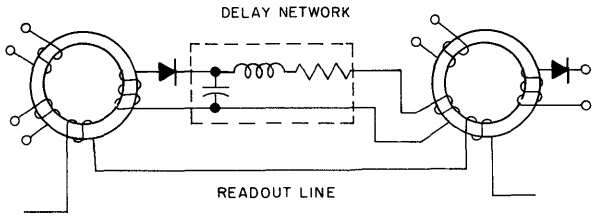


Figure 3—35. Interconnection of Magnetic Cores

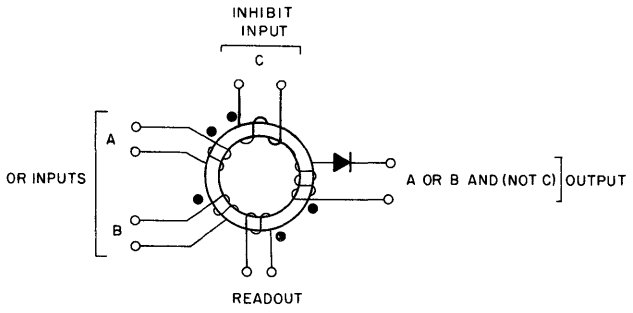


Figure 3—36. Magnetic Core Inhibit Arrangement

inhibit windings can be placed on a single core, space permitting. In this case, a pulse on either inhibit winding (or both) prevents the core from being set to 1 by an OR input.

In practical computer logic circuitry, the AND operation is difficult to perform reliably using a single core. It seems a relatively simple matter to wind two or more AND input coils in such manner that all must be energized simultaneously to provide enough magnetizing force to switch the core, and it is true that this can be done under controlled conditions, such as in the main memory element. In actual logic circuits, however, the amplitudes of the information pulses cannot be rigidly controlled because the diodes and delay networks attenuate them, often by differing amounts, and any two cores may produce output pulses of somewhat different amplitudes. For OR and inhibit cores, all that is necessary is to insure that every pulse has at least sufficient amplitude to switch or inhibit a core; oversize pulses are no problem. For a core with AND input windings, however, oversize pulses on two out of three inputs might set the core to 1, while undersize pulses on all three might not. Also, the hysteresis loops may vary somewhat from core to core, further complicating the problem. One reliable means of performing the AND operation requires three inhibit cores and three simple pulse generators. A pulse-generating core can be made to put out a continuous stream of pulses (1's) by placing a steady d-c voltage level on the input winding and applying a readout pulse each bit-time. The readout winding must be large enough to switch the core back to 0, each time a readout pulse appears, against the opposition of the field set up by the input voltage level. As soon as the readout pulse passes, the input level sets the core to 1 again.

Performing the AND operation by using cores with other functions depends on two facts. The first is that a quantity inverted twice is returned to its original state. When a 1 is inverted, for example, it becomes a 0. Invert it a second time and it becomes a 1 again. In logic terms:

$$\text{NOT (NOT A)} = \text{A}$$

The second useful fact is that two inhibit inputs to a single circuit bear an AND relationship to each other. To get an output from such a circuit, as diagrammed at (a) of figure 3–37, the normal input, C, must be present, and both the inhibit inputs, A and B, must be simultaneously absent. If a pulse generator is connected to input C, 1's are always present on C, and the circuit produces an output only when the inputs are (NOT A) AND (NOT B).

Because of the relationship between the two inhibit inputs, this arrangement is the same as that shown at (b) of figure 3–37, two inverters feeding an AND circuit (sometimes called a NEITHER . . . NOR circuit).

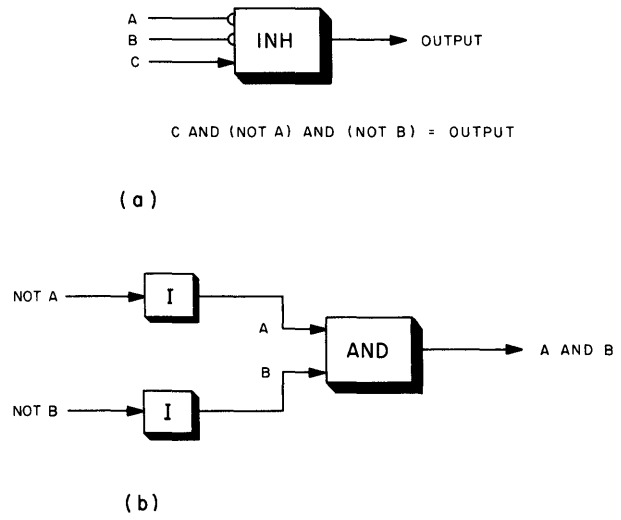


Figure 3–37. Effect of Two Inhibit Inputs

The output is A AND B, as required, but the inhibit circuit (core) must be fed with NOT A and NOT B. This means that A and B, which are represented in the magnetic core circuitry by pulses, must be inverted before being sent to this 2-inhibit-input core. This can be done for each signal by an inhibit core and pulse generator combination, by connecting the signal line to the inhibit winding. The complete arrangement for performing the AND operation in this manner is diagrammed in figure 3–38. The inhibit cores are represented by the blocks labeled INH, the pulse generator cores by the encircled 1's. When either A or B is not present (no pulse), the corresponding inhibit core puts out 1's (inserted by the pulse generator core) to one of the inhibit windings of INH 4. With a 1 at either of its two inhibit windings, INH 4 cannot produce an output 1. When A and B are present at the same instant, however, no output pulses are produced by INH 2 and 3. No pulses appear at the twin inhibit windings, and INH 4, therefore, puts out a pulse representing A AND B. The AND operation is thus achieved through the use of six magnetic cores. Once the three simplest logic functions can be handled, as shown here, complex

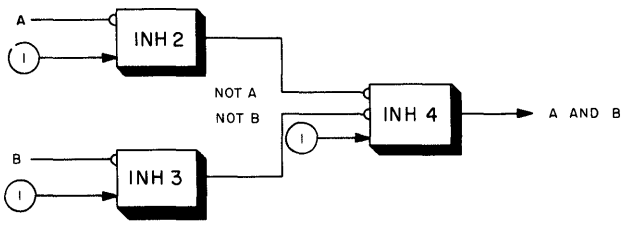


Figure 3–38. Magnetic Core AND Circuit

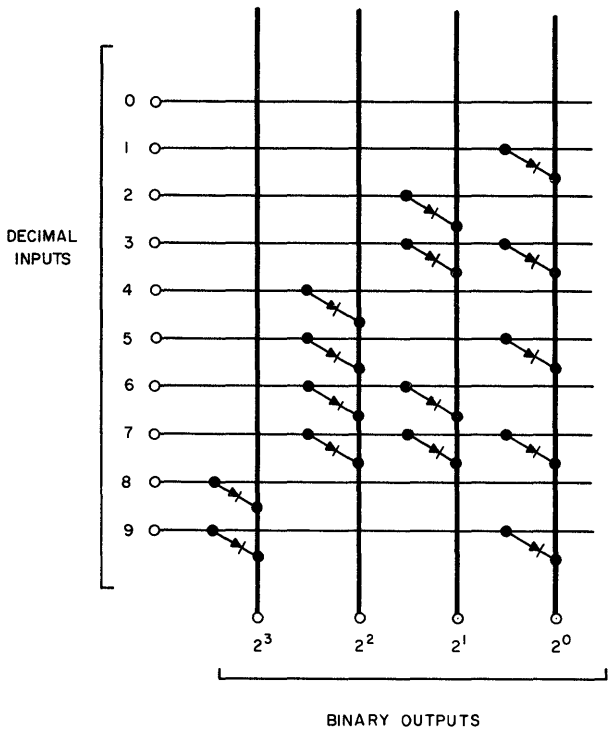


Figure 3–39. Diode Matrix

chains of core logic circuitry can be built up by combining them.

2.1.6 Matrices

The logic circuits and circuit arrangements considered thus far have been types producing a single output, usually from two or more inputs. Another type of importance is the multiple-output switching network, especially the category called *matrices*. A multiple-output network produces a different output for each different input or combination of inputs. The *matrix* performs the same job but gets its name from the manner in which it is drawn schematically (and often built), with components and connections arranged in rows and columns. A very simple, but typical, matrix of diodes is shown in figure 3–39. This matrix is capable of translating any of 10 positive input signals representing the 10 decimal digits into parallel binary *tetrads* (groups of four bits). At first glance, it appears that the diodes are unnecessary, but actually some sort of isolating, 1-way device is needed at the interconnection points to prevent the signal on one line from backing up on others. In operation, a level or pulse appearing on input 5, for example, puts signals on the 2^0 and 2^2 output lines. The resulting output is 0101, the binary tetrad representing decimal 5.

In the example of figure 3–39, a signal on one input results in a particular combination of output signals. Figure 3–40 shows a small matrix, using logic circuits, in which each desired combination of inputs produces a separate output. In this *array* (which means regular or symmetrical arrangement), the input signals go through intermediate OR circuits before being combined in the matrix proper (the six AND circuits). This is only one of many possible matrices. Diode matrices of several varieties and matrices constructed of OR and AND circuits are especially common, but they can be

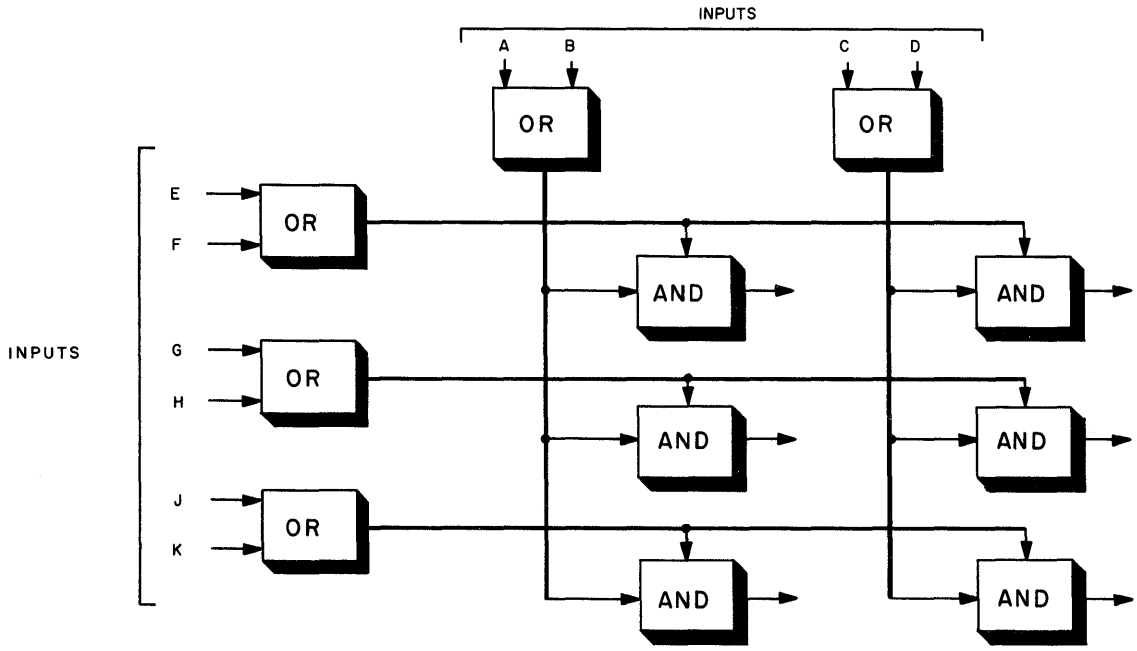


Figure 3–40. Matrix of Logic Circuits

made up of nearly any type of switching device or logic circuit.

2.2 SMALL-SCALE STORAGE CIRUITS

The fact that all arithmetic and data-processing operations in a digital computer are accomplished by switching and storing electrical signals has been mentioned several times. Now, the switching circuits that perform the logic operations have been examined in detail, alone and in simple combinations satisfying various conditions. It is easy to see that if signals representing 1's and 0's must be combined in certain logic circuits and if a signal available now is needed a little later, some means of *storing* this signal must be found to hold it until it can be used. This necessity was pointed out back in the discussion of figure 3–22, in showing how logic circuits are put together. Each of the delay circuits in (b) of that figure can be thought of as a temporary storage circuit, capable of storing or holding one pulse-type signal for one-quarter or one-half of a bit-time.

This type of storage device (usually some form of electromagnetic delay line) works nicely for pulse signals and for brief storage periods of a few bit-times. But what if voltage levels are used and what if the storage period must be 50 bit-times, or 10,000, or indefinite? It is certainly impractical to send signals from all parts of the computer to the main memory or storage element. The resulting circuitry would be an impossible maze, for one thing, and the memory would have to be adapted to handle single bits as well as words. What is needed is small-scale, on-the-spot storage for use during operations.

A little thought indicates that the answer is a bistable device or circuit that can be set to 1 or 0 by the signal it receives and that will then remain in that state, after the input signal disappears, until it is reset. It must, of course, be able to indicate its 1 or 0 state to other circuits by means of one or more outputs. This indication may be continuous or it may be supplied only when demanded, as in the case of a magnetic core which indicates the bit stored only when a readout pulse is applied. With such a device or circuit, it becomes an easy matter to store a single bit until it is needed. And in parallel transmission, it is equally easy to store a complete word (number) simply by providing storage places in parallel for each bit of the word. A group of devices or circuits for storing a complete word is called a *register.*

Where serial transmission is used, a somewhat different storage method is needed because, obviously, a series or train of bits fed to a single storage device would set and reset it and only the last bit would actually be stored. One possible solution would be to switch the serial word into parallel form to store it, but, except for certain special purposes, this is clumsy and requires extra circuitry. Another, more commonly used, method will be described.

2.2.1 Bistable Circuits

With the possible exception of semiconductor diodes, all the switching devices used in logic circuits can be easily adapted to circuits for bit storage. Magnetic cores were originally developed for this purpose, and their use in logic circuits came later. The idea of using switching devices to create storage circuits may seem a little strange, but consider figure 3–41. This is a bistable storage circuit made up of switching circuits and a feedback loop.

When 0's are on both inputs, the 0 at the input of inverter I2 produces a 1 at its output. The 1 passes through OR 3 to the input of I3 which produces 0 at the circuit output. This 0 is also fed back around the loop through OR 2 to the input of I2, so with 0's at output and input the circuit is stable and remains in this state. The circuitry leading to inputs A and B is so arranged that 1's never appear on both lines at the same time. If a 1 now appears temporarily on input A, which is labeled as the *set input*, it goes through OR 2 to I2 and causes I2 to put out 0. This 0 goes to I3 which then places 1 on the output line. Since this 1 also is fed back to the input at OR 2 and I2, the circuit is again stable. Thus, a 1 on the *set* input switches the circuit to the 1 state where it remains after the input has

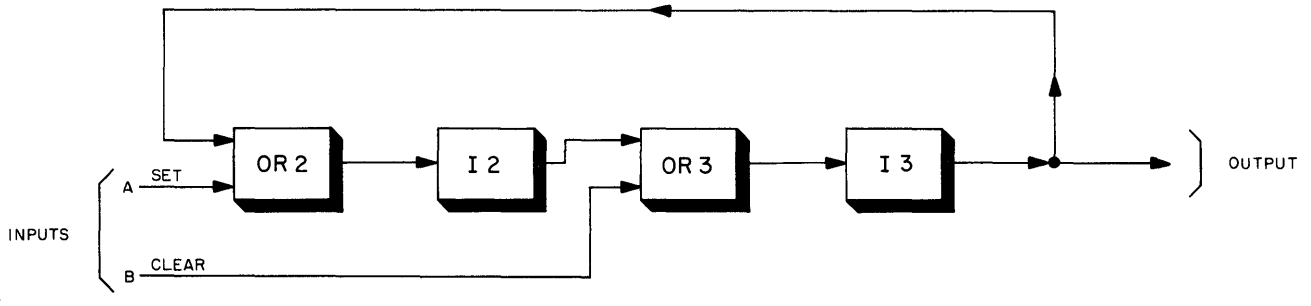
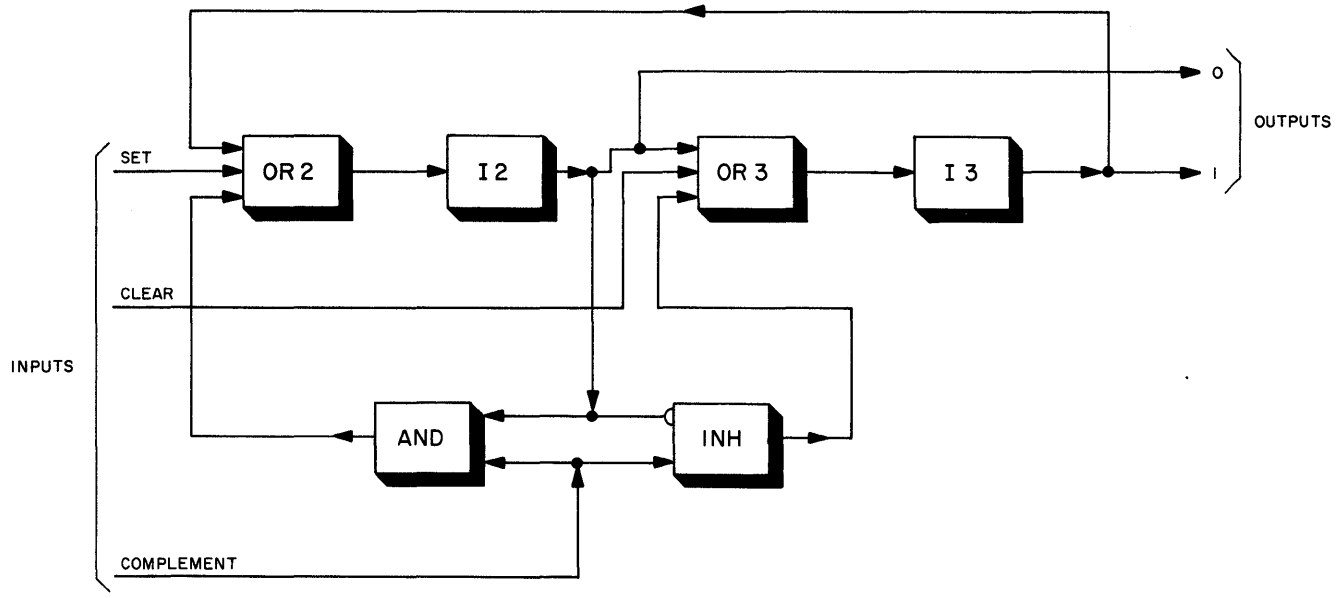


Figure 3–41. Basic Bistable Storage Circuit



Fiaure 3—42. Complete Logic Circuit Flip-Flop

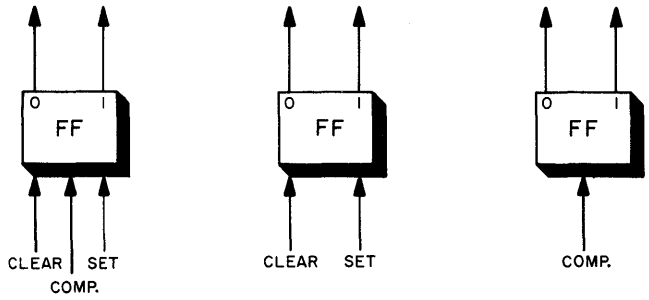


Figure 3—43. Flip-Flop Circuit Symbols

disappeared. A 1 appearing on input B, called the *clear* or *reset input*, causes I3 to produce a 0, switching the circuit back to the 0 state.

An arrangement of this sort is called a *flip-flop* (abbreviated FF), and various types are widely used in digital computers for bit storage. The inputs are normally pulses, although it is possible to use levels; the outputs are voltage levels.

The flip-flop output shown in figure 3—41 is always called the *1 output*, whether it is indicating 1 or 0. Notice that a second output from the circuit can be taken from I2 and that this is opposite in state to the 1 output. In other words, when the 1 output carries a down level (indicating that the flip-flop contains 0), the output from I2 is up, and vice versa. This output from I2 is called the *0 output*. In this manner, it is possible to indicate not only the state of the flip-flop but its complement (opposite) as well, which is often useful.

The use of only one output line (either one) is called *single line transfer*; using both is *double line transfer*. It is also possible—and common practice—to add a third input, so arranged that a pulse on this input switches the flip-flop, regardless of its state. This *complement input* can be added as shown in figure 3—42. The extra circuits, the AND and the INH, are used to switch the complement input to the proper OR circuit to either set or clear the flip-flop, depending upon its state at the time the pulse is received. (It is assumed that there is sufficient delay in the logic circuits to prevent switching I2 before the input pulse has disappeared.) For some purposes, flip-flops do not require separate set and clear inputs and, therefore, have only the complement input.

Flip-flops have their own circuit symbols, which may resemble any of the three shown in figure 3—43. The designation, FF, is often omitted, but the 0 and 1 output sides are always shown. The input leads are not usually labeled. Intead, for ease of understanding, the set input is always drawn on the same side of the block as the 1 output, the clear input on the same side as the 0 output. Thus, a pulse entering on the 1 side sets the

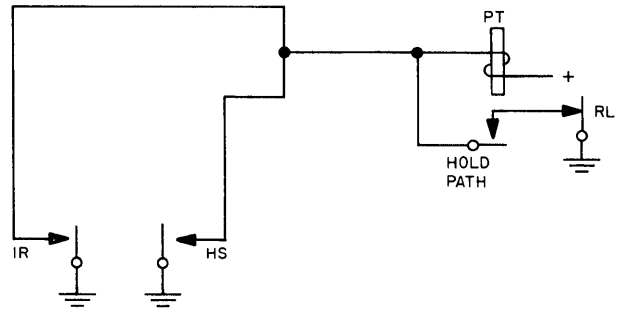


Figure 3–44. Relay Storage Arrangement

flip-flop to 1. The complement input enters at the center of the block.

Constructing a flip-flop of logic circuits is instructive but not very practical, of course, since bistable storage circuit such a multivibrators can be made directly from vacuum tubes, transistors, etc.

2.2.1.1 Relay Storage

Even the lowly relay can be made to remain in the operated (1) state after the passing of the operating signal. One of the simplest methods of accomplishing this (there are several) is shown in figure 3–44. This is essentially the same circuit as that in figure 3–14, but with the addition of a hold path to make the PT relay act as a storage device. The PT relay operates when either the IR or the HS relay is operated, closing the corresponding set of contacts and completing the circuit to the PT coil. As soon as PT operates, however, one pair of its own contacts closes the hold path to ground through the normally-closed contacts of RL. If IR or HS now releases, removing the original operating signal, relay PT nevertheless remains operated through its hold path. When the time comes to release PT (return it to the unoperated state), relay RL is operated and its contacts open the hold path of PT. By this means or others, relays used in logic circuitry can be made to store information indefinitely.

2.2.1.2 Vacuum Tube Flip-Flops

The basic bistable multivibrator circuit appears in figure 3–45. Actually, this circuit is bistable only if the component arrangement and values are correct; this, however, is a design problem. In operation, the cross-coupling between either plate and the opposite grid means that one triode at a time can conduct, but not both. The decrease in plate potential that occurs when one tube conducts is coupled to the grid of the opposite tube, driving it to cutoff. With V1 conducting and V2 cut off, for example, the circuit is in one of its two stable states. The plate potential of V1 is relatively low, placing a down level voltage on the 1 output. The plate potential of V2 is high because the tube is not conducting, so an up level voltage is on the 0 output and the circuit is said to contain a 0.

To change the state, a positive pulse is applied to T1 through the set input. This pulse is inverted through the transformer action, and a negative pulse is placed on the grid of V1. This drives the tube toward cutoff; the plate potential rises as conduction decreases; and the rising potential is coupled to the grid of V2. As V2 begins conducting, the decrease of voltage at its plate is coupled to the grid of V1, helping the initial pulse to cut off V1 completely. With V1 cut off, its plate potential is high, so an up level voltage is now on the 1 output, while conduction through V2 places a down level on the 0 output.

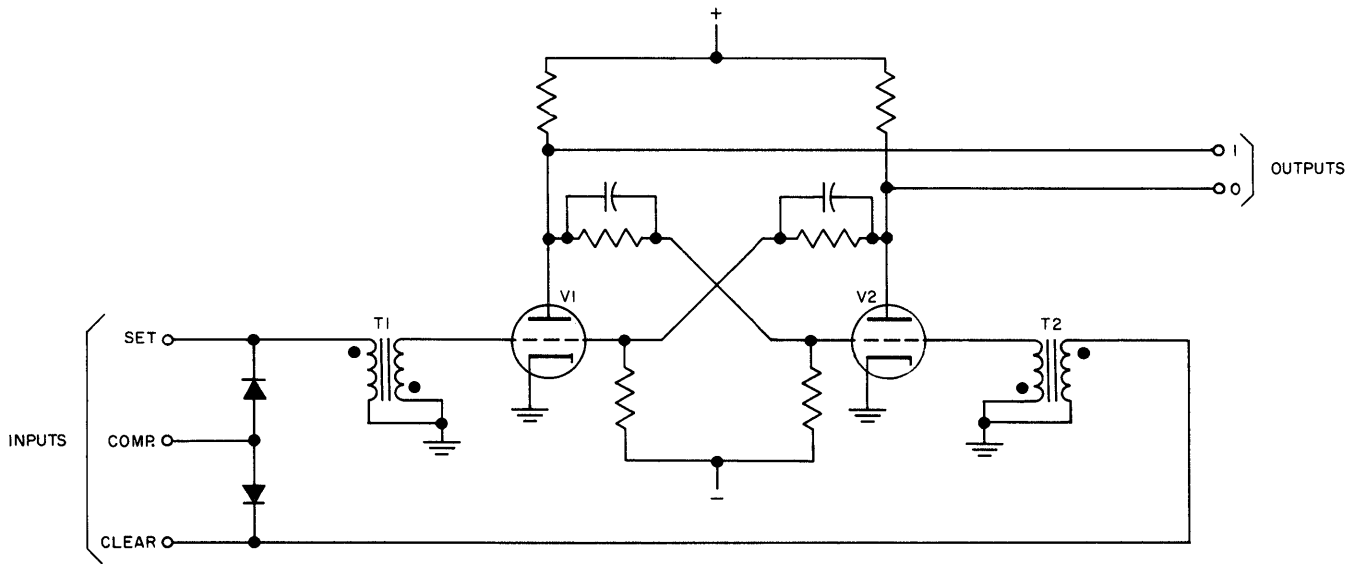


Figure 3–45. Basic Vacuum Tube Flip-Flop

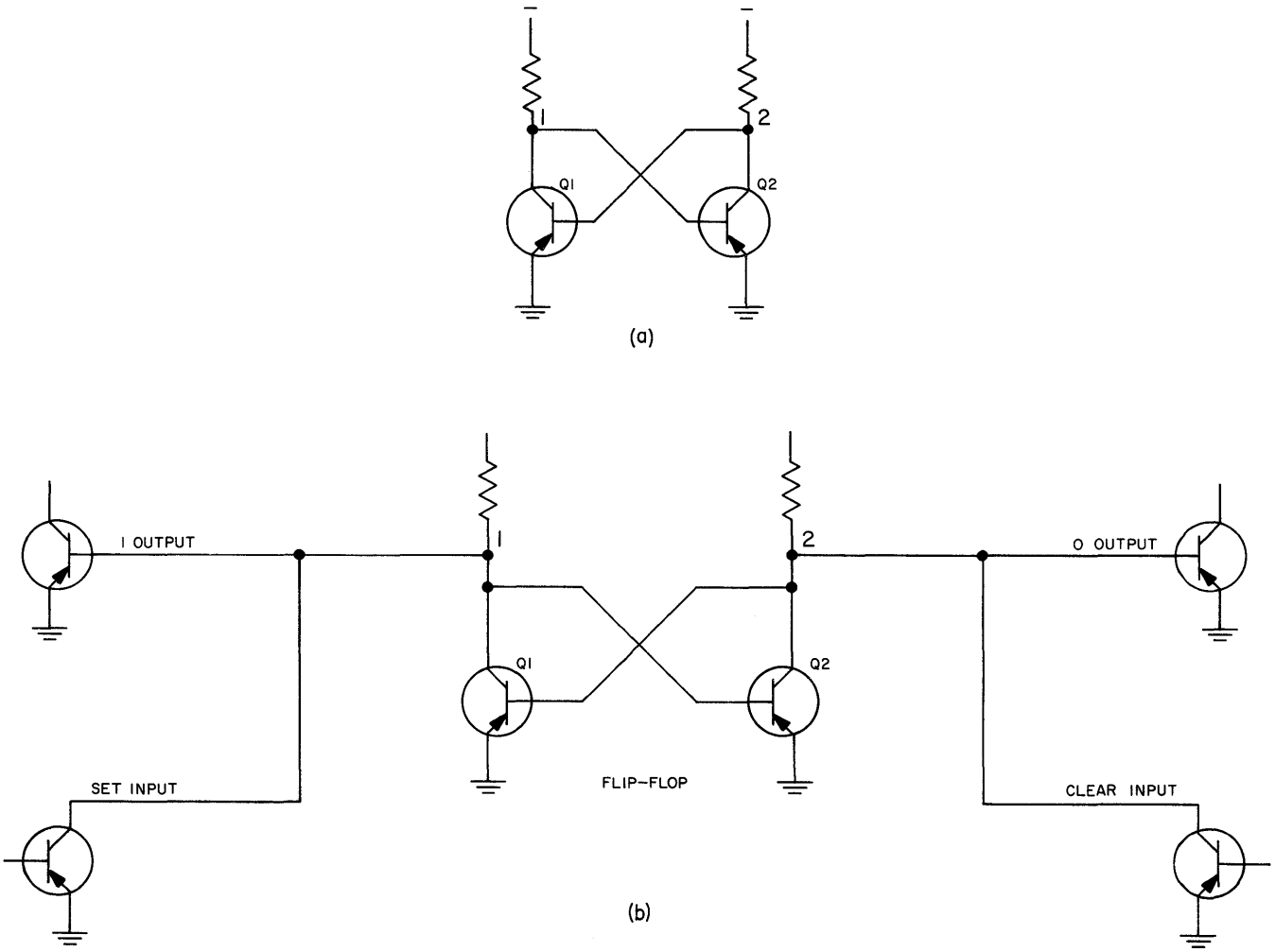


Figure 3–46. Transistor Flip-Flop

The circuit is now in its second stable state, indicating a 1 on the 1 output line. If a second pulse is received on the set input, nothing happens because V1 is already cut off. To clear the circuit and return it to the 0 state, it is necessary to apply a pulse on the clear input. This cuts off V2 and starts V1 conducting again.

A pulse on the complement input changes the state of the circuit regardless of whether it contains 0 or 1. This puts pulses on both inputs simultaneously. The one reaching the grid of the cutoff tube has no effect, but the other causes the switching action to take place by cutting off the conducting tube. The diodes are necessary to prevent pulses on the set or clear inputs from reaching both grids and complementing.

The vacuum tube flip-flop is comparatively simple and stable and can be made quite fast in its switching action.

2.2.1.3 Transistor Flip-Flops

Transistors can be used to replace vacuum tubes in multivibrator circuits similar to that shown in figure 3–45, resulting in savings in power and space. As in the logic circuits, however, they may also be used in the form of the grounded-emitter switch. The resulting direct-coupled transistor flip-flop appears at (a) of figure 3–46. Here, again, either transistor conducts if its base is driven negative. When a transistor is turned on, its collector swings in a positive direction almost to ground potential. If Q2 is turned on, for example, node 2 is almost at ground, and this relatively positive voltage on the base of Q1 keeps Q1 shut off. At the same time, node 1 is about 3 volts negative, supplying the currents to keep Q2 conducting. This can be called the 0 state of the circuit, Q1 off and Q2 on. To switch the flip-flop to the opposite state, it is necessary to drive more positive the base of the transistor that is turned on, Q2. As Q2 shuts off, its collector goes negative, turning on Q1. This causes node 1 to swing up toward ground, keeping Q2 shut off. Now, with Q1 on and Q2 off, the flip-flop is in the second of its stable states, the 1 state. To return it to the 0 state

again, the base of Q1 must be driven up toward ground.

The manner in which the transistor flip-flop is used in direct-coupled circuitry is shown in skeleton form at (b) of figure 3–46. The inputs, typically, are supplied by transistors whose collectors are connected to the flip-flop nodes, while the outputs drive the bases of one or more transistors associated with other nodes. Again, it may be assumed that the flip-flop is in the 0 state, with Q1 off and Q2 on. The potential of the 0 output line is near ground; that of the 1 output is about — 3 volts. To switch the flip-flop to the 1 state, the transistor connected to the set input line is turned on. Its collector (and node 1) swings up toward ground, shutting off Q2 by applying a positive-going voltage to its base. Node 2 returns to about —3 volts when Q2 is turned off, switching on Q1 and holding node 1 almost at ground potential. The flip-flop is now in its 1 state, Q1 on and Q2 off. The 1 output is near ground and the 0 output is at —3 volts. Even if the transistor (or logic circuit) supplying the set input line is turned off, the circuit remains in the 1 state. Notice that this transistor can be turned on again without any effect, since Q2 is already turned off.) To clear the flip-flop, or return it to the 0 state, Q1 must be turned off by driving its base more positive. This is done, of course, by turning on the transistor in the clear input line.

This direct-coupled transistor flip-flop provides an extremely small, reliable storage circuit.

2.2.1.4 Dynamic Flip-Flops

The so-called *dynamic flip-flop* is not really a flip-flop at all, but it serves the same purpose — that of storage — in circuitry that uses pulse-type signals exclusively. When set to the 1 state, the dynamic flip-flop, puts out a continuous stream of pulses, one each bit-time, until it is reset, or cleared, to the 0 state again. In the 0 state, it produces nothing. Thus, it somewhat resembles a pulse generator that can be turned on or off, as required, and is sometimes referred to as a pulse generator (PG) or circulating memory (MEM) circuit. The flip-flop is simple, consisting of an inhibit circuit with OR inputs plus a delay circuit located in a loop that feeds the INH output back to one of the inputs, as shown in figure 3–47. The total delay in the loop — that of the delay circuit plus that of the INH — is made to equal one bit-time. When a pulse is applied to the set input (and there is no-pulse on the clear input at that instant), the INH circuit produces a pulse on the output line. This pulse is fed back through the delay circuit to the other OR input. Since the total delay around the loop equals one bit-time, a second output pulse appears exactly one bit-time after the first. The pulse continues circulating in the loop (and producing 1's at the output) until a clear pulse is applied

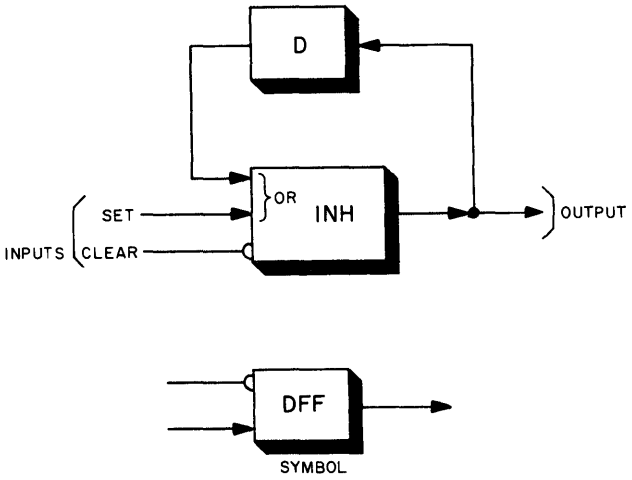


Figure 3–47. Dynamic Flip-Flop

to inhibit it, whereupon the dynamic flip-flop returns to the 0 state and puts out nothing.

Diodes or other suitable switching devices can be used in the INH circuit, but the circulating pulse will be attenuated and eventually drop below limits unless some sort of amplifying and reshaping circuit is included in the loop.

The dynamic flip-flop is used to store a 1 bit for longer periods than is practicable with delay circuits, which must be lined up in series usually with amplification between) to obtain longer delay times. If, for example, a certain pair of pulses must fire an AND circuit, but one pulse appears 42 bit-times later than the other, the use of delay circuits is not practical. Since it is difficult to achieve a delay of more than about 5 bit-times (5D) in a single circuit, a string of at least 9 delay circuits and several amplifier-reshapers would be required to hold up or store one pulse until the other showed up.

The problem may be solved instead, as shown in figure 3–48, by feeding the first pulse to the set input of a dynamic flip-flop which has its output connected to one AND input. The other pulse is fed directly to the other AND input. Now, when the first pulse appears,

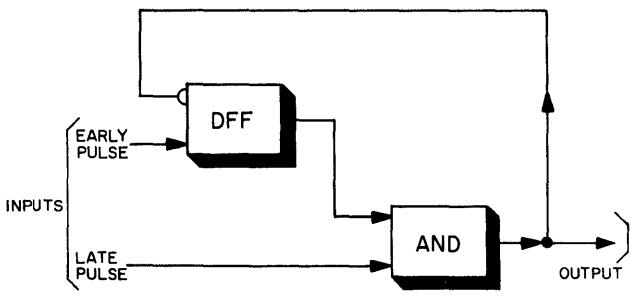


Figure 3–48. Use of Dynamic Flip-Flop

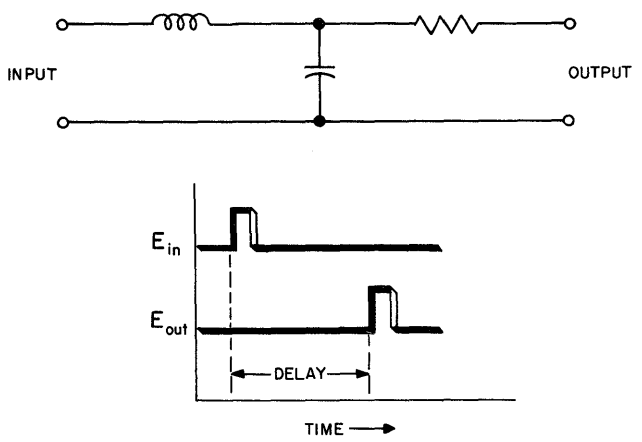


Figure 3–49. Basic Delay Line Section

it sets the flip-flop to putting out 1's. These have no effect until 42 bit-times later, when the other pulse comes along. Since this pulse appears at the same instant as the 42nd pulse from the flip-flop (ignoring fractional delays), the AND is fired. Once this is accomplished, another pulse can be sent to clear the flip-flop; in fact, the output pulse from the AND circuit could easily be fed back for this purpose, as shown. It is possible, of course, to drive the inputs of a number of other circuits from the output of a single dynamic flip-flop.

2.2.2 Delay Circuits

The need for delay circuits to compensate for small differences in the arrival times of pulses has been brought out in connection with other matters, and the manner of using the delay circuits has been shown. All that remains is to show the makeup of these temporary storage circuits.

One basic delay circuit (fig. 3–35), suitable for briefly delaying the output pulse from a magnetic core, has been shown. Most delay circuits – or *delay lines,* as they are commonly called – are patterned on the arrangement shown in figure 3–49, which is electrically similar to a long piece of transmission line.

A pulse travelling through a 1-mile length of transmission line might require 5 usec to complete its trip. Obviously, it is not practical to coil up a mile or two of transmssion line between two circuits in a computer, but the artificial transmission line of figure 3–49 offers similar characteristics lumped conveniently in the form of coils, capacitors, and resistors. A single section such as this offers only a very short delay, but by using special techniques of coil construction and connecting several line sections in series, delays of up to 4 or 5 usec can be obtained.

2.2.3 Word-Length Registers

A register, as mentioned earlier, is a group of storage devices or circuits used for storing a complete word. Since information is usually moved about and operated upon in the computer word-by-word, rather than bit-by-bit, a number of registers will be needed. (Don't understand this word-by-word idea; the circuits have to handle each individual bit, but the computer is so arranged that groups of circuits usually handle a word at a time in response to an instruction or a command.) Since the two principal methods of information-transmission are parallel and serial, some registers are designed to take words in parallel form, some to take serial words. A third type, sometimes useful, is the serial-parallel register, which accepts a number (word) in parallel form and feeds it out in serial form, or vice versa.

Registers have uses aside from simple word storage and are frequently built for such specific jobs as counting or shifting. Counting properly belongs under the heading of arithmetic and is covered in Chapter 3. The present discussion is restricted to storage and shifting registers.

2.2.3.1 Storage Registers

The simplest registers are those used solely for word storage. One type of parallel flip-flop registers appears at (a) of figure 3–50. This is nothing more than a set of flip-flops with no connections between them, one for each bit in the word. Since five stages are shown, this register could handle only 5-bit words, each flip-flop representing a different bit position.

There are two methods of writing words into this register. The first, and least likely to be used, is to place a pulse on the set input of each flip-flop that must store a 1 and a pulse on the clear input of each flip-flop that must store a 0. (This is essentially double line transfer.) Thus, to store the word 11001, pulses would appear on the following inputs, left to right in the illustration:

2^4	2^3	2^2	2^1	2^0	(Bit Position)
1	1	0	0	1	(Input Word)
Set	Set	Clear	Clear	Set	(Inputs Pulsed)

The drawback to this method lies in the fact that for each bit position containing a 0, a pulse must be generated and switched to the clear input line. This, of course, means added circuitry. The second, and easier, method begins by placing a pulse on each clear input before the word to be stored arrives. This clears all the flip-flops in the register to the 0 state, wiping out any word that may have been stored previously. Then the word to be stored is applied in parallel form to the set inputs. (This is single line transfer.) The pulse in each bit position where there is a 1 sets the corresponding flip-flop to 1. No pulses appear in the bit positions where there are 0's, so the flip-flops in these positions remain at 0, and the correct word is stored.

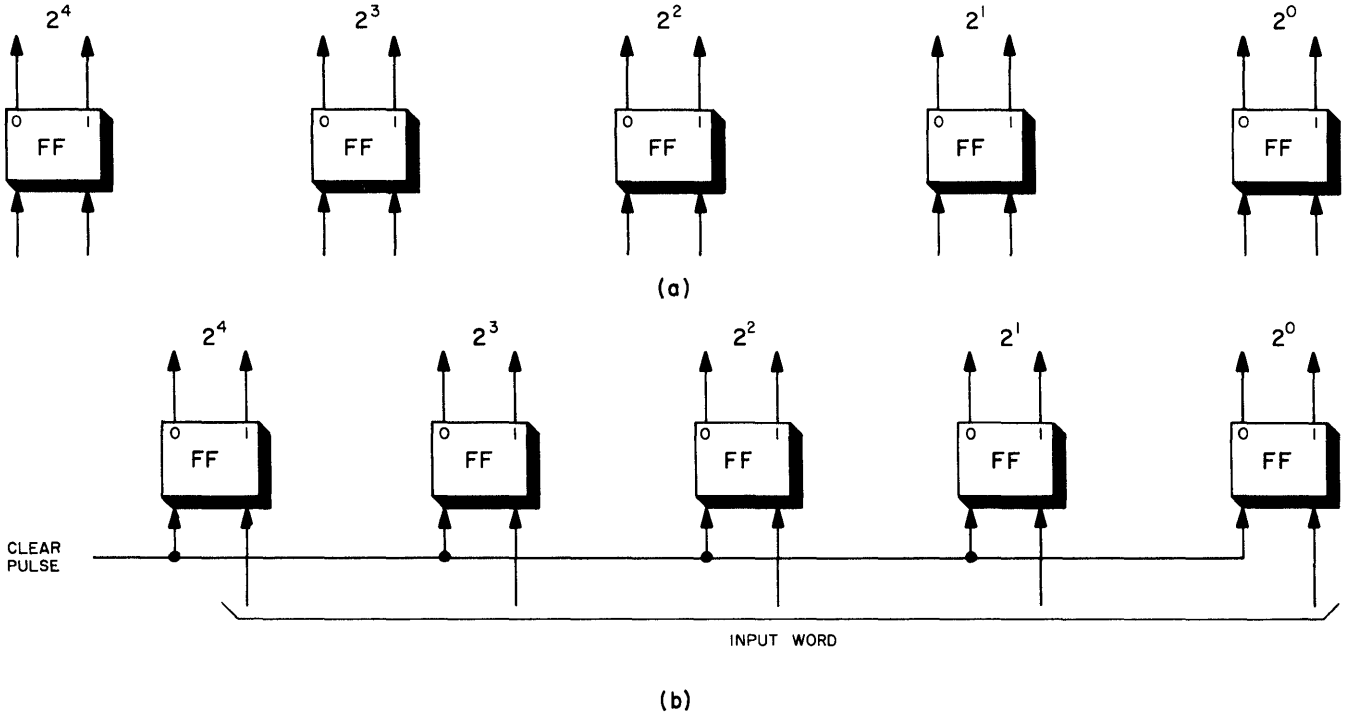


Figure 3—50. Parallel Flip-Flop Storage Register

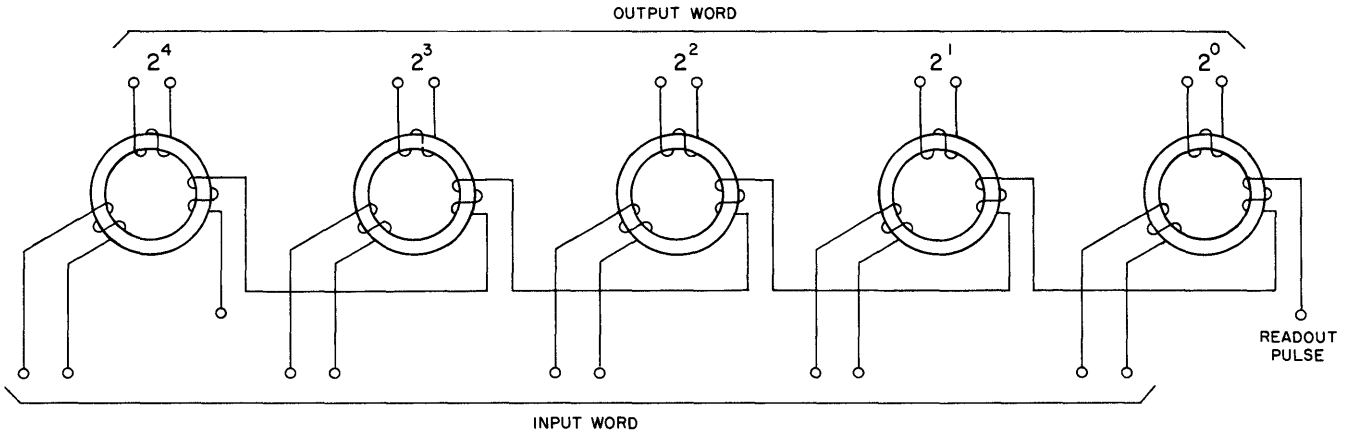


Figure 3—51. Parallel Core Register

Instead of generating a separate pulse to clear each flip-flop before storing a new word in the register, the practical thing to do is use a single pulse, as shown at (b) of figure 3—50. By connecting all the clear inputs in parallel, a single pulse can be made to clear the whole register just before the new word to be stored is due to arrive. A parallel storage register of magnetic cores can be built up in a manner similar to this. Such a register appears in figure 3—51. Each bit of the input word is applied to the input winding of a separate core. The readout windings, connected either in series or in parallel, serve a dual purpose. When a readout pulse is applied, it not only reads out (in parallel form) all the bits stored in the individual cores, but it also clears the register by resetting all cores to 0. The register is then ready to store a new word. When cores are used for parallel word storage, like this, the readout pulse does not appear each bit-time but is more in the nature of a control pulse generated only when the stored information is needed.

These are the basic parallel storage registers. What about storage of words in serial form?

This appears rather difficult, at first thought, but, actually, it is quite simple. A clue to one often-used

method lies in the dynamic flip-flop, or circulating memory, shown in figure 3—47. There, a feedback loop with a total delay of one bit-time keeps a single pulse circulating until it is desired to stop it. Why not extend this idea and keep a whole word circulating until the time comes to take it out of the loop? All that is needed to do this is a total delay in the loop equal to the number of bits in the word to be stored, plus the means of switching the words into and out of the loop, when necessary. The arrangement is shown in figure 3—52. The loop itself consists of the OR circuit and the long delay line (which must include an amplification and reshaping circuit). If 5-bit serial words, for example, are to be stored, the total delay here must be 5 bit-times. This might be divided as 1/4D in the OR and 4-3/4 D in the delay line. The two AND circuits are used as gates to switch the words into and out of storage (gate tubes could be used just as well).

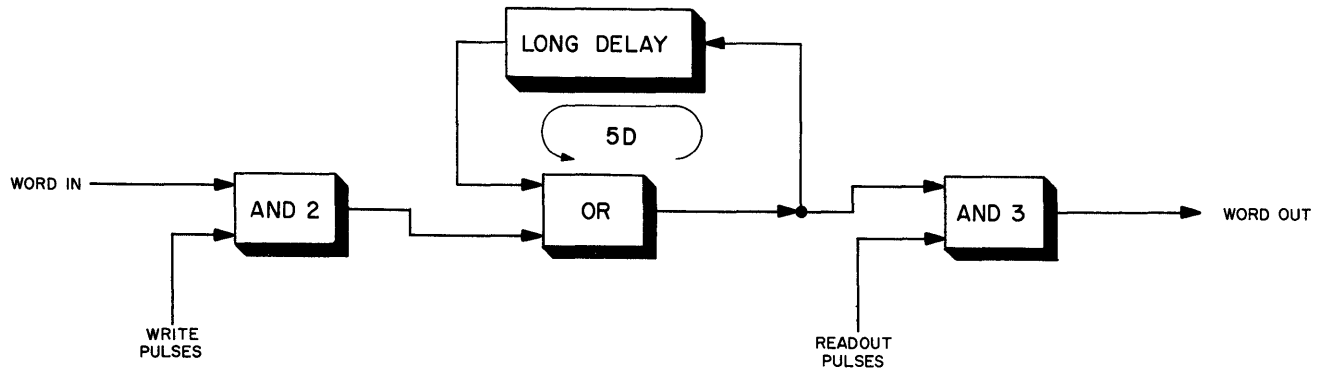
The word to be stored arrives as a train of five bits at the input of AND 2. To gate it into the storage loop, a train of five control pulses (often called write pulses) is generated and timed so that the first write pulse reaches the AND at the same instant as the first (least significant) bit of the input word. If this bit is a pulse, or 1, the AND fires and puts a 1 into the loop; if the bit is a 0, the AND cannot fire so a 0 is put into the loop. In this manner, as shown in the small chart on figure 3—52, the five write pulses gate the five bits of the word through AND 2.

Once through the OR circuit, the first bit of the stored word cannot get through AND 3 because no readout pulses are yet applied to this circuit, which is the output gate. Therefore, the bit follows the feedback path and enters the delay line. The second bit of the word enters the delay line one bit-time later, and the others follow in sequence. The first bit emerges from the delay line 4-3/4 bit-times after it entered. It requires another 1/4 bit-time to pass through the OR circuit again, so it is back at the beginning of the feedback loop exactly 5 bit-times after it was last there. The other four bits of the words are spaced at 1-bit intervals behind it. By thus providing a closed loop with a delay exactly equal in time to the length of the word, it is apparent that the word will continue to circulate and, therefore, will be stored as long as desired. To take the word out of storage, it is only necessary to wait until the first or least significant bit emerges from the OR circuit, then apply the first of a train of five readout pulses to the second input of AND 3. The readout pulses then gate the stored word out of the loop in the same fashion as the write pulses gate it in.

Notice, however, that reading the word out of the circulating register does not clear the latter, since a pulse at the input of AND 3 is also sent back over the feedback path whether it is gated through the AND or not. So the word still remains in the loop, even after it has been read out to other circuits. This, incidentally, is called *nondestructive readout* because the information is not lost from the storage circuit through the reading process. *Destructive readout* is the type occurring in magnetic cores, where the readout process clears the cores.

To be able to store a new word in the register, the old information must be cleared out; otherwise, for example, a pulse (1) in the old word that was left in the register might take the place of a blank (0) in the new word being fed in. The register can be cleared easily if an inhibit circuit is used in place of the OR circuit. Simply placing a string of five clearing pulses on the inhibit input stops each of the circulating pulses, and the loop then contains 0's.

The circulating register is practical and widely



WORD IN	1	1	0	0	1
WRITE PULSES	1	1	1	1	1
AND OUTPUT	1	1	0	0	1

Figure 3—52. Circulating Register for Serial Words

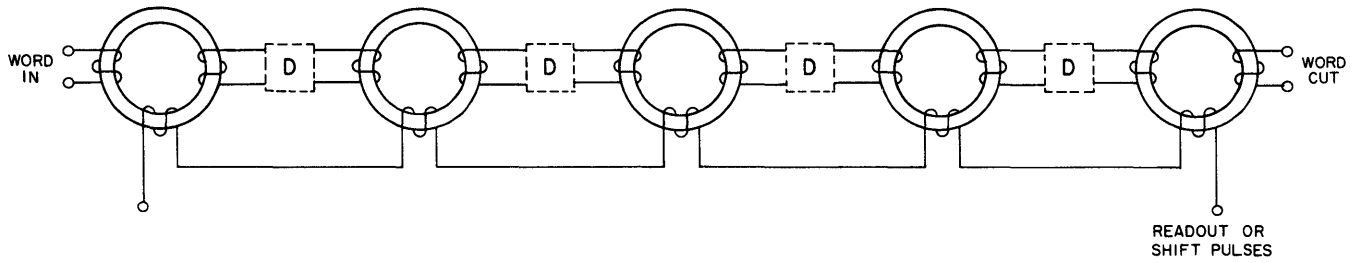


Figure 3–53. Core Register for Serial Words

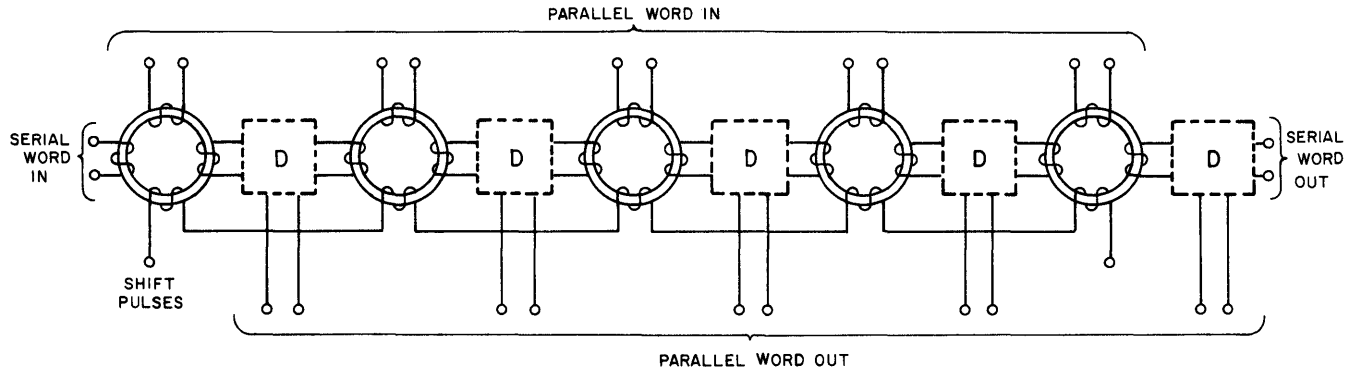


Figure 3–54. Core Shifting Register

used in serial mode computers. Another useful type of register for serial words can be made with magnetic cores, as shown in figure 3–53. To follow the operation of this register, consider the effect of applying a single pulse (1) to the input coil of the core at the left and then applying a series of readout pulses, one each bit-time. The input pulse sets the first core to 1. The first readout pulse immediately resets this core to 0, producing an output pulse which is applied to the input of the next core. The delay circuit between cores (which includes the necessary diode to control polarity) prevents the pulse from setting the second core to 1 until the readout pulse has died away. So, the application of one readout pulse shifts the 1 from the first core to the second. The next readout pulse resets the second core to 0 and shifts the 1 to the third core. This effect continues with the third and fourth readout pulses, shifting the 1 finally to the core on the right. If the readout pulses (which may in this use be called shift pulses) are stopped at this point, the 1 remains stored in the right-hand core. With a complete, 5-bit word fed into the register, instead of a single bit, the same shifting action takes place upon application of the readout or shift pulses. The shift pulses are applied at intervals of one bit-time so the first bit that enters the left-hand core is shifted out in time to clear it for the second bit of the input word, etc. When it is desired to read the word out of the core register, all that is necessary is to apply a string of five shift pulses, at 1-bit intervals. This shifts the bits of the word to the output line in the proper relationship. The principles of *shifting* in core registers can be seen clearly in this example. Shifting is simply a matter of moving all the bits of a word in step, one or more places to the left or right. The register of figure 3–53 would seem to qualify as a shifting register, but actually does not. It performs the shifting operation only as a means of getting serial words into and out of it; its function is simple storage.

2.2.3.2 Shifting Registers

A shifting register is built with the intention of shifting any numbers stored in it for a purpose other than that of ordinary storage. The purpose may be to convert words from serial to parallel form, from parallel to serial, or it may be to multiply or divide the numbers by some power of 2. (Remember from Part 2 that shifting a binary number one place to the left multiplies it by the radix, 2; one place to the right divides it by 2, etc.)

One means of changing the register of figure 3–53 to a true shifting register is to add a second input coil to each core, as shown in figure 3–54. If a word in parallel form is applied to these inputs, the cores are set according to the pattern of 1's in the word. Then,

applying a string of five shift pulses shifts the bits of the word to the output line in serial form.

By taking outputs in parallel off the capacitors in the delay networks between cores (plus that in the output line), a serial word can be written into the register and then taken out in parallel form upon the application of a single readout pulse. With parallel outputs available, the use of shifting for arithmetic purposes can easily be shown. The serial input and output can be ignored for the moment (consider them disconnected). A word is written into the regiser on the parallel inputs and stored momentarily. The word might be, for example:

0.1110

(which equals decimal 14/16).

If a single shift pulse is now applied, each bit of the word is shifted one core to the right. The bit in the right-hand core, however, has no place to go, with the serial output line disconnected, and thus is lost. The left-hand core shifts the bit it contained to the core on its right, but there is no incoming bit to replace it, so this core is reset to 0 by the shift pulse and remains in the 0 state. Therefore, the register now contains:

0.0111

(which equals decimal 7/16).

The shift of one place to the right has thus divided the number in the register by 2. Notice that during the shift the original number appeared on the parallel output lines taken from the delay networks. The core switching can be so arranged, however, that these outputs can be ignored at this time. They will be used only when it is desired to take the shifted number out of the register, by applying one more readout or shift pulse. Notice, also, that any remainder left by the division is lost because the right-hand (least significant) bit is lost. If the sample number shown above is shifted once more, the number left in the register is:

0.0011

(which equals 3/16), instead of 0.00111, the exact answer. Since the lost remainder in division by shifting is always smaller than the least significant bit, however, dropping it has only a minor effect on the accuracy of computations.

Multiplication by shifting requires a shift to the left, instead of the right. To accomplish this, the series-

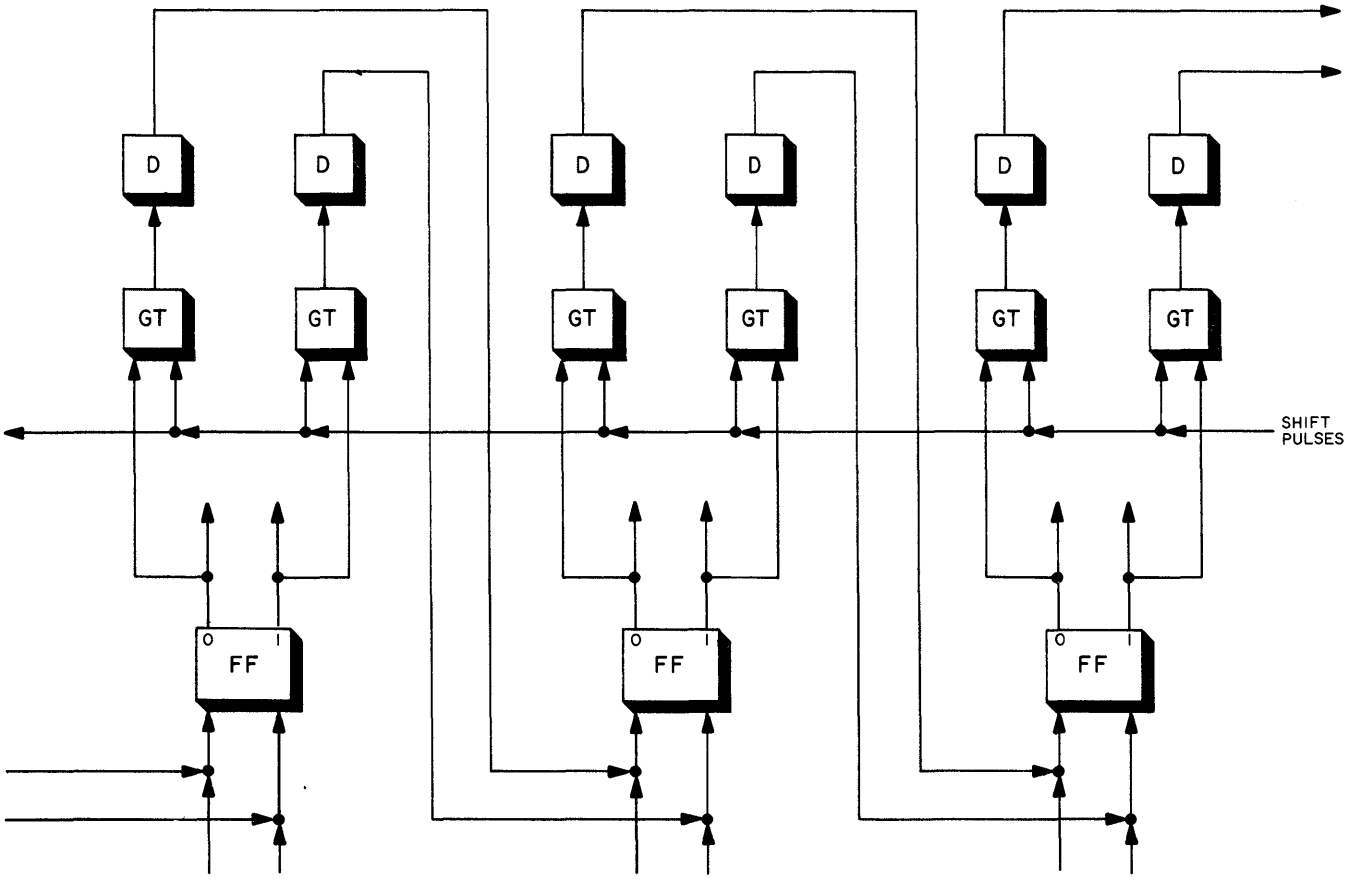


Figure 3–55. Flip-Flop Shifting Register

connected input and output coils of the cores in figure 3–54 must be reversed so that the output of each core goes to the one on its left. Now, the most significant bit of the shifted number is lost, and the least significant bit is replaced by 0.

Although it is perhaps easiest at first to follow the operation of a shifting register in terms of numbers written in and read out in parallel form, the same operations can be performed on serial words. A serial word is written into the register, the input and output lines are opened or switched off, the shifting operation is performed, and then the shifted word is read out.

When shifting registers are made up of flip-flops and logic circuits, there are many possible arrangements. Which one will be used in a given case depends upon the exact operations to be handled, plus such factors as the requirements of associated circuitry.

One of the simplest of such shifting registers appears in figure 3–55. Only three stages are shown, for clarity, and the connections are arranged for a shift to the right. Each flip-flop is set to either 0 or 1 when a word is written into the register, and each flip-flop output then conditions the corresponding gate tube. When the time comes to shift the word in the register, a single pulse is applied on the shift line. This pulse can be passed only by the one gate of each pair that is conditioned by the flip-flop. The pulse, therefore, is gated to the transfer line corresponding to the state of the flip-flop and sets the next flip-flop at the right to that state. The delay circuits are needed to prevent the possibility that the state of a flip-flop might be changed before it had sent an adequate signal to the next one.

By connecting the transfer lines from each flip-flop to the one at its left, instead of the one at its right, the register could be made to shift numbers to the left. In some applications, it is desirable to be able to shift in either direction. A register to handle this can be made by using two sets of gate tubes and transfer lines, one set leading to the right, the other to the left, each set having its own shift pulse line. A pulse on one line would provide a shift to the right; a pulse on the other would result in a shift to the left.

The shifting registers of figures 3–54 and 3–55 both provide what is called *broadside shift*, or *simultaneous shift*; i.e., all bits of the word in the register are shifted at the same instant because the shift pulse hits all conditioned gates at the same time. It is also possible to shift one bit at a time, in rapid succession, a process called *ripple shift*. Starting at the right-hand end of the register if the shift is toward the right (at the left end for a shift left), the output of each flip-flop is gated in turn. As shown in figure 3–56, this is done by switching the shift pulse through the conditioned gate tubes in series instead of applying it to all of them in parallel. When the shift pulse enters, it gates the output of the 2^1 flip-flop to the corresponding input of the 2^0 flip-flop. (The bit originally in the 2^0 flip-flop is lost, as mentioned earlier, in a shift to the right, unless special provisions are made to save it, outside the register.) The gated pulse passes through the OR circuit and is applied as the shift pulse to the next set of gates, shifting the 2^2 bit into the 2^1 flip-flop, etc. In this manner, the bits shift one after the other in a wave or ripple down the register. Although the ripple shift is not as fast as the simultaneous shift, it can often save operation time. In an arithmetic operation involving carries, for example, a ripple shift can be started while carries are still being transmitted from stage to stage down the register. A simultaneous shift, on the other hand, can be made only when all other activity has ceased and any transients have been given time to die out.

Numerous other circuit arrangements can be used in constructing shifting registers but they differ little from the basic types described here.

2.3 ELECTRICAL CONSIDERATIONS AND NONLOGIC CIRCUITS

The information signals that have been illustrated thus far in this part (figs. 3–1, 3–3, and 3–4) have stood tall and square – as, in theory, they should. In actual circuitry, however, as mentioned earlier, there are many factors that act to attenuate or knock down the amplitudes of the signals and to make them round-shouldered instead of square. Pulses are more subject to these difficulties because they appear and die out quickly – usually in less than a microsecond – but levels also lose amplitude and their leading edges become rounded. As a result, the typical pulse taken from a digital computer and displayed on an oscilloscope *might* look almost as good as those in figure 3–4, but is far more likely to have the appearance of that in figure 3–57. In this figure, the actual pulse is seen to have an amplitude considerably smaller than that of the ideal pulse and to last somewhat longer. The actual pulse also rises more slowly and overshoots the baseline on its return to zero (the ideal rise time is instantaneous and therefore impossible).

None of these seeming drawbacks of the actual pulse may be severe enough to make it an unusable one. Although the ideal pulse for a given machine might be, for example, 30 volts in amplitude and 0.1 usec in duration, it is necessary in practice to set limits for the actual pulses. Thus, the circuits in this example might be designed to be definitely triggered by any pulse between 20 and 40 volts in amplitude and 0.08 and 0.12 usec in duration. If the pulse shown in figure 3–57 were between these limits, it would be a good, usable pulse. In some types of computer circuitry, a limit is set on rise time. That is, a pulse must rise to a certain am-

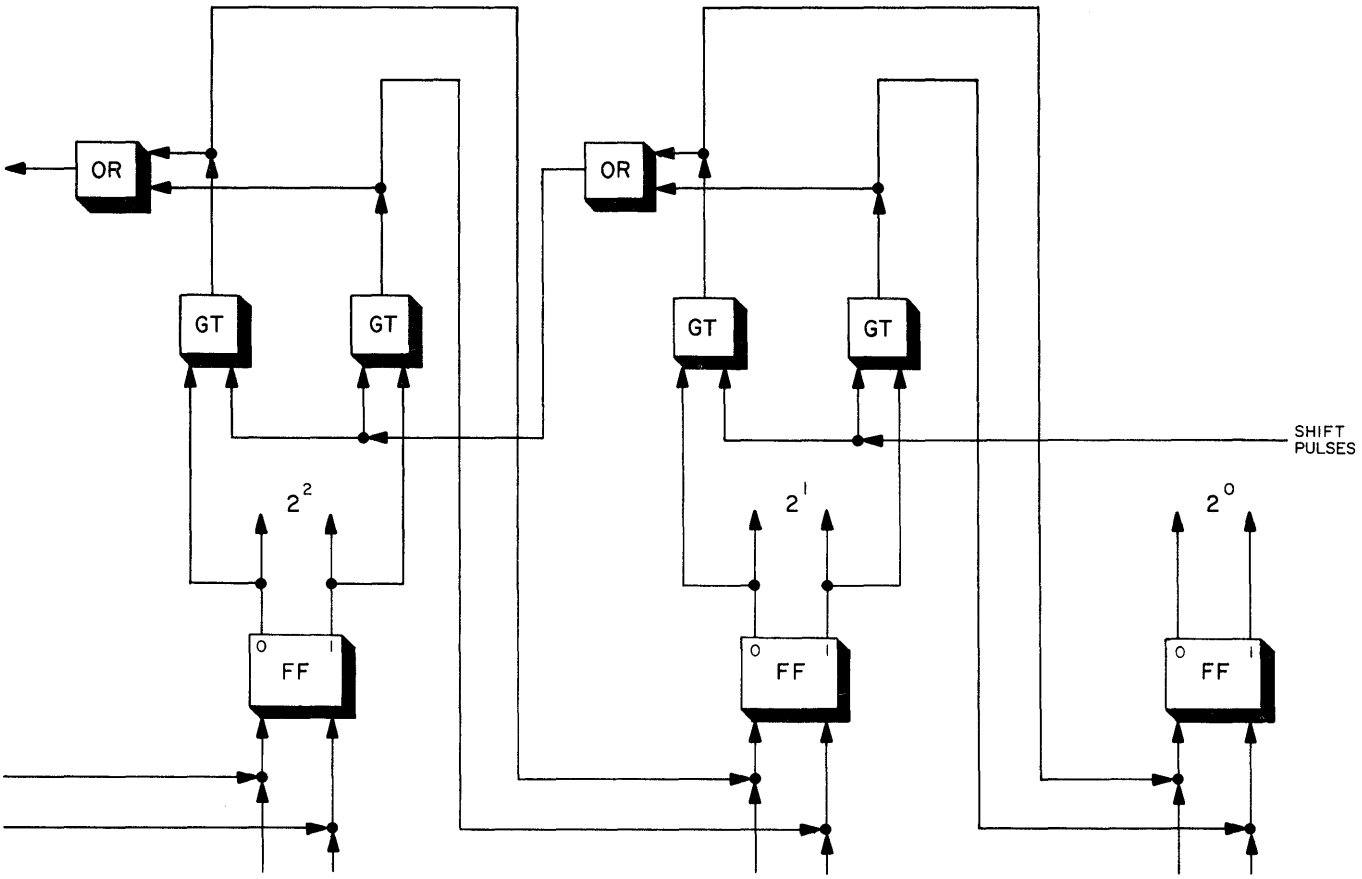


Figure 3—56. Register Using Ripple Shift

plitude within a certain time. There may also be a limit on the amount of overshoot permitted.

Whenever a pulse or a level must travel through a number of circuits or devices without amplification, loss of amplitude must be expected. Such attenuation occurs in diode logic circuitry, for example. Circuits or signal paths with poor high-frequency response lengthen the rise times of pulses or levels, rounding the leading edges. Excessive capacitance in signal lines or circuits causes attenuation, distorts pulses, and may upset timing because of undesired phase shifts. And impedance mismatches are particularly serious and difficult to locate. They may result in misshapen pulses, severe attenuation or complete loss of signals, or phase shifts affecting signal timing. Mismatches of this sort can be caused by improper resistive values terminating coaxial signal lines (setting up reflections or standing waves on the lines); changing values of circuit components, often due to aging; leakage or partial shorts, and other factors. These are some of the electrical or electronic problems encountered in digital computers by both designers and maintenance technicians. The designer must include circuits to amplify and reshape pulses or reset levels wherever it appears possible that the pulses or levels may be forced outside limits. He may also have to provide impedance-matching and power-amplifying circuits to drive loads too large for the ordinary logic circuit to handle. Circuits such as these, included because of electrical necessity but not performing operations necessary to the logic of the computer, are called *nonlogic circuits.* Examples are cathode followers for both impedance-matching and power amplification, pulse amplifiers, level setters, register

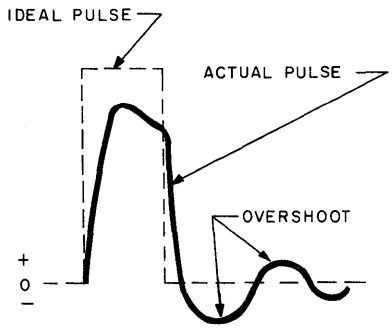


Figure 3—57. Typical Pulse in Computer Circuitry

drivers, etc. Pulse generators may produce an output of one pulse or a series of pulses and may be used either for reshaping or for logic.

Nonlogic circuits of a particular type are usually identical throughout a given computer; therefore, they can be drawn as circuit blocks on the machine schematics, like logic circuits. Since they contribute nothing to the logic operations, they are normally omitted from the simplified or "pure" logic block diagrams used to describe the theory of operation, but must be included, of course, in the complete diagrams of the equipment.

2.4 CIRCUIT PACKAGING

It has been mentioned that digital computer circuits are usually packaged, either individually or in small groups. The circuit connections are made by some sort of plug-in arrangement, so once a trouble has been localized to a given circuit, replacement is a simple matter of pulling out the defective circuit package and plugging in a good one from a supply of spares. The computer is then ready to run again, much sooner than it would have been if the circuit had to be repaired in the machine. Business expense or military necessity makes it important to keep almost every computer (commercial or military) running and solving problems as continuously as possible. Thus, the maintenance time saved by using pluggable packaged circuits is important.

It would not be practicable to attempt to describe all of the many different packaging methods used. Among the principal aims in all designs are making the circuit packages as small as possible while maintaining efficient cooling for reliable operation, making wiring simple and uniform, and simplifying repair (except in types designed to be thrown away if they fail). Printed circuits and miniaturized circuit components are widely used.

In the AN/FSQ-7, -8 computer, for example, circuits are constructed in rectangular metal forms of uniform size, illustrated in figure 3–58. These *pluggable units,* as they are called, are designed to be plugged into rack assemblies, one above the other, like drawers in a bureau. The circuit components are mounted on etched cards, one or more to a circuit, which are then inserted in vertical slots in the pluggable unit. The vacuum tubes are mounted horizontally on the front of the unit, and standard wiring is used to connect the cards and the tube socket pins. When all the units are in place in a stack, conditioned air is blown up from the base and escapes through openings around the tube sockets in each unit, cooling the cards and the tubes.

These pluggable units are fairly sizable, but convenient for one man to handle. By way of contrast, one transistorized digital computer for airborne use has its circuits individually packaged in small plastic cubes, each less than half the size of a cigarette package. The circuit packages are interconnected by printed wiring on the cards upon which groups of circuits are mounted, and the cards, in turn, are plugged into air-conditioned cabinets. The entire computer occupies less space than the average office desk.

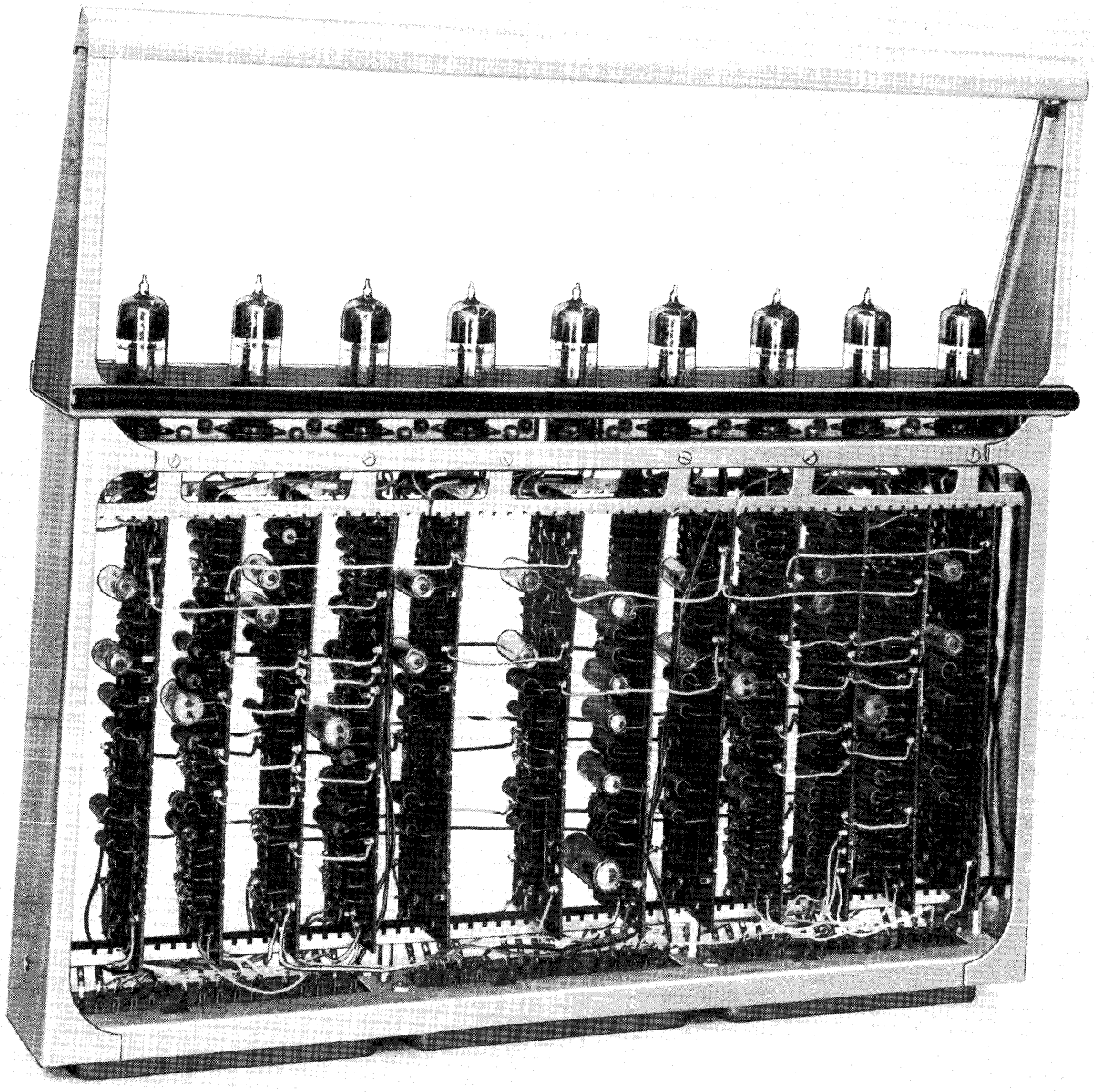


Figure 3–58. Circuits Packaged in Pluggable Unit

CHAPTER 3
ARITHMETIC AND CONTROL

The previous chapter has described the basic types of logic and storage circuits, their operation, and some of the ways in which they can be arranged to perform combinations or sequences of logic operations. The digital computer is built by assembling networks of these basic circuits to perform the operations of arithmetic, to handle the input and output of information, and to control the internal working of the machine.

This chapter will examine some of the networks that can be used to accomplish the functions of arithmetic and internal control in computers working with the binary system. These networks will be shown by means of logic block diagrams because it is the logic of operation, rather than the electrical theory, that is most important to a clear understanding. For this reason, nonlogic circuits will not appear, although it is known that they are often needed for amplification, level setting, etc.

Some types of the basic circuits described in Chapter 1 could be used to construct these networks without change, but others, for electrical reasons, would require modifications of the networks to make them work. This, however, would not change the overall principles of operation. It can be assumed, for most of the networks to be described, that either voltage level or pulse signals, or both, could be used in different parts of the network.

There are usually several different network arrangements that will perform a given arithmetic operation and it is not possible or practical to treat all of them here. What is intended is to give a clear picture of the basic ways in which logic and storage circuits are put together to accomplish the various jobs that must be done in the computer. Remember, therefore, that the approaches to be shown here are not by any means the only ones possible.

3.1 COUNTING

The cyclic nature of the counting process makes it easy to design networks or circuit arrangements that can count input signals. Such circuits, called *counters,* are used for various purposes, such as counting steps in the program as they are executed. In any of these uses, a signal — usually a pulse — is generated each time the event to be counted occurs. The counter then counts these signals. For instance, a signal is generated and sent to the program counter each time a step of the program is completed. In this manner, the counter keeps track of the progress of the program by counting the individual signals.

The cycling of the bits in a binary number can be seen by examining a simple count:

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

Note that the bit in each binary place (order) changes state from 0 to 1 and back to 0 again as the count progresses. Each time the bit in a given place changes back to 0, the bit in the next more significant place (to the left) also changes state.

3.1.1 Binary Counters

This cycling count suggests that a group of flip-flops might be arranged as a counter, using one flip-flop to represent each binary place, as shown in figure 3–59. When a signal is applied to the complement input, a flip-flop switches from one of its stable states to the other. Since each flip-flop is driven by the 0 output of the one in the next lower place, each changes state only when the bit in that place goes from 1 to 0. (The capacitor in each output line represents a differentiating

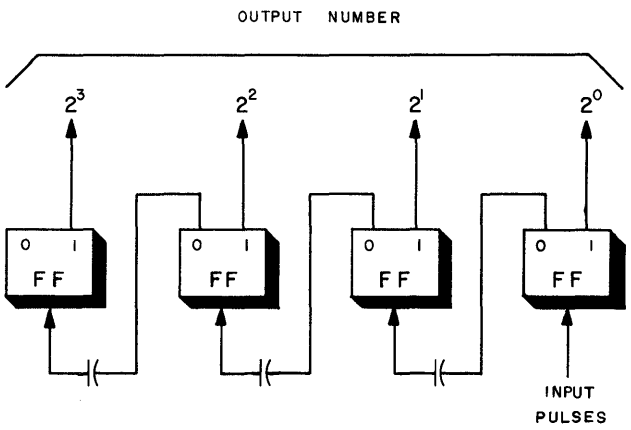


Figure 3—59. Basic Flip-Flop Counter

circuit that changes the voltage level rise into the necessary pulse when the 0 output level goes up.)

The pulses to be counted are fed to the flip-flop in the least significant place. The first pulse (assuming the counter has been cleared to 0000) sets the 2^0 flip-flop to 1, and the counter output is 0001. The second input pulse resets the 2^0 flip-flop to 0 and the differentiated voltage level rise on its 0 output line sets the 2^1 flip-flop to 1, so the output is now 0010. The third input pulse sets the 2^0 flip-flop to 1 a second time, but this has no effect on the 2^1 flip-flop. The counter output is 0011. When the fourth input pulse arrives, the first two flip-flops are reset to 0 and the 0 output from the 2^1 flip-flop sets the 2^2 flip-flop to 1, producing an output from the counter of 0100.

Counting continues in this manner until the 15th input pulse has produced an output of 1111. The 16th pulse then resets the counter to 0000 and it is ready to begin the counting process again. It is possible to clear this counter at any time by applying a pulse to all the flip-flop clear inputs, which can be connected in parallel for this purpose. Notice that the input pulses do not have to appear at regular intervals. This group of flip-flops is storing a number, between input pulses, so it also may be called a *counting register.*

The counter of figure 3—59 has one drawback — the input pulse often has to switch a number of stages in series, so the counter may be slow to respond. This is especially true where a count to 12 or 16 or more places may be required. An arrangement providing faster operation, but requiring more switching equipment, is shown in figure 3—60. In this counting register, the input pulses are applied almost simultaneously to the flip-flops of all places to be switched.

When the counter is cleared to 0000 and the first input pulse arrives, it sets the 2^0 flip-flop to 1 but cannot get through GT 2 because the gate has not been conditioned. (The delay circuit is needed to prevent the flip-flop from changing state and conditioning the gate tube before the pulse ends.) After the first pulse disappears, the counter is at 0001, and the 1 output of the 2^0 flip-flop is conditioning GT 2. The second input pulse passes through GT 2 and sets the 2^1 flip-flop to 1, at the same time resetting the 2^0 flip-flop to 0 and removing the conditioning voltage from GT 2. Now the counter output is 0010 and GT 3 is conditioned but GT 2 is not. The third pulse, therefore, can only set the 2^0 flip-flop to 1 again, making the output of the counter 0011. When the fourth input pulse appears, it now finds both GT 2 and GT 3 conditioned, so it changes the state of the first three flip-flops and the counter output becomes 0100.

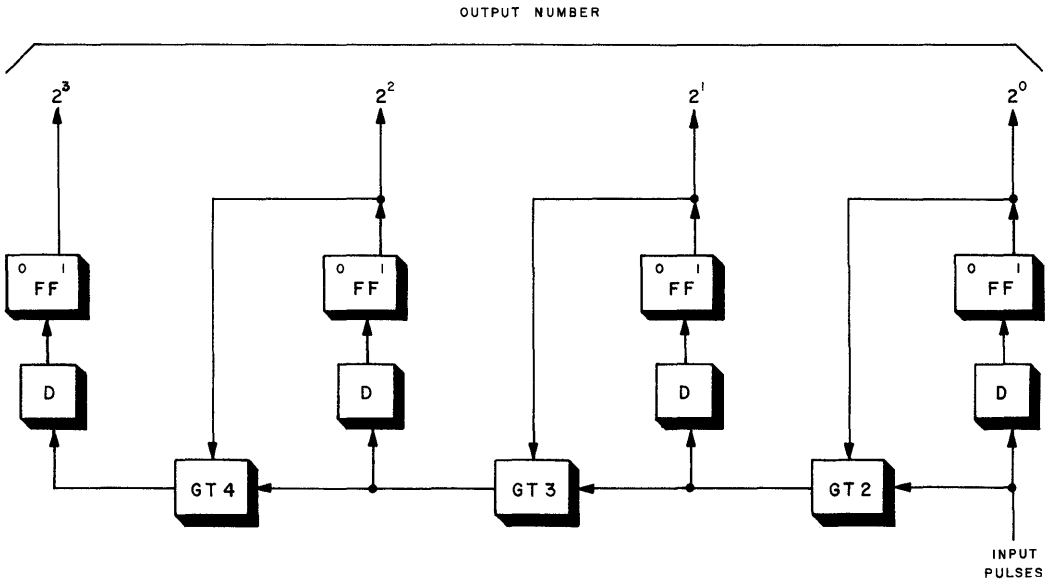


Figure 3—60. Higher Speed Counting Register

In this manner, the count continues to 1111, whereupon the next pulse received resets the counter to 0000. It can, of course, be cleared at any time by pulsing the clear inputs of all flip-flops.

Again, the input pulses do not have to appear at regular intervals. If they do, however, the arrangement can also be used as a *frequency divider*. Consider, for example, an input of 1000 pulses per second (pps). Then, since only one pulse in two gets through GT 2, the output of GT 2 is 500 pps, a frequency division of 2:1. Similarly, the output of GT 3 is 250 pps, or 4:1, that of GT 4 is 125 pps, 8:1, etc. In other words, each additional stage provides an additional division by 2 of the input frequency.

3.1.2 Ring Counters

The two counters described thus far are both *binary counters* – that is, their outputs are parallel binary numbers. Often, however, all that is needed is some definite indication of the progress of the count, such as one output line at a time energized (called a *one-hot* indication).

In such a case as this, it is apparent that only one stage of the counter at a time need be in the 1 state. This state is passed along from one stage to the next on the receipt of each new input pulse to be counted.

Counters of this type are called *ring counters* and they are often in the form of closed rings, with the last stage connected to the first in such fashion that the count automatically starts over. In some cases the ring is not closed and there is some separate means of turning on (to the 1 state) the first stage.

There are a great many possible circuits for ring counters, as there are for the other types. Four stages of one such circuit are shown in figure 3–61.

If FF 2 is on, or in the 1 state, the other stages off, only output 1 is hot, carrying an up level voltage. The next pulse arriving on the input line clears FF 2 but cannot affect the other flip-flops since they already contain 0's. As FF 2 changes stage, the voltage level rise on its 0 output line is differentiated in the circuit represented by the capacitor symbol, yielding a pulse that sets FF 3 to 1. Again, the delay circuit is here to allow the input pulse to die out before FF 3 is turned on.

In this manner, each input pulse turns off the one flip-flop that is on and this change in state is used to turn on the next stage. One output after another becomes hot in turn as input pulses "step" the on stage around the ring. Thus, as input pulses continue to come in, the outputs are:

0001

0010

0100

1000

Etc.

It is common practice to use two or more ring counters together, with the last stage of one supplying an input pulse to the next counter, as shown in figure 3–62. The tandem setup shown here has five stages feeding three – other combinations can be used, of course, and the second counter can feed a third, etc.

In this arrangement, five input pulses must be received by the first counter to produce one input pulse to the second. The 3-stage counter, then, counts by 5's (the number of stages in the first counter), so the complete tandem counter can count to 15, although it has a total of only 8 stages. A 16th input pulse starts the count over again.

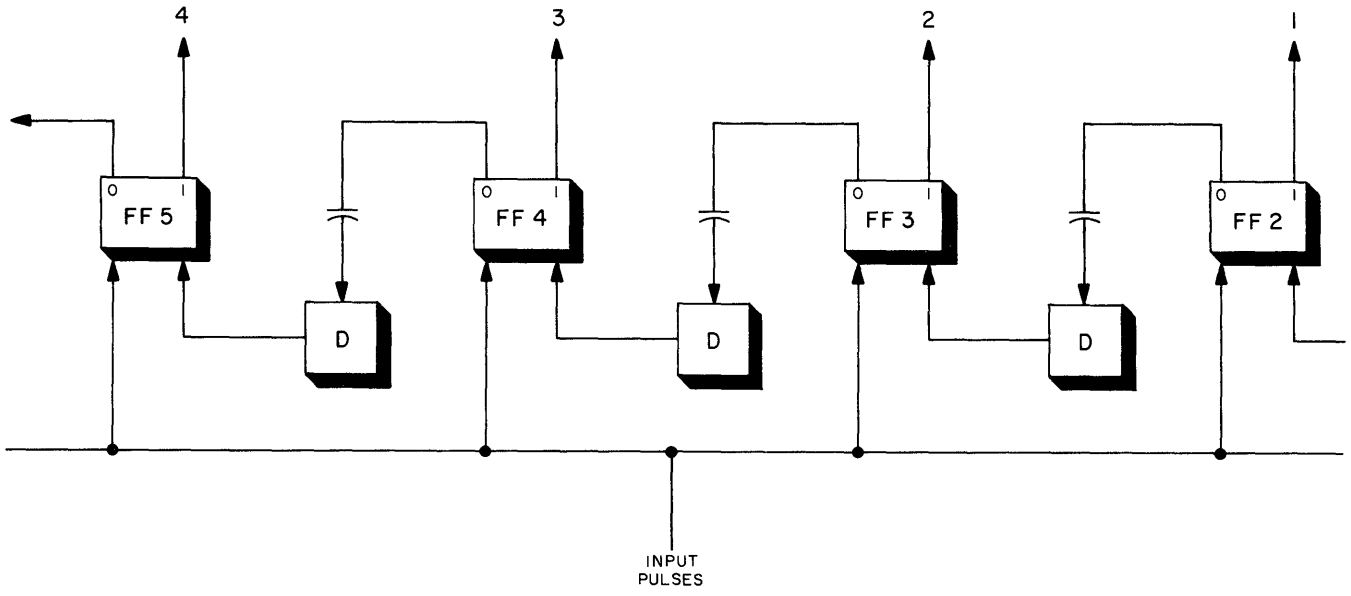


Figure 3–61. Ring Counter Stages

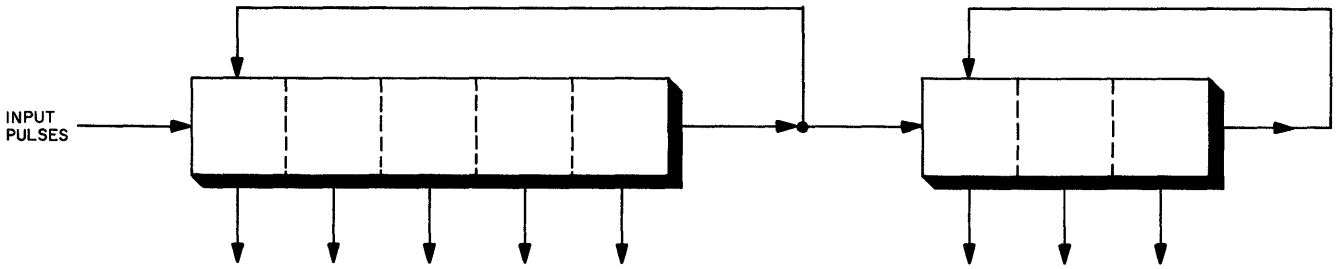


Figure 3–62. Using Ring Counters in Tandem

There is always one hot output from each counter. Assuming that the first pulse received turns on the left-hand stage of each counter, the output indications change as shown in table 3–1. To produce a signal indicating that a certain-numbered input pulse has been received, therefore, it is only necessary to bring together in an AND circuit the two output lines that are made hot by that pulse.

If it is desired to obtain an output after the eighth input pulse, for example, table 3–1 shows that the center output from each counter is hot, so these two lines are combined in an AND. In fact, a three-by-five matrix of AND circuits, similar to that shown in figure 3–40 (but without the OR's), can be used to give a separate output for each of the 15 input pulses on the common line. Thus, outputs can be obtained for any or all of the pulses to be counted.

TABLE 3–1. OUTPUT CHANGES OF TANDEM RING COUNTERS

INPUT PULSE	OUTPUT INDICATIONS	
1	10000	100
2	01000	100
3	00100	100
4	00010	100
5	00001	100
6	10000	010
7	01000	010
8	00100	010
9	00010	010
10	00001	010
11	10000	001
12	01000	001
13	00100	001
14	00010	001
15	00001	001
1	10000	100

3.2 ADDITION

Simple arithmetic addition is nothing more than a short-cut method of counting up from smaller numbers to larger ones. Since the only binary digits are 0 and 1, binary addition is just a matter of counting in columns and properly handling the carries between columns.

Although it is common practice in pencil-and-paper work, decimal or binary, to add a whole column of figures at one time, this has not been found practical in digital computers. Instead, a computer adds the first two numbers, then separately adds the third to the sum of the first two, then the fourth to the sum of the first three, etc., always taking the numbers two at a time.

3.2.1 Adders

The circuit requirements for binary addition can be calculated by examining the rules for addition of any two bits. There are only four possible combinations of two bits:

0	1	0	1	Augend Bit
+0	+0	+1	+1	Addend Bit
0	1	1	0	Sum Bit
(0)	(0)	(0)	(1)	Carry Bit

(The carries will henceforth be placed in parentheses to avoid confusion with the sum bits.)

So what is needed is a logic circuit arrangement that will take any two bits and produce the proper sum output and carry output for each of the input combinations shown above.

The easiest starting method is to consider separately the requirements for the sum bit and the carry bit. If the augend bit is called A and the addend bit B, then the sum bit is 1 when:

(A OR B) AND NOT (A AND B) = 1 (Sum)

The carry bit is (1) when:

(A AND B) = (1) (Carry)

Since the requirement for the carry bit is the same as one term of that for the sum bit, it should be possi-

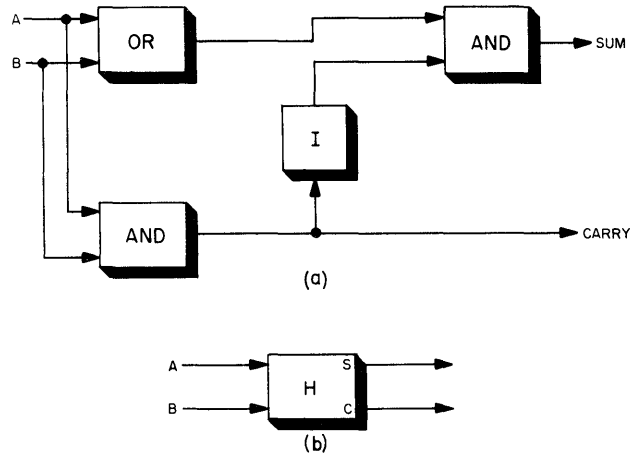


Figure 3–63. Half Adder

ble to use only one AND circuit for (A AND B), taking the carry output directly from it. An inverter and a second AND will take care of the AND NOT (or an inhibit circuit, if pulse transmission is used), and an OR circuit handles the (A OR B). The arrangement is shown at (a) of figure 3–63.

This arrangement for adding two binary bits and producing the proper sum anl carry bits is called a *half adder.* The common circuit symbol appears at (b) of figure 3–63, and it should be remembered that the symbol may represent any of several possible circuit arrangements, as well as the one at (a).

In operation, if only one input, A or B, is 1, the output 1 from the OR circuit is applied to one input of the following AND. The first AND circuit cannot produce an output with only one input present, so the carry is (0). This carry is inverted to 1 and applied to the other input of the second AND circuit, resulting in a sum bit of 1. If A and B are both 1's, the input AND circuit produces a carry of (1), which is inverted to 0 and applied to one input of the second AND circuit. With a 0 on one input, this AND cannot provide an output, even though a 1 comes from the OR circuit, so the sum bit is 0. Finally, if A and B are 0's, both the sum and carry bits are 0's.

An arrangement of this type is called a half adder because it cannot by itself add two binary numbers, although it can add two bits. The reason can be seen by performing a sample binary addition, showing the carries from one column to the next:

	(1)	(1)	(1)		Column Carry
0.	0	1	1	1	Augend
0.	0	0	1	1	Addend
0.	1	0	1	0	Sum

Although only two numbers are being added, some columns require that three bits be summed because of the carries from the previous columns. A single half adder cannot do this, but some means must be found.

First of all, it is necessary to establish a fixed and definite pattern that will cover all possible addition problems. The example above used 5-bit words, but in an actual computer the words might be of any length and carries could occur in any column. (When the 1's complement system is used to represent negative numbers, remember that the end carry, if it occurs, must be taken around and entered into the least significant column.) To make the addition pattern regular, the practical thing to do is consider that there will always be a carry into each column, but the carry may be (1) or (0). Thus, the example above becomes:

(0)	(1)	(1)	(1)	(0)	Column Carry
0.	0	1	1	1	Augend
0.	0	0	1	1	Addend
0.	1	0	1	0	Sum

The carry in the least significant column is always (0) at the beginning of an addition, although an end carry may have to be entered there later. Unlike the column carries, the end carry (not shown) is usually ignored unless it is a (1). Therefore, it is commonly said that there is no end carry unless it is a (1).

Now, there are three bits to be added in every column. One half adder will add two of them, but a second half adder must be used to add the third bit to the sum of the first two. And, since there will be a carry of (0) or (1) from each half adder, some additional logic circuitry is needed to handle the carry to the next column.

Probably the most common method, shown at (a) of figure 3–64, is to add the augend and addend bits (A and B) in the first half adder, producing a temporary sum bit and a first carry.

1	Augend Bit
1	Addend Bit
0	Temporary Sum Bit
(1)	First Carry

Then, in the second half adder, the carry from the column to the right is added to the temporary sum bit, producing the column sum bit and a second carry.

0	Temporary Sum Bit
(1)	Column Carry
1	Sum Bit
(0)	Second Carry

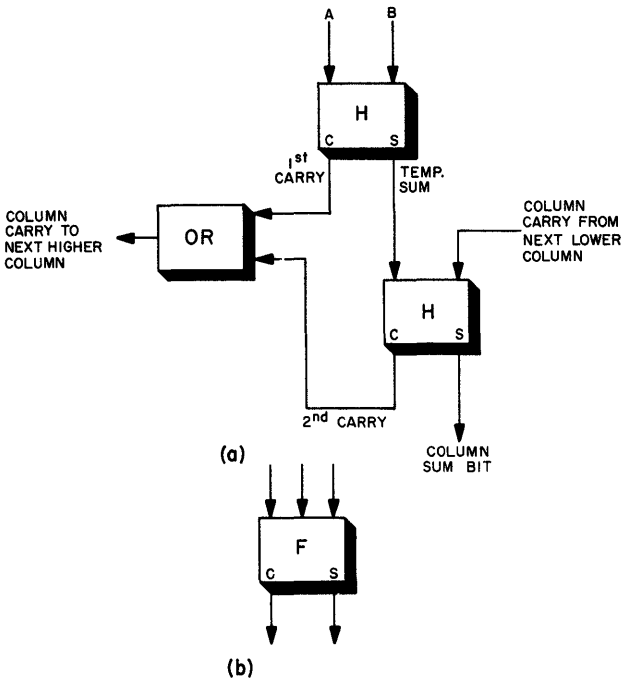


Figure 3–64. Full Adder

If either the first or the second carry is (1), this must be the column carry, since any carry resulting from addition in a given column must be sent to the next. Notice, however, that both carries cannot be 1's— if the first is a (1), as in the example above, the second can only be (0). So all that is necessary to produce the column carry is to feed both first and second carries to an OR circuit, as shown at (a) in figure 3–64.

The arrangement in this figure 3–64 is a *full adder,* so called because it can properly handle the carries between columns. It is only one of a number of possible arrangements. The usual symbol is shown at (b) of figure 3–64.

Consider now the previously-given addition of two binary numbers, 0.0111 + 0.0011, as it would be performed by five of these full adders in parallel, shown in figure 3–65.

Notice that the column carries must travel (or "propagate") through all the stages in sequence, from the least to the most significant. That is, the addition in the second column from the right cannot be completed until the carry from the first column is received, the addition in the third column depends upon the carry from the second, etc. The time required for this *carry propagation* definitely slows down the addition process, especially if the computer word contains many bits. A number of methods have been devised to speed up carry propagation, but it remains a problem.

The complete picture of the addition problem represented in figure 3–65 can be seen by examining what happens inside the adders.

First, the augend and addend bits are brought from registers where they have been stored into the first half adders.

0.	0	1	1	1	Augend Bits
0.	0	0	1	1	Addend Bits
0.	0	1	0	0	Temporary Sum Bits
(0)	(0)	(0)	(1)	(1)	First Carries

Next, beginning in the least significant column, the temporary sum bit is added to the column carry from the next lower column. Remember that this action takes place in sequence from right to left, although it appears below as if simultaneous. In the least significant column, the carry is automatically made (0). The second half adders produce the sum bits and the second carries, which are combined with the first carries in the OR circuits.

0	0	1	0	0	Temporary Sum Bits
(0)	(1)	(1)	(1)	(0)	Column Carries
0	1	0	1	0	Sum Bits
(0)	(0)	(1)	(0)	(0)	Second Carries

Thus, the sum is 0.1010, taking the adder outputs in parallel. Of course the binary points do not appear in the machine words.

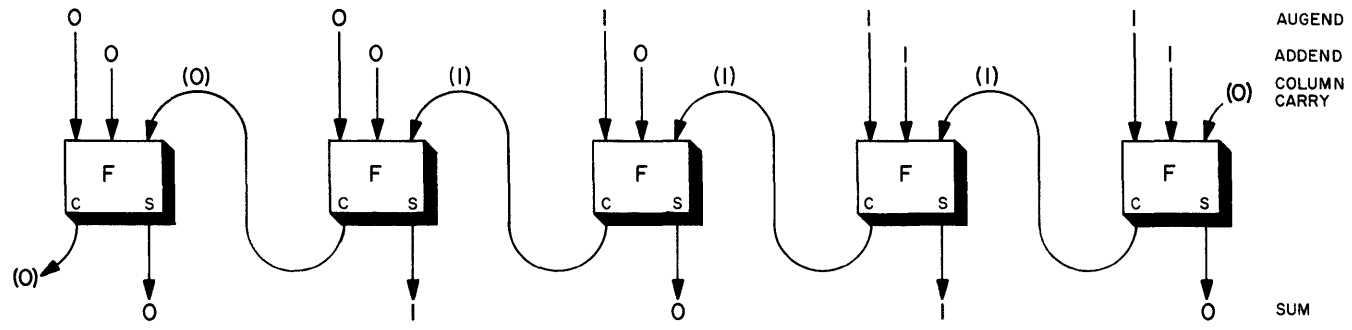


Figure 3–65. Parallel Adders

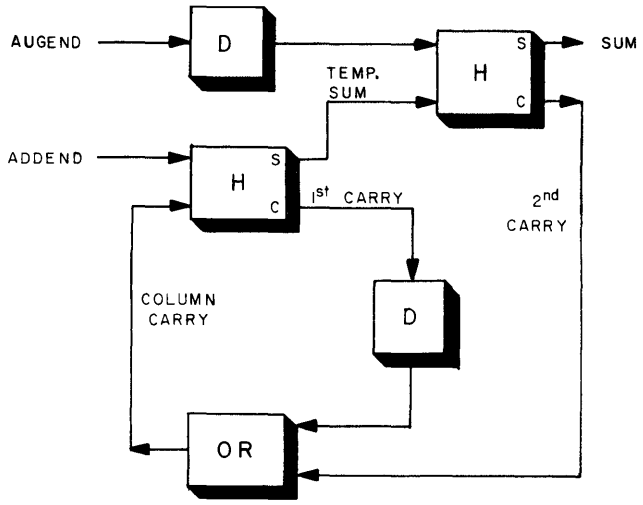


Figure 3–66. Full Adder for Serial Operation

In this manner a digital computer can perform addition of numbers in parallel form. What about the addition of numbers transmitted serially?

The principles underlying the two methods are the same, so a switch to addition of serial binary numbers is a matter of adapting the circuitry to the requirements of serial transmission. Half adders and full adders will still be necessary. Assuming that the numbers are trains of pulses and blanks, transmitted least significant bit first, notice that the columns to be added are separated in time instead of space ("space" meaning the separate wires of the parallel system). This means that only one full adder is needed to add two serial binary numbers of any length, instead of one adder per column.

The basic half adder of figure 3–63 is satisfactory for serial use, except that the inverter and second AND circuit would probably be replaced by an inhibit circuit, which is better suited for pulse work.

The full adder arrangement is changed to that shown in figure 3–66. Although the adder of figure 3–64 could be made to work for serial numbers, the handling of carries would become involved with longer delays.

The operation of the serial full adder is almost identical to that of a single parallel full adder, except that the addend bit and the column carry are added first, then the augend bit is added to the temporary sum bit that results. Assuming that each successive pair of bits of the augend and addend arrive at the same instant, the bits of the augend must be delayed to arrive at the second half adder at the same time as the corresponding temporary sum bits. In other words, the augend delay must be equal to that in the first half adder. Correspondingly, the first carry must be delayed to arrive at the OR circuit at the same instant as the second carry.

The timing of the entire arrangement must be such that the column carry gets back to the input of the first half adder just as the next addend bit arrives.

With these details arranged satisfactorily, the serial adder accepts any two binary numbers, adds them, and feeds out the sum in serial form, least significant bit first. The total delay in the adder is usually one bit-time, so the time required for addition of computer words containing N bits is N + 1 bit-times. Two 5-bit words, for example, are added in 6 bit-times.

Bit-time		6	5	4	3	2	1
Inputs	Augend		0	0	1	1	1
	Addend		0	0	0	1	1
Output	Sum	0	1	0	1	0	

The parallel method of addition is normally much faster than this, even allowing for carry propagation, but on the other hand the serial method requires much less circuitry.

3.2.2 Accumulators

The adders that have been described above simply add two numbers and send out their sum. The usual practice is to feed this sum to a register for temporary storage. If another number is to be added to it, the sum must be removed from the register and sent with the other number to the inputs of the adders, after which the new sum is stored in the register.

An *accumulator* is defined as a device that stores a number, adds to it any new number received, and then stores the sum. The number previously held in the accumulator is wiped out in the process. Thus, for example, if an accumulator is cleared and eight numbers are fed into it, one after another, the number stored at the end of the process is the sum of the eight.

It is entirely possible to combine a register with the parallel adders previousy shown to form an accumulator. And in the case of the serial full adder, it is

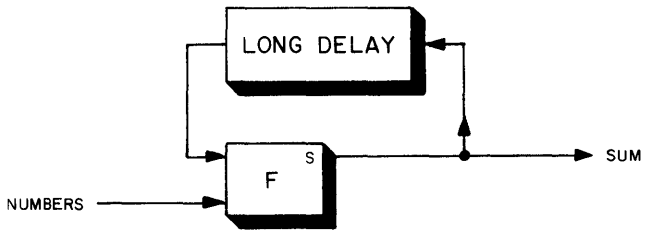


Figure 3–67. Serial Accumulator

extremely simple. All that is necessary is to make a word-length loop from the sum output line back to either the augend or addend input line, as shown in figure 3–67.

The word length loop through one or both half adders acts as a circulating register. When a number is fed in, it begins circulating in the loop (a gate in the output line is kept closed until it is desired to read out a sum). Each time through the adder, the number is added to zero and thus remains unchanged until a new input number is applied. Timing is very important, as in all serial circuits, and the arrival of each input number must be timed so that the least significant bit is added to the least significant bit of the circulating number. When this is properly done, the new number is added to the stored number and the sum remains in the loop. The number originally stored is wiped out in the addition process. In this manner, the accumulator stores the sum of all numbers fed to it.

Parallel accumulators using flip-flops to add by a counting action in each column are found in many digital computers. There are a large number of different circuits for these, but the chief differences are in the methods of handling carries.

One of the simpler arrangements appears in figure 3–68, which shows four stages of a parallel accumulator. The bits of the number (the addend) to be added to that in the accumulator may be stored in a flip-flop register (not shown). The 1 outputs from the register are applied to the gate tubes as inputs to the accumulator. Thus, the only gate tubes conditioned are those receiving the 1's in the input number. (It would be possible, as always, to use AND circuits in place of the gate tubes.)

When the addend has been loaded into the storage register and the gate tubes are conditioned, the add pulse is applied. Pulses are gated through the conditioned GT's and applied, through the OR circuits, to the complement inputs of the corresponding flip-flops. These flip-flops change state, "counting up" by 1 in each column that receives an addend 1.

When the flip-flop in any column changes to the 0 state, a carry pulse is sent to the next higher column (delayed until the input pulse dies out). It is possible, of course, that some of these carries may cause other carry pulses from the stages receiving them. Thus the carries travel automatically through all the columns. At the completion of the carry propagation process, the number left in the accumulator is the sum of the number previously stored there and the input number.

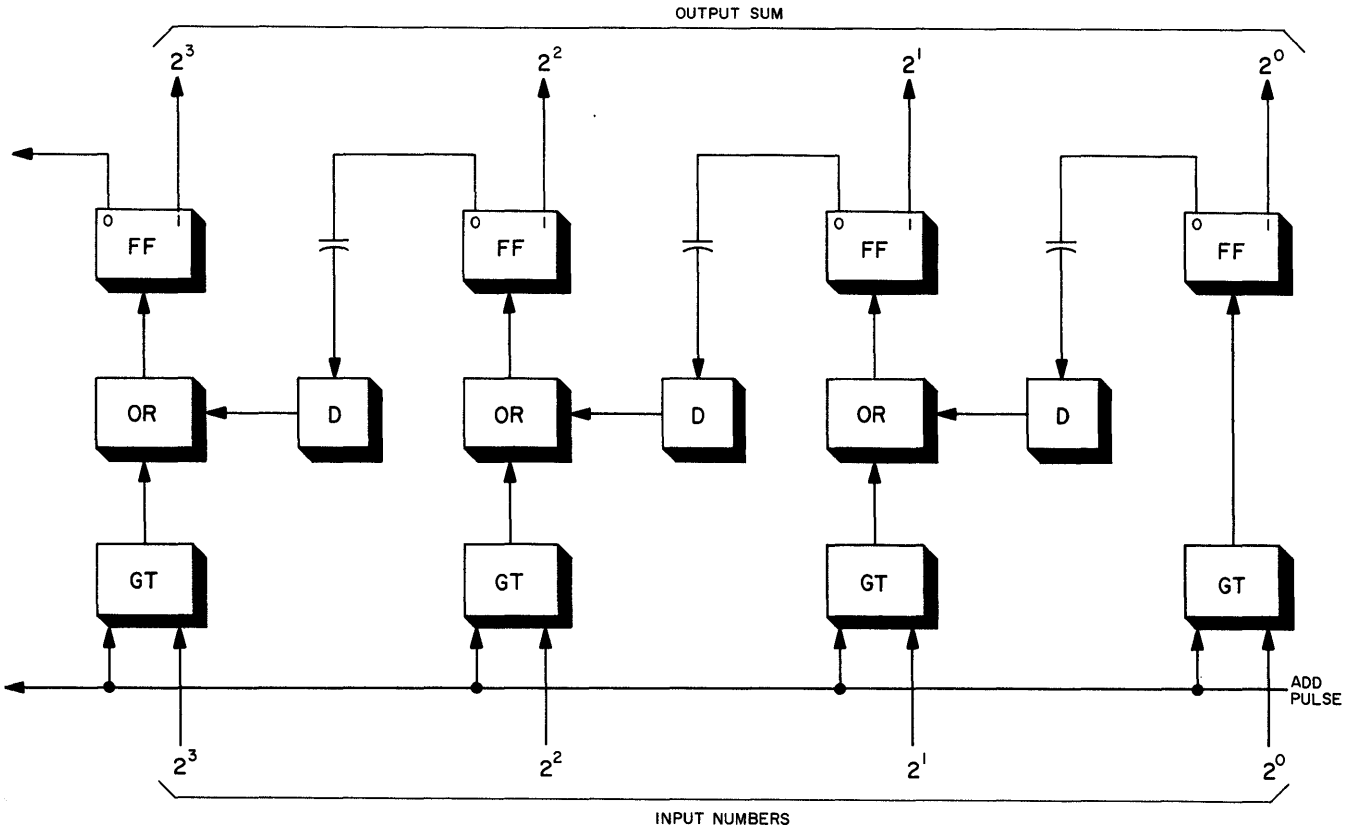


Figure 3–68. Parallel Accumulator

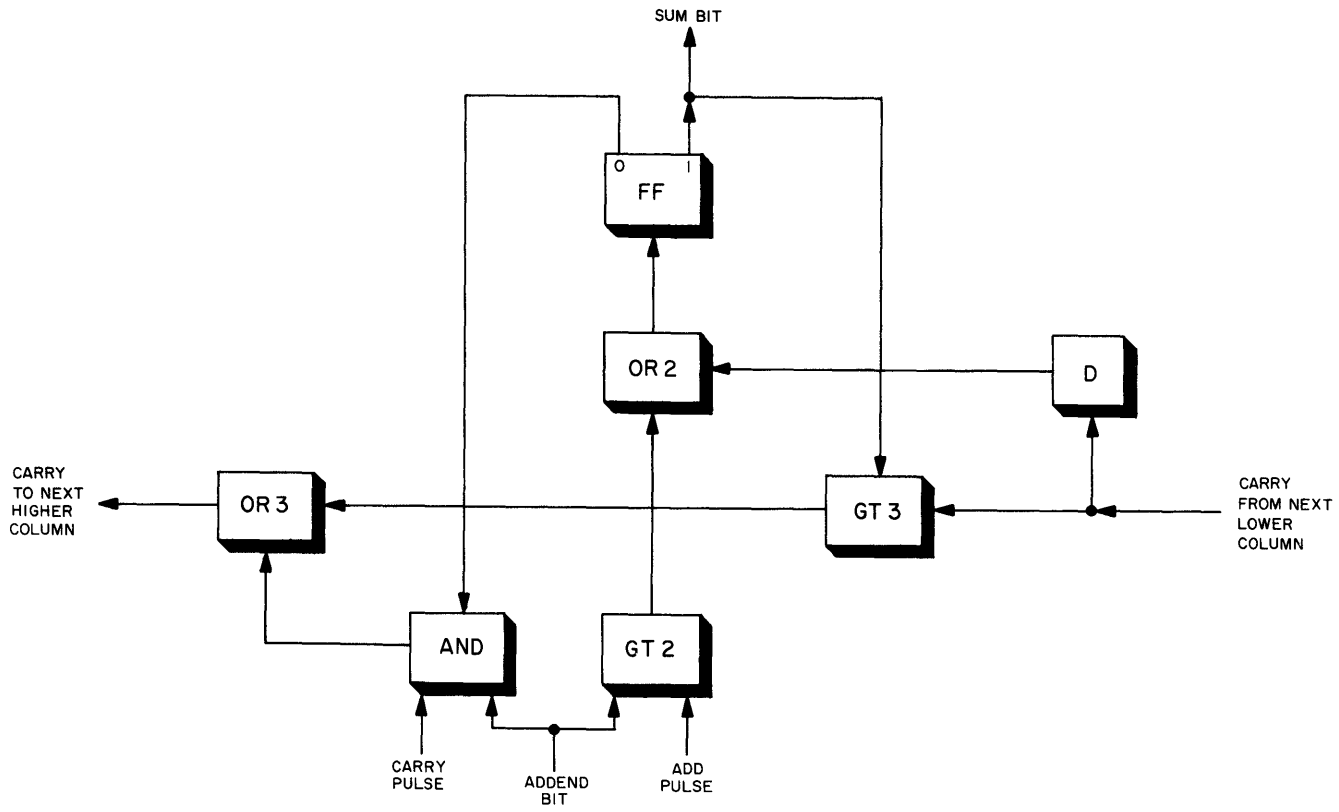


Figure 3–69. Stage of Accumulator With Faster Carry Propagation

This sum can be read at any time on the output lines shown. The entire accumulator can be cleared by pulsing the clear inputs of all the flip-flops.

Although the carry propagation in this accumulator is automatic, it is quite slow because the carry delays act in series. In other, faster methods, carries are switched to the proper columns without passing through each flip-flop; in some cases they are generated by sampling the addend and augend bits before addition begins, or even added simultaneously, although the latter method requires so much switching it is seldom practical for more than three or four stages.

One stage of a fairly common type of parallel accumulator appears in figure 3–69. Again the bits of the addend are assumed to be stored in a flip-flop register. If the addend bit applied to a given stage is a 1, GT 2 of that stage is conditioned and the add pulse is gated to OR 2 to switch the flip-flop.

As soon as the flip-flops have been switched, the logic circuits are prepared for the carry-handling process, which cannot occur, however, until a carry pulse is applied to the AND circuits of all stages. After the addition of the addend, if the flip-flop is in the 0 state and the addend bit is a 1, then the addition in this column must have been $1 + 1 = 0$ and a carry of (1) has to be developed. This is done by the AND circuit when the carry pulse arrives. The pulse passes through OR 3 to the stage representing the next higher column.

If, on the other hand, the flip-flop is in the 1 state, any carry from lower columns must switch it to 0 and develop a new carry. To save propagation time, however, GT 3 is conditioned by the 1 side of the flip-flop and a carry pulse arriving under this condition is gated without delay through GT 3 and OR 3 to the next higher column. In this manner, a carry pulse can quickly be switched through several consecutive columns (stages) indicating 1's. This is called "ripple-through" carrying.

In each stage where a carry is received (or passed through), it is delayed and then fed through OR 2 to switch the flip-flop. The delay allows the pulse on the carry line to die out before the flip-flop changes state, preventing the development of a false carry through the AND circuit or GT 3.

With accumulators operating along lines similar to these, carry-handling is considerably faster than in the first type shown.

3.3 SUBTRACTION

The rules of binary subtraction can be tabulated like those of addition. The four possible cases are:

0	1	0	1	Minuend Bit
−0	−0	−1	−1	Subtrahend Bit
0	1	1	0	Difference Bit
(0)	(0)	(1)	(0)	Borrow Bit

It is possible to build a half subtracter that will accept any two bits and produce a difference bit and a borrow bit according to these rules. A full subtracter can be made with two half subtracters and an OR circuit.

In practice, however, subtracters are almost never used. Compare the above table to the table for addition and it becomes apparent that the difference bits are exactly the same as the sum bits for the same cases. The only change is that the two right-hand borrow bits are the opposites of the corresponding carry bits.

In addition:

0	1	
+1	+1	
1	0	
(0)	(1)	Carry Bits

But in subtraction:

0	1	
−1	−1	
1	0	
(1)	(0)	Borrow Bits

Since these are the only differences between addition and subtraction, it is comparatively easy to modify an adder into an adder-subtracter, a device that can either add or subtract in response to the control signals sent to it. An accumulator can be modified for the same purpose. In the adder, it is only necessary to suppress one possible carry and generate one borrow when the command to subtract is received. Two inhibit circuits with AND inputs accomplish this. In an accumulator, a borrow should be produced when a flip-flop changes to the 1 state (and the subtrahend bit is 1) and borrows should be passed through any stages that are in the 0 state. For subtraction, therefore, switching is added to interchange the 0 and 1 output lines from each accumulator flip-flop.

Probably the most common subtraction method of all, however, is to add the complement of the subtrahend to the minuend. Straightforward adders or accumulators are used and the only extra switching required is to complement the numbers. If the subtrahend is fed to the accumulator from a flip-flop register, addition or subtraction can be handled very easily by inserting gates in all the 1 and 0 output lines from the register flip-flops, as shown in figure 3–70. If addition is to be performed, the 1 gates are pulsed and the number in its true form is sent to the accumulator or adders. For subtraction, the 0 gates are pulsed, feeding out the 1's complement of the number in the register.

Another common practice is to complement the subtrahend right in the register and then add. Only the normal set of gates is required.

The 2's complement is more difficult to obtain directly. The effect of using it can be accomplished by generating the 1's complement, as above, but adding 1 as the carry to the least significant place in the accumulator and allowing any resulting carries to propagate.

In many computers, negative numbers are kept in complement form at all times and are even stored in memory in this form. This makes addition and subtraction very easy, although there are some pitfalls. For example, when a positive number is subtracted directly (not by complementing and adding) from another smaller than itself, the subtraction circuitry always gives the negative difference as the 2's complement of the true answer. This is due to the cyclic nature of the numbering system. The 2's complement can be changed to the 1's complement by bringing around and subtracting the end carry. It is often easiest to keep such a negative number in complement form and, by so doing, the problem of keeping track of changing signs is handled automatically.

The comparative advantages of the 1's complement against the 2's complement are difficult to evaluate and the choice of one or the other for use depends upon the overall requirements of the computer.

With the 1's complement system, complementing is easy, but end carries must be handled (requiring extra propagation time) and there are two values for zero, as explained in Part 2. The *positive zero* value, 00000, never occurs as a result of addition or subtraction; instead, it is always *negative zero,* 111111, that appears in these cases. This can be awkward since it is often desirable to have zero appear as a positive number. In a computer using serial transmission, the end carry can be handled only by passing the sum through the adder a second time, which makes addition or subtraction comparatively slow.

The end carry does not occur with the 2's complement system and zero is always positive, 00000, so despite the greater difficulty of forming complements, this system offers certain definite advantages, especially for serial operation.

3.4 MULTIPLICATION

The binary multiplication table consists of only the first two places of the decimal table, and hence is very simple.

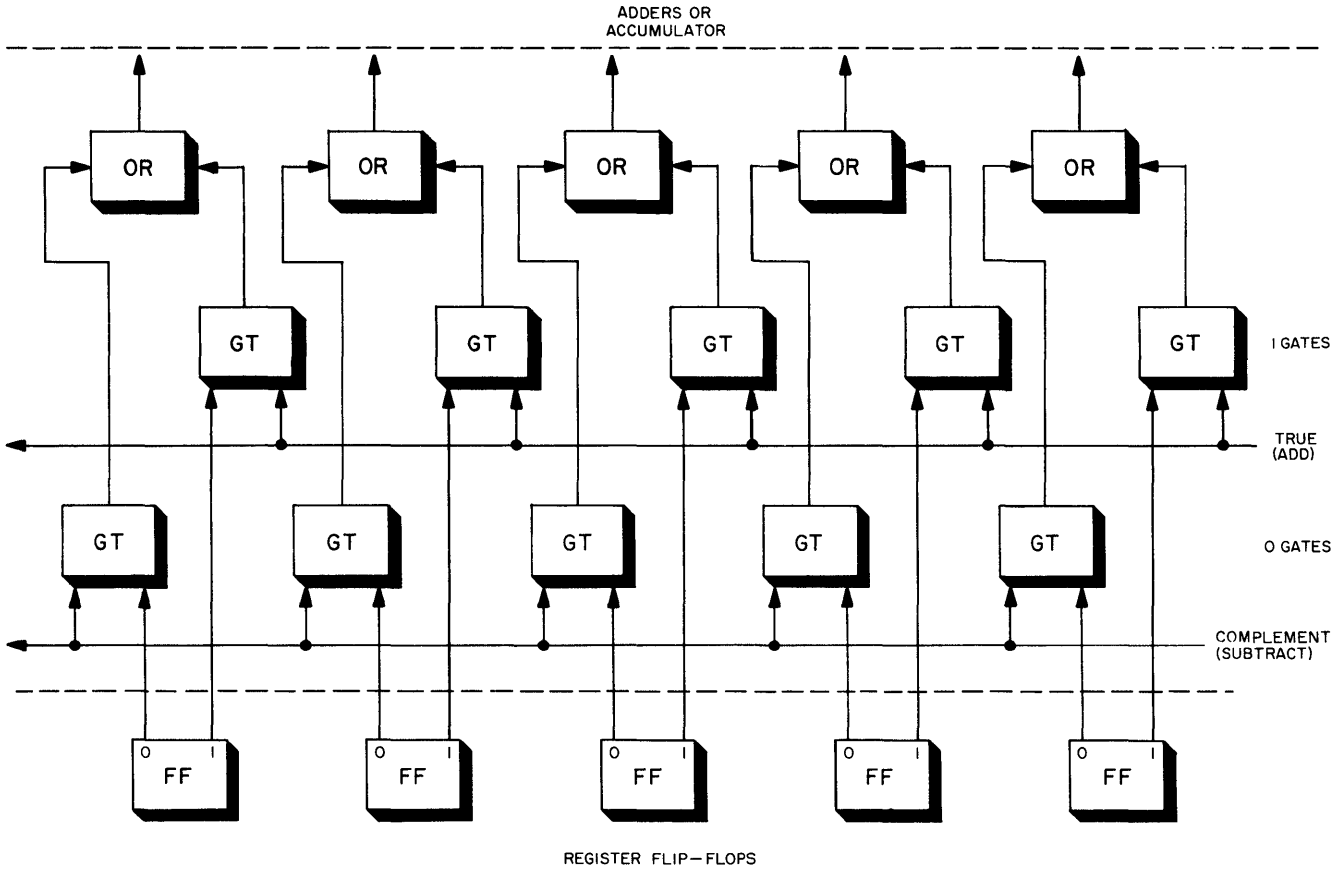


Figure 3–70. Arrangement for Transfer of True or 1's Complement Number

		0	1	Multiplicand Bit
Multiplier	0	0	0	
Bit	1	0	1	

From an example, it can be seen that the whole process is relatively uncomplicated:

DECIMAL		BINARY	
11	=	01011	Multiplicand
14	=	01110	Multiplier
44		00000	
11		01011	Partial
		01011	
		01011	Products
		00000	
154	=	010011010	Product

Each partial product, in binary as in decimal multiplication, is the product of the multiplicand and one bit of the multiplier. Since the binary bits can only be 0 or 1, each partial product must therefore be either zero or the multiplicand. The only arithmetic process actually involved, therefore, is addition, so it should be possible to perform multiplication with adders or, better, with an accumulator. Note, however, that each partial product must be shifted a number of places equal to the position of the multiplier bit.

The zero partial products in the binary example seem to be unnecessary, but a shift must occur for every multiplier bit, even if nothing is added to the partial products. This becomes especially important when it is recalled that the arithmetic circuitry cannot add a column of numbers such as these partial products simultaneously, but instead must add them two at a time.

One problem that complicates the picture is the fact that the maximum length of the product equals the length of the multiplicand plus the length of the multiplier. Most computers are built to handle words of some fixed length, so for a word length of N bits, the

product must contain 2N bits. The example given previously showed a product of 2N—1 bits, but only because the multiplicand and multiplier were small in comparison to the largest possible 5-bit number, which is 11111. When larger whole numbers are multiplied, a carry occurs from the last place on the left, making the product 2N bits long. For example:

10110	Multiplicand
11001	Multiplier
10110	
00000	Partial
00000	
10110	Products
10110	
1000100110	Product

A great many computers, however, use fractional binary numbers and carry negative numbers in complement form. In this case, the most significant bit is the sign bit and the binary point always appears immediately to the right of it. Again using 5-bit numbers as an example, the largest possible positive number is 0.1111 and the largest negative number is either 1.0001 (2's complement) or 1.0000 (1's complement).

The location of the binary point in the product is determined in the same manner as in decimal multiplication—by counting off the number of fractional binary places in the multiplicand plus the number of fractional places in the multiplier. For words of N bits (including sign), since each contains N—1 fractional binary places, the product must contain 2N—2 places to the right of the decimal point. Adding one bit in the most significant place to indicate the sign, the total length of the product becomes 2N—1 bits when fractional numbers are used.

0.1011	Multiplicand (N)
0.1110	Multiplier (N)
00000	
01011	Partial
01011	
01011	Products
00000	
0.10011010	Product (2N-1)

As long as both multiplicand and multiplier are positive numbers, no carry beyond the most significant place can occur. This brings up the question of how to handle negative numbers in complement form, which obviously cannot be multiplied according to the rules for positive numbers since, for one thing, these rules do not allow for filling out partial products with changing signs. For example:

—0.1111	Negative Multiplicand
0.1001	Positive Multiplier
—0.0000 1111	1st Partial Product

The first partial product is negative, according to the rule of signs which states that a negative times a positive yields a negative. The 2's complement of this first partial product is 1.11110001. But, following the regular rules of multiplication and using the 2's complement of the multiplicand:

1.0001	Negative Multiplicand
0.1001	Positive Mutiplier
0.00010001	1st Partial Product

So, this partial product is incorrect because it is not the 2's complement of the true value. One solution is to devise new multiplication rules, which can be done quite easily for the 2's complement system but is somewhat more complicated when the 1's complements are used.

Actually the easiest method, and probably the most commonly used, is to put all numbers in true (positive) form before multiplying. This can be done by testing the sign bits of the multiplicand and multiplier, complementing either or both of these numbers to make them positive, and then complementing the product after the multiplication, if it should be negative.

The sign of the product of any multiplication is determined as follows:

MULTIPLICAND & MULTIPLIER	PRODUCT
Both positive	Positive
One negative	Negative
Both negative	Positive

In the arithmetic circuitry, this is easy to handle. A sign flip-flop is cleared before the signs of the multiplicand and multiplier are sensed (tested for value). If either number is found to be negative, the flip-flop is complemented at the same time as the number. Thus, after the process, the flip-flop indicates 0 if both numbers were positive or if both were negative (in the latter case, it is complemented twice). It indicates 1 if one number was negative, the other positive. The output of the flip-flop, therefore, can be used to indicate whether or not the product must be complemented.

Another problem is introduced by the fact that the product of a multiplication is nearly twice the fixed

word length that can be handled by the computer circuits. With the sample 5-bit words given previously, for example, the products are 9 bits long if the numbers are fractional.

An obvious answer in the case of fractional numbers is simply to chop the product to length, retaining the proper number of most significant bits. Thus, in the last full-length example, the product, 0.10011010, would be chopped down to 0.1001. Actually, however, this is not very accurate because the part being thrown away makes this product closer in actual value to 0.1010 than it is to the retained value of 0.1001. What is needed to avoid sizable errors, then, is a method of *rounding off* to the nearest value.

This is done exactly as in the decimal system, except that it is somewhat easier. A decimal number such as 33.X is rounded off to the nearest whole number value by observing whether 0.X is more or less than half the corresponding power of the radix. If less than half (0.0 to 0.4), the actual value is nearest 33.0; if half or more (0.5 to 0.9), the nearest whole number is 34.0. This can be done mechanically, without thinking, by adding half the corresponding power of the radix in the most significant place to be dropped, then chopping.

For example:

33.X	=	33.2	
		+ .5	round off
		33.7	
		33	chop
33.X	=	33.6	
		+ .5	round off
		34.1	
		34	chop

In the binary system, half a given power of the radix is a 1 in that place. To round off the binary product shown above, therefore, a 1 is added to the most significant place to be thrown away, which is the fifth place to the right of the decimal point:

0.10011010	product
+ 1	round off
0.10100010	
0.1010	chop

The round-off of whole numbers becomes involved with the question of significance of digits, and round-off methods are usually suited to the particular application, with variations too great to be covered here.

The foregoing are some of the major problems encountered in preparing to "mechanize" binary multiplication; that is, to design circuitry capable of performing it.

3.4.1 Parallel Methods

Certainly the most common approach to multiplication in computers handling numbers in parallel form is the repeated addition of the multiplicand, with appropriate shifts, in a set of adders or an accumulator. The multiplier is stored separately and its bits are used in sequence to determine whether the multiplicand or zero is added to the accumulated sum of the partial products.

(It is possible to build a "simultaneous multiplier" that accepts the multiplier and multiplicand simultaneously and produces signals representing the product, but this requires so much equipment that it is seldom practical for numbers of useful length.)

The problem of shifting can be attacked in either of two ways. That is, the multiplicand can be shifted to the left between entries into the accumulator, or the accumulated sum of the partial products can be shifted to the right after each addition.

Since the multiplicand is usually held in a register and gated into the accumulator, one solution is to use a shifting register of extra length. In multiplying two numbers, each N bits long, N−1 shifts must be made, so the register must contain N + (N−1) = 2N−1 places. There must be a gate for each place, so the "gate string" is the same length as the register.

At the start of the multiplication problem, the multiplicand is loaded into the N places at the right of the register and shifted one place left after each entry into the accumulator. This type of arrangement, using 5-bit numbers as an example, appears in figure 3–71.

The gating of the multiplicand into the accumulator is done by the multiplier bits, stored in another register and fed into the accumulator one at a time, least significant bit first. When the multiplier bit is a 0, the gates are not opened and nothing (a zero partial product) is added to the accumulator contents. The multiplier bit (inverted if it is a 0) is delayed long enough to allow proper gating and then applied to shift the contents of the shifting register one place to the left. (The diode prevents an inverter output from getting back to act as a false gate signal.)

The accumulator itself must also be increased in length – to 2 N places if the computer uses whole numbers, 2 N−1 places if fractional numbers are used. If the extra place on the left (not shown in fig. 3–71) is needed, its only input is a possible carry from the most significant place shown here.

The rounded-off product is taken from the N most significant places of the accumulator. The rounding-off is accomplished by adding a 1 in the place shown after the product has been formed (or at any convenient time

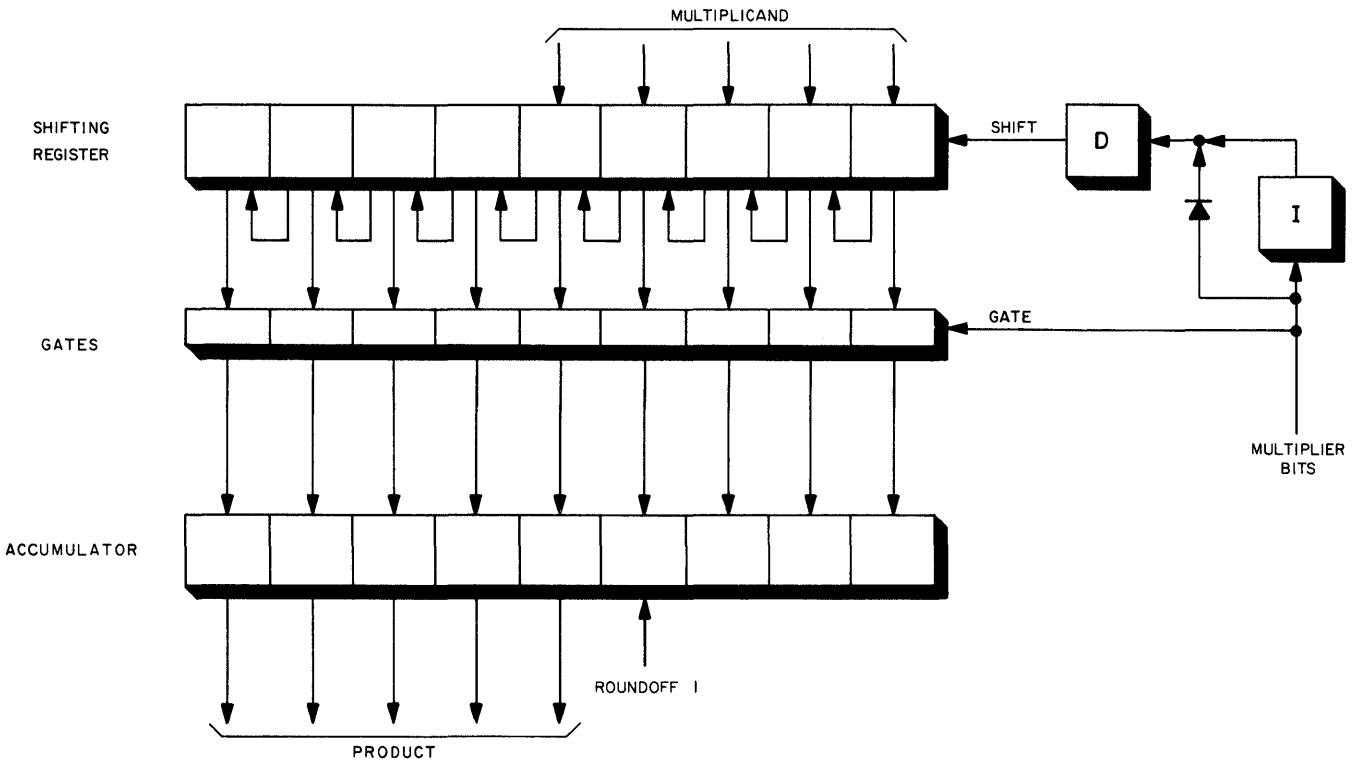


Figure 3–71. Shifting Register Feeding Accumulator for Multiplication

during the summing of the partial products).

In operation, the register and accumulator are both cleared and the multiplicand is loaded into the shifting register while the multiplier is placed in a register of its own. (If necessary, each is complemented to put it in positive form and a sign flip-flop is set.) Assuming the multiplicand and multiplier to be 0.1011 and 0.1110, respectively, the shifting register and accumulator at this point contain:

01011	Reg
000000000	Acc

(The unfilled spaces in the register contain 0's; they are omitted so the position of the multiplicand can be clearly seen.) Now, the least significant bit of the multiplier, 0, is brought in. It cannot fire the gates, but it does cause a shift after a short delay. The result of this "add zero and shift" is:

01011	Reg
000000000	Acc

The next multiplier bit is a 1, which has the effect of a command to "add multiplicand and shift." This gates the multiplicand down into the accumulator before another shift is made. Following this first 1:

01011	Reg
000010110	Acc

The next two multiplier bits are also 1's, each of which also adds the multiplicand into the corresponding places of the accumulator and then shifts the multiplicand. The effects are:

01011	Reg
001000010	Acc
01011	Reg
010011010	Acc

The last multiplier bit is a 0. Furthermore, it must always be a 0, because these fractional numbers are always placed in positive form before multiplying. Therefore, the multiplier sign bit can have no effect on the product, which is already in the accumulator. To save multiplication time, the sign bit can be suppressed or dropped out of the circuitry and the time that would otherwise be spent in uselessly shifting the multiplicand again can be used to round off. If the multiplier sign bit is dropped, the most significant place in the shifting register and gate string can be eliminated.

The round-off process is:

010011010	Acc
1	Add 1 to round off
01010	Rounded product

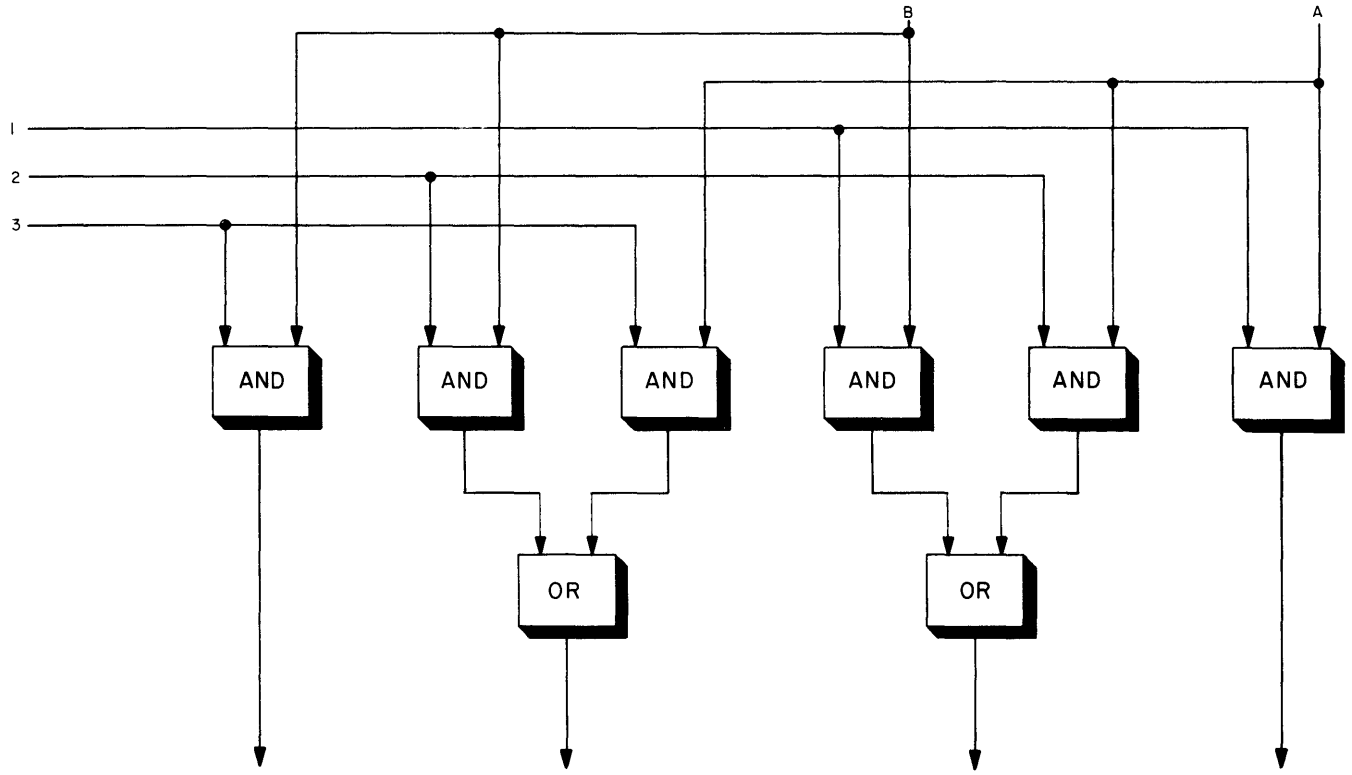


Figure 3—72. Gating and Shifting by Switching

This is the basis of one common approach to parallel multiplication. There are, of course, a number of different ways to accomplish the same results. One fault of the arrangement shown in figure 3—71 is that it uses a considerable amount of extra circuitry, since the shifting register, gate string, and accumulator must all be extra-length.

One possible modification would be to use a word-length storage register for the multiplicand and to use switching networks to accomplish both gating and shifting at the same time. An arrangement of this sort, capable of shifting two bits in parallel to any of three places on receipt of a signal on the 1, 2, or 3 line, is shown in figure 3—72. A control signal on the 2 line, for example, causes a shift of two places to the left. The number of AND circuits alone, however, amounts to the product of bits and shifts, so the method is not practical for large numbers.

An alternative, mentioned earlier, is to shift the accumulated sum instead of the partial products themselves. To do this requires a shifting accumulator, which is easily made by adding shifting lines and gates to the usual accumulator.

The shift must be to the right, to line up the accumulated sums properly with the multiplicand, so the accumulator must be extended to the right. The multiplicand can be held in a simple, word-length storage register since it is not shifted. This means that the multiplicand is always entered into the same places of the accumulator; hence the accumulator extension to the right need only be a shifting register, since no addition is performed there.

The arrangement is shown in figure 3—73. If the computer uses whole numbers instead of fractional numbers, one additional accumulator place on the left must be provided to take care of possible carries out of the most significant place. These do not occur with positive fractional numbers. If the product is to be rounded off, the place where the round-off 1 is added must also be an accumulating place to handle the addition.

Since the accumulator and shifting register are cleared at the beginning of the multiplication operation, there is no reason why the multiplier cannot be stored in the shifting register. It is true that one bit of the multiplier is lost after each shift, but only after it has been used to determine whether zero or the multiplicand should be entered, so the loss makes no difference.

The shift can be made to occur automatically after each addition or, if the multiplier bit is a 0, after it is determined that zero is to be added.

The operation from this point on is illustrated bebe followed by observing the contents of the combined accumulator-shifting register, since the multiplicand does not shift. The multiplier and multiplicand are complemented to positive form, if necessary, before the

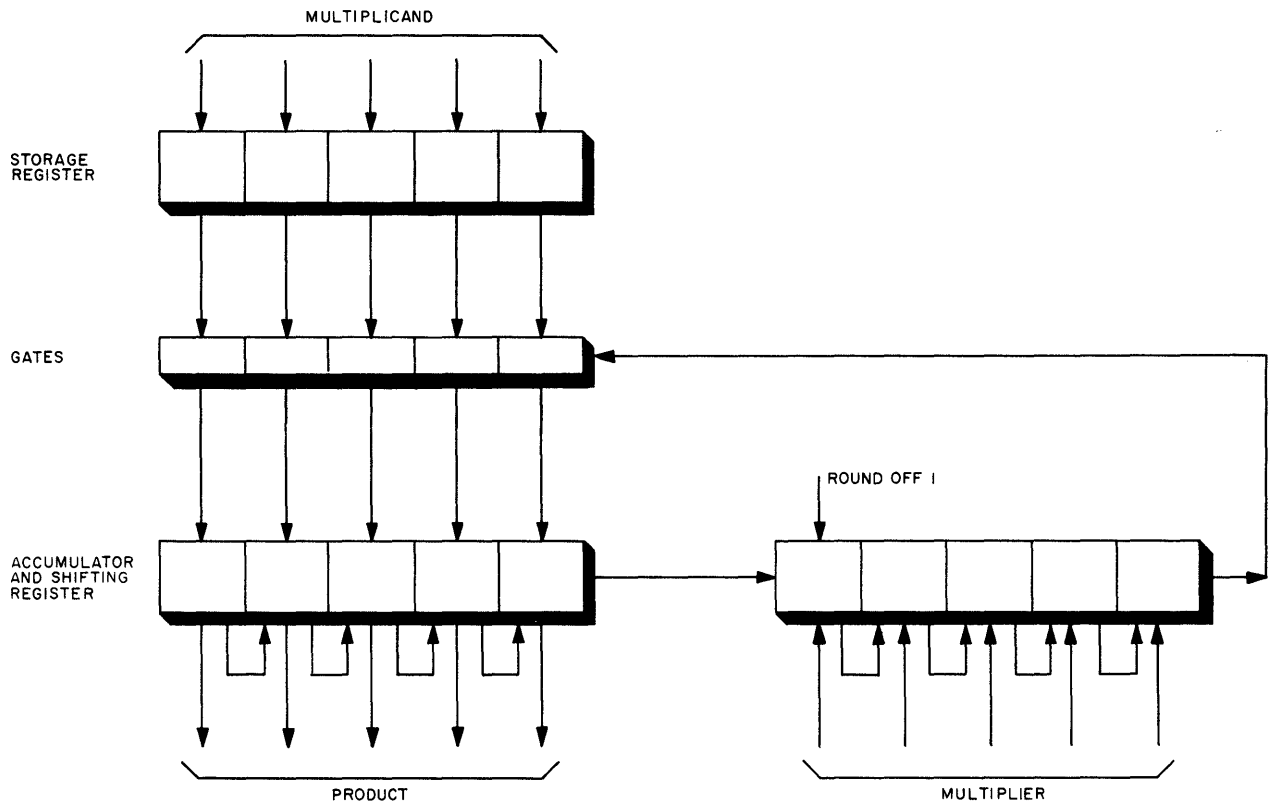


Figure 3–73. Shifting Accumulator Used for Multiplication

multiplication begins. This may be done before they arrive in the arithmetic circuitry, or after they are in the registers. The sign flip-flop is set during this process to indicate the sign of the product.

The operation from this point on is illustrated below by multiplying the same two numbers as in the previous example, 0.1011 and 0.1110. The multiplier bits are enclosed in parentheses to make it easier to keep track of them in the shifts.

0. 1 0 1 1	Multiplicand
0 0 0 0 0 (0. 1 1 1 0)	Accumulator
0 0 0 0 0 (0. 1 1 1 0)	Add zero
0 0 0 0 0 0 (0. 1 1 1)	Shift
0 1 0 1 1 0 (0. 1 1 1)	Add multiplicand
0 0 1 0 1 1 0 (0. 1 1)	Shift
1 0 0 0 0 1 0 (0. 1 1)	Add multiplicand
0 1 0 0 0 0 1 0 (0. 1)	Shift
1 0 0 1 1 0 1 0 (0. 1)	Add multiplicand
0 1 0 0 1 1 0 1 0 (0)	Shift
1	Round off
0. 1 0 1 0	Product

Again, the sign bit of the multiplier can be suppressed and an unnecessary shift eliminated. Had either the multiplier or the multiplicand (but not both) been negative, the sign flip-flop would have been set to 1 and this would be used as a signal to complement the product.

The use of a shifting accumulator to perform multiplication in this manner is very common. There are a number of possible variations in circuitry, of course, but the basic approach is the same.

3.4.2 Serial Methods

Although the principles remain the same, the multiplication of numbers transmitted serially involves different methods and, again, timing is very important.

It is apparent that the repeated additions can be performed only by passing the multiplicand and accumulated partial products through an adder the proper number of times, shifting each time. Thus, the serial methods are slower than the parallel.

A serial number can be shifted to the left simply by delaying it, as shown in figure 3–74. This is purely a matter of relative timing, of course, and the shift is not noticeable or important except in comparison with an unshifted number or control signal. In the figure, a number is shown being fed directly to one input of a full adder and, in a branch circuit, being delayed one

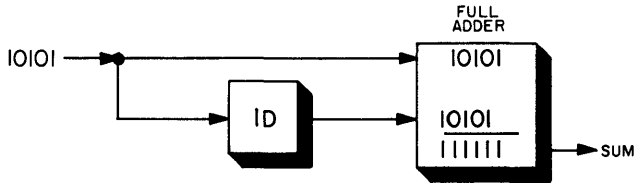


Figure 3—74. Shifting by Delay of Serial Number

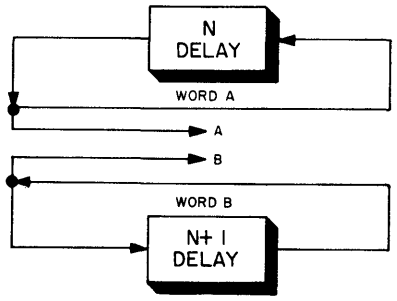


Figure 3—75. Shifting With Circulating Registers

bit-time before being applied to the other input. On the direct line, the least significant bit reaches the adder at a bit-time that can be called T1. In the branch, however, the 1-bit delay holds up the least significant bit until T2. All the following bits are also delayed in this branch, so the 1-bit delay shifts the number 1 place to the left in comparison to the undelayed number.

A shift to the right would mean advancing a number in time, which is impossible. This is unimportant, actually, since any shift must always be relative to some other number and shifting one to the left has the same effect as shifting the other to the right. This suggests the use of circulating registers of different lengths to obtain regular shifts, such as those required in multiplcation. Figure 3—75 shows two registers with delays of N and N + 1 bit-times, respectively, where N equals the computer word length.

If words A and B are inserted in the registers simultaneously, word A makes a circuit of the register in N bit-times, while word B requires one extra bit-time to make a complete circuit. Therefore, if they start at the same instant and are of the same length, word B shifts one place to the left relative to word A for each circuit of the registers. Using 3-bit words as an illustration, the results are shown in table 3—2. The bit positions are numbered in order of significance and time reads from right to left to show more clearly the manner in which the words shift.

Only two shifts are possible with 3-bit numbers, of course, but only N—1 shifts are required in any multiplication, so this arrangement can provide the necessary number. Since word B shifts to the left, this must represent the multiplicand and word A must represent the accumulated sum of the partial products. (The bits of word B, therefore, do not change, but the A bits may change after each addition; remember that A1, A2, etc., show bit positions, not the actual bits occupying them.) To obtain the accumulated sums, the N register has to be looped through a serial full adder, as shown in simplified form in figure 3—76. The N + 1 register, containing the multiplicand, must feed the other adder input. A gate controlled by the multiplier stored (elsewhere) lets through the multiplicand or zero, depending upon the multiplier bit.

TABLE 3—2. WORD SHIFTS IN CIRCULATING REGISTERS

BIT-TIME	12	11	10	9	8	7	6	5	4	3	2	1
N Reg	A3	A2	A1	A3	A2	A1	A3	A2	A1	A3	A2	A1
N + 1 Reg	0	B3	B2	B1	0	B3	B2	B1	0	B3	B2	B1

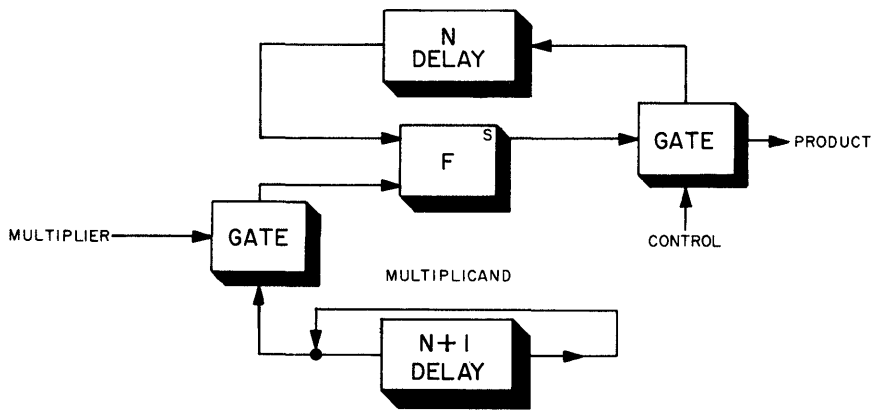


Figure 3—76. Basic Arrangement for Serial Multiplication

The N register must be cleared at the start of the multiplication process. Although the delay itself is marked N, this actually must be the total delay around the loop, including the full adder and the gate. The multiplicand, complemented if necessary to place it in positive form, is loaded into the N + 1 register. The multiplier, also in positive form, is stored in a register from which it can control the multiplicand gate.

The first time through the adder, the multiplicand is added to the zero in the cleared N register, resulting in the first partial product. The next time around, the multiplicand is shifted and added to this, giving the accumulated sum of the first and second partial products. After each addition, the least significant bit of the accumulated sum is actually a product bit because the multiplicand will be shifted before being entered again. This product bit is taken out and stored elsewhere, since the N register will hold only N bits. An example will show this more clearly.

MACHINE		PENCIL AND PAPER	
000		101	
+101	Multiplicand	111	
101	A	101	A
+101	Multiplicand	101	B
111(1)	A + B	101	C
+101	Multiplicand	100011	
1000(11)	A + B + C		

This is done in the arrangement of figure 3–76 by first adding the multiplicand to zero in the cleared N register. Table 3–3 shows the complete timing of this serial multiplication.

A3	A2	A1	000	N Reg
B3	B2	B1	101	Multiplicand
(A4)A3	A2	A1	(0)101	1st Partial Product

The least significant bit in the A1 position of this first partial product (which is the first accumulated sum) is the least significant bit of the final product. It is not involved in any further additions and therefore is gated by a control pulse out of the N register to a separate register that stores the product bits. This occurs immediately after the A1 bit is formed (bit-time 1 in table 3–3). The bits in places A2 and A3, plus the carry (in the place left vacant by the removal of the A1 bit), are sent through the N register. The bit in the A2 place reaches the adder input again at bit-time 5, along with bit B1 of the shifted multiplicand, making its second pass.

(A4)	A3	A2	(0)10	
B3	B2	B1	1 01	Multiplicand
(A5) A4	A3	A2	(0) 1 11	Acc Sum

This time, the new bit in the A2 place is gated to the product register and the new carry (A5) is returned at bit-time 8 to fill out the accumulated sum to N bits. On the last pass of the multiplicand:

(A5)	A4	A3	(0)11	
B3	B2	B1	1 01	Multiplicand
(A6) A5	A4	A3	(1) 0 00	Acc Sum

These are the final product bits and all are gated out to the register, the exact method depending upon the control circuitry. Rounding off and chopping the product to length can be done in the adder and gate, if desired. Notice that bit (A6) occurs at bit-time 12 as a carry but is actually the most significant bit of the product. It is treated as a normal product bit, of course.

This is the basic approach to serial multiplication. Many variations are possible, as in all the arithmetic methods, but all are comparatively slow due to the need for passing the multiplicand repeatedly through the adder. Since the multiplicand plus shift is N + 1 bit-times in duration, and it must go through the adder N times, the minimum time required for serial multiplications is N (N + 1) bit-times. If fractional numbers in true (positive) form are used and multiplication by the sign bit is suppressed, N^2 bit-times are required, so the saving is minor in comparison to the speed of parallel methods.

Obviously, the multiplication could be speeded up enormously if a method could be found that required only one passage of the serial multiplicand. It is equally

TABLE 3–3. TIMING OF SERIAL MULTIPLICATION

BIT-TIME	12	11	10	9	8	7	6	5	4	3	2	1
N Reg	(A6)	A5	A4	A3	(A5)	A4	A3	A2	(A4)	A3	A2	A1
N + 1 Reg	0	B3	B2	B1	0	B3	B2	B1	0	B3	B2	B1
Product	A6	A5	A4	A3				A2				A1

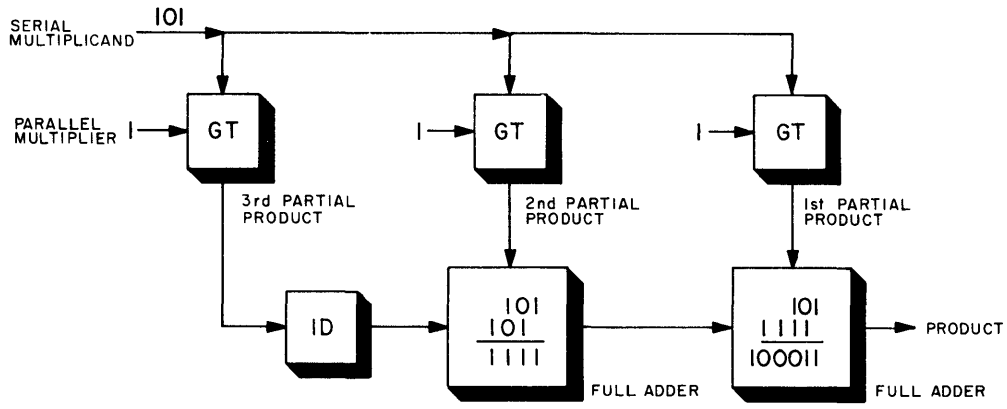


Figure 3–77. Serial-Parallel Multiplication

obvious that this cannot be done with a single adder, but adders are comparatively cheap and speed is very important in many or most computers.

The clue to such a method, often called serial-parallel multiplication although it is basically a serial method, appeared in figure 3–74. This figure illustrates the possibility of shifting a number and adding it to itself, which is what must be done many times over in multiplication.

Consider that the delay in a serial full adder is usually (or can easily be made) one bit-time, and it becomes apparent that the sum shown in figure 3–74 has been shifted one place to the left in comparison with the input number. This must be the case, since the first bit of the input word reaches the adder input at bit-time 1 on the direct line, but the first sum bit does not emerge until bit-time 2 because of the delay through the adder.

BIT-TIME	7	6	5	4	3	2	1
Number			1	0	1	0	1
Delayed		1	0	1	0	1	
Sum	1	1	1	1	1	1	

This immediately suggests that the input number might be fed simultaneously at bit-time (1) to one input of a second adder and the shifted sum to the other input, to produce a second shift and addition. Figure 3–77 shows two adders used in this fashion to multiply the same pair of 3-bit numbers shown in the serial method. Through gates controlled by the bits of the multiplier, held in a serial-parallel register, all the partial products are entered simultaneously. The 3rd partial product is shifted by the 1-bit delay circuit and added to the 2nd. The sum of these two is shifted by the delay in the first adder and added to the 1st partial product.

To multiply numbers N bits long, N–1 adders are required and the necessary N–1 shifts are obtained in the series adder string. The only delay in producing the final product is that introduced by the last adder, one bit-time, so the product begins emerging at bit-time 2. Since the longest possible product is 2N bits, the maximum time required for this serial-parallel multiplication method is 2N + 1 bit-times.

3.5 DIVISION

Division is a process of repeated subtractions. It appears, at first glance, to be the direct opposite of multiplication, but there are important differences, including the factor of trial and error.

The division operation consists of repeated attempts to subtract the divisor, first from the dividend and then, with appropriate shifts, from the successive remainders. The element of trial and error occurs because the divisor either "goes" or "does not go" into any given remainder, depending upon whether the remainder is larger than the divisor.

In pencil-and-paper work (binary), if the divisor "goes," a 1 is recorded in the quotient and the subtraction is performed, leaving a positive balance as the new remainder. If the divisor "does not go," a 0 is recorded in the quotient since the divisor is larger than the remainder and, if the subtraction were performed, the balance would be negative (in 2's complement form). Actually, a variety of subtraction is done mentally in comparing the sizes of the remainder and divisor. When the divisor does not go, the previous remainder must be restored, the divisor shifted one place to the right, and a new trial made. The necessity for backing up to the previous remainder gives this process its name, the *restoring* method.

This method is satisfactory for pencil-and-paper division because the remainder and divisor can be inspected and no subtraction is made if the divisor is larger. In computer circuitry, it would be possible to compare the two numbers before subtracting, but this is

time-consuming and would make division even slower than is otherwise necessary. It turns out to be just as practical to go ahead and subtract. If the balance is positive, a 1 is placed in the quotient, and the balance becomes the new remainder from which the shifted divisor is subtracted in the next step.

If, however, the balance after subtraction is negative, a 0 goes in the quotient and the previous remainder must be restored for the subtraction in the next step. The simplest way to restore the remainder is to add the divisor back in. For example:

0110011	Remainder
01110	Subtract Divisor
(—)1111011	Balance (Quotient Bit 0)
01110	Add Divisor
0110011	Previous Remainder
01110	Shift and Subtract
(+)0010111	Balance (Quotient Bit 1)
Etc.	

This process can be performed, of course, in a set of parallel adders, in a shifting accumulator, or in a serial adder with circulating registers. In short, by adding the proper control circuitry, it can be done in any of the arrangements shown for multiplication, except the serial-parallel. The necessary shifts can be accomplished by shifting the divisor to the right or the remainder to the left, whichever is most convenient. The usual practice is to shift the remainder.

The restoring method of division is slow because of the extra time required to add the divisor back in each time a negative balance occurs. Analysis of the process shown in the above example indicates that the divisor is added to the negative remainder, then shifted one place to the right (which divides it by 2), and subtracted to obtain the next balance. Adding the divisor and then subtracting half of it has the same effect as adding half the divisor, which could be done in one step instead of two.

Thus, the restoring process can be eliminated and the same result obtained by shifting and adding the divisor when the balance from the previous subtraction is negative. Since there is no need to back up to the previous remainder, this is a *non-restoring* method of division. Using the same example as before, it works like this:

0110011	Remainder
01110	Subtract Divisor
(—)1111011	Remainder (Quotient Bit 0)
01110	Shift and Add
(+)0010111	Remainder (Quotient Bit 1)

This is considerably faster for machine computing than the restoring method.

In the case of computers using fractional binary numbers, a special restriction is placed upon division. Since a machine of this type cannot hold (without error) a number as large as +1, the dividend must not be equal to or larger than the divisor, for the quotient would then be +1 or greater. The computer would perform such a division but the answer would be completely misleading, due to the nature of the numbering system. It is therefore up to the programmer to make certain that such a situation does not occur. Because of this restriction on fractional numbers, there are two possible approaches to starting the division process with these numbers.

The first step in some computers is to line up the binary points in the divisor and dividend and subtract. If the balance is negative, this is proof that the dividend is smaller than the divisor and the quotient will fit the machine. A positive balance from this first subtraction, however, indicates the impossible situation. The sign of this balance can be used to stop the division process or warn the operator or, in some cases where an approximation will serve, write the largest positive or negative quotient the machine will hold.

Other computers have no built-in protection against an incorrect division. In these cases, division may start with an initial shift since it is assumed that the subtraction of the lined-up divisor and dividend would always yield a negative balance.

The easiest way to obtain a clear picture of the mechanization of division is to follow a problem as it is handled by a typical machine. Figure 3—78 shows, in simplified form, a parallel arrangement for division using a shifting accumulator.

In this case, the accumulator and shifting register connected to it are set up to shift to the left, since the dividend (and therefore all succeeding remainders) will be held and operated upon in the accumulator. There is no actual need for a connection between the least significant place of the accumulator and the most significant of the shifting register, since no information need be transferred from the register to the accumulator. On the other hand, it is easier to shift both with one control signal and no harm is done to the division process by the connection.

As in multiplication, if either number involved in the division is negative, it is complemented to put it in true or positive form before division and the sign of the quotient is corrected afterward (that is, the quotient is complemented). If, however, subtraction is performed by adding the complement (as is most often the case in an accumulator), the divisor may have to be in true form for one step, complement form for the next. For this reason, the divisor in this arrangement is placed in

true form in the storage register and a string of complementing gates is used to obtain either form when needed.

Since the method of nonrestoring division calls for subtraction when the remainder is positive, a sign bit of 0 from the most significant place of the accumulator is used to gate the complement of the divisor, after a delay to allow time for shifting. As an alternative, it is possible to complement the divisor right in its register and then gate whatever the register contains. Both the complementing gates and the easiest method of register complementing (a pulse on the complement inputs of all flip-flops) produce the 1's complement of the divisor, which means extra time for propagation of an end carry. To save this time, a carry of 1 is added into the least significant place of the accumulator whenever the complement of the divisor is used, converting to the 2's complement form.

The sample problem to be solved in the circuitry of figure 3—78 is 0.0101 divided by 0.0110. Using the nonrestoring, pencil-and-paper method, the solution looks like this:

$$\begin{array}{r} 0.1101 \\ 0.0110\overline{)0.0101} \end{array}$$

$$\begin{array}{rl} -00110 & \\ \hline 00100 & \text{A} \\ \\ -00110 & \\ \hline 00010 & \text{B} \\ \\ -00110 & \\ \hline 11110 & \text{C} \\ \\ +00110 & \\ \hline 00010 & \text{D} \end{array}$$

On the assumption that the dividend will always be kept smaller than the divisor, making the sign bit of the quotient at 0, the first step is a shift of one place and a subtraction. Direct subtraction is used here, although in the arrangement of figure 3—78 it will be done by complementing and adding. Remainder A, the result of this first subtraction, is positive, calling for subtraction after another shift. The complement of its sign bit is the first quotient bit to the right of the binary point. Note that in additions or subtractions, carries beyond the most significant place of the shifted divisor do not affect the results and hence can be ignored.

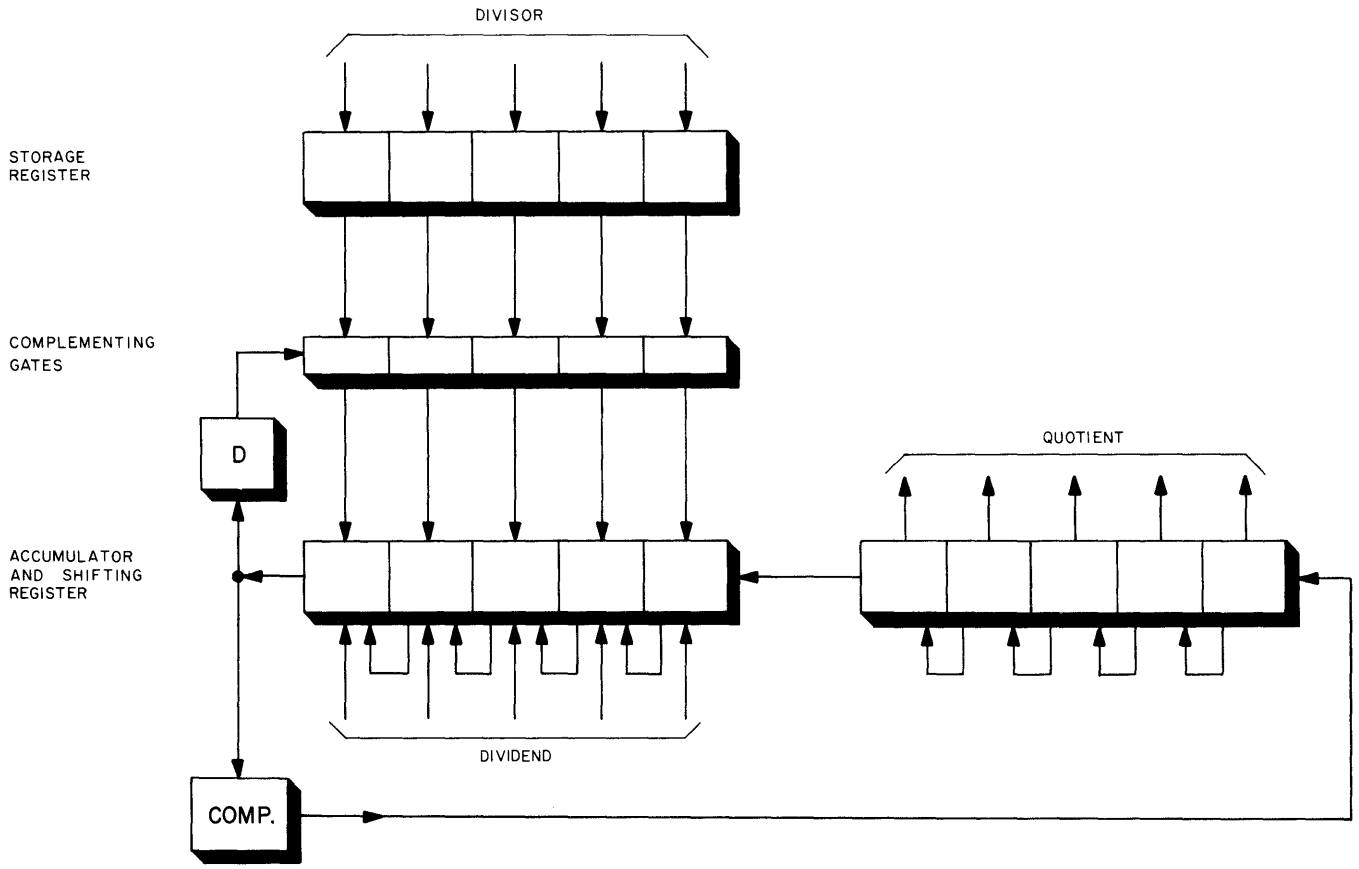


Figure 3—78. Shifting Accumulator Used for Division

In this manner the division process continues—on paper—until a sufficient number of places in the quotient have been filled. The quotient is normally required to have the same length as all other computer words, so in this 5-bit example division is carried to 5 places. The quotient here is 0.1101, but there is also a *final remainder* of 0.00000010 (remainder D). This is simply another way of saying that the division does not come out even at this number of places and that the final remainder is the amount left over. Proving it by multiplying the 5-place quotient, 0.1101, by the divisor, 0.0110, does not equal exactly the dividend.

0.1101	5-place Quotient
$\times$0.0110	Divisor
0.01001110	Product
+0.00000010	Final Remainder
0.01010000	Dividend

The final remainder is ordinarily just discarded, unless there is need for great accuracy in the results of division. In this case, it is possible to store the final remainder separately. When it is thrown away, the quotient may be rounded off by the method described under multiplication, but this requires that the division be carried out to one extra quotient place, and it also means that the quotient must be sent to the accumulator so that the addition can be performed. Other round-off methods, such as forcing or "stuffing" a 1 into the least significant place whether the actual quotient bit there is 1 or 0, do not require arithmetic operations but are less accurate.

Now, to the mechanization of this division problem in the circuitry of figure 3—78. To begin the process, the dividend and divisor are put in positive form, if necessary, by complementing, and a sign flip-flop is set, as in multiplication, to determine whether the quotient must be complemented after it is formed. The accumulator and registers are cleared and the dividend is placed in the accumulator (often by gating it in through the divisor register). The divisor is then entered into its register.

Again assuming that the dividend will always be smaller, there is no need to subtract the divisor from it, so the accumulator and shifting register contents are immediately shifted one place to the left. The sign bit of the dividend is pushed out of the most significant place in this shift, but it is not complemented and sent to the quotient register, since the sign of the quotient is known to be a 0. The complementing gate (labeled COMP.) prevents this from happening. The least significant place of the accumulator picks up a 0 from the shifting register, so the accumulator after the shift contains:

01010

The dividend sign bit, although not sent to the quotient register, is used to start the first subtraction after the shift by gating the complement of the divisor into the accumulator.

01010	Accumulator
11001	Divisor 1's Comp
1	Carry for 2's Comp
00100	Remainder in Acc

The resulting number in the accumulator is remainder A (compare with the pencil-and-paper example shown earlier). As soon as the subtraction is completed, another shift occurs. The 0 from the most significant place of the accumulator is spilled out, complemented to a 1, and sent to the least significant place of the quotient register, which now contains 00001. This most significant 0 is also used to start the next subtraction:

01000	Shifted Acc
11001	Divisor 1's Comp
1	Carry for 2's Comp
00010	Remainder B

Again the shift is made. The quotient bit in the least significant place of the shifting register is moved one place to the left, so room is made for the new quotient bit shoved out of the accumulator and complemented. After a short delay to allow any transients to settle, the next subtraction begins when the complement gates are opened.

00100	Shifted Acc
11001	Divisor 1's Comp
1	Carry for 2's Comp
11110	Remainder C

The shift this time pushes the most significant 1 out of the accumulator. This becomes a 0 in the quotient, and the 1 is used to open the normal gates, bringing the divisor in true form into the accumulator for addition.

11100	Shifted Acc
00110	Divisor
00010	Remainder D

A final shift complements the 0 in the most significant place and sends it around to the quotient register. The bit this time is prevented from gating either the divisor or its complement, however, so the division process ends here.

The entire process can be seen in capsule form by observing the contents of the accumulator and quotient register after each step. The remainders are lettered, as above.

	Accumulator	Q Register	
	00101	00000	Start
	01010	00000	Shift
A	00100	00000	Subtract
	01000	00001	Shift
B	00010	00001	Subtract
	00100	00011	Shift
C	11110	00011	Subtract
	11100	00110	Shift
D	00010	00110	Add
	00100	01101	Shift
			End

The correct quotient is now in the quotient register. If this were supposed to be a negative number, the sign flip-flop would be set to complement it, either in the register or when it is removed.

The mechanization of division in a serial arrangement similar to that shown for multiplication in figure 3–76 follows along the same basic approach as the parallel method described here, with appropriate modifications for the serial requirements.

3.6 CONTROL CIRCUITRY

In all of the arithmetic arrangements, the need is clear for control pulses and signals fed to the proper places at the proper times to open or close gates, to start and stop operations, to transfer numbers from one place to another, etc.

The circuitry that generates, times, and distributes these control signals—usually called *commands*—may be as complex as the actual arithmetic circuitry, especially in a large computer. The generating, timing and major distributing circuits, taken together, are most often called the *control element.* This element must provide commands not only to the arithmetic portion of the computer, but to the main storage, input, and output parts as well.

For a complete understanding of the role played by the control element, it is necessary to recall some of what was said in Part 1 about the manner in which the program govern the operation of the entire computer. The program, of course, is a set of instructions, coded in the form of numbers (words) that are stored in the main storage element or memory in stored-program computers along with the numbers representing information or data.

Regardless of where the program is kept, each instruction word is taken in a separate step (in the proper sequence) by the control element and decoded to see what major operation must be performed next. The control circuitry then develops the complete set of commands or control signals that enable all the small parts of that operation to be carried out by the other elements of the machine.

An instruction, for example, gives the order, "Add." (Usually it also gives the storage address of the number to be added to the contents of the accumulator.) The control element must then put out a series of commands which may be single pulses, levels, pulse series, or combinations. There are as many possible variations in a set of commands to perform the addition operation as there are variations in adder-accumulator circuitry. A sample set, for example, might be a sequence of single pulses meaning:

"Clear addend register"

"Accept number from storage"

"Gate addend into accumulator."

The first would be a pulse on the clear inputs of all flip-flops in the addend storage register, the second a pulse to a set of AND switches to take the number into the register, and the third a pulse to the set of gates between the register and the accumulator. Properly timed and applied, these commands make the circuitry carry out the addition process, using the number at the storage location specified by the address portion of the instruction.

The operation portion of an instruction may, of course, call for any of a number of jobs to be done in the computer, instead of an arithmetic operation. The variety of operations depends upon the capabilities of the machine. One major job in nearly every computer is transferring numbers from place to place, another is control of input-output devices, etc.

3.6.1 Program Control

When one instruction (one program step) has been carried out, the control element must obtain the next from storage. But how does it know the location of the next instruction? Although there are several possible solutions to this, one of the easiest is to set aside a block of memory addresses in which the program is always kept and use an *instruction or program counter* (usually a binary counter) to keep track of the progress of the program and the instruction addresses. An arrangement of this type appears in figure 3–79.

In this (or any other) arrangement, the memory or main storage element must be capable of storing many numbers, including the input information with which the computer is to work, partial results to be held for later use, final results, and—in stored-program

computers—the program itself. There are several types of storage devices (described in Ch 3) but it is easiest to think of the memory as a large bank of pigeonholes or mailboxes, numbered 0, 1, 2, 3, 4, etc., and each capable of holding one computer-length word. The number of each is its address. There must also be selection circuitry to translate a binary address into an actual electrical connection to the correct pigeonhole so that a number can be put in or taken out.

Ordinarily, a block of the lowest-numbered addresses is set aside for the program. Assuming, for example, that addresses 0 through 99 were reserved for this purpose, any program would be stored with its first instruction in address 0, its second in address 1, its third in address 2, etc. Thus, a 67-step program would be stored in sequence in addresses 0-66.

The instruction counter shown in figure 3—79 is cleared before the program begins. Its indication of 000 . . . 000 is sent to the storage selection circuits which quickly make a connection to address 0. The first instruction is taken out and sent to the control element through switching circuits operated by commands issued for that purpose. The time spent in this process of obtaining the instruction is sometimes called a *program cycle*, or an *instruction cycle.*

Once in the control element, the instruction is placed in a temporary storage register called either the *instruction register* or the *operation-address register.* The control circuits issue commands to carry out the operation called for by this first instruction during what is called the *execution* or *operation cycle.* The address part of the instruction goes to the memory selection circuits to obtain the number representing the data to be operated upon. (Or the address may call for a connection to one of the input or output devices to obtain or send out information.)

Toward the end of the operation being performed, a pulse is sent from the control circuits to advance the instruction counter by 1. When the operation cycle ends, control is turned over again to the instruction counter and a new instruction cycle begins. The counter now sends 000 . . . 001 to the storage selection circuits, the second instruction of the program is taken from address 1 to be sent to the operation-address register, and this instruction is then executed.

The process of bringing each instruction in sequence from the memory, executing it, and stepping the instruction counter by 1 continues in this manner until the entire program has been performed or until a *Branch* instruction is encountered. This type of instruction (sometimes called *Transfer* or *Jump)* makes it possible to change a program or repeat parts of it, either unconditionally or under control of the results that have been computed. For example, a branch may be ordered only if the number left in the accumulator is negative. If sensing (checking its state) shows it to be positive, the branch is not made.

When a branch is to be made, the address part of the instruction is taken from the address register and loaded directly into the instruction counter, replacing whatever number was previously there. Therefore, the next instruction taken from memory is not from the next address in the sequence that was being followed, but from the address given by the *Branch* instruction.

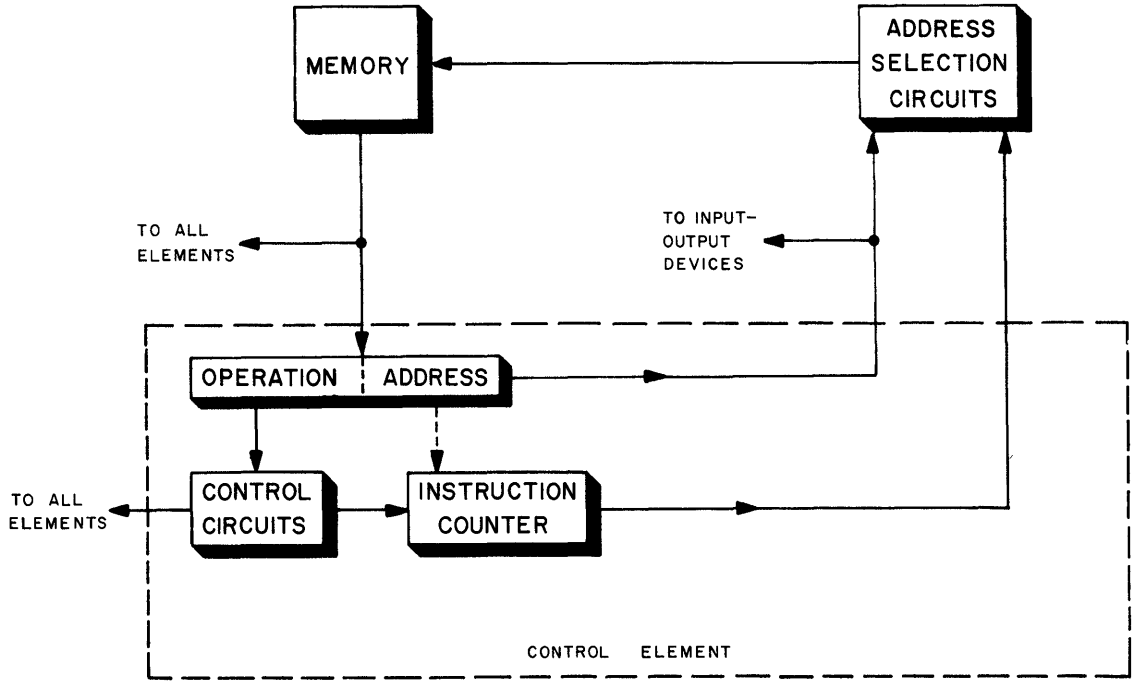


Figure 3—79. Instruction Control

From this point on, the instruction counter is again pulsed once for each instruction carried out, so a fresh numerical sequence of instructions is followed until another branch is ordered. A branch can be made to either a higher or lower numbered step of the program.

In other types of computers—that is, those not storing the program instructions in the memory element—different methods of instruction control may be used. These depend to a great extent, of course, upon the nature of the device or arrangement used to hold the program. Usually, the instructions are made available in sequence and, when each operation is completed, the control element senses the next instruction. This procedure is somewhat similar to the one described for stored-program computers, except that such useful programming techniques as branching are difficult and expensive.

3.6.2 Operation Control

The commands necessary to carry out an instruction have been described as a set of pulses (sometimes other types of signals) sent to the proper places at the proper times. Different sets are needed, of course, to carry out different operations, although certain individual commands may well be used in a number of operations.

There are two basic approaches to the problem of handling operations in sequence. First, it is possible to use a timing arrangement of some sort and rigidly time each separate operation, issuing each command at the proper instant. The timing arrangement is called the *clock* and, because all operations performed in the computer are synchronized by the clock, this is a *synchronous* control method.

In the second method of control, called *asynchronous*, no clock is used for timing operations (although there may be a clock for other purposes). As soon as one operation is finished, a signal is provided to start the next. The timing of commands is done by starting a pulse through a long delay line (when each operation begins) and tapping it off as commands at the proper intervals.

Some computers use one system, some the other, and quite a number of computers use combinations of the two, with synchronous control for short operations but asynchronous handling to speed up the longer ones. Although it is true that operations can often be performed faster under asynchronous control because no fixed time intervals (cycles of the clock device) are used, this type of circuitry is generally more complex than the synchronous.

3.6.2.1 Synchronous Control

The first item of interest in this system is the clock and the method of timing. There are certain time intervals of importance in any computer, of which one is the period required to transmit a single bit of information and to allow the circuits to recover from transients. This period, of course, is one bit-time.

In a serial-mode computer, the bits of a word follow each other on a single wire, and exact timing is so essential (at the inputs to an AND circuit, for example) that a clock must be used to set the basic pulse repetition frequency or bit-time interval. If pulse-type signals are used, clock pulses at bit-time intervals must be distributed throughout the circuitry to provide the frequent reshaping and retiming of information pulses that is necessary. This holds true even if the control element itself is asynchronous in operation and does not depend upon the clock.

In parallel-mode machines, the bit-time interval is usually less vital to successful operation, so the clock pulses need not occur every bit-time. They may instead be produced at some longer or shorter interval more useful in synchronizing the operations of the computer. It is not often that a different interval is selected, however, since the bit-time is the basic measure of the speed with which numbers can follow one another in the circuitry and therefore is one controlling factor in obtaining the fastest possible operation.

Whatever the basic interval selected, the clock pulse intervals, of course, bear some relationship to *real time*—that is, time in the outside world. Many types of problems solved by computers involve keeping track of real time. Military weapon-control computers, for instance, must solve time-speed-distance problems against an incoming enemy, while computers operating various types of machines must often time the operations. So, it is frequently valuable to select a clock pulse interval that can be easily converted to real time. A fairly common clock rate, for example, is 1 megacycle, which means pulses at 1 microsecond intervals. Using decimal counters or other circuits for frequency division, it is possible to obtain pulses at 1-second intervals for useful time-measurement in the computer.

The device which produces the clock pulses is an accurate oscillator of some sort, generally crystal-controlled, followed by amplifying and pulse-shaping circuits.

It becomes apparent, however, that the continuous stream of clock pulses thus produced must still be counted or somehow kept track of. How many clock intervals are required to perform addition or division, for example? How many for an instruction cycle, for transferring a number? It is obviously impractical to run a continuous count of the clock pulses—in 10 seconds of operation, the count would be between 5 and 20 million for most computers.

A cycling count, on the other hand, proves eminently practical. By counting a given number of clock pulses and then beginning the count over again, the

passage of time in the computer is divided into intervals of useful length that can be called *clock cycles*, or *machine cycles*. A simple ring counter can do this and provide a different one-hot output for each step of the count.

The number of clock pulses to be counted in the clock cycle may depend upon several factors. In a serial-mode computer, the word length is quite closely related to the speed of operation since a 20-bit word, for example, requires one word-time, or 20 bit-times, to be transferred past a given point in the circuitry. Thus, one word-time might be chosen as a useful length for the clock cycle—or it might be 2, 6, or 10 word-times if one of these lengths were more convenient. A word-time is of no importance in a parallel-mode computer, since it is the same as a bit-time.

A factor that must be considered in all machines, parallel or serial, is the time required to transfer numbers into or out of memory, called *access time*. This is the principal limitation on the speed of operation because each number to be used in computation and each result must go through this transfer.

The access time in most computers is somewhat longer than the time needed for the shorter operations such as addition, subtraction, shifting, etc. By making the length of the clock cycle equal to the access time, the all-important transfers of numbers and the shorter operations can be performed in one clock cycle each. When one of the longer operations such as multiplication or division must be done, the required number of complete clock cycles is allowed for it.

Using an arrangement of this sort, illustrated in figure 3—80, two complete clock cycles are the minimum required to carry out any instruction. The first must be a program or instruction cycle, to get the instruction out of memory and load it into the operation-address register. The one-hot signals from the various stages of the clock ring counter are gated or otherwise switched to provide commands controlling the address selection circuits, etc., to obtain the instruction.

When this instruction cycle is nearly done, a command sets a flip-flop or switch to make the next an operation cycle. Now, the circuits gating the clock ring outputs come under control of the signal representing the operation called for and the result is the set of commands necessary to perform that operation. The commands must be timed to allow the maximum time for each part of an operation to be completed, plus a safety factor. In binary addition, for example, time must be allotted for the propagation of a possible carry from each place, even though no carries at all may occur in some problems. If more than one operation cycle is required, a simple counter keeps track of them and changes the gating for each cycle.

When the operation is almost completed, the switching is reset for another instruction cycle and the instruction counter is advanced one step, to take the next instruction from memory. In this arrangement, all program or instruction cycles are identical, but the operation cycles depend upon the operation to be performed. All commands are provided by the timed pulses from the clock ring, switched through the gating circuitry under control of either the operation signal or

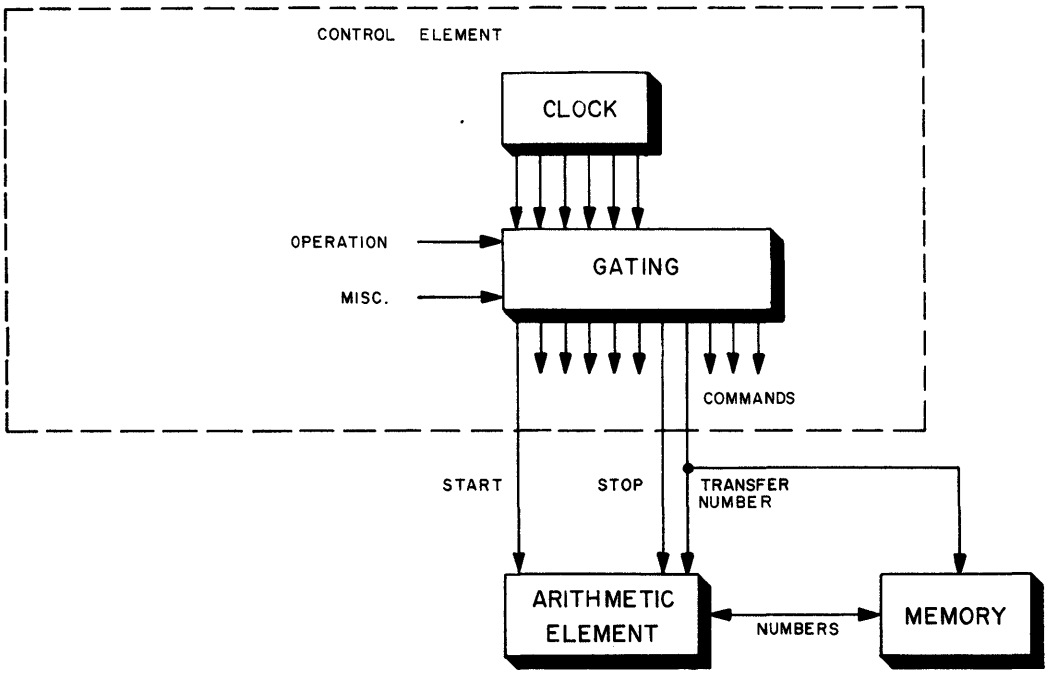


Figure 3—80. Synchronous Control of Operations

various miscellaneous signals, such as the one that determines whether an instruction or an operation cycle is required. The operation signal is translated (usually in a matrix) from the operation portion of the instruction, placed in the operation-address register (fig. 3–79).

3.6.2.2 Asynchronous Control

There are a number of possible variations of the asynchronous control method. The basic approach appears in figure 3–81, showing how the timing of commands is accomplished through the use of a long delay line, instead of a clock.

The delay line consists of a number of long series paths of delay circuits (and switching circuits where necessary). Commands are tapped off between circuits at the required intervals.

An operation requiring 24 bit-times for its operation cycle, for example, could be handled by a string of delays totaling 24 bit-times. Commands might be taken off at the end of 1, 9-1/4, 16, 20-1/2, 21, 22-3/4, and 24 bit-times, as shown at (a) of figure 3–82.

A pulse is inserted in the delay line when the operation signal appears. One bit-time later, it emerges from the first delay circuit, goes out on the first command line, and also enters the next delay section. Perhaps this first command might clear a storage register and start the address selection circuitry through the process of taking a number out of memory. The pulse emerges from the second delay section 9-1/4 bit-times after the operation was started, and is sent out on another command line to perform some other control function. In this fashion the pulse travels through the entire length of the delay line and is tapped off at intervals to form the commands. From the end of the line the pulse goes to an OR circuit at which all lines meet and emerges from that as a command calling for the next instruction.

Although it might seem necessary to have one delay line for the process of obtaining instructions and one for each different operation, this would require an unnecessary amount of circuitry. Actually, many operations are quite similar and the parts of longer ones are often repeated, so it is possible to use what amounts to one long delay line with switching circuits, alternate paths, and feedback loops to produce commands for all operations.

One small portion of such a line is shown at (b) of figure 3–82. The heavy, downward-pointing arrows are commands and the input leads labeled X are controlled by the operations called for. Some, but not all, of the X leads are energized when a given operation is to be performed. The operation of the circuitry is straightforward and should be clear enough, except possibly for the feedback loop involving the counter. If a pulse enters this loop through the OR circuit and finds

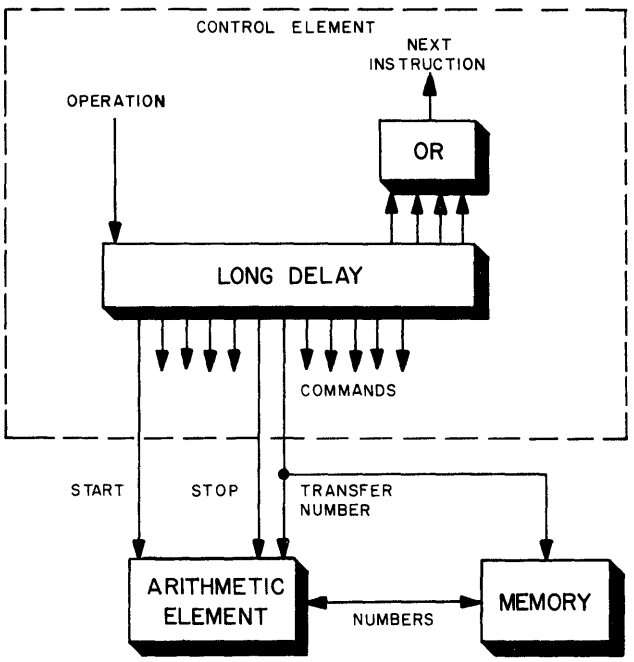
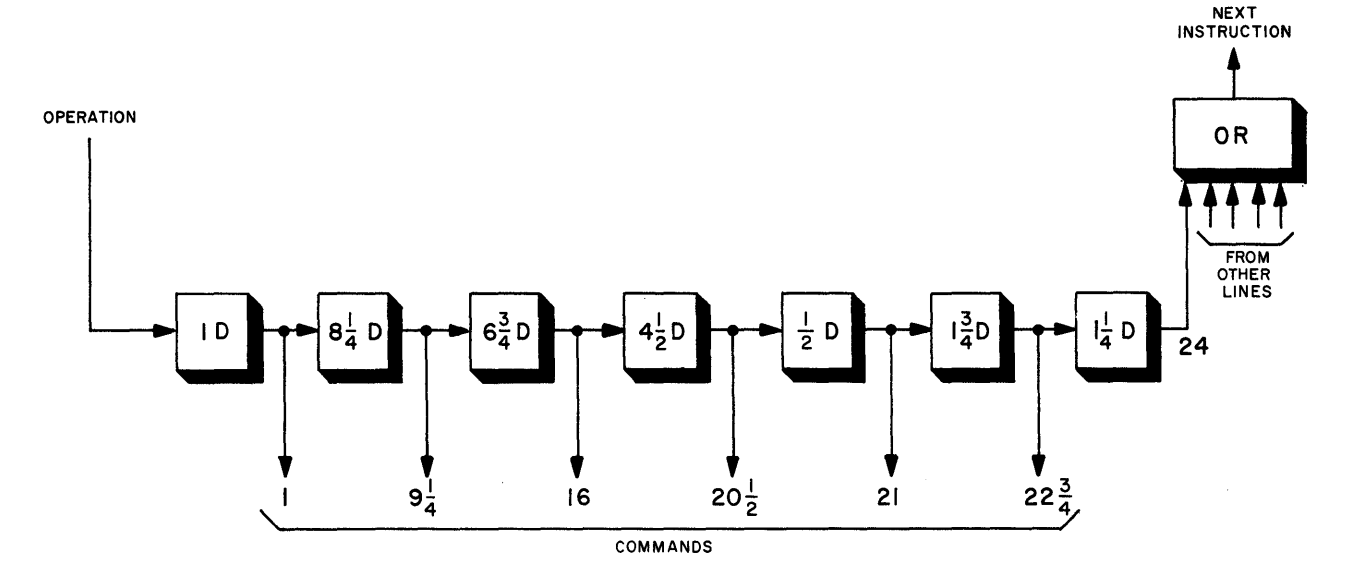


Figure 3–81. Asynchronous Control of Operations

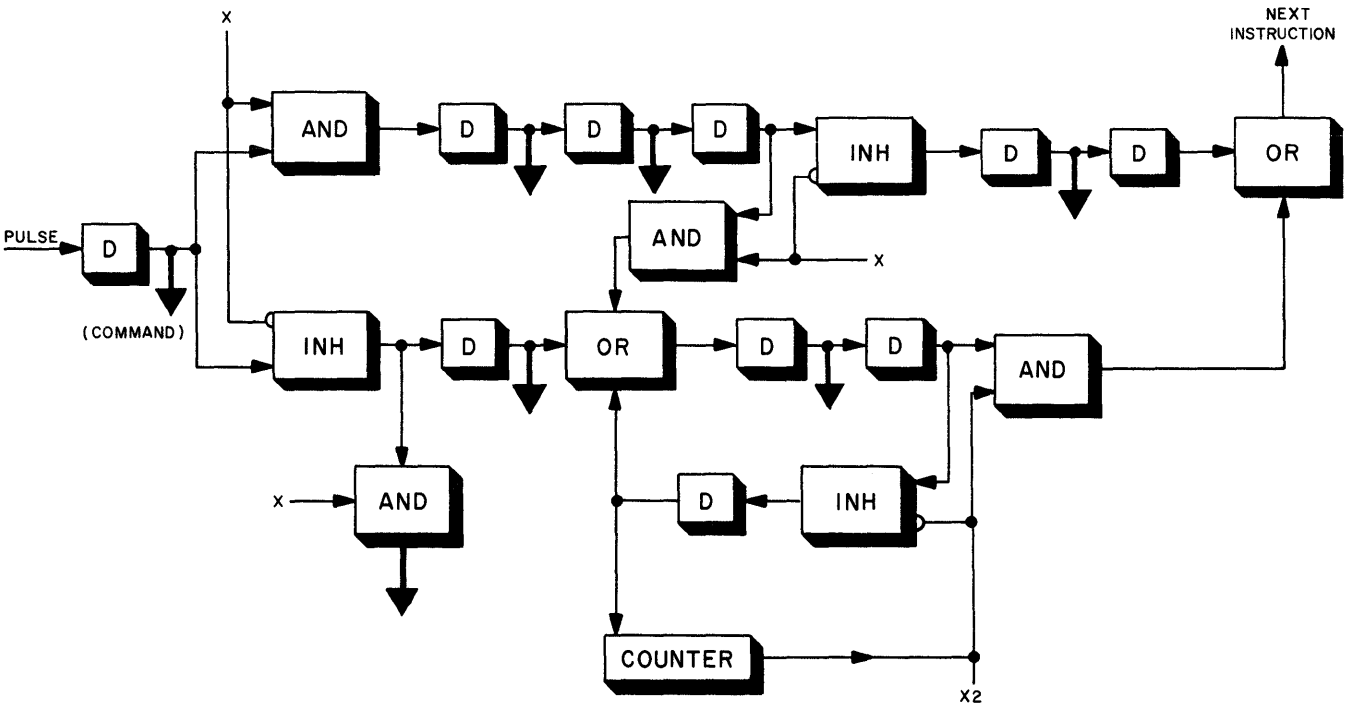
that operation signal X2 is present, the feedback loop through the INH is opened and the pulse goes out without additional delay through the AND circuit. If X2 is not present, however, the pulse cannot get through the AND but must enter the loop through the INH and circulate, being delayed by the amount of time needed to travel the loop. Each time the pulse returns to the OR circuit, it also steps the counter. When a predetermined count is reached—meaning the pulse has circulated and been delayed this many times—the counter output opens the loop and enables the pulse to exit through the AND circuit.

Thus, one long delay line with many possible paths and loops enables the commands for any operation to be produced. A common path at the beginning of the delay line produces the commands necessary for procuring each instruction. The timing of each operation is exactly what it requires, not a number of fixed-length cycles (with the possibility of wasted time if the actual operation ends before the end of the last cycle).

This idea of asynchronous control can be extended in some types of circuitry by letting the control pulse run through the circuits themselves as the operation is being performed. An asynchronous adder, for example, can be built to allow the control pulse to run with the carries. As soon as carry propagation is completed, the control pulse is returned to the control element as a "next instruction" command. Multiplication can be handled in similar fashion. In this type of arrangement, the control element tells the arithmetic element, "Start," and then waits for a signal to come back saying, "Finished."



A



B

Figure 3–82. Delay Lines for Asynchronous Control

CHAPTER 4
LARGE-SCALE STORAGE AND MEMORY

4.1 REQUIREMENTS OF MEMORY ELEMENT

The simple description in Chapter 3 of the memory or main storage element of the digital computer referred to it as a bank of pigeonholes or mailboxes, each with its own, separate address and each capable of storing one computer-length word.

Without a memory such as this, the automatically sequenced computer could not perform a long string of arithmetic computations and logical decisions without stopping. It must have some place to keep all the input information it has to work with, the intermediate results that will be used in later computations, and the final results that have to be fed out when the program calls for outputs. In a stored-program computer, the principal type under consideration here, the memory must also provide enough pigeonholes (actually word-length storage registers) to hold all the instructions of the program.

The first requirement of the memory element, then, is size. It must contain enough storage registers to hold all the data and all the instructions of the program. In a very large computer, this may not be feasible and auxiliary storage space may be provided outside the main memory element. In this case, the memory must be sufficiently large to hold information and instructions enough to keep the computer running for a reasonable length of time.

While it would be entirely possible to build a memory of flip-flop registers, like those used elsewhere in the computer for temporary storage purposes, this brings up the question of physical size, especially if vacuum tube flip-flops were to be used. From a few hundred to many thousand registers may be needed in the memory, and the sheer bulk of many flip-flops, plus the power required to operate them, makes their use impractical. Some early computers did use this type of storage, but newer devices require far less space and power for the same amount of storage.

Another important requirement of the memory element is the speed with which numbers can be put in or taken out. As mentioned in Chapter 3 this *access time* largely controls the speed of operation since many operations can be performed faster than the numbers can be obtained to work with. So, a fast-access memory is required and, once obtained, another reason for storing the program instructions in memory becomes apparent. For maximum operating speed, the instructions must be made available just as fast as the numbers to be operated upon, so the logical place to keep them is with the data numbers.

Part of the access time (sometimes called the *memory cycle*) must be used to translate the address (also in number form) and set up electrical connections to the desired storage register, in order to write in or read out a number. *Writing* or *storing* is the process of putting a number into a storage location; *reading* is the process of taking it out.

Translating the address in the address selection circuits rarely takes long, but actually reaching the proper storage register, when some types of storage devices are used, may require much time. The effect is as if the registers were seats on a merry-go-round or cars on a roller coaster and it were necessary to wait for the desired one to come by. Thus, the computer may be forced to wait for the information it is to work with. Certain techniques in preparing programs can be used to cut this access time to a minimum, but these techniques often make the problem of writing the program very complex.

A better solution is to use fast-access storage devices for the main memory and use the slower devices as auxiliary storage facilities outside the computer proper, reached through input-output circuits. ("Memory" usually means the main storage element inside the computer.) Then, large groups of numbers at a time can be sent back and forth, as required, and stored in consecutive storage registers. Sometimes the computer can continue its computations during the transfer. Instead of having to locate individual registers in the auxiliary storage, the access is made to large blocks of registers.

Only the main memory is used for all operations going on inside the computer. When the memory fills up with intermediate results, instructions send a large block of them out to the auxiliary storage and may bring back in some fresh data or even additional program instructions, as required.

Four types of storage devices are in most common use today: magnetic (ferrite) cores, magnetic tapes, magnetic drums, and electrostatic storage tubes. Others, such as the acoustic delay line, have been used in some machines and will be described briefly, along with punched paper cards and tapes.

The internal organization of a computer is affected to some degree by the type of storage device (or de-

vices) used, not only because of the physical differences between devices but because of the access problems.

Of the magnetic storage devices (cores, tapes, and drums), the cores offer the easiest and by far the fastest access to any storage location. In fact, ferrite cores and electrostatic storage tubes are probably the most satisfactory of present storage devices for use in the internal computer memory, since all registers are equally accessible. The other types of storage devices generally require the computer to wait for the transfer of information, but they offer such other advantages as low cost, fast serial operation, or easy changing of the stored information by an operator.

4.2 MAGNETIC STORAGE

Magnetic storage takes two principal forms: one stores the individual bits in separate cores, as described in 2.1.5 and figure 3–33 of Chapter 2; the other stores each bit by magnetizing a separate, tiny spot of a magnetic material coated on the surface of a plastic tape or a metal drum. In both forms of storage, the magnetic field that is left (remanent flux) after writing the information indicates by its direction (polarity) whether a 1 or a 0 is stored.

The magnetic material coated on the surfaces of tapes or drums must, like the core materials, have a nearly rectangular hysteresis loop so that it will retain most of the flux impressed upon it after the magnetizing force has been removed. Thus, the magnetized portion of the material acts like a permanent magnet, the direction of whose field can be reversed by applying a second magnetizing force of sufficient strength. This external force is usually a temporary magnetic field about a coil through which a pulse of current is passed. The magnetic coating for tapes is normally one of several iron oxides, finely powdered and mixed with a binder or adhesive that is dried under controlled conditions to hold the oxide particles in a thin, even film or coat. The drum surface is usually a plating of a metallic alloy, such as nickel-cobalt.

A coating of such a material on a surface that is relatively flat does not form a closed magnetic circuit for small fields (as the closed ring of a core does), so separate areas of the surface can be magnetized in opposite polarities without interfering with each other, as long as there is sufficient distance between them. If the applied magnetizing force is kept in a very small field, only a correspondingly small spot of the tape or drum coating is magnetized and more bits can be stored on a surface of given size. To magnetize different spots, either the coil providing the magnetizing force or the coated surface could be moved, but in practice the surface is always moved past the stationary coil at a constant speed, and writing and reading are done with the surface in motion. Long lengths of magnetic tape are wound on compact reels and pulled past the coils used for reading and writing (called *magnetic heads*). A magnetic drum revolves on its axis, passing its coated cylindrical surface under fixed heads.

The problem of holding the applied magnetizing force or field to a very small area is solved in the following manner. It is known that if a coil is wound on one side of a rectangular or ring-shaped core of magnetic material, as in figure 3–83, the core forms a closed magnetic circuit. When the coil is energized, the field set up finds it much easier to complete its circuit through the core material than through the surrounding air; hence virtually all the lines of force remain in the core. If the core is cut through at one point, forming a gap, the flux lines jump across the gap to complete their circuit.

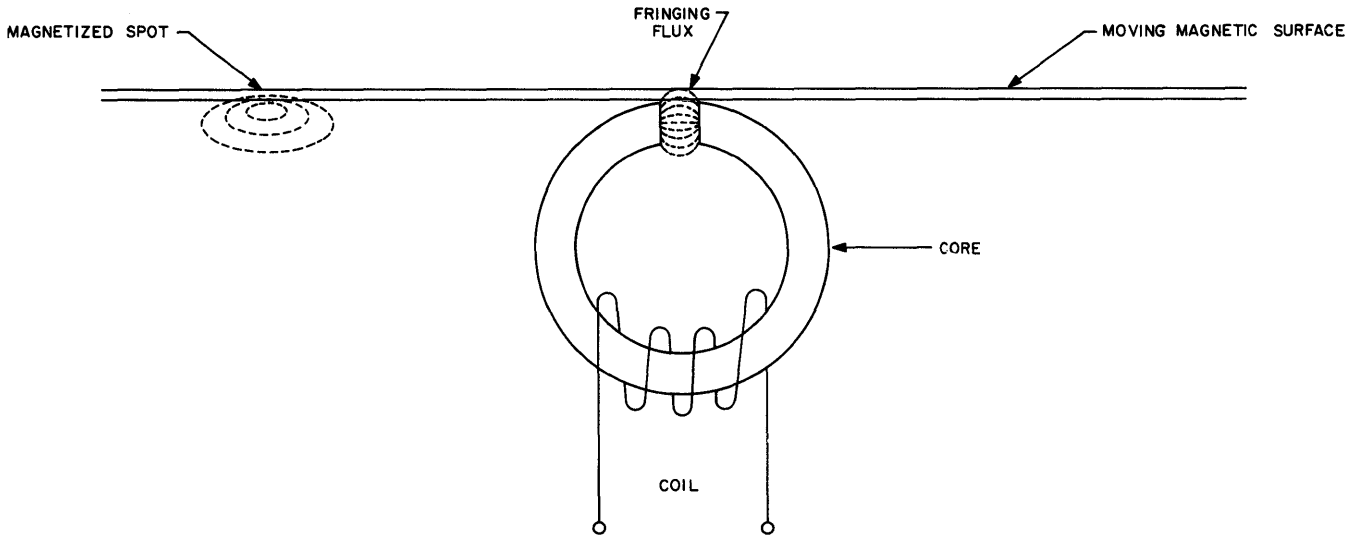


Figure 3–83. Magnetic Head

As shown in figure 3–83, not all the flux lines jump straight across the gap. Instead, because the air offers greater reluctance (magnetic resistance) than the core, the field tends to spread out, or expand in cross-section, through the gap. The portion of the field that arcs out beyond the core area is called the fringing flux. Now if a magnetic surface is pulled past the gap, very close to it, the fringing flux produced by a pulse of current in the coil magnetizes a very small spot of the surface. When the short-duration pulse disappears, the motion of the surface pulls the spot away and then another current pulse can magnetize another spot, etc.

The magnetic head is made in this manner. The gap is usually about 0.001-inch wide, so the fringing flux is held to a very small area and, in practice, up to about 100 bits can be stored as magnetized spots in an inch-length of surface.

When the small magnetized spots representing stored bits of information are passed again under the head, each tiny magnetic field enters and travels quickly around the magnetic circuit of the core, inducing a voltage pulse cycle into the head coil. The bit is identified (by reading circuits) as a 1 or a 0, depending upon whether the negative or the positive peak appears first. The magnetized spots are unchanged by reading, so this is nondestructive readout; i.e., the stored information remains on the tape or drum.

Any stored bit can be changed from 1 to 0 or from 0 to 1 simply by writing over it. If a 0 is stored by passing a negative current pulse through the head, it is changed to a 1 by applying a positive current pulse the next time this spot passes under the head. The resulting applied field, opposite in direction to that of the spot, switches the remanent flux in the spot just as a core is switched, leaving a 1 stored where the 0 had been.

It is also possible to remove all stored information from a tape or drum surface by *erasing*, leaving a blank, unmagnetized surface. This is done by applying to the surface an a-c field strong enough to produce saturation, then reducing the field strength gradually to zero. At saturation, the field overcomes all previously written magnetized spots and fills the entire surface under its influence with flux, switching the flux direction rapidly as the field reverses in polarity. As the field strength is reduced, it applies less flux to the surface on each reversal—enough to switch the remanent flux but less than the amount previously applied. The result is less remanent flux retained in the surface after each polarity reversal, until both the applied field and the remanent flux in the magnetic surface are reduced to zero.

Erasing of drums is not usually necessary to change the stored information, which can be simply written over (if it can be located easily), but is used to rid the surface of noise caused by stray magnetic flux picked up over a period of time. On tapes, where a single word or bit is difficult to locate, erasing is used to wipe out old information, a complete block at a time, to make way for new data.

4.2.1 Magnetic Tapes

The oxide-coated plastic tapes used in computer work may be the standard quarter-inch widths used in home recording and other applications, or they may be special types up to about an inch in width.

In any case, there must be a tape transport or drive mechanism to hold the full and empty reels and to pull the tape past the head (or heads) used for reading and writing. This mechanism must be capable of moving the tape at a rigidly controlled speed for writing or reading operations, which generally require careful timing. It must also be capable of very fast starts and stops and of high-speed forward and backward winding to locate a block of stored information or a blank space into which new information is to be written. All of this must be accomplished with minimum danger of damage to the thin tape.

There must be control circuitry to make the tape drive mechanism perform the desired operations when called for by the computer. A few manual controls are generally necessary, as well, since the operator must be able to take control for changing tapes and other operations.

In addition to the mechanical operations, there must be circuitry for the electrical functions of writing and reading, synchronizing, etc. One basic arrangement of a tape storage element appears in figure 3–84.

Magnetic tape is excellent for storing large amounts of information whenever rapid access is not required. If the information desired is at the opposite end of the tape from that under the heads, it is generally a matter of seconds before it is reached, during which time the average computer could perform thousands of operations. For this reason, access is usually programmed and blocks of information may be stored in such manner that they can be counted to locate a given one.

Ordinarily, neither reading nor writing is done continuously, so the tape movement is stopped after each operation to avoid wasting long lengths of tape. Accordingly, spaces of blank (unrecorded) tape are left between blocks of information to allow for starting and stopping times, since the drive mechanism cannot react instantaneously to start or stop commands.

When information recorded on the tape has to be changed, the common method is to erase and rewrite the complete block of information in which changes must be made, because of the difficulty of locating individual bits or words.

Since tape reels can easily be changed by an operator, the tape element is often used as an input device.

Arithmetic programs and data, programs for checking the operation of parts of the computer, lengthy mathematical tables, and other input information can be made available to the computer in this manner.

4.2.2 Magnetic Drums

Though tapes are valuable for storing large amounts of information, when it is essential to write and read information at frequent intervals and in random order, magnetic drums offer much faster access times, commonly ranging from 10 to 40 milliseconds. Because the information is stored on the surface of a cylinder revolving under fixed magnetic heads, the drum provides a form of *cyclic storage* (once written, a word comes back under the heads on every revolution).

As the drum rotates, the area in which a single fixed head can write or read is only a very narrow strip—called a *track* or *channel*—running around the circumference of the drum. Information can be stored in serial form simply by sending serial words to the single head while the drum revolves (translating 0's and 1's to current pulses of the proper polarities). The bits of each word are then stored as a sequence of magnetized spots along the single channel running around the drum.

Another common storage method is parallel storage, shown in figure 3–85. To store a 5-bit word by this process, five heads are lined up side by side, each writing in a separate channel. The translated current pulses representing the bits of the word are sent in parallel form to the heads and the bits are written simultaneously. Now, the bits are stored as a row of magnetized spots in adjacent channels. So, in this method, the registers are strips of drum surface running toward the ends of the drum and including as many channels as there are bits in the computer word. In the example of figure 3–85, a register stretches across five channels. The band of registers extending completely around the drum is called a *field.*

Typical drums used with the AN/FSQ-7, -8 measure 10.7 inches in diameter and 12.5 inches long. Using 33-bit words, one of these drums holds six fields of 2,048 registers each, for a total storage capacity of 12,288 words.

Locating a given register or group of registers on the rotating drum to read or write information requires some means of keeping track of the drum position. One common method uses a special *timing channel* in which is written either a series of 1's or a regularly repeated combination of 1's and 0's. These bits are read by the timing channel head and used to synchronize the access circuitry with the drum rotation and to locate registers by a cycling count. A special combination of 1's and 0's at one point on the track can be used as an index mark to tell the circuits that a new revolution of the drum is beginning.

In this method, each register in a field has its own address, and a given register is located by first selecting the proper field (by switching connections to the heads), then selecting the register by address. The method, sometimes called *address selection*, requires a circuit arrangement similar to that shown in figure 3–86.

It is entirely possible, of course, to write information on the drum with one set of heads and to read it with another set positioned over the same field. One reason for doing this may be to use the drum as a time buffer, or isolating device, between the fast-operating computer and much slower input or output devices.

With this method, writing is a matter of looking for a register in which to put new information and

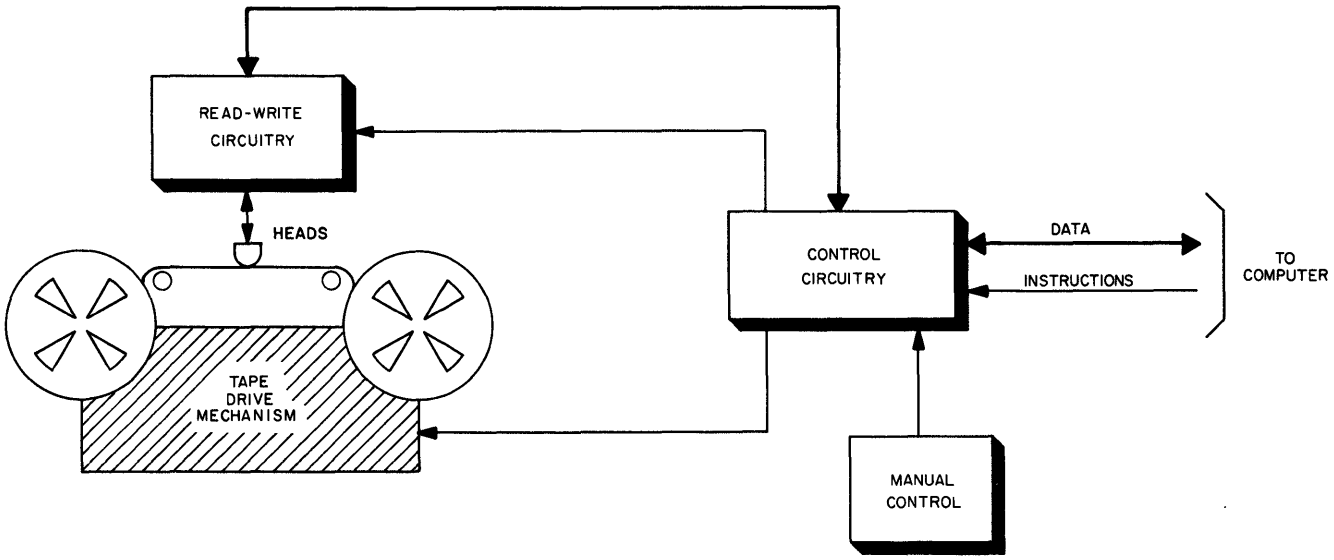


Figure 3–84. Basic Tape Storage Arrangement

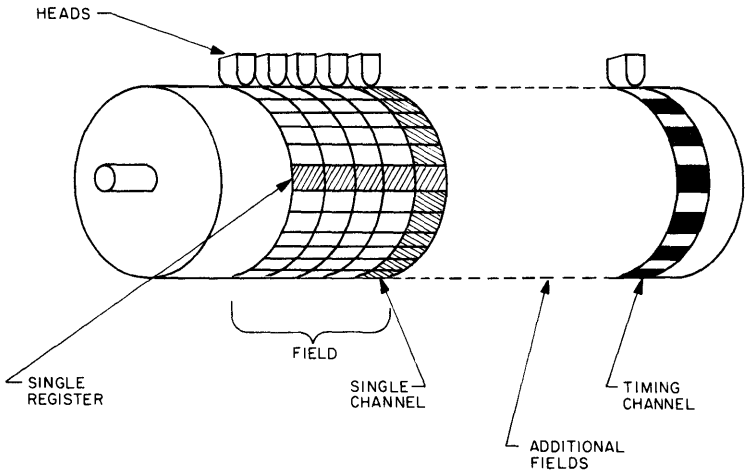


Figure 3–85. Storage on Magnetic Drum

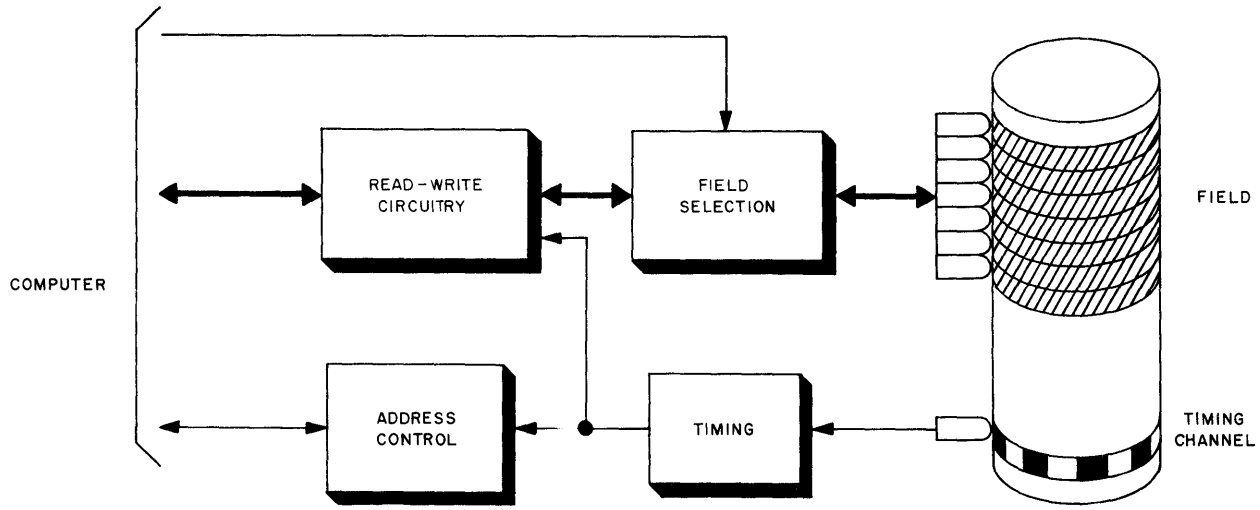


Figure 3–86. Address Selection of Drum Registers

reading is concerned only with taking out information that has not been read before. Each set of circuitry, therefore, is interested in the *status* (state or condition) of each register in the field. Furthermore, each side is able to tell the other what it needs to know. This can be done by using two extra drum channels for control, a write channel and a read channel (fig. 3–87).

When the writing side has information to store, it must locate one or more registers that are empty; that is, contain information that can be written over because is has already been read. To indicate the latter, the reading side inserts a 1 in the write channel each time its reads a register. Thus, when the writing side finds a 1 in the write channel, it is free to write fresh information into the corresponding register. As it does this, it inserts a 1 in the read channel, telling the reading side that this register now contains information that has not been read. A 0 in either channel tells the circuits that the corresponding register should not be written in or read.

This process is called writing or reading by *status*. Timing is necessary, as shown in figure 3–87, but in this case only to synchronize the access circuits to the drum. Writing and reading are not done continuously. When input information is available, the writing side finds empty registers and writes it into them. When the computer wants more input information, the reading side locates registers containing it and reads it out to the computer. In output operations, of course, the computer side does the writing and the output side the reading.

Status and address are two of the principal methods of reading and writing on drums, using them either as the main memory or as auxiliary storage. There are a great many possible variations in the details, using either method.

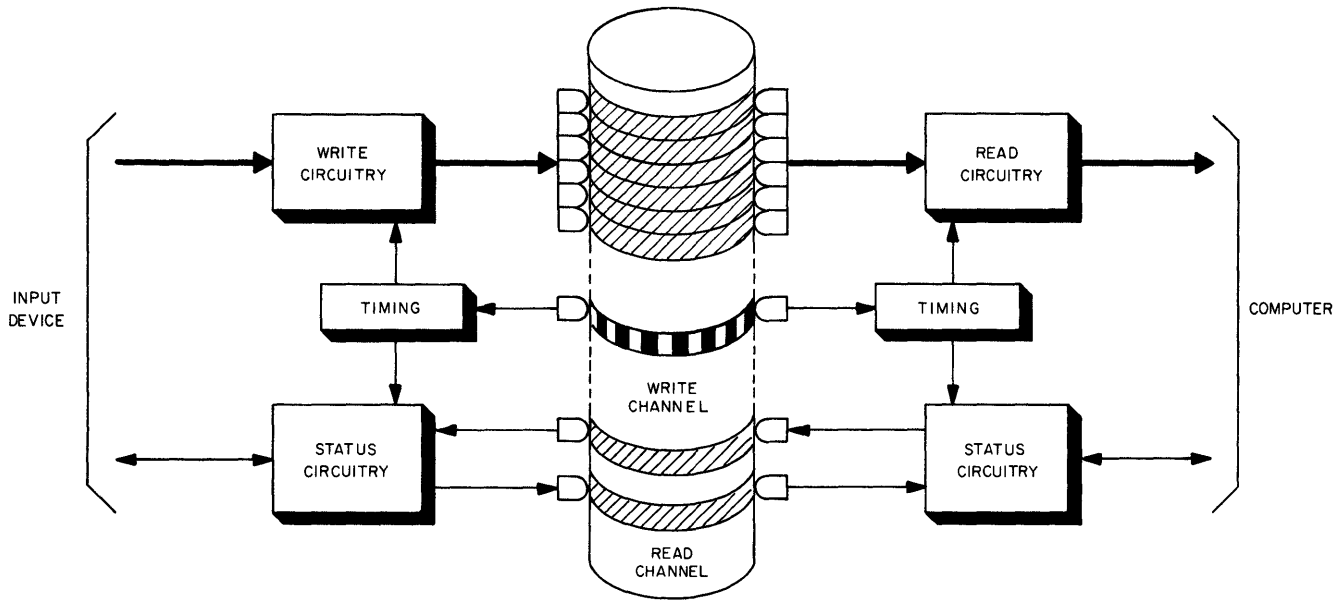


Figure 3—87. Writing and Reading by Status

4.2.3 Magnetic Cores

Ferrite cores are probably the best present-day devices for use in comparatively large, fast-access memory elements. No other device offers faster access time and the core registers can all be reached with equal speed and ease.

The theory of operation of cores has been covered in Chapter 1, including the description of core registers. These discussions were concerned primarily with tape cores, but all the principles apply equally well to the ferrite cores used in the memory element. The two types differ physically in that the ferrite core is a ceramic-like material, rather than a metal, and can be made very small for memory use. The required coil can be replaced by a single wire threaded through the open center of the core.

When a number of core registers are grouped together, as must be done in the memory element, the problem of reaching the individual cores of a given register to insert or remove a word becomes a little more difficult than in a single register. The principle emphasized in Chapter 1 is used to solve this—the principle that a magnetizing force of H is more than enough to be certain of switching a core, while a force of H/2 definitely will not switch it. These forces must be closely controlled, which can be done by controlling the currents used to set up the fields. If a current, I, sets up a field of force H about a conductor, then a current of I/2 will set up a field of H/2.

Using this principle, four registers grouped side by side in a 2-dimensional array (sometimes called a *memory plane*) are shown in figure 3—88. This is called a 4-by-4 array because there are four cores on a side.

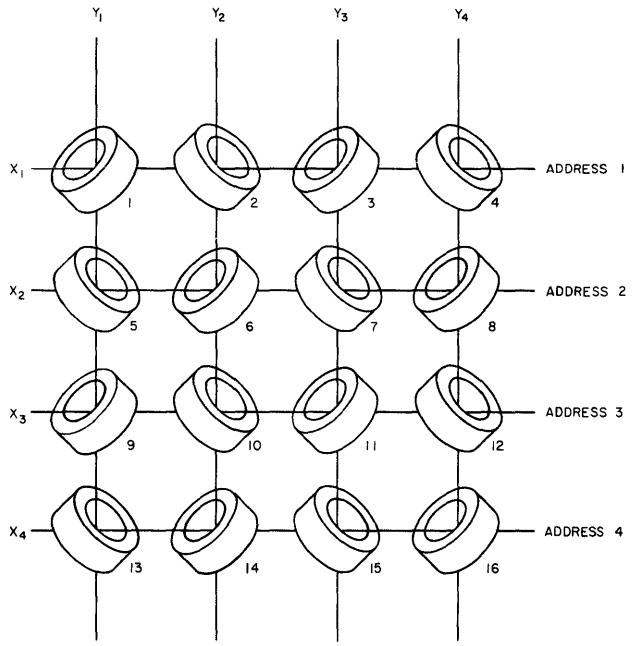


Figure 3—88. Core Memory Plane

Each horizontal row of cores is a separate, 4-bit register. The leads labeled X and Y are the wires that make up the core windings. The X windings of all cores in a single register are in series, while the Y windings of the cores in each vertical column are series-connected. Each X line, therefore, selects a particular register, and each Y line selects a certain bit position in all registers.

To write the word 0001 in the register at address 3, for example, a current pulse of I/2 is placed on the X_3 line to select the desired register (all registers are as-

sumed to be cleared). At the same time, the word is applied in parallel form to the Y lines, using current pulses of I/2 to represent 1's, blanks to represent 0's. The word 0001, therefore, is entered as a single half-current pulse on the Y_4 line.

Now, both the X and Y windings of core 12, at the intersection of X_3 and Y_4, receive half-current pulses, setting up a field about each winding of force H/2. The windings are so arranged that these fields add, producing a total magnetizing force of H, which switches core 12 to the 1 state. Notice that cores 9, 10, and 11 on the X_3 line each received a half-current pulse in its X winding, but none in its Y winding; hence these cores were not switched. Cores 4, 8, and 16 had their X windings energized but not their Y windings, so these were not switched either. Since the write operation was being performed, a half-current pulse on any line can be called a *half-write* pulse. In reading, it is called a *half-read* pulse.

To read a word out of a register, one additional winding is needed in each core. This is the *output* or *sense* winding. (For the sake of clarity, it is not shown in fig. 3—88.) Since only one register at a time is read, all sense windings from a given bit position can be connected in parallel to a single output terminal. That is, the sense windings from cores 4, 8, 12, and 16 on the Y_4 line, for example, are paralleled, so that the least significant bit from any register being read appears at this output terminal (Y_4). Similarly, the sense windings from all cores on the Y_3 line are brought to a separate output terminal, those on the Y_2 line to another, etc.

For the reading process, the desired register is selected by a half-read pulse placed on the proper X line, and half-read pulses are applied to all the Y lines. These half-read pulses are opposite in polarity to the half-write pulses. In other words, the magnetic fields set up by the half-read pulses tend to set the cores to the 0 state.

Again using the register at address 3 as an example, to read out the word previously written in, half-read pulses are sent to the X_3 line and to all the Y lines. Each core in register 3, therefore, receives a half-read pulse in both its X and its Y winding. The resulting fields add in a direction that sets the core to 0. The word stored in register 3 is 0001, so cores 9, 10, and 11 are already in the 0 state and are not switched. Core 12, however, is storing a 1 and the combined half-read pulses switch this core to the 0 state, producing a pulse on the output line from the Y_4 bit position. Thus, the word 0001 that was in register 3 appears at the ouput terminals as a single pulse in the least significant place. The register itself is cleared (all cores reset to 0).

The cores in registers 1, 2, and 4 are all half-selected by the half-read pulses on the Y lines, but since no pulses are sent through their X windings, they are not switched. Notice that the process of writing also must produce pulses in the sense windings of any cores that are switched to 1. These pulses, however, are opposite in polarity to those produced in the reading process and can be either blocked or ignored at the output terminals.

The fact that a register is cleared by the process of reading a word out of it is not good. This is destructive readout, as it is usually desirable to continue storing the word in the register after readout, since any one piece of information might be needed a number of times in the course of a program. The solution is to write each word back into the register from which it came immediately after reading it out. At the same time, of course, it is also sent to the arithmetic element or wherever it is needed. The necessity for rewriting each time a word is read out slows the access time since the memory circuits cannot handle another operation until rewriting is completed. Nevertheless, the total access time using ferrite cores is usually 10 microseconds or less.

An arrangement like the single memory plane shown in figure 3—88 is sometimes used in other parts of the computer to store a group of words and read them out in sequence, on demand, to some slower-acting circuitry or output device. This can easily be done by connecting the outputs of a ring counter to the X lines. The ring counter is designed to produce half-current pulses as its outputs. Another circuit is arranged to distribute half-read pulses to all Y lines each time the one-hot output of the ring counter is shifted during reading. In writing, these are suppressed (inhibited).

Using this arrangement, the computer (or other circuitry) loads the memory plane by pulsing the ring counter once as each word to be stored is sent to the Y lines. Each time the ring counter is pulsed, it applies a half-current pulse to the X line of a different register, so words are written into one register after another until the plane is filled. The ring counter serves to select the registers in sequence. During this writing process, the half-read pulses are suppressed.

When the memory plane is loaded and the output device or other slow circuitry is ready to receive the first stored word, it sends a control pulse to the ring counter. (Since the pulse is sent to obtain data, it is often called a "demand pulse.") The counter output shifts back to the first register in the plane and half-read pulses are gated to all the Y lines, so the word in the first register is read out. When this word has been processed in whatever manner is necessary, another demand pulse is sent to the ring counter and the word in the second register is read out. The memory plane, used in this fashion, can accept words at the fast rate dictated by the computer and give them to other cir-

cuitry at a much slower rate. Tape cores can be used instead of ferrite cores.

Returning to the main memory, the fact that words must be rewritten because of the destructive readout does lead to one advantage, which is that each memory operation, reading or writing, can consist of the same sequence of events with minor variations, somewhat simplifying the access and control circuitry.

For a reading operation, the desired register is read and the output word is sent where it is needed and also written back into the same register. On a writing operation, when a new word is to be placed in a register, the desired register is read to clear it but the word it contained is discarded and the new word is written into it instead. (Of course, a register is not selected to receive new information unless the programmer knows that the word in it is no longer needed, or it is empty.) During each memory cycle, therefore, the selected register is read and then written into, whether the operation called for is reading or writing.

The single memory plane under discussion thus far is not large enough to provide the amount of storage space needed in a big computer, although—depending upon word length — a large number of registers can be wired into a plane of reasonable size. To provide the required storage space in a compact arrangement, memory planes are stacked in a 3-dimensional array, as shown in figure 3—89.

Now, there are two possible ways of arranging the registers. If each memory plane contains a group of registers, as previously described, each address must specify the number of the X line and the number of the plane. This method, however, is likely to lead to complicated wiring within the plane itself. The alternative method is the one shown in figure 3—89. Here, each bit position of a register is in a different plane.

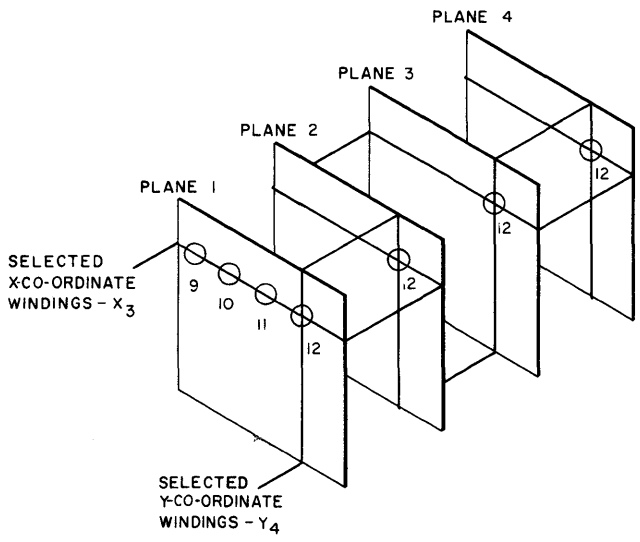


Figure 3—89. Stacked Memory Planes

To see this clearly, consider core 12 in figure 3—88, which is selected by half-current pulses on lines X_3 and Y_4. Stack up four planes identical to this one and connect the X_3 lines of all planes in series, then do the same with the Y_4 lines, as illustrated in figure 3—89. Placing half-current pulses on the X_3 and Y_4 lines now selects core 12 in each of the four planes, so the four core 12's can be used as a single register.

The drawback to this is immediately apparent. It is fine for reading, when all cores in the selected register are set to 1, but how are words written in? Obviously, each bit of a word in parallel form must be applied to a different plane, to the core of the selected register in that plane. Since selection (in the writing process) sets all the cores to 1, the logical way to write a word is to inhibit those cores that must represent 0's in the word. This can easily be done by adding an inhibit winding (not shown) to every core.

Then, to write a word into storage, half-write pulses are placed on the proper X and Y leads, selecting the register and tending to write 1's into all the cores in it. At the same time, a half-current inhibit pulse is developed for each 0 in the input word and applied to the inhibit winding of the core in the corresponding bit position. The inhibit pulse and winding are so arranged that the resulting magnetic field cancels the field set up by one of the write pulses, so the core cannot be switched to 1.

Since only one core in a given memory plane belongs to the selected register, the inhibit windings of all cores in the plane can be connected in series. An inhibit pulse on this lead can affect only the core that is fully selected. Similarly, in the reading operation, only one core in the plane is selected and can produce an output pulse, so this means that the sense windings (not shown) of all cores in the plane can also be series-connected. Neither the inhibit pulse nor the output pulse can have any important effect on unselected or half-selected cores in the plane.

In the reading operation, of course, the half-read pulses are of a polarity that sets all cores of the selected register to 0. The cores that were storing 0's are unaffected, those that were storing 1's are switched, producing output pulses that are taken off in parallel form from the sense windings of the separate memory planes. The word is thus read out as a parallel set of pulses and blanks. As described earlier, it is then rewritten into the register from which it was taken.

The address selection circuitry used with such a core memory is usually simple in nature. The address received from the operation-address register in the control element must be decoded or translated into a pair of one-hot signals, one to select the X line, the other the Y line. This decoding is commonly done in a matrix of some sort. If a new word is to be written into stor-

age, it is held temporarily in a register (often a complementing register) in which an inhibit pulse is developed from each bit position containing a 0.

For either reading or writing, a half-read pulse is first gated to the selected X and Y lines. This clears the desired register and reads out the word (if any) previously stored there. For the writing process, this word is discarded since a new one will be stored. In reading, the word is sent to the circuits needing it and also gated into the register that develops the inhibit pulses. In either case, this register now contains the word to be written into the core register. Half-write pulses are now gated to the proper X and Y lines and, at the same time, inhibit pulses are gated to the inhibit windings of the bit positions containing 0's, thus writing or rewriting the word into the selected core register. All control circuitry is then cleared in preparation for another memory operation, completing the memory cycle.

In this manner the ferrite core memory array handles the storage requirements of the computer quickly and efficiently.

4.3 ELECTROSTATIC STORAGE

Electrostatic storage is accomplished by storing 1's as positive charges and 0's as negative charges on tiny separate areas of a dielectric plate in a device similar to a cathode-ray tube. The dielectric plate structure is mounted behind what would be the face of an ordinary CRT and consists of a screen grid on the cathode side (the dielectric plate) and a conducting plate of metal on the tube-face side. The electron gun and deflection plates are practically the same as those in any CRT.

Although there are several possible methods of control, the effect is similar to that of storing a charge in a capacitor. It is achieved by changing the potential of the conducting plate to make the dielectric area temporarily positive or negative with respect to the screen grid as the electron beam strikes a small spot on the dielectric. If the dielectric area is more positive than the screen grid at this instant, there is practically no secondary emission and the spot soaks up a surplus of electrons, thus holding a negative charge when the beam is removed and the potential of the nearby conducting plate returns to normal.

If, on the other hand, the dielectric area is more negatively charged than the screen, the electron beam knocks out a large number of electrons from the dielectric (secondary emission) and these are picked up by the screen. The beam actually knocks out more electrons than it puts in, so the spot is left lacking in electrons and hence is positively charged.

Since the dielectric is a nonconductor, these small areas of charge are held after the beam is removed and they can be placed close together. By using selected deflection voltages, the beam can be moved to any spot on the plate. The address of a register, however, is a binary number representing the location of the whole register, so it must be decoded in such manner as to develop a set of deflection voltages in steps that will select all the bit locations of the register, one after the other. This is often done by deciding the address to an analog current, then passing the current through a precision voltage divider to develop the deflection voltages.

Reading a bit out of the electrostatic storage tube is done by aiming the electron beam at a selected bit location and making the potential of the conducting plate somewhat negative. If the bit location is positive, its potential is slightly negative with respect to that of the screen and there is a sudden increase in secondary emission that can be detected as an increase in screen current to the tube. If the beam is striking a negative bit location, however, secondary emission is already high and there is no momentary change when the conducting plate swings negative. Both writing and reading must be done in serial form, since the beam can strike only one bit location at a time.

Electrostatic storage is fast of access, but not as fast as magnetic cores. The principal reason lies in the need for accuracy in the deflection voltages to aim the electron beam at the selected bit location and because it takes time to develop these voltages properly.

Reading information out of the electrostatic storage tube destroys the information. With these tubes, however, high resistance on the dielectric plate tends to discharge the bit locations gradually and it is therefore necessary to rewrite all the stored information at regular intervals, whether it is read out or not. The time used in rewriting the information is lost to the computer as far as normal reading and writing operations are concerned.

4.4 ACOUSTIC DELAY LINE STORAGE

Information is stored in acoustic (or sonic) delay lines in the form of mechanical vibrations traveling along a length of some solid or liquid material, ordinarily a column of mercury. (Actually, despite the name, the vibrations are far above the frequencies of sound.) Since the computer signals are electrical, there must be, at each end of the delay line, a *transducer*, a device that converts energy from one form to another. Blocks of quartz crystal are used to convert between the electrical pulses and the mechanical vibrations.

The acoustic delay line is used in the closed-loop type of storage arrangement, as shown in figure 3–90. This means the words to be stored must be in serial form. The mercury tank is capable of holding several hundred bits, so a sequence of pulse-type serial words is stored.

When a serial word is written into the loop as a train of pulses and blanks, each pulse in turn hits the

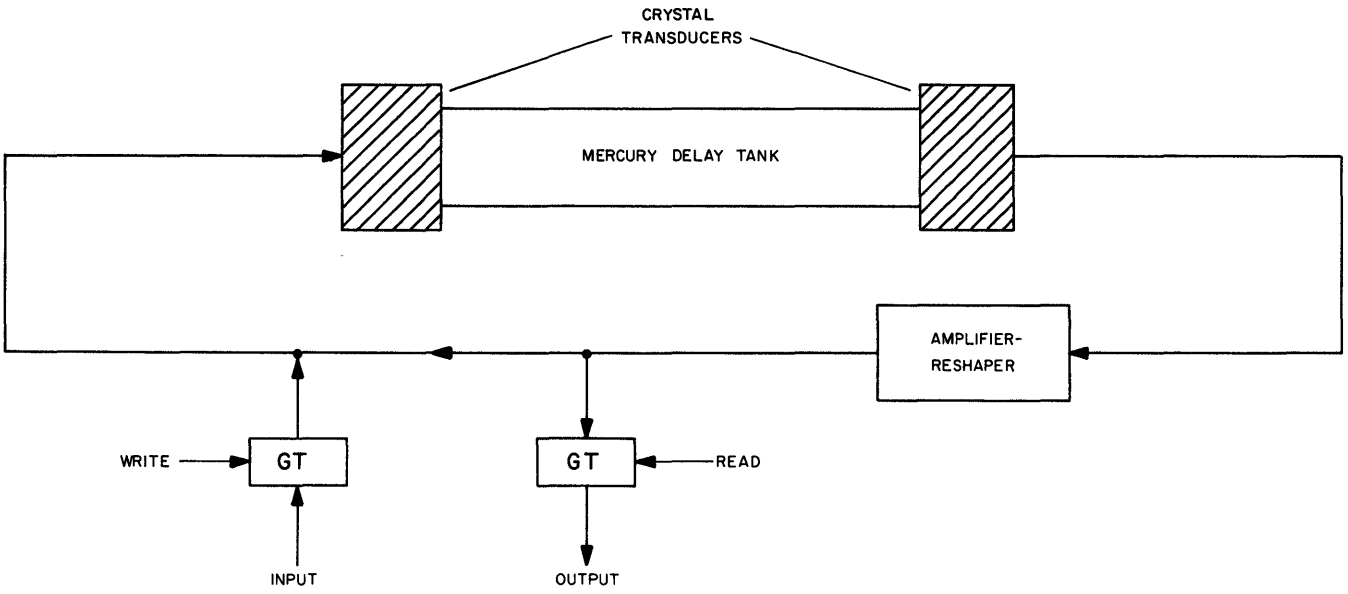


Figure 3–90. Acoustic Delay Line

quartz crystal transducer at the input end of the mercury column. Through the piezoelectric effect, the impact of the electrical signal is translated into a mechanical vibration of the crystal that is sent into the mercury column. After traveling the length of the column at a comparatively slow speed—which introduces the delay — the wave (vibration) shocks the crystal at the output end into producing an electrical pulse corresponding to the one that was put in.

Due to attenuation and reflections in the tank, the output pulses must be amplified and reshaped after they emerge into the external loop. Or they may simply be used to gate new, fullsized pulses (usually clock pulses) into the loop to take their places.

The problem of reading a desired word out of the loop is one of timing, just as in a word-length circulating register, except that the desired word is delayed much longer in getting through the acoustic delay line to a point in the loop where it is available for reading. The total delay around the loop is known, so this time becomes the length of one memory cycle. That is, if a pulse entered in the loop emerges 400 bit-times later, for instance, then 400 bit-times is one memory cycle.

During operation, then, a continuous cycling count from 0 to 399 is made (using the above example). If the least significant bit of a new word being written into storage enters the loop at bit-time 50 of a memory cycle, it will emerge from the delay line and become available for reading at bit-time 50 of every memory cycle thereafter, until it is finally erased from the loop. Thus, 50 is the address of this word.

Several methods can be used to locate and read out this (or any) stored word. One of the easiest is to place the address 50 in a counter and then, at the start of the next memory cycle, use clock pulses to count down toward zero. When the counter reaches zero, the read gate is opened for one word-time and the desired word is read out. Where more than one delay line is used, each address must also identify the proper line.

Notice that this is nondestructive readout; that is, the word continues to circulate in the loop even after it is read out. To remove words, another gate or an inhibit circuit can be placed in the main loop. This can be closed for one word-time or for one memory cycle to remove a single word or all stored information.

Although the computer must wait for a desired word to be read out, the pulse repetition rate can be very high (up in the megacycles) in these acoustic delay lines, so they have been found practical for use in serial mode computers. One drawback is that the line must be temperature-controlled because the velocity of mechanical vibrations in mercury, like the velocity of sound in air, varies with temperature. The longer the line, the more accurate the control must be to avoid serious timing difficulties.

4.5 MECHANICAL STORAGE

4.5.1 Punched Hole Storage

The idea of storing information by punching or not punching holes in specified locations on tough paper cards or tape has been in existence a long time. By arranging regular rows or columns on a card or along the length of a tape, binary numbers can be permanently stored by the punch (1)-no punch (0) system.

When a card or tape is properly aligned and drawn between a metal plate and a set of metal wipers or fingers, a brief electrical contact is made through each hole. None is made where there is no punch. Thus,

punch-no punch is translated to pulse-no pulse, which is understood by the computer.

A newer and faster method of reading punched information is to draw the card or tape between a light source and a set of photocells (one per row or column). Each punched hole lets through a flash of light to the corresponding photocell, which produces an electrical pulse, so again the informamtion is translated to computer language.

Punched cards and tape are not well suited for use as either main or auxiliary storage mediums in modern computers, because of the necessity for frequent handling by an operator as well as because of the slowness of access. For this reason, punched card and tape machines are most often used as input-output devices and, as such, are described in Chapter 4.

4.5.2 Control Panel Storage

A telephone switchboard might be considered to be a type of memory device. Each time the operator plugs in one of the patch wires, she sets up a connection that will "remember" which circuits are connected. In computer usage, a similar storage device is the control panel, often called a plugboard. A device of this type, sometimes used to store programs and data by means of plug-wires, is shown in figure 3–91. Information is stored in it by connecting certain hubs (holes) together with pluggable wires.

Two general types of control panel storage are used. In some machines, the control panel is used purely for storage of control information (the program). In this case, the control panel usually consists of two types of hubs: exit hubs and entry hubs. In general, the exit

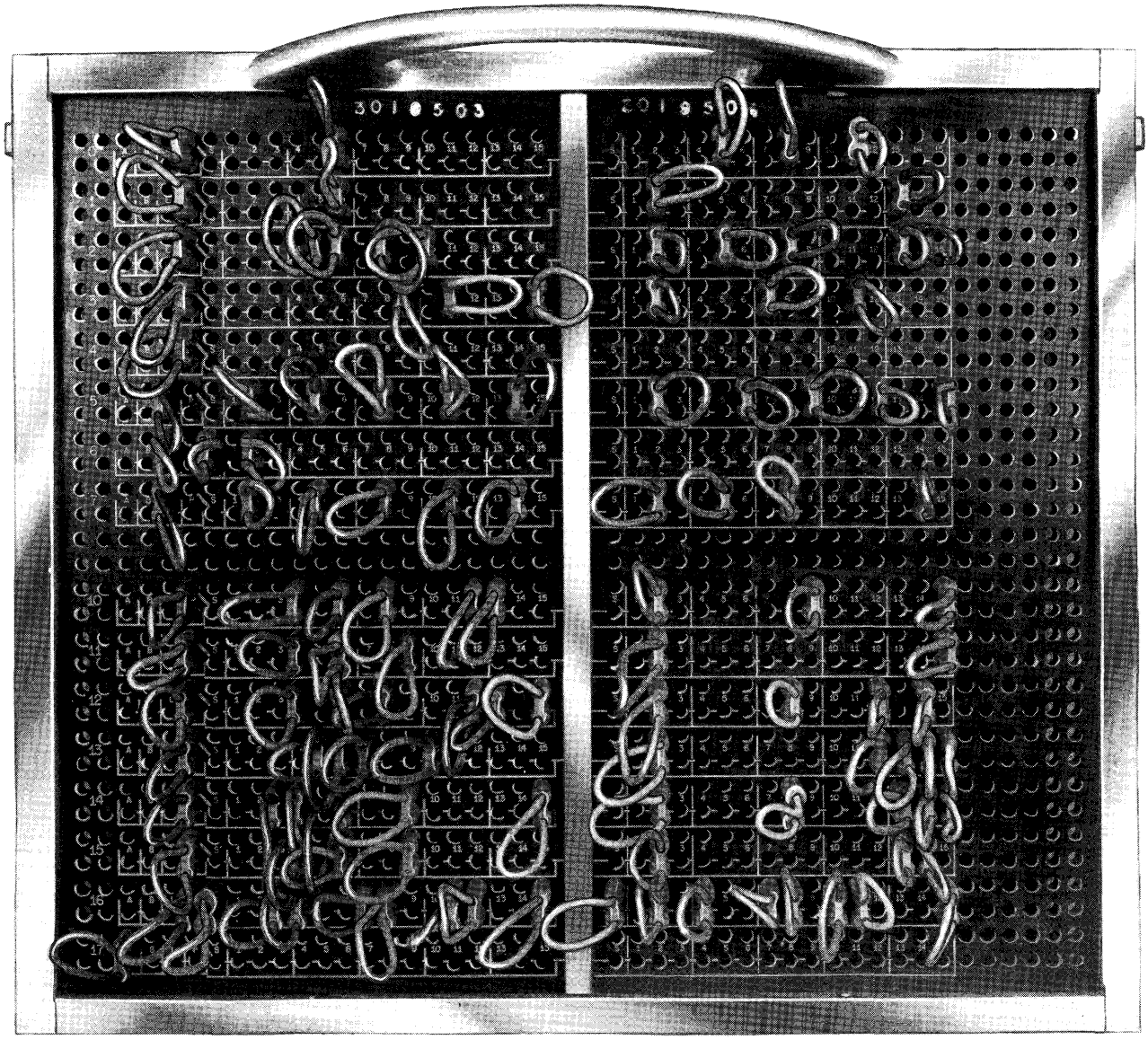


Figure 3–91. Control Panel

hubs are hubs which emit pulses. Entry hubs are connected to circuits controlling various machine functions. Therefore, when they are pulsed, the functions they control are initiated. During each machine cycle, the various exit hubs are pulsed in certain sequences by the machine. If one of these exit hubs is wired to an entry hub controlling a particular function, the pulse, when it appears, initiates this function. For instance, a exit hub might be wired to the entry hub which initiates the addition function. Then, when the machine pulses this exit hub, it will cause the machine to add.

Another type of control panel storage is more like the register storage which has been mentioned before. In this case, the control panel is divided into registers, and each bit position of each register consists of a pair of hubs. In such a system, a 1 is usually represented by a wire connecting the two hubs of a pair, a 0 by no connection between the two hubs. A 5-bit register, for example, has five pairs of hubs. If the first, third, and fifth pairs were connected together and the second and fourth left unconnected, the register would contain 10101.

CHAPTER 5
INPUT-OUTPUT EQUIPMENT

5.1 EQUIPMENT, GENERAL

5.1.1 Introduction

This chapter discusses in general common types of input-output (IO) equipment and covers in detail how IO devices are used in digital computing systems.

Many types of IO devices are available and there are also devices which could be adapted for preparing and receiving information for digital computer operation. This chapter will consider those devices which are actually used.

5.1.2 Definition of Input-Output Devices

The equipment that introduces input information into the computer and receives output information from the computer may be the same device. The function of this device may vary between its input and output utilization. For example, a tape unit may be used as a memory storage, as an input source, or as an output device. However, the general characteristics and capabilities of the tape unit are fixed and do not vary with its function.

An input or output device may be defined as equipment for communication between the computer and the external sources or destinations of information and data; devices may be classified as input-output equipment if they can translate external information into computer information or vice versa.

5.2 DESCRIPTION OF INPUT-OUTPUT EQUIPMENT

5.2.1 General

Information transfer between a computer and a person is generally through magnetic or paper tapes, card machines, typewriters, visual displays, and line printers. The automatic inputs of the AN/FSQ-7 are all-electronic equipments which translate information from the form in which it appears in the telephone line receiver (serial pulses) to the form used in the computer (parallel pulses).

5.2.2 Tapes and Tape-Handling Equipment

5.2.2.1 General

There are two types of tapes available for use with digital computer systems, paper and magnetic. Both types can be used for either input or output functions. Each type of tape has its own distinctive processing equipment and the tapes are not interchangeable. A paper tape must be used with a paper punch and reader, and magnetic tape must be used with equipment designed for magnetic tape preparation and processing.

5.2.2.2 Paper Tape Equipment

Paper tape is usually used as an IO device, although it has been used for memory storage, and is similar to teletype tape. Information is coded on the basis of a hole-no-hole code. In a typical system, the holes are punched on the tape by a manual keyboard and paper-tape punch procedure or by the computer and paper punch arrangement. Normally a computer-prepared tape is an output procedure and a keyboard prepared tape is an input procedure. Paper tape with the associated paper-tape punch and reader is shown in figure 3–92.

The paper-tape reader converts the punched paper-tape code into electric impulses by means of a photoelectric system or by sensing the pattern of hole-no hole with brushes.

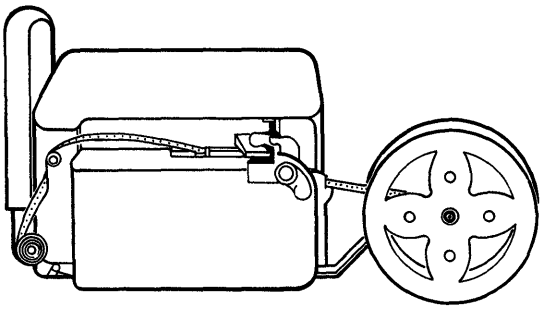
The paper-tape reader interprets the punched tape and can be linked to a printout device as well as to the input phase; it can also be used to verify the correctness of a keyboard-prepared tape before the tape is processed as input information. In the input phase, tape is sensed by the reader and the information, expressed in electrical pulse form, is fed into the computer.

A paper-tape system is a relatively slow process of information processing (although it is less expensive than a magnetic tape system) and is generally used with a special purpose computer solving scientific problems of a fixed type.

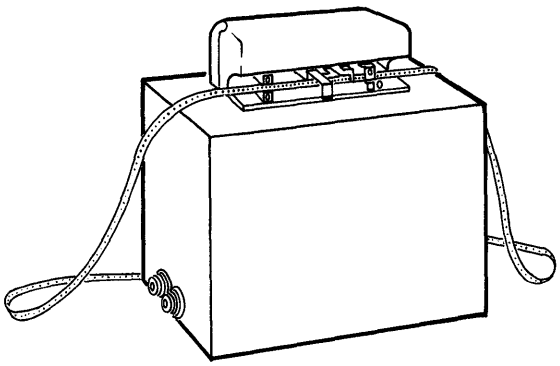
5.2.2.3 Magnetic Tape and Tape-Handling Equipment

Magnetic tape usually is a coated plastic tape about ½-inch wide similar to the tape used in home style tape recorders. The coating varies with the commercial processes used to manufacture tape. The coating has magnetic properties, that enable the tape to be magnetized in discrete units (very small magnetized spots).

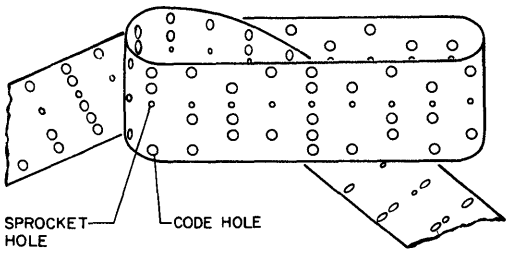
Information is represented in the form of a pattern of magnetic bits. In one form of tape recording, a magnetized spot or bit may represent a binary one; a non-magnetized spot on the tape may represent a binary zero. A more common system of writing on tape requires that both 1's and 0's be expressed as magnetized bits. This is accomplished by recording 1's with one north-south magnetic alignment and 0's with south-north alignment. A large amount of information can be stored



PAPER TAPE PUNCH



PAPER TAPE READER



5-HOLE CODE PAPER TAPE

Figure 3–92. Paper Tape with Associated Reader and Punch

on a length of tape. A typical tape is about 2,000 feet in length and has a word density of 41 computer words per inch.

Information may be transferred to a magnetic tape by means of special typewriters, by card-to-tape converters, by card machines, from magnetic drum, or directly from a computer.

Tape-handling equipment consists of a tape drive with associated electronic reading and writing circuitry. A typical tape drive is shown in figure 3–93.

If a program has been coded and is ready for translation into a form of information usable in a computer, the first step in the procedure is "to write" the program on the tape using the equipment available in a given system. This equipment may be a computer or card machine. When the information is on the tape, the reel of tape is placed in a tape drive unit. Then, the information can be read by the read-write head of this unit and the pulse pattern transferred to the computer.

Ttapes and tape-handling equipment may also be used to receive and record outputs from the computer. The outputs are operationally processed from the computer to the tape unit. Generally the information is left on the magnetic tape in its magnetic form since it will probably be used again by the computer and there is no need for human monitoring (no printout is necessary). Sometimes a common program is stored on tape so that when it is needed again it will be very easy to insert. When human monitoring of the output is necessary it is often advantageous to record the output on tape and then use an auxiliary device (entirely separate from the computer) to print the results from the tape. This system permits a very high speed tape output which can be monitored.

Tapes and the associated equipment are generally used as large-capacity slow-access memory storage. They may be considered IO equipments since they are used to initially load information into the computer. As storage unit, tapes are memory type equipments, slow in relation to the other memory units.

5.2.3 Card-Handling Equipment

5.2.3.1 General

The term card machines includes all units of a card handling system that uses the holes punched in paper cards to represent information. Such a system must be capable of punching information on cards and reading the information from the punched cards and printing it in a form that can be directly read without further decoding. A card-handling system implies at least three units; a card punch, a card reader, and a printer.

There are many possibilities of coding information

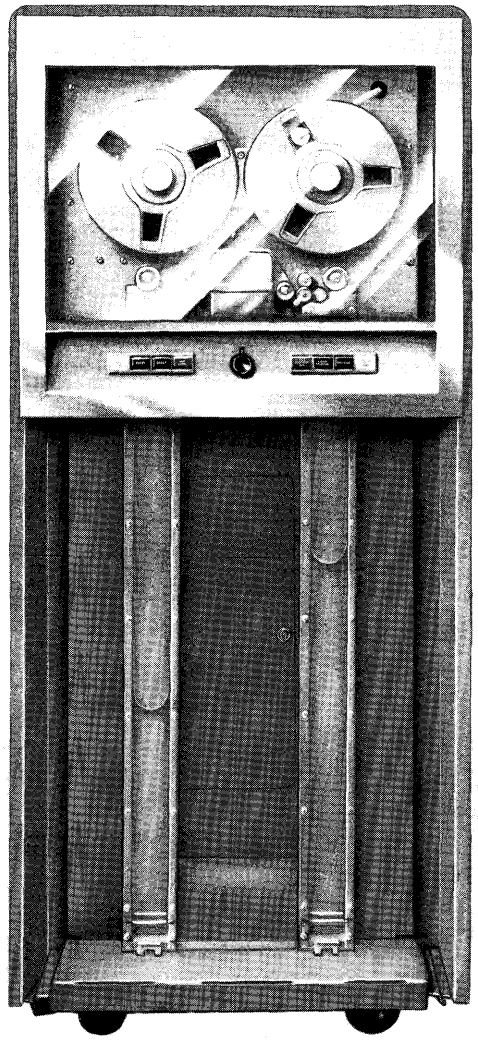


Figure 3–93. Magnetic Tape Drive Unit

on cards using numeric or alphanumeric codes. Among the more important codes are the binary and the Hollerith.

5.2.3.2 Cards and Card-Punch Equipment

A typical card (fig 3–94) is of fixed dimensions and is made of a specified quality of paper. A card may be of varying format in accordance with the best techniques for translating information from the card to the computer of any given computer system. The card punched in Hollerith code is of one definite format which, despite disadvantages, is suited for use with mechanical computing equipment. Another similar card, arranged in other format, is the binary card; this is suited to electronic computing equipment.

The equipment used to manually transfer information onto a card is a card punch. A manual card punch is an electromechanical device which punches information on cards and prints out the information on the top of the cards. The punching operation is performed at a keyboard, similar to a standard typewriter keyboard, by an operator. A typical example of a manual card punch is illustrated in figure 3–95. This particular card punch also performs the additional function of reading the punched cards and converting the information expressed into a pulse-no pulse pattern for direct computer input. This type of punch is called a computer-entry punch. First, the card is punched; then the card is automatically sent to the reading station. At the reading station mechanical feelers sense the hole-no-hole pattern and generate voltages corresponding to this pattern, which in turn are fed to the computer. Such a system allows the operator to check the card punches before the information is actually entered in the computer.

A card punch for transferring outputs from the computer to cards is illustrated in figure 3–96. This card punch is operated by the computer through the memory. The computer generates voltages which operate punch-selecting electromagnets. When the punches operate, they punch out a card-hole pattern corresponding to the computer output information. No manual operation is possible. The computer-operated card punch is faster than the manual operated punch; the output punch can process about 100 cards per minute while the manual-operated punch is limited to the skill of the operator. A processing of about 3 cards per minute is within normal operator capabilities.

In most computers it is possible to punch information on cards in any of several codes. This may be a built-in feature or, if not, it may usually be accomplished by special programming techniques.

5.2.3.3 Card Reader

The card reader shown in figure 3–97 is used to transfer information from punched cards into the Central Computer. The card reader is directly linked to the computer. Cards may be read at the rate of about 250 cards per minute. Reading is accomplished by contact with brushes which sense holes in the card and complete an electrical contact; holes in the cards become pulses and the intact area becomes the no-pulse in the binary code arrangement. The reader can be used to process information from cards directly to the memory unit of the computer. This information is available for use in the computer operation or for transfer to magnetic tape for storage.

5.2.3.4 Line Printer

A line printer records output information, usually in alphanumeric form (fig. 3–98). The term line designates that the printer is capable of printing a line of characters at a time.

Printers vary in speed of the printout from about

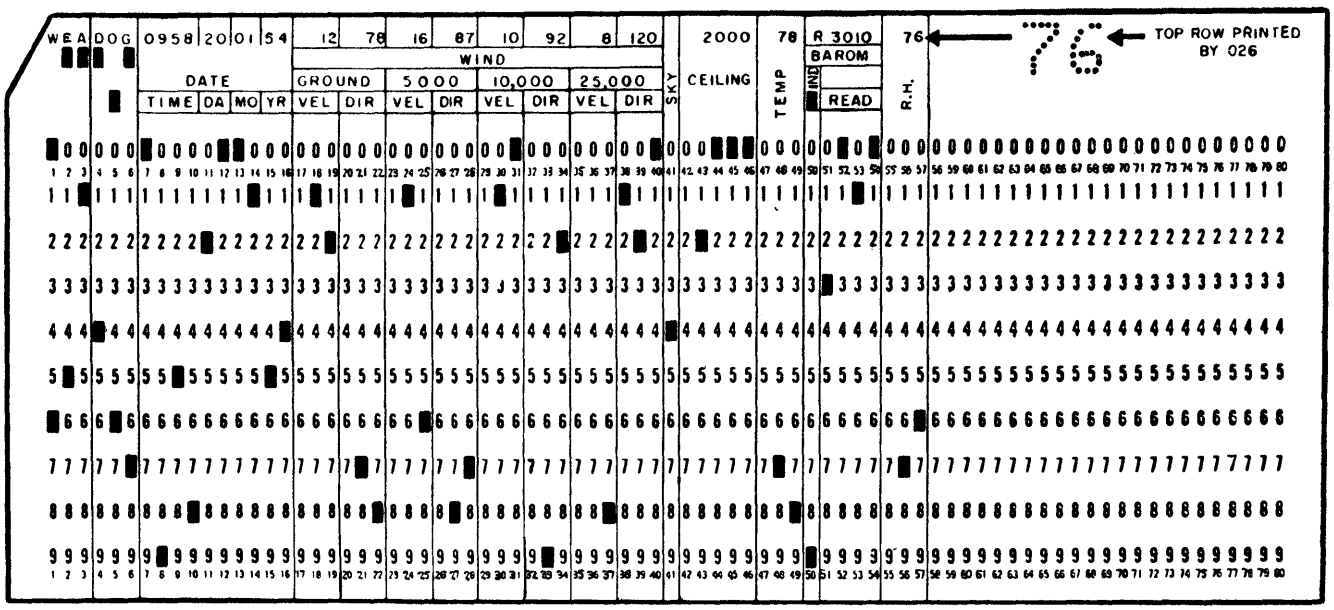


Figure 3–94. Card Arranged in Hollerith Code Format

100 lines per minute to over 1,000 lines. The number of characters per line also varies. The printer in the figure above can print out 150 lines per minute, 64 characters per line.

The exact technique of how the printing is accomplished is not within the scope of this manual. In general the computer generates a pattern of voltages which are used to energize electromagnets. These electromagnets than select the proper letter or number on a typewheel and the printer prints. The printer illustrated in figure 3–98 prints one line at a time by the simultaneous positioning of the type wheels (120 in number). Each typewheel is not positioned in every line of printout–only those 64 typewheels which can be selected within the cycle of line printout are used.

5.2.4 Typewriter

In some systems, typewriters are available for transferring input and output data into and out of the memory element of the Central Computer. These typewriters have an alphanumeric keyboard and may be manually and computer operated. When typing inputs, typewriters are manually operated and transmit one character at a time to the Input System. The keyboard character is translated into coded electrical pulses for entry into the memory by means of contacts on each key. Manual operation is a relatively slow input method and is usually restricted to inputs of a local nature in a computer system with a fixed program. Outputs are typed out one character at a time under computer control and selection. The outputs also are transmitted from the memory.

During output operation the IO typewriter is operated by a series of electromagnets and solenoids mounted beneath the keyboard. The magnets and solenoids automatically actuate keyboard functions of the typewriters, including carriage return, spacing, tabulation, ribbon color control, and others. When used as input devices, electrical impulses are transmitted from the typewriter by depressing a key. These electrical impulses may be generated in a coded pulse-no-pulse pattern by a preset action of the key in relation to a group of switches.

5.2.5 Visual Displays

Information may be transmitted from a digital computer and displayed visually in a direct-read form. The equipment used to display such output information is generally known as visual displays. A typical example of a visual display equipment is shown in figure 3–99. Some display a picture similar in physical appearance to a television picture of printed or pictorial information. Other display outputs may be merely graphical, such as might be seen on the usual cathode-ray oscilloscope.

The main component of a display equipment is a cathode-ray tube (figure 3–99). The special purpose tubes such as those used in the AN/FSQ-7 have provisions for writing actual alphabetic characters. Figure 3–100 shows a simplified diagram of a typicial display tube. Writing on the viewing screen is accomplished by forming a character and positioning the character on the tube face. Characters may be generated by a stenciling process; that is, the electron beam is directed through a selected aperture in a character-forming matrix and directed by electrostatic deflection plates to a selected position on the tube face. The face of the tube

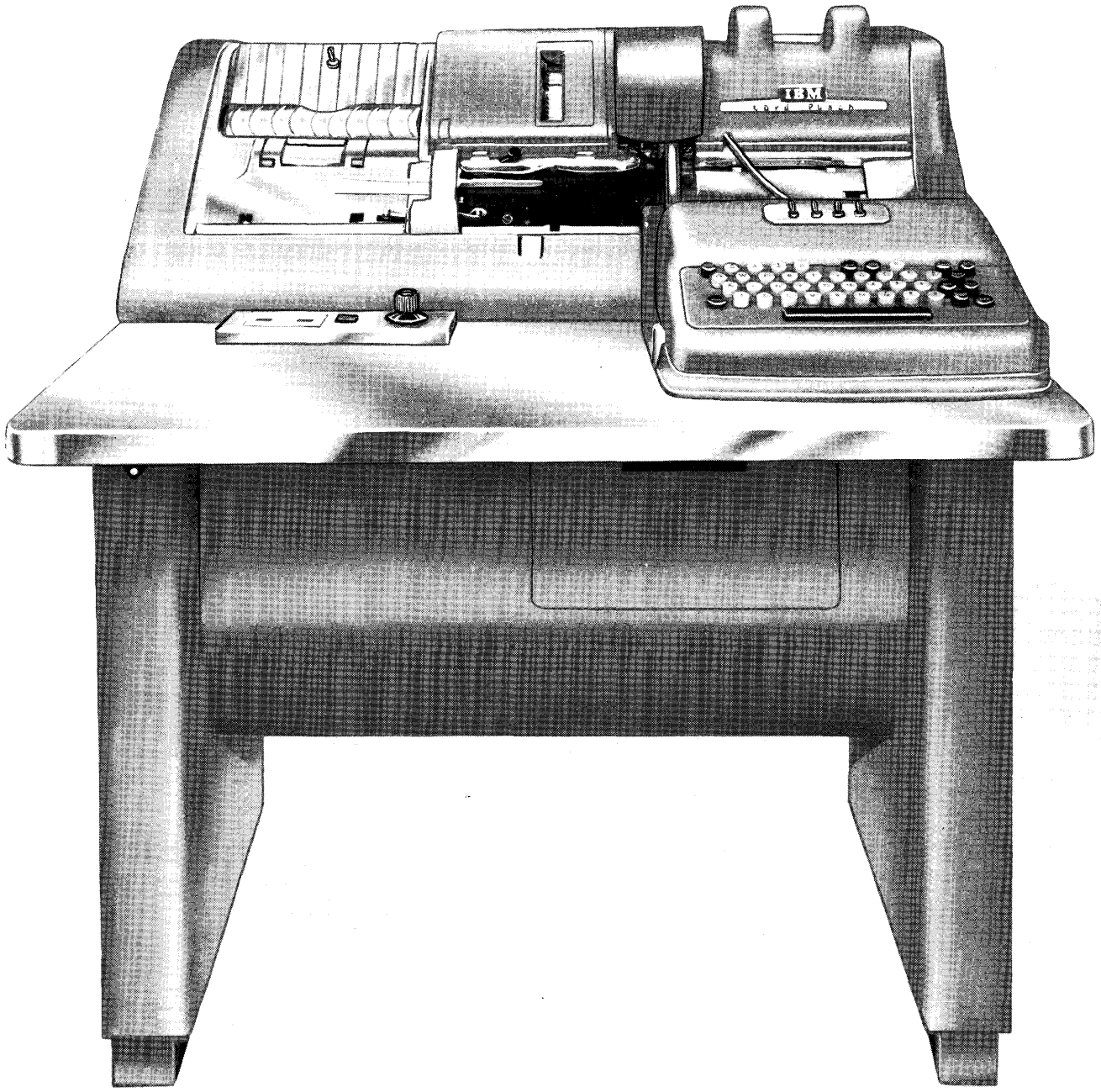


Figure 3–95. Computer Entry Punch

is phosphor-coated. The phosphor emits a blue flash when illuminated by a writing beam, followed by a yellowish afterglow which persists long enough for human perception. In some display systems, it is possible to retain the character on the tube face by using another beam of electrons.

Information in binary-coded pulse form is transmitted from the storage element of the computer or from the Input System to the display equipment. There, it is converted to analog deflection voltages which generate the desired visual displays. Since the information on the tube face does not persist for a long period of

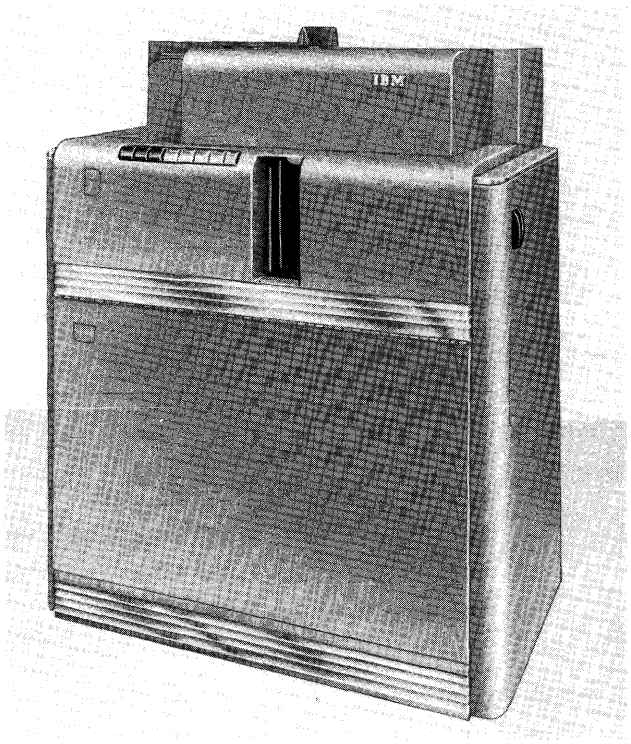


Figure 3–96. Computer-Operated Card Punch

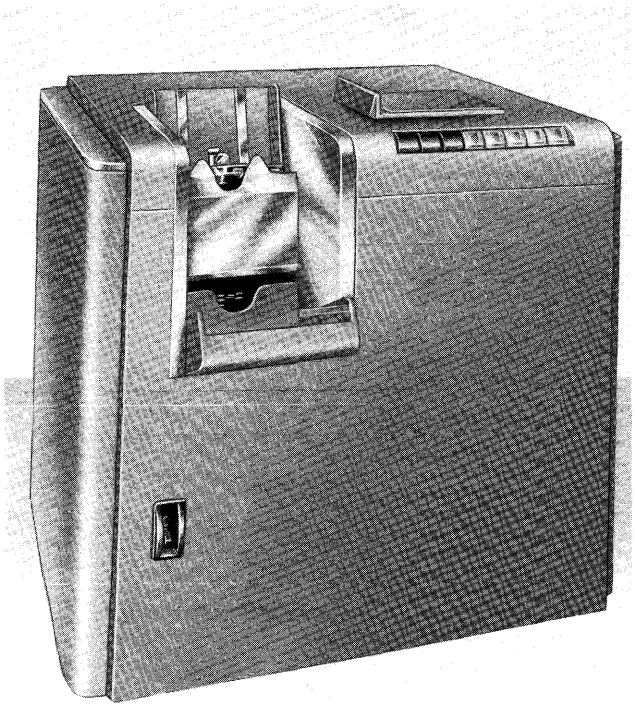


Figure 3–97. Card Reader

time, it is sometimes desirable to record this information in a more permanent form. Camera systems are available which automatically photograph the display tube face and process the film to provide a positive transparency. When required, the transparency can be projected on a screen and the information is visually available for as long as required. Other information recording systems photograph the information on the display tube face and project the images on sensitized paper where the information is reproduced.

5.2.6 Other Input-Output Equipment

The IO equipments discussed in the preceding paragraphs are commonly identified with general purpose digital computers. Possible types of IO devices associated with special purpose computers are limited only by the capability of any device to transmit or to receive information in a form that can be converted for use in a digital computer.

Input information can be transmitted from teletype machines, telemetering devices, analog computers, and even by human voice. In some way, this information must be converted to a pulse no-pulse binary number code, after which it is compatible for transmitting into the computer system as data in a specific problem.

The results of computer computation can be translated into a form that can be transmitted to guided missiles, control systems, and wide-spread inventory and business offices. The devices used to transmit computer results are teletype systems, telephone lines, radio, and so forth. These devices may be directly linked to the computer system and the outputs require no additional processing by human operators. A dramatic type of output is the speech output from the computer. This technique is termed speech communication with the computer and has been used in diagnostic troubleshooting for locating causes of failure.

The choice of IO devices for special purpose computers depends on the sources of information for the solution of specific problems. These sources are sometimes termed data-links. The digital computers are not restricted to accepting information only from the traditional tapes, card-machines, and keyboards. They will use data from any source as long as the data is translated into the binary code of pulse no-pulse.

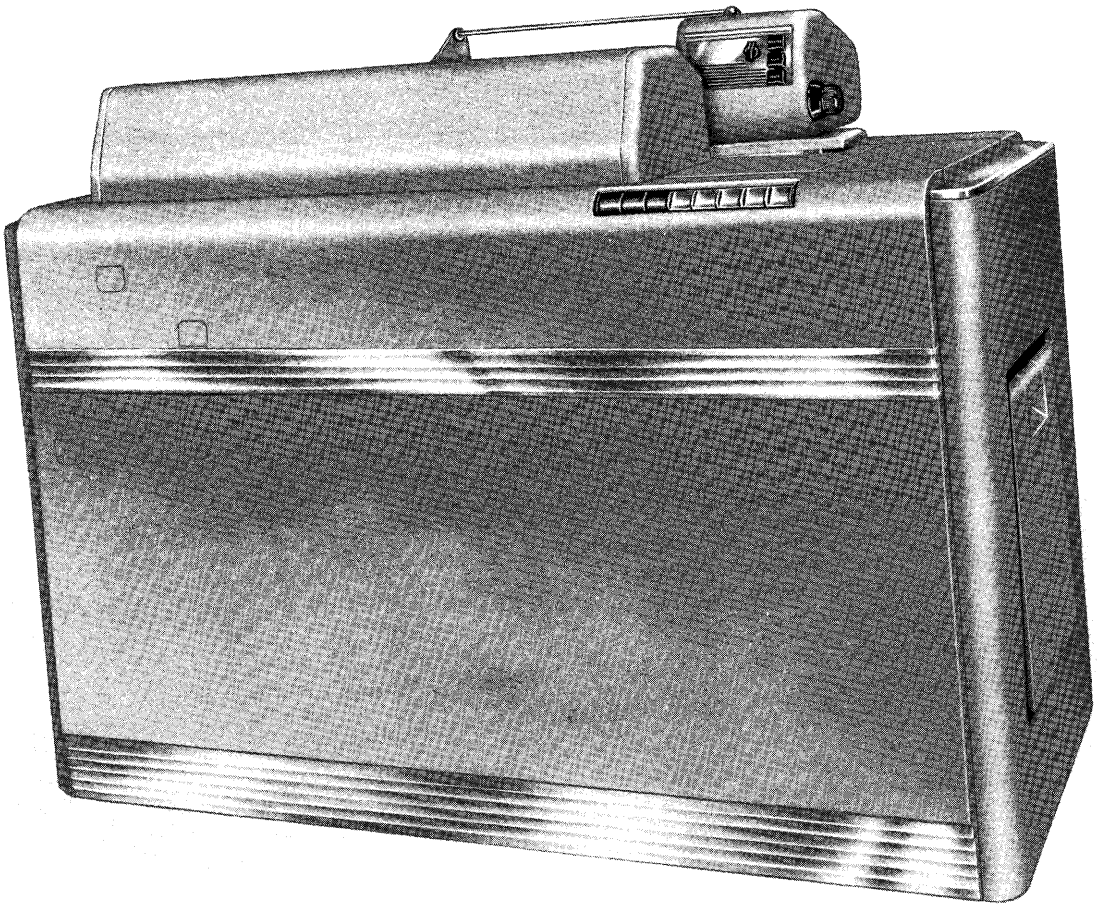


Figure 3–98. Line Printer

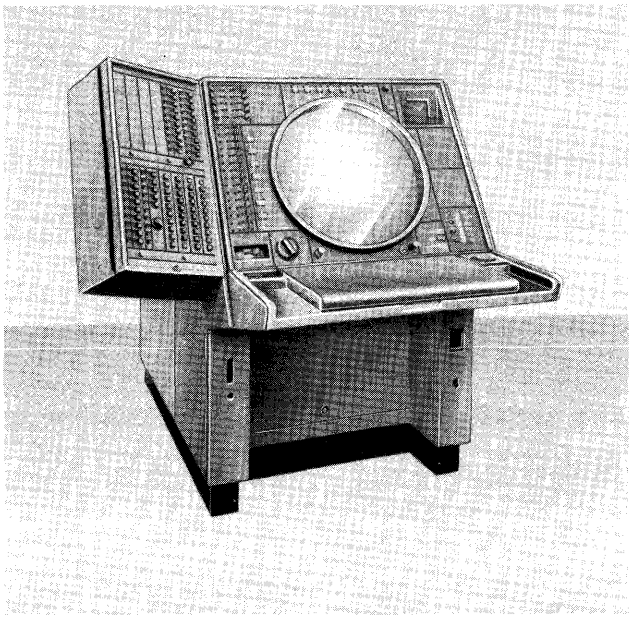


Figure 3–99. Visual Display Unit

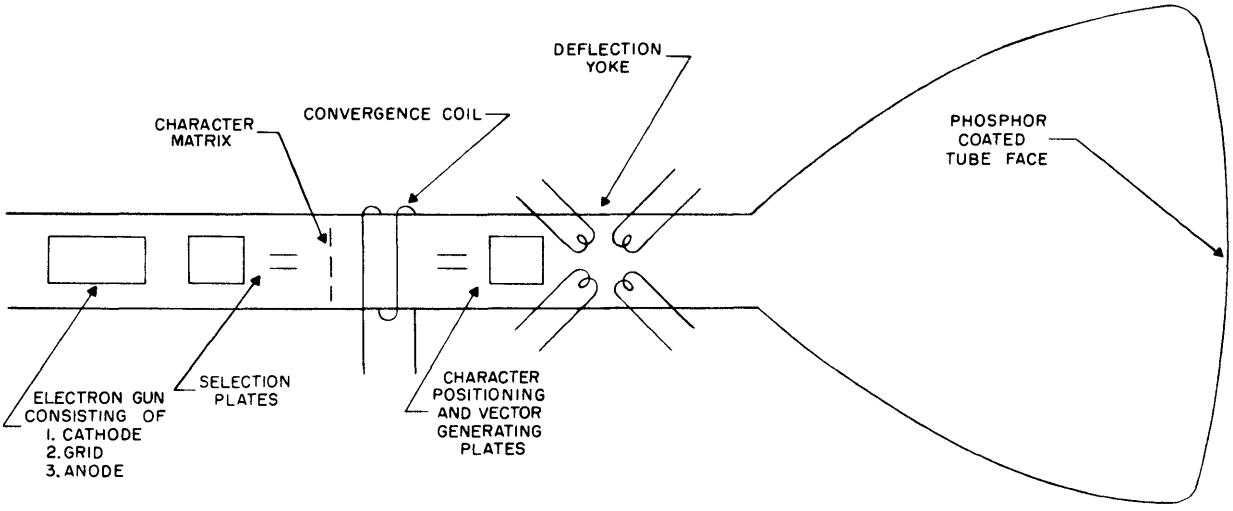


Figure 3–100. Display Tube, Simplified Diagram

PART 4
COMPUTER ORGANIZATION

CHAPTER 1
INTRODUCTION

1.1 GENERAL

Up to this point, this manual has attempted to show only a very general picture of computer organization, along with the specific circuits and arithmetic techniques which make such organization possible. This part gives a more specific picture of computer organization. In particular, the interaction of the various elements working within a complete computing system will be shown. A typical computer, which uses many of the techniques and components developed in previous parts, will be described.

The computer to be described is non-existent but is patterned after a simplified version of the AN/FSQ-7. It will be obvious later that theoretically this computer could be refined and expanded to meet the needs of the SAGE function.

1.2 SAMPLE COMPUTER DESCRIPTION

1.2.1 Requirements

Despite the fact that some data processing machines are said to be general purpose computers, some consideration of the use to which they will be put is necessary if an efficient system is to be designed. It would for instance, be foolish to build a large data processing system to compute the payroll of a company employing only 10 men. Consider the requirements of an aircraft-control computer. Such a computer obtains information on aircraft position and movement over a wide area from several radar sets. This information is automatically entered into the computer for processing. The processing consists of calculations of the planes' velocity, position, and so forth. The processing also includes a means of presenting this information to human operators, so that they can decide what ground control action (shooting down, redirecting, etc.) is necessary. Of course, there must be a means of taking the resulting ground control action into consideration by the computer. There must also be a means of furnishing computer information to users other than the actual operators of the computer; i.e., weapons bases or other ground control stations. A consideration of the requirements of such a computer brings out at least the following:

a. The problem is a real-time-control problem. That is, the computer controls a process which is continuously changing while the computer calculates. While the computer is computing, the position of an airplane is moving. This implies the need for high-speed computation if the results of the computer's work are to be useful in, for instance, shooting down the plane.

b. In the control of an air battle spread out over a wide area, it must be possible to determine aircraft positions quite accurately. The hypothetical computer, then, should have high accuracy as well as considerable precision.

c. The use to which the computer is put requires the ability to solve a great variety of problems. This seems fairly obvious when all the different types of situations possible in an air battle are considered.

d. The fact that the problem is to obtain real-time control also requires that a considerable data be continuously entering and leaving the machine.

e. As is usually the case, the input and output devices of the computer will be slow compared to the computing section. Therefore, the input-output system requires as much refinement as possible to get the best speed with the devices available.

1.2.2 General Description

Before going into the specific details of the computer to be developed, it is necessary to consider some basic concepts. It is necessary to determine those general characteristics which affect the design of the complete computer.

1.2.2.1 Analog or Digital

First, what general type of computer is required, analog or digital? In the requirements given for the hypothetical computer, it has been stated that high de-

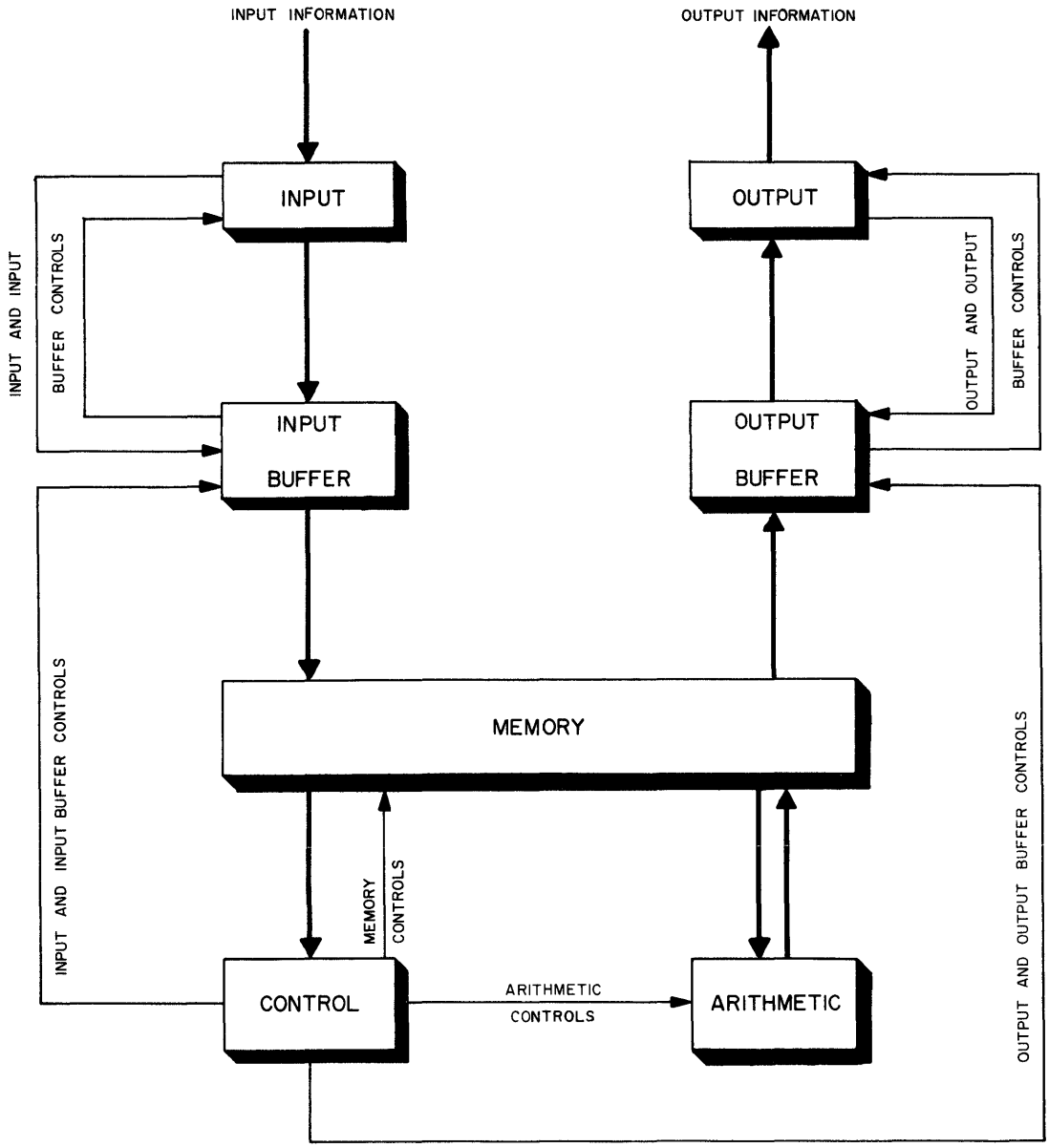


Figure 4–1. Elements of the Sample Computer

grees of accuracy and precision were required. This implies the use of a digital rather than an analog computer.

An analog device can be made quite accurate, but its precision is limited by fundamental obstacles which cannot be completely overcome. For instance, since the data in an analog device may be represented by the position of a mechanical shaft rotation about its axis, the slop of the gears and the difficulty of making precise angular measurements limit the precision possible. Furthermore, if voltages are used to represent data, any power supply changes may have a definite effect on accuracy of results. Such considerations limit the accuracy and precision possible to a few decimal places.

On the other hand the precision of a digital device is limited only by the length of the word which the device can handle. If the device can handle three decimal digits the precision possible will be to two decimal places. If it can handle 10 decimal digits the precision can be held to nine decimal places. The accuracy is theoretically almost infinite. It can, therefore, be assumed that a digital computer must be used if the problems to be solved demand high degrees of accuracy and precision.

1.2.2.2 Fundamental Elements

The nature of digital computing dictates several more general characteristics. In figure 4–1 the basic elements of a digital computing system, as well as control and information transfer lines, are shown. The thin

lines symbolize control functions, the heavy lines, information transfer. Any digital computing system must comprise at least five elements: memory, arithmetic, control, input, and output. This is true whether the system consists of a man using pencil, paper, and an adding machine or is a complex electronic computer. The other elements shown, the buffers, are refinements necessary to attain the high-speed system required.

In the main-adding-machine example mentioned in the previous paragraph, the operator's memory and the paper serve as information storage media. Memory is necessary to store the data to be operated upon as the results of the operations. In digital computing this information storage is required because the step-by-step operation makes it necessary to store some numbers while they await their turn to be operated upon. For instance, when adding three numbers together with an adding machine, two numbers are first added, while the third is retained in the memory. The final result is found by adding the number in the memory to the sum of the first two.

Sometimes, memory is used for a second very important function, that is, to store the program. Just as a man operating an adding machine must know when to push what buttons for the solution of a problem, so the computer must know the proper sequencing of its operations. In some computers (control-panel-programmed), this proper sequencing is accomplished by means of actually changing the wiring of the computer. In a stored-program computer, however, the instructions are actually retained in the memory. There, the computer can automatically refer to them and can then perform them in sequence. This type of memory, in which both data and program are stored in memory, allows great versatility.

If one thinks again of the computing system comprising a man using an adding machine, he realizes that it is in the adding-machine element that the actual manipulation of the data takes place. It is here that any calculations are performed. The adding machine is the switching mechanism through which transfers of data are made and, consequently, by which the actual changes to the data are made. Just as this system needs an arithmetic unit of some kind, so an actual computer needs an arithmetic element. It has been stated that the computing process consists of nothing more than a controlled manipulation and transfer of data between storage devices. In a standard computer the arithmetic element is the channel of transfer during the actual computing. It is the device which actually performs the work of changing the data during transfer.

The function of the control element is to direct the operation of the other elements of the computer. In a man-adding-machine computing system, the man acts as the control element. He directs the transfer and manipulation of any data. The control element in a computer does the same thing. (Of course, it must be originally programmed by the operator.) It initiates transfers, and it directs the arithmetic element in its manipulations upon the data transferred, and the memory element in its data storage functions. The control element also controls the input and output elements. It tells what data will be entered from the computer input devices or readout to the output devices. In other words, the control element controls the overall operation of the whole system; it co-ordinates all elements of the system.

A device for entering information into the system is necessary. Somehow, the information as it is understood by the operator of the computer must be entered into the machine in such a form that it can be used by the computer. That is, there must be a method of converting the numbers understood by man into their electrical equivalent (pulses, levels, etc.) used in the computing system. In the man-adding-machine example the operator performs this function; in a computer an input element is provided to do it. The input element furnishes the transfer path from the outside world to the computer, and the means of translating the data from the language of the external world to that of the computer.

Consider again the man-adding-machine example. All the calculations would be of little use if the results could not be made available to the outside world. To make the results available, an output device such as a paper-tape printer (on the adding machine) must be furnished. Similarly, any other computing system needs an output device or system. Analysis of an Output System shows that it has two important functions: it furnishes an output data path; and it translates the results from computer language to the language of the human being or other user. At first thought, it appears that a printer to print results for human operators would be all that is necessary. For computers where only human beings use the results, this is true, but in others it is not. For instance it may be necessary to connect the output to control a machine, a guided missile, for instance. In this case the output device would translate from the language of the computer to the control pulses of the missile guidance system. Such an Output System would still be performing its fundamental purposes, transfer from computer to outside world and translation of data from one language to the other.

The Input and Output Systems have one important consideration in common; they both transfer information between a computer and an outside device. This has a very important effect upon the speed and capacity of the computer. If the computer is to be able to accept information from the outside world, it must be controlled to do so. The control pulses for accomplishing

this may come only at predetermined times. On the other hand, there is no way of predetermining the availability of information from the outside world. There is, then, little chance that the demand of the computer for information will coincide with the availbility of that information from the outside world. Therefore, some sort of storage must be provided between the input-output (IO) devices and the computer. In the case of inputs, this device must be capable of storing input information at any time, regardless of what is going on in the rest of the computer. It must also be capable of transferring the information to the computer immediately upon demand of the computer. This is the purpose of the memory buffers between the IO system and the computer.

The input buffer will operate somewhat as follows:

a. It always is attempting to read from the input.
b. It collects all data from the inputs and stores this for quick transfer to the computer.
c. It transfers this data to the computer upon demand.

What has been said for the input buffer memory also can be applied to the output; the computer can read into the buffer at its own pace, and devices can read from the buffer at their own pace. In either buffers the amount of information going in should not exceed that coming out, or information will be lost; i.e., a new message might be written over an old message not yet read. A buffer's main purpose is to match transfer times of the many slow, intermittent devices (input and output) to the transfer times of the one rapid device (the computer).

1.2.2.3 Program Control

It is possible to classify program control according to the medium used to store the program. Based upon such a classification, three general types of computer program are available: external, control panel, and stored program. Each has its advantages and disadvantages. All three accomplish the same function: they tell the machine what to do. However, the method of program storage has a great effect upon the usefulness of each in the solution of a given type of problem.

A computer which executes individual instructions as soon as they are received from an input device is said to be externally programmed. In this case, the program is stored externally to the computer, usually on cards or tapes. The advantages of this type of program storage are its low cost and its simplicity. However, external programming presents two major difficulties: the speed of program execution depends on the speed of the input device (particularly during repetitive operations), and externally stored programs are not easily changed by the computer during the course of a program. The ability to execute repetitive programs and to change a program while it is in progress are extremely important characteristics of the modern digital computer.

A higher-speed and more versatile computer must be used to satisfy the requirements set forth at the beginning of the chapter. The control-panel-program computer might be used. In this type of control, a control panel somewhat similar to a telephone switchboard is used to control the computer. Wires may be inserted between different control points on the panel to initiate the various functions of the machine.

In control panel program storage, all steps of the program are stored within the computer at the same time (when the control panel is inserted). Therefore, the program control is not limited in speed by any input device as was true in the externally programmed computer. The fact that all steps of the program are available to the computer also makes repetitive programming much simpler. However, there is a practical limitation to the size of the program which could be stored on the control panel. As more and more steps are required (more plug wires required) the board becomes too large and complicated to be practical.

Because there are to be long programs and a wide variety of problems, the best type of program control to use is the stored program. In this kind of storage, the program is stored in the machine's internal memory in the form of "numbers." These numbers can be decoded by the machine to direct its operations.

The stored program is the most versatile and highest-speed program control yet devised. The program is loaded into internal memory; this means that the speed of operation does not depend upon the input device once the program is in operation. Since all the steps of the program are available to the computer at any same time, it is very simple to accomplish repetitive operations. Another important advantage of this type of control is that the instructions are stored in the same memory (usually) and in the same form as regular data. Consequently the computer can operate upon the instructions of the program just as easily as it can on the data of the problem. In other words, the computer can change its own program as it goes along. There are two main reasons, then, for using this type of control in the sample computer, high speed and great versatility.

1.2.2.4 Single Address or Multiple Address

In Part 3 it was pointed out that the instruction word always consists of at least two parts, the operation part and the address part (fig. 4–2). The operation part of the instruction is a coded number which stands for the operation to be performed; e.g., ADD. The address part of the instruction indicates the location where the number to be added is stored. A single-address machine is one in which each instruction word specifies the address of just one item of data. By con-

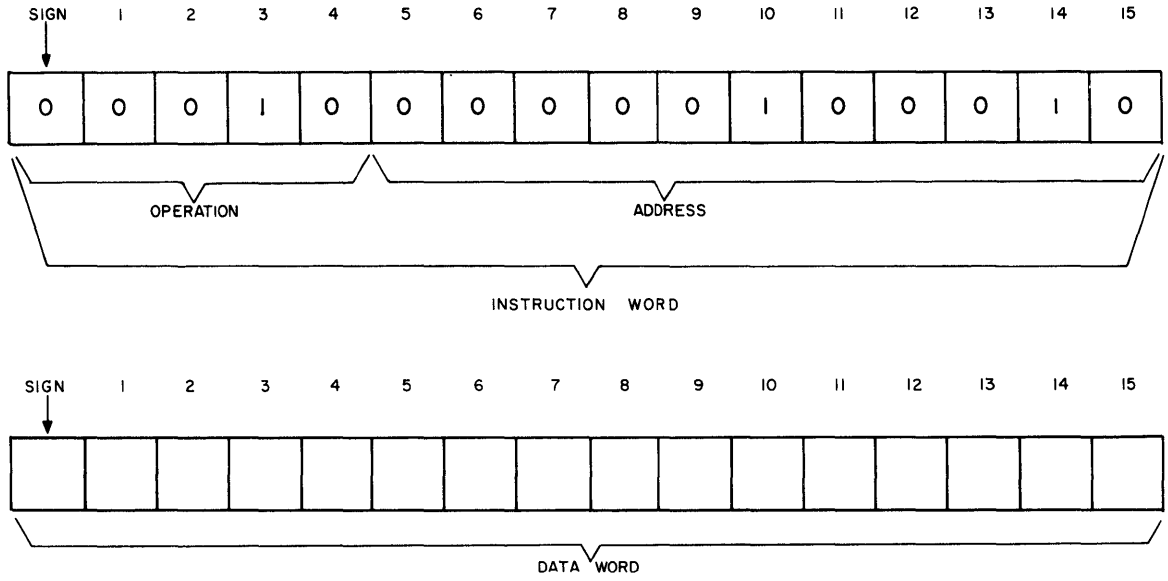


Figure 4—2. Word Format

trast, a multiple-address machine is one in which each instruction word can specify more than one address.

In a single-address machine the addition of a number, a, to a second number, b, and the storage of the sum, a+b, might be indicated as follows:

OPERATION	ADDRESS
CAD	13
ADD	14
FST	15

where: a is stored at address 13; b is stored at address 14; and address 15 is assigned for storage of the sum, a+b. The *Clear and Add* instruction *(CAD 13)* would cause a to be transferred from storage location 13 to the previously cleared accumulator. The *Add* instruction (*ADD*) would then cause b to be transferred from storage location 14 and to be added to the number in the accumulator; and, finally, the *Store* instruction *(FST 15)* would cause the sum, a+b, to be transferred from the accumulator to storage location 15.

The same addition could be specified in a multiple-address machine by the simple instruction:

OPERATION	ADDRESS
ADD	13 14 15

where it is the convention that the first two numbers following the instruction code are the addresses of the operands, and the last number is the address of the location in memory where the result is to be stored.

It would appear from the example that exactly three times as many instructions are required to execute any routine using the single-address system as are required using the three-address system. However, this is not the case. Suppose, for example, that the addition a + b + c + n is to be performed. The single-address machine requires one instruction for each number involved and a final instruction to store the total. The three address machine requires one instruction to add a to b, a second to add (a + b) to c, a third to add (a + b + c) to d and so on. Since the last of these instructions can also specify a storage location for the total, two instructions are saved in adding a sequence, regardless of its length. If the computer is to be used in long operations made up of long sequences this saving is hardly substantial enough to justify the longer word which is required to specify three addresses instead of one. If, however, the operations of the computer tend to be made up of many short disconnected sequences, the three address system might be used.

In deciding which addressing scheme to use, the following advantages and disadvantages should be taken into account. In some programs, the single-address scheme is slightly slower than the multiple-address scheme. However, because the single-address scheme is divided into more basic operations, it is considered to be a more versatile system. It also has the advantage of using a shorter instruction word, especially when a large number of addresses must be differentiated. For instance, a 4-address scheme requires four times as many address bits in each instruction as does a single-address scheme. This, in turn, implies simpler circuitry in the single-address machine. The single-address scheme, therefore, will be used in the sample being developed.

1.2.2.5 Word Length

The length of the computer word is determined upon by consideration of two general requirements, those of the data word and those of the instruction word. The length of the data word directly affects the precision of computing. The longer the word, the more precise the calculations may be. The length of the instruction word is fixed by the number of separate instructions the computer will be required to perform and by the number of memory locations that will be contained in the total computer memory. In the sample computer the length of either word will be 15 bits plus a sign bit. Figure 4–2 shows the makeup of instruction and data words.

1.2.2.6 Arithmetic

Another requirement to be considered in the design of a computer is the type of arithmetic section desired. To a great extent this requirement determines the complexity of circuitry. The binary system, in general, requires the least complicated and most flexible circuitry in the computing and storage sections of the computer, but this system usually requires complicated input-output translation equipment (the equipment to change the decimal numbers to the binary numbers of the computer).

Consider the requirements of the sample system again. At the beginning of the chapter, it was stated that some input information would be from a decimal system manual device; however, the majority of the input information would be from telephone lines. It was also stated that output information will go to telephone lines and to a cathode-ray-tube display. Digital information sent on telephone lines may be sent in binary form. It is also generally true that comparatively simple circuitry can be used to translate binary information into analog information, which can be used for the deflection voltages of a cathode-ray-tube display. Any other number system may require more complicated circuitry.

If the binary system is used internally, the only place complex translation is required is in the manual inputs section. Since there is comparatively little information entered manually, it is practical to do the decimal to binary conversion required, within the computer by means of an easily prepared program. If this is done, the binary system appears to be the best system to use. It is economical, simple, flexible and reliable; it is the system used in the sample machine to be described.

1.2.2.7 Type of Logic

One more requirement related to the arithmetic system to be used is the type of logic. Shall the machine be of the parallel or the serial transfer type? As has already been explained in Part 3, serial logic is generally less expensive in equipment and perhaps more reliable than parallel logic. However, since the computer to be described will be used in real-time control applications, the high speed possible with parallel logic will be a great advantage. The system to be considered, therefore, uses parallel logic.

1.2.2.8 Input-Output System

It has been stated in the requirements that the computer requires a high-capacity input and output facility. To provide this the inputs of the system consist of direct telephone line input and a manual input. The direct input continuously enters data such as radar range and azimuth which comes in over telephone lines in binary form. The manual input allows the computer operators to enter information at the computer site. This is a form of typewriter input.

The outputs consist of a direct telephone line output and a cathode-ray tube display. The automatic phone line output transmits data in binary form to the user (perhaps a remote weapons base). The display equipment will display such things as plane positions on a cathode-ray tube.

1.2.2.9 Summary of General Considerations

A review of the general characteristics of the computer required shows that the more accurate and versatile digital type must be used, not the analog type. The fact that a digital computer is to be used automatically implies the need for the five elements: storage, arithmetic, control, input, and output. Buffer storage is also provided to increase the speed and efficiency of the whole system. To provide the high speed and versatility the stored-program, single-address control will be used. Requirements for simplicity and for high circuit speed will be met by use of binary arithmetic in parallel transfer logic. High data input and output requirements demand the use of direct telephone line inputs and outputs, as well as a manual input and a display output.

CHAPTER 2
SAMPLE SYSTEM STORAGE

2.1 INTRODUCTION

The general system specifications having been stated, it is possible to go into the individual elements of the system. The memory is described in this chapter with particular emphasis upon its interaction with other elements of the computing system.

2.2 GENERAL REQUIREMENTS OF A STORAGE SYSTEM

A computing system must have one, and may have all, of these types of storage. Regardless of the type of storage used, the storage device has certain basic requirements. Briefly, these are:

a. The device must be able to retain information. Since the information to be stored is digital, the ability to store information implies that the device must be able to assume at least two stable states.

b. There must be a way of inducing these states under outside control; that is, there must be a way of writing the information into the device and there must be a way of reading the information from the device. When the storage element is made up of more than one register, it must be possible for an outside control element to select a particular register from many. It should also be possible to read and write at such speed that the rest of the computing system is not delayed by a reference to storage.

c. The storage device should store information in the radix which is used in the internal computing section) except in systems where external storage is used). Otherwise it is necessary to provide a radix translating device (such as the Input System in some external storage systems).

2.3 TYPES OF STORAGE

There are three general types of storage in a computing system. The first is some high-speed storage device such as a magnetic core memory or an electrostatic memory. This storage device stores information to which the computer must have direct access; that is, it stores the data as it is being processed. A second type auxiliary storage, consists of some intermediate-speed storage device such as a magnetic drum. This type of storage is for information which is used fairly often, but not continuously, by the computer; e.g., some tables or portions of the program infrequently used might be stored in auxiliary memory. Finally, there is the external storage, consisting of the magnetic tapes, punched cards, and so forth; these are only available to the computer through an input device. In general, this type storage is for seldom used information.

2.4 TYPES OF STORAGE IN SAMPLE SYSTEM

The requirements stated in Chapter 1 of this part (calling for very high speed, great versatility, and a large capacity) forces the use of a very large direct-access, high-speed memory or a medium-size, direct-access, high-speed memory working in conjunction with a large, intermediate-speed auxiliary memory. These two types of memory can be made almost equal in speed. Therefore, since it is generally true that the slower storage media are a good deal cheaper to build than the faster media, the system chosen is a compromise between requirements of high speed and economy. A combination of direct-access and auxiliary memory is used.

Figure 4–3 shows the relationship of the two internal memories and external storage to each other and to the rest of the computing element. From this figure it can be seen that the auxiliary memory and external storage are not accessible to the computer except through the direct access memory. From there it will be processed like other information in direct access memory. External storage will be furnished. Since this may be in the form of cards, written records, or even operator's memories, there is little point in discussing external storage further here. Just remember that externally stored information enters the computer from an input device and that it usually has some manual operation (such as typing of the information on the computer entry typewriter), connected with it.

2.5 GENERAL REQUIREMENTS OF SAMPLE COMPUTER DIRECT ACCESS MEMORY

2.5.1 Access

A digital computer operates in step-by-step fashion; that is, it executes one complete instruction at a time, and the instruction execution is done in steps. All instructions are stored in the memory, and the execution of most instructions involves the use of data which is stored in memory. The step-by-step nature of computer operation always requires at least one refer-

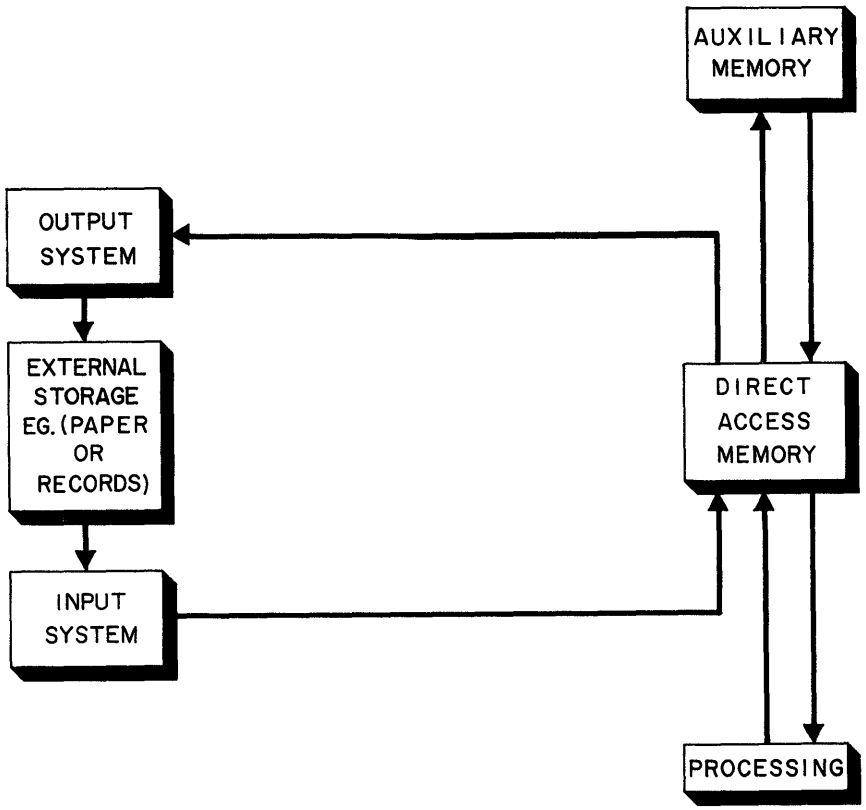


Figure 4–3. Storage Function Relationships

ence to memory during the execution of any instruction, and usually requires two references. The first reference obtains the instruction and the second, data. The operating speed of the computer, therefore, depends as much upon the access time (the time required to withdraw a number from memory) as it does upon the speed of the arithmetic elements.

"Access" can be defined as the method that the computer must use to select a word stored in memory. Access will significantly affect the speed of the memory operation (access time) and, consequently, the speed of the computer.

A computer is said to have a random-access memory when the access times to all registers are equal. This means that it takes exactly the same time to select a memory location with a large address as it does to select one with a small address. Random access is possible when the computer can always "see" all registers of the memory equally well. A core memory and a cathode-ray-tube memory are examples of random access memories.

Some computers have a memory which must be continuously scanned if information is to be withdrawn. That is, when the computer requires a word from memory, it must examine each address in sequence until it finds the proper location and can withdraw the word required. This is called non-random, cyclic access, or block-access. The addressable drums of AN/FSQ-7 furnish an example of this type of access. In these drums, the computer can select a particular memory location only by reading each address as it passes under the read head.

The random-access memory is much faster than the non-random type built with circuitry of comparable speed. This is because the computer does not have to wait for the memory searching process each time a reference is made to memory.

2.5.2 Size

The overall speed of computation also depends to a great extent upon the size of the direct-access memory. If the direct-access memory is small and the program large, parts of the program which should be directly accessible have to be held (for instance, in auxiliary memory). Consequently, computing begins to depend heavily upon the comparatively slow auxiliary memory.

When time is a factor, the versatility of the computer also depends on both the speed and size of the internal memory (direct-access and auxiliary combination). The speed, particularly that of direct-access memory, determines whether a problem of a very great number of steps can be done within a practical time.

The size of the internal memory must be large enough to store all the instructions and data to be used in any problem. If it is not, the speed of execution of the program (part of which requires external storage space) depends to a great extent upon speed of input-output equipment.

2.5.3 Storage Medium

The storage medium affects both the speed and organization of the computer. There are two different classifications of storage medium, volatile and permanent (non-volatile). A volatile storage device is one whose stable states are not truly permanent but depend upon some force to keep them in a particular state. For instance, information may be stored in the form of a charged spot on a cathode-ray tube. If the information is to be preserved, this spot must be constantly renewed or the charge will leak off and be destroyed. Long term storage using a volatile medium can be accomplished as long as some form of automatic rewriting is supplied. However, such storage requires a good deal of extra circuitry in the computer and also uses time which could be used in computing. Another disadvantage of the volatile type of storage medium is the fact that when power is shut off, or when the rewrite process stops for any reason, the information is lost. This can not be tolerated in any system that must be highly reliable.

Some outside force, such as an electric current, is required to set a permanent storage medium in one of its stable states, but once the medium is set, it changes state only if a new force, such as a current of opposite polarity, is applied. An example of a permanent storage medium is the magnetic core used in the AN/FSQ-7 memory; it is set to one state by magnetizing it in one direction. It then retains that state until it is forcibly magnetized in the opposite direction. There are many advantages to such a system. It saves computing time since no regeneration is necessary. It saves circuitry for the same reason. Finally, the fact that information is not lost during a power failure allows high reliability.

2.5.4 Memory Controls

The control of the various functions of the memory can be centered within the memory unit itself or within the control element of the computer. Where large memories are used, it is usual to accomplish this control within the memory unit. All that is then required of the computer control element, when memory must be referenced, is a pulse to tell the memory ot start the read operation. Once this start pulse is received at the beginning of a transfer, the memory circuits control the rest of the operation. Such a control system allows the computer to carry out some of its functions while the memory is in the process of transferring information, increasing the speed of the computer. Autonomous memory control is, therefore, the most usual type of memory control in large-scale memory systems.

2.5.5 Summary of Requirements

The sample computer is to be very large, fast, versatile, and reliable. A review of the points stated for memory indicate what points to seek in selecting a suitable memory system.

To get a large, fast memory at a reasonable cost, a combination of medium-speed and high-speed memory should be used. It is easiest to obtain the small access time required if a random-access memory is used for primary storage. Furthermore, speed and versatility require a large primary memory; i.e., it must hold as much of the program and data at one time as possible. Both the speed and reliability of the primary memory are best if a permanent storage medium is used. A greater computer system speed can be had if the memory operates under autonomous control.

All of these requirements suggest that the magnetic core memory discussed in Part 3 should be used for primary storage. A random-access memory using magnetic cores has been constructed with a 6-microsecond access time. Moreover, this same memory is sufficiently large and reliable to perform the primary memory job for the AN/FSQ-7. A similar memory is to be used in the sample computer.

The sample memory to be described has the following general characteristics:

a. It will store 1,024, 16-bit words in a core array.
b. It will be a random-access memory with a 6-microsecond access time.

2.6 MAGNETIC CORE STORAGE

2.6.1 Operation of Array

In order to understand the requirements for logical circuitry to implement the transfer of information into and out of the core memory, it is necessary to specify the following:

a. The form in which information is stored in the cores
b. The mechanism by which information is written into the cores
c. The mechanism by which information stored in the cores is sensed during the readout process
d. The organization of the core array into individual storage locations

In the discussion which follows, a core array like the one described in Part 3, 4.2.3, is assumed. A review of the operation of this array follows.

Information is stored in the cores in binary form. Magnetization in one direction is interpreted as a 1; magnetization in the other direction is interpreted as a 0.

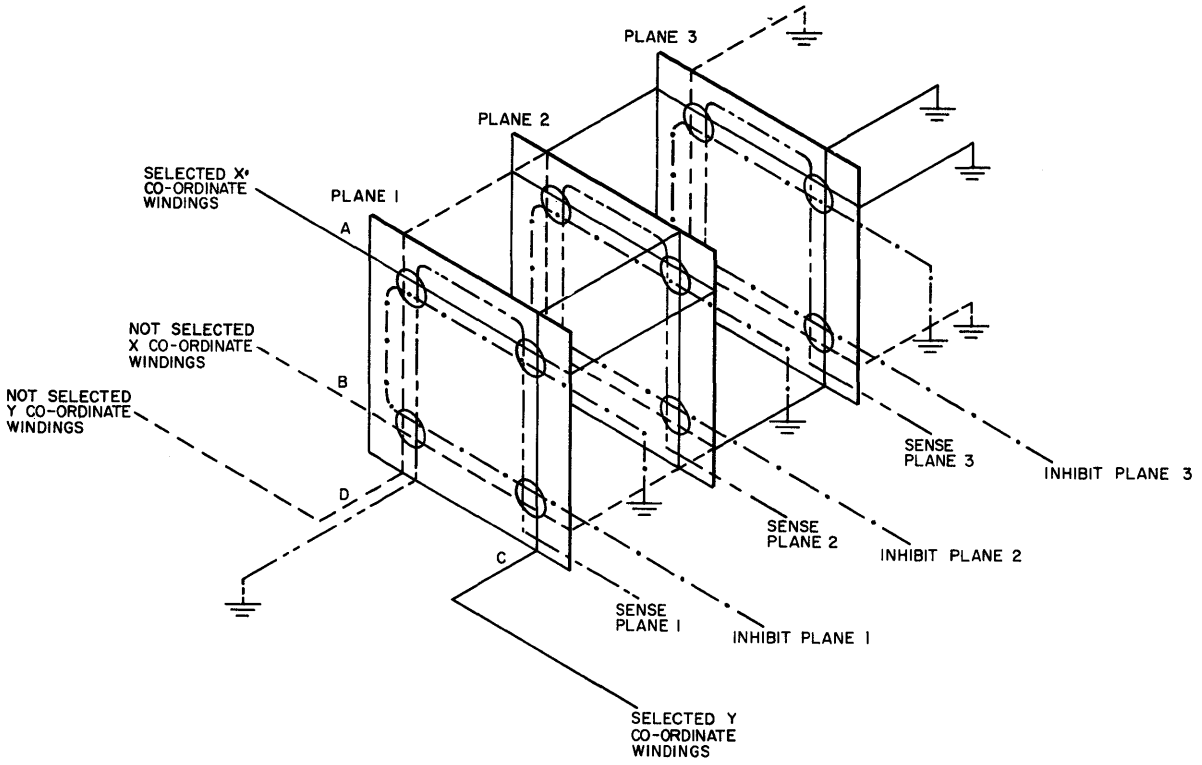


Figure 4—4. 4-Location 3-Bit Register Core Array

Figure 4—4 illustrates the basic selection principles of a 4-location, 3-bit-register core array. In order to write 1's into all the cores of a particular memory location, so-called half-write pulses are applied to two coils (designated as X and Y co-ordinate windings) wired as shown in the figure. If the half pulses are applied to winding A and winding C, the cores where the two windings meet have two half-write pulses applied at the same time. In these cores the fluxes induced by the currents add up to a full-write pulse, which is enough to switch the core from a 0 to a 1. Therefore, a core on each plane, located where the A and C windings cross, will have a 1 written into it. The state of the other cores on the X and Y windings is not changed since they only received half-write pulses. The core on each plane which received a full-write pulse constitutes a single-bit position of the location selected by the A and C line combinations.

In general, of course, a word of information contains 0's as well as 1's. If all the cores of a storage location are cleared (i.e., driven to 0) prior to the writing of a word into that location, the 0's of the word may be produced simply by inhibiting the writing of 1's on those cores where 0's are to be stored. This can be done by applying an inhibit-current pulse to a third coil (inhibit plane winding) associated with each of the cores where a 0 is to be stored, just at the instant when the half-write currents are applied. The polarity of the flux caused by the inhibit current must be opposite to that of the write current fluxes so that the cancellation inhibits the writing of a 1. For instance, to write 101 in location A-C of figure 4—4, the location is first cleared to all 0's. The A and C lines are pulsed with half-write pulses in an attempt to write 1's into all positions. At the same time, an inhibiting pulse is applied to the inhibit winding of plane 2. This prevents the writing of a 1 on the selected core of plane 2. The stored number would then be 101. An important point to understand is that this system implies that all the cores in a location must be set to 0 before a word consisting of 1's and 0's may be written.

In order to sense a word stored in a particular storage location, so-called half-read-current pulses are applied to the X and Y lines associated with the cores of that location. The half-read currents are equal in magnitude but opposite in polarity to the half-write currents mentioned earlier. The effect of the simultaneous reception of two half-read pulses is, therefore, to drive a core into the magnetic state representing 0. If a 0 is stored in the core, there is no change of magnetic state. If, on the other hand, a 1 is stored in the core, it

reverses the core's magnetic state. This polarity reversal induces a voltage pulse in the sense winding associated with the plane of cores. The readout process, therefore, causes pulses to appear on the output lines associated with each of the cores in the selected location that contains a 1. Notice that all the cores are left in the 0 state. In other words, the 1's were all destroyed in the read process. This is called destructive readout.

The half-read pulses, like the half-write pulses, perform a selection function. The form of the word read out is, on the other hand, established by the signals appearing on the sense windings.

In summary, four coils are associated with each core: the X and Y (selection) windings, the inhibit winding which forms the pattern of the word being written, and the sense winding which senses the outputs of the cores on readout. The selection scheme involving X and Y windings is based upon the organization of the core array into a number of X groups and a number of Y groups. The coils of any one X group or any one Y group are connected in series. A set of cores which belongs to a certain X group and a certain Y group comprises a storage location. Thus, to select a storage location for reading or writing it is merely necessary to apply half-read or half-write current pulses to two input terminals, namely the input terminals of the particular X and Y groups which specify the particular storage location. It should be understood that cores of many locations belong to each X group and each Y group. However, when a core is supplied with only one half-write or half-read current its magnetic state is not affected. For this reason it is said to be only half-selected.

Since information is transferred into or out of only one location in the array at any one time, groups of sense windings or groups of inhibit windings can be series connected on a bit basis. For example, the cores which store the first or most significant bit in each of the locations of the array can have their sense (or output) windings connected in series. They can also have their inhibit (or input) windings connected in series. The same thing is true of the cores which store the second bit, and the third, and so on. This means that the same set of output lines can be used to sense the bits of every location in the array. Also, one set of input lines may be used to perform the inhibit function for every location in the array.

Now that the organization of the array has been reviewed, it is worthwhile to return to two characteristics of the read-write scheme which were mentioned earlier:

a. A memory location must be cleared of 1's prior to the writing of information on it.

b. Readout from the core memory is destructive.

Since readout is destructive, a read cycle can be used to clear a location prior to the performance of a write operation. Also, since it is usually desirable to retain in a memory location a "copy" of the word read out of it, a write cycle normally follows the destructive read operation. These two facts can be restated as follows: Regardless of whether the purpose of an operation is the transfer of information out of the core memory or the transfer of information into the core memory, the same basic cycle must be performed; i.e., a read operation followed by a write operation.

2.6.2 Sample Computer Memory Element Operation

Figure 4–5 illustrates the operation of the memory during both read (solid lines) and write (dotted lines) cycles of a word transfer operation. The core array is assumed to be a typical ferrite core array capable of storing 1,024 16-bit words. The timing and control section shown is a delay line control such as that described in Part 3, Chapter 3, paragraph 3.6.2.2. The sequence of memory-control pulses generated by this control device is initiated by a read-operation or write-operation pulse from the computer control. The selection section is composed of diode matrix decoders. These matrices decode the address information which specifies the location to which, or from which, a word is to be transferred. The sense section consists of amplifiers and gates which can sense the output of the core array. The memory buffer is a flip-flop register which is actually under control of the computer control rather than the memory timing and control section. Finally, the inhibit section is a series of gates conditioned by the zero sides of the memory buffer and pulsed by the timing and control section.

The operations of the memory are similar whether a transfer out of memory or a transfer into memory is called for. In both cases, dual cycles will be required, one cycle to read from the array, the other to write into the array. Since the transfer operations are very similar whether information is entering or leaving memory figure 4–5 may be used to show both operations. In this figure, the operations accomplished during a read cycle of any transfer operation are shown in solid lines. The operation which occurs during a write cycle is shown in dotted lines.

When the purpose of the cycle is to transfer a word out of the core memory, the memory control circuits transfer the word from the array to the memory buffer register. The process is as follows: The cycle is started by the computer control element. One clears the timing and control section and the memory buffer so that they are ready to start the memory cycle. Two control pulses are sent to the memory element. The other, the read-operation pulse, starts the memory and specifies the

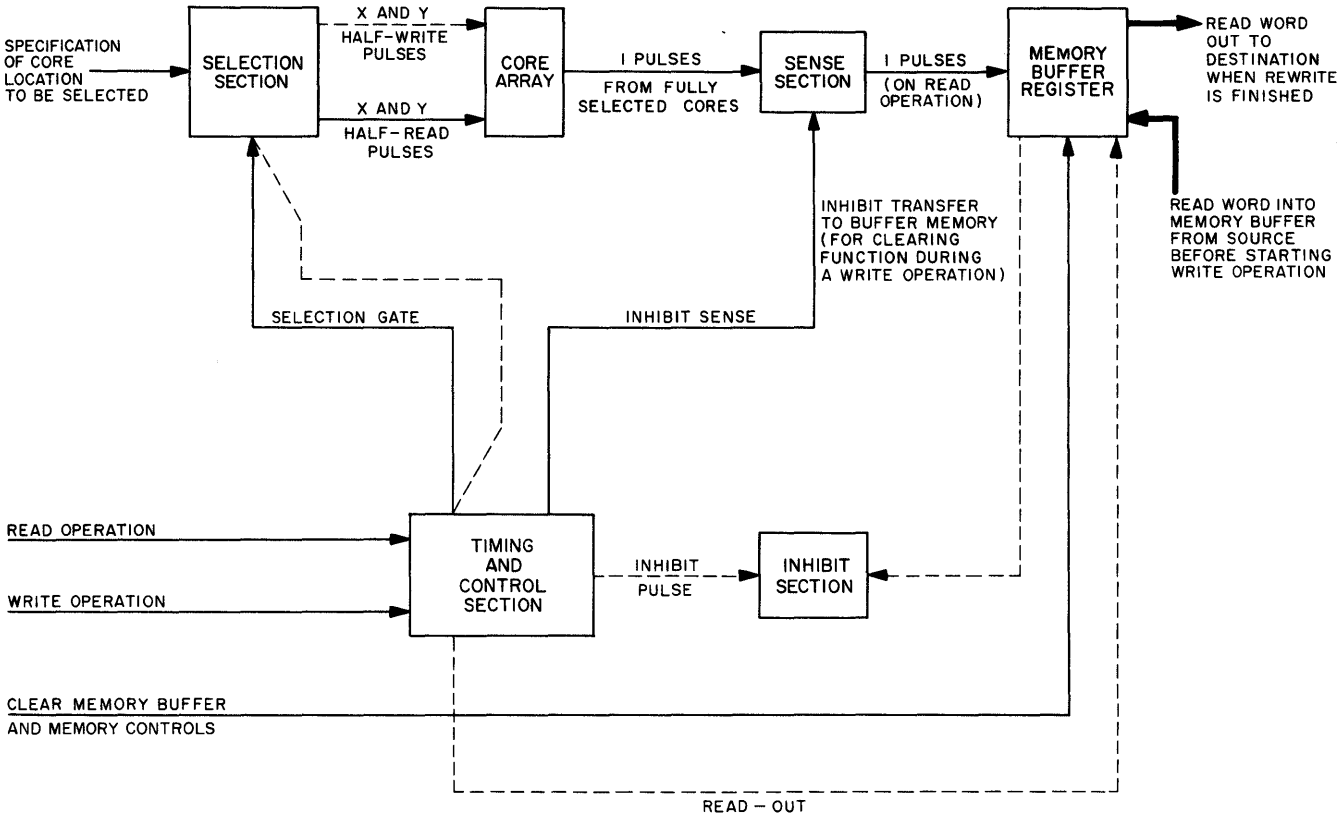


Figure 4–5. Read and Write Operation of Memory

purpose (read) of the operation. At the same time the computer control element sends the address of the word to be selected to the selection section of the memory. From then on, the completion of the transfer operation depends upon the internal controls of the memory itself.

Half-read-current pulses are supplied to the X and Y coils of the specified storage location, causing all those cores containing 1's to be driven to the 0 state. The polarity reversal of these cores is sensed and is caused to condition the previously cleared buffer register, which now stores the information temporarily. (It is now apparent that the buffer register had to be cleared prior to receiving the sense pulses for, otherwise, it might contain some 1's which did not originate in the core location.) The read cycle of the operation is followed by a rewrite cycle. During this cycle, the 0's of the number temporarily stored in the memory buffer register are gated through the inhibit section to furnish inhibit pulses to the location which is again fully selected, this time by X and Y half-write pulses. (In fig. 4–5, the operations in this rewrite cycle are indicated by dotted lines.) The result is that the word removed from storage is rewritten in the same location. However, a copy of it remains in the buffer register ready for transfer to some other part of the computer.

When the purpose of the cycle is to transfer a word into the core memory, the memory control circuits transfer the word from the memory buffer register to the array. The process begins with the control element causing the word to be transferred into the memory buffer register from the source (e.g., the arithmetic element). This time the operation is started by a write-operation pulse from the computer control element. The memory, therefore, knows that the transfer is to be into the memory. As in the transfer from memory, the control element also sends address information to the selection section of the memory. From then on the operation depends on the memory control circuits for completion.

The storage location is first fully selected by X and Y half-read currents from the selection section just as in the previous case. However, in this operation, the voltage pulses induced in the sense windings by the cores containing 1's are of no interest, because the operation is performed merely to clear the location for the receipt of the new word. Moreover, since the buffer register is temporarily storing the word to be read into the storage location, the contents of the buffer register

must not be destroyed. Consequently, the output of the sense windings must be prevented from conditioning the buffer register. This is done by conditioning the inhibit sense line. By referring again to the dotted lines of figure 4–5, it can be seen that, after the storage location has been cleared by the half-read pulses, it is fully selected again, this time by means of X and Y half-write pulses. Simultaneously, the 0's of the word in the memory buffer, which is to be entered into storage, are used to generate inhibit pulses. Thus, the new word is written into the selected location just as in the write cycle of the read operation.

In summary, it can be seen that, because readout of a core memory is destructive and because writing requires a clearing of the memory location, it is convenient to make the read and write operations as nearly alike as possible. A read cycle and a write cycle are provided regardless of whether the operation is a transfer into or out of memory. The difference between the two types of transfer lies in the operation of the memory buffer and the sense section. Table 4–1 summarizes the difference.

2.6.3 Operation of Memory in Computing System

The operation of memory in a stored program computing system is based upon the fact that both instruction and data words are stored. The two types of information are used for quite different purposes: One is used to tell the computer what operation to perform; the other gives the computer the data that the operation is to be performed upon.

Despite the difference in the use of the two types of words, they are indistinguishable in form. That is, they are both binary numbers and, in most cases, are of equal lengths. The computer distinguishes between the two by keeping track of where the two types of information are stored. In a single-address computer distinguishing is usually done by storing the instructions in sequence in the first addresses of memory while the data words are stored toward the end of the memory addresses. For instance, the first 300 locations of a 1,024 location memory may contain the instructions sequence 0, 1 through 299 of the program while the data is kept in the last 724 locations. With such a system, the computer can be built to automatically read out the instructions in sequence. The data address is obtained simply by reference to the address part of each instruction word.

Consequently, the execution of an instruction requires obtaining data from memory, at least two memory references must be made. The first is the reference to obtain the instruction from the program sequence; the second is to obtain data from the location in memory specified by the address part of the instruction.

In figure 4–6, the contents of 16 locations in memory are shown. The contents consist of a simple 4-step program in locations 000 through 003 and the data which the program is to operate upon in locations 013, 014, and 015. At the end of the programs execution, location 015 will contain the result of the program (1600). The problem can be stated $1000 + 600 = 1600$. The execution of the program would proceed as follows:

a. When the computer is started, the computer control automatically calls for the information in location 000 to be transferred from memory to the control element.

b. The control element decodes the instruction thus obtained. Since a reference to memory is required, the control element sends address information associatd with the instruction (location 013) back to the memory selection circuits and calls for a memory read operation.

c. The read operation transfers the data in location 013 (1000) to the accumulator of the arithmetic element.

d. At the completion of this instruction, the computer automatically refers to the next location in the sequence of instruction (location 001).

TABLE 4–1. SUMMARY OF DIFFERENCES BETWEEN READ AND WRITE CYCLES OF A READ OPERATION AND A WRITE OPERATION

CYCLE	TRANSFER INTO MEMORY	TRANSFER OUT OF MEMORY
Read	1. Started by write-operation command.	1. Started by read-operation command.
	2. Clearing operation.	2. Transfer operation.
	3. Contents of memory location are not sensed.	3. Contents of memory location are sensed.
	4. Contents of memory buffer are not changed.	4. Contents of memory location transferred to memory buffer.
Write	1. Rewrite former contents of memory location back into memory location.	1. Write word from outside source into memory location via the memory buffer.

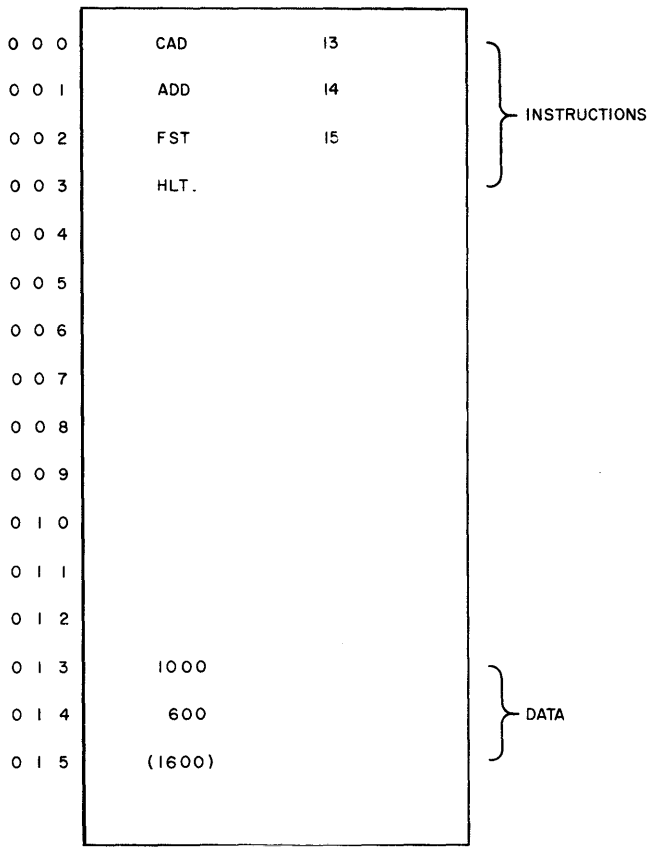


Figure 4–6. Contents of Memory During Execution of Program

e. The instruction in this location is transferred to the control element, where it is decoded.

f. Again, the computer control ascertains that a reference to memory is required. Therefore, address information (location 014) is sent to the memory selection circuits and another memory operation started. The word in location 014 (600), is transferred to the arithmetic element and added to the contents of the accumulator. (As a result the accumulator contains 1600.)

g. The computer then automatically refers to the next address in the program sequence (at location 002). The information in this location is transferred to the control element to be decoded.

h. The control element determines that the reference to memory is a storing operation. Therefore, the control element sends the address information associated with the instruction (location 015) to the selection circuits of memory, together with a store operation control. The information in the accumulator is then transferred into location 15 of memory via the memory buffer register.

i. When the *Store* instruction is completed, the computer automatically refers to the next location in the program sequence. This instruction is a *Halt* instruction which stops the computer so that no more memory locations will be referred to, either automatically or by instruction.

This last instruction, it will be noted, distinguishes the instruction words from the data words in the memory. If no *Halt* or similar instruction were given, the computer would automatically continue reading successive memory locations until the machine was shut off. This means that eventually (steps 13 and those following) the computer would try to use the data words as instructions.

2.7 AUXILIARY MEMORY

2.7.1 General

It has been stated that the internal memory of the sample computer consists of primary memory and auxiliary memory. It may be recalled that the primary and auxiliary memory combination is designed to provide a very large memory, almost as fast as a single, large, high-speed memory would be. This system has the desired characteristics if two provisions are fulfilled:

a. The two storage systems are compatible in speed and operating characteristics.

b. The information to be stored is organized so as to take advantage of the total memory system.

2.7.2 Choice of Auxiliary Memory Medium

In the example computer, core memory is used as direct-access memory. Choice of the auxiliary memory medium must, therefore, be limited to one which is compatible with the core memory. The auxiliary memory must also be one which has a potentially large storage capacity. It must also be as reliable as possible since the computer can only be as accurate as the data entered into it. Drum storage can be made to fit all of these requirements and so is used as auxiliary memory in the sample computer.

2.7.3 Operation of Sample Computer Auxiliary Memory

2.7.3.1 General

Figure 4–7 shows a typical drum and indicates the configuration of bits stored. On the right is a timing track of bits permanently recorded around the drum. This timing track, as the name implies, is used to time the reading and writing operations of the drums. The head, which reads the bits of the timing track, puts out a pulse which initiates the read or write operation at each register location around the drum. It can be seen, then, that the timing track determines the position of each register around the drum.

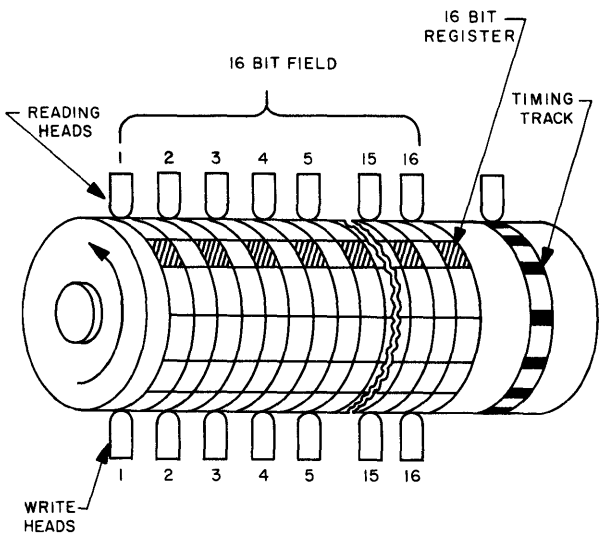


Figure 4–7. Auxiliary Memory Drum

When the timing track is originally put on the drum (during manufacture), it is recorded as evenly spaced bits around the drum. However, one bit is left out to indicate a beginning and end of the drum field. This is called the index point of the drum. Because the beginning of the field is indicated, it is possible to address the different registers of the drum with a counter. The address of a register is a count of the number of registers (timing track bits) between the index point and the register.

Figure 4–7 shows a drum designed for parallel read and write operation. When a single word is to be read out, all the read heads of the field are conditioned to read simultaneously. Then, as the magnetized spots of the register to be read pass under the heads, they are read out simultaneously.

2.7.3.2 System Operation

Figure 4–8 shows the drum system in more detail. The drum with its timing track and its read and write heads is the same as that just described. The read circuit consists of amplifiers and gates which transfer a word being read from the read heads to its first destination, the drum read-and-write register. The write circuits perform the same function in reverse for words to be read into the drum. The drum read-and-write register is a temporary storage register for words waiting to read into the drum from core memory or into core memory from the drum.

The timing and control section is a long delay line control. In this device the timing cycle is started by a pulse from the timing-track head. This pulse is routed through a line of delay units and gates to properly sequence the control function required. The control (gates) of the timing and control section are conditioned for a write or read operation according to the type of start pulse received (start-read or start-write).

The angular position counter and the compare section are the selection circuits of the storage system. The angular position counter counts timing track bits after being reset to zero by the timing track index point. The count of timing track bits indicates the drum register address. The compare section compares the angular position counter contents (drum register address) with the register address specified by the computer instruction. When these addresses compare correctly, the compare section initiates the transfer operation required.

A read operation begins with the instruction *Read Auxiliary Memory Address 10 (RAM10)* which is decoded by the computer control section. It sends the address information (10) to the drum compare section and then sends a read-from-drum pulse to the drum timing and control section. The timing and control is conditioned to cause a read operation to start. The timing and control section then allows the compare section to compare the address specified by the program with the contents of the angular position counter (the address of the register under the read heads). When the location addresses compare, the sought for register is about to come under the read heads. The compare section, therefore, conditions the read circuits to transfer the word read to the drum read-write register. Here, it is held until the program directs it to be read into memory. When the compare section conditions the read circuits, it also sends a transfer-complete pulse to the computer. This pulse notifies the computer control element that the word is in the drum read-write register, ready for transfer to core memory. The program, which was automatically stopped by the *RAM* instruction, can then proceed to the next instruction.

The write process operates in the same way except that the transfer is from core memory into the drum read-write register and then into the drum by way of the write circuits. The write operation is started by a write-on-drum pulse. In figure 4–8 the dotted lines indicate write operations which differ from the corresponding read operations.

2.7.3.3 Program Operation

Auxiliary memory information is accessible to the computer only through direct-access memory. The particular (much simplified) system of transferring information from the computer auxiliary memory to the primary memory requires two instructions. One instruction (*RAM* drum address X) specifies the reading of a word from a specified auxiliary memory register into the drum read-write Register. A second instruction, (*SAM* memory address X) must then be given to

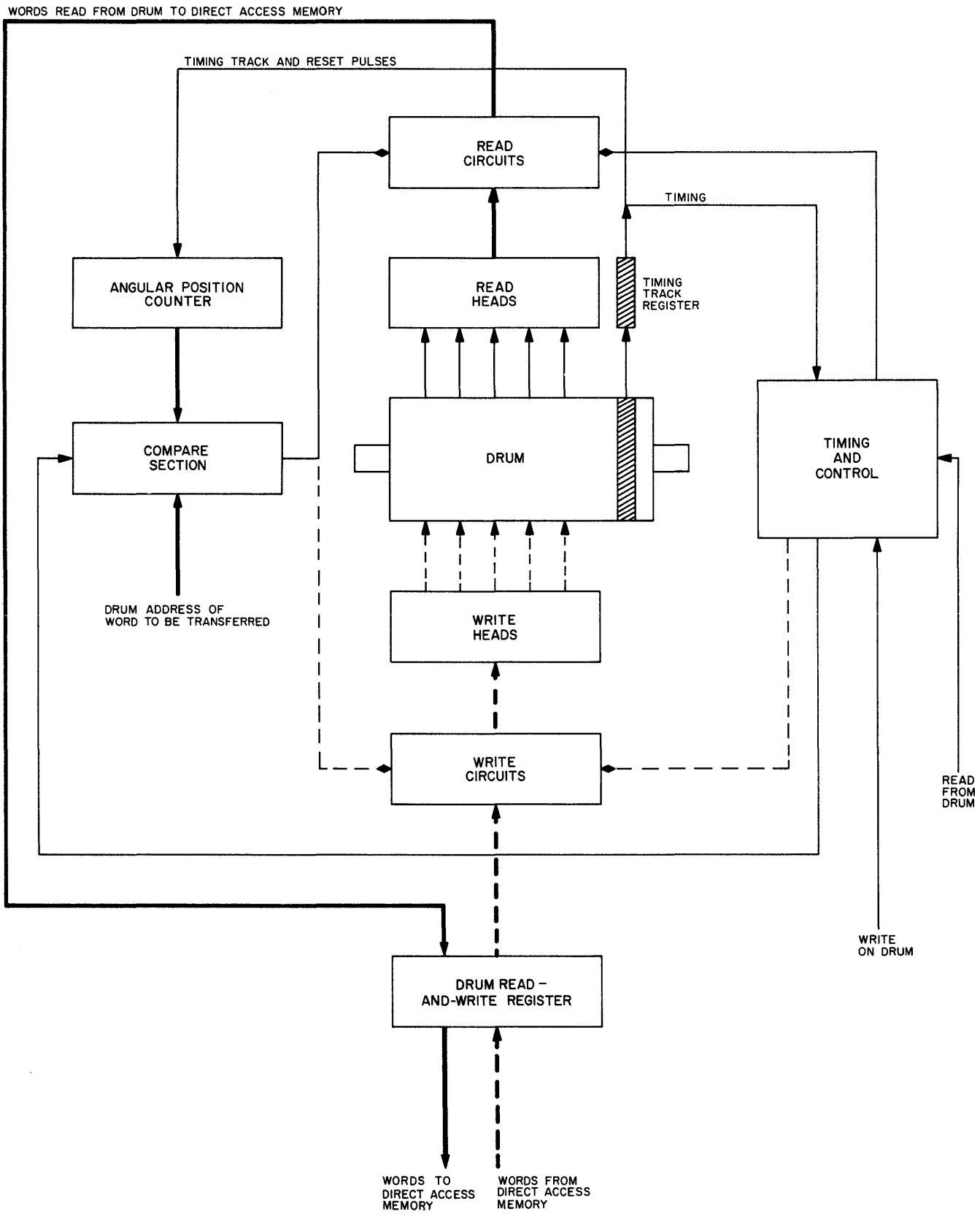


Figure 4–8. Auxiliary Memory Drum System Operation

cause the word in the drum read-write register to be stored in a specified core memory register.

When it is desired to transfer the word in auxiliary memory register 21 into core memory register 77, the program appears as follows:

RAM 21 – word in drum register 21 to drum read-write register

SAM 77 – word to memory location 77

One thing should be pointed out about the *RAM* instruction. Most instructions take a specified length of time to complete. This instruction may require a few microseconds or several milliseconds, depending upon the position of the specified drum register when the instruction is given. To take care of this condition, this instruction automatically puts the computer into a pause condition (stops progression of program) until the transfer to the drum read-write register is complete. When the transfer is complete, the *SAM* instruction will be executed, but not before.

One more thing should be noted about this transfer operation. Normally information is transferred in blocks (several words at a time) by means of a repetitive program whose principles are discussed in Part 5.

When it is required to transfer a word from direct-access memory to auxiliary memory, two instructions are again required. One instruction (*WAM* core memory address X) specifies the writing of the word at core memory address X into the drum read-write register. The other instruction (*TAM* drum address Y) specifies the transfer of the word from the drum read-write register to drum register Y. The same type of program is used to accomplish this operation as in the previous example.

2.7.3.4 Operation of Direct Access and Auxiliary Memory

The direct-access memory stores any data or program which is in immediate use by the computer. Data processing is essentially a process of transferring information between two storage devices through a switching mechanism. The direct-access memory serves as the eventual source and destination in this process. In other words, direct-access memory is actually a part of the processing machinery. It is the only major storage system to which the computer has direct access.

The auxiliary memory is used to store information which is not of immediate use to the computer. However, when any is needed, a great deal is required at high speed. For instance, auxiliary memory is often used to store tables of data which are needed in certain computations. A table of friendly aircraft identifications could be stored in auxiliary memory. Such a table probably would be used only occasionally. However, when it was used in an identification, it would be necessary to use the whole table, consisting of, perhaps, identity information for 50 airplanes. Auxiliary memory is also used to store parts of the program. Suppose, for instance, the program is too large to fit completely in direct-access memory. In this case, part of the program would be stored in direct-access memory and part in auxiliary memory. Such a program might be executed in three phases. First the part stored in direct-access memory would be completed, next, the part stored in auxiliary memory would be completely read into direct-access memory, and, finally, the new part in direct-access memory would be completed.

The speed and practicality of the memory system which consists of a combination of high-speed, direct-access memory and intermediate-speed auxiliary memory are directly dependent upon the type of information stored. Information in auxiliary memory should be of the type which is useful in blocks. Another requirement of the information is that it should be of the type used in repetitive operations, that is, it should be information which is used many times, once it is called in from auxiliary storage. If it is not, the proportion of time spent loading the information into direct-access memory could become so great that it would be more practical to use a single, fairly high-speed, very large, direct-access memory in place of the combination. As an example, consider a table of four numbers which is stored in auxiliary memory. Assume a 6-microsecond access time for direct-access memory and a 24-microsecond transfer time for auxiliary-memory-to-direct-memory transfers. Forget the time required to find the first drum register. The total transfer time for the four words of the table would then be approximately 18 (3 transfer initiating instructions x 6 microseconds) + 96 (4 transferred words x 24 microseconds), or 114 microseconds. Now, if the reference to the table, once it was in direct-access memory, took three instructions (18 microseconds), the total time for reference to the table would be about 132 microseconds. This is also the average access time to the table. On the other hand, if the table were to be referred 100 times in some repetitive program the average access time would be much less. If this were the case, the time of loading the four numbers would be the same, 114 microseconds. The total time consumed in reading the table 100 times after it was loaded would be 1,800 microseconds. The average access time to the table would then be $\frac{1,914}{100}$ microseconds or about 19.1 microseconds per table reference.

In summary, it can be said that a combination of auxiliary and direct-access memory is used to provide a high-speed, large capacity memory at reasonable cost. The speed and practicality of the system depends directly upon the type of information stored and upon its organization.

CHAPTER 3
CONTROL

3.1 BASIC CONTROL ASSUMPTION

3.1.1 Sequential Operation

A digital computer functions step by step. The operation, therefore, depends upon a sequential control of various functions performed by the computer. To provide this sequential control, a control element is included. This element is primarily used to control the interaction of the operations performed by each of the other elements so that together they can operate as a system. In some cases, the control element can also be used to control the operation within an element. In any case, however, the control device performs a similar function; it determines the sequence in which the various parts of the computer operate.

3.1.2 Types of Control

In general, three types of control are possible: synchronous, asynchronous, and a combination of these.

3.1.2.1 Synchronous Operation

In synchronous operation the beginning and end of each operation performed and the timing of that operation are determined by some central source. This means that each operation is done in a cycle or in an integral number of cycles, usually all of equal length so that it takes the same time to complete two operations no matter how long one operation is or how short the other is, if both can be done in the same number of cycles.

The synchronous mode of operation requires a central timing source to determine the length of a cycle and the number of cycles to be used throughout the computer. Without a central timing source, the various parts of the computer could not be synchronized. Usually, this timing source is in the form of a pulse-output oscillator of fixed pulse repetition frequency (PRF), together with a time pulse distributor. The oscillator furnishes the basic timing pulses to the machine. The time pulse distributor separates these time pulses into cycles of equal length and distributes them where they are needed. For instance, it might split the time pulses into consecutive trains of 15 pulses each to make successive cycles 15 pulse-times long. These pulse cycles can then be sent to the control element to furnish the time-sequencing necessary to execute the instructions.

3.1.2.2 Asynchronous Operation

Asynchronous operation is that in which the beginning time of each new operation is determined by the ending time of the previous operation. In other words each operation may have a final pulse which is used to start the next operation. In this case, there is no set cycle time. Those operations whose execution require very few different operations do not take as long to do as those which require many operations. An example of an asynchronous operation is the operation of the auxiliary memory drum mentioned in the previous chapter. When an instruction is given to transfer a word from the drum, the program progression stops until the transfer is complete, at which time the drum controls restart the program progression.

Theoretically, in a completely asynchronous parallel machine no timing other than that furnished by the delays inherent in each operation would be required. Usually, however, some source of central timing is required to furnish time pulses.

3.1.2.3 Synchronous-Asynchronous Combinations

Synchronous operation is generally somewhat slower than asynchronous operation. However, the fact that a synchronous system is easier to design and maintain than an asynchronous system makes it more suitable for computing circuits where the difference in comparative speeds is small. Computing circuits are usually designed to operate synchronously. However, where the possible time saving is great such as when a computer is transferring information into or out of mechanical equipment, asynchronous operation is often used. In this case, the two systems might operate synchronously within themselves, but their operation together would be asynchronous. That is, the computer would arrive at its results using synchronous operation. The transfer of the results to the output would be on demand, or an asynchronous operation.

Actually there is little difference in a control function whether it be accomplished synchronously or asynchronously. The function is always to make sure that the various computer operations occur in proper sequence. Therefore, since the synchronous system is much easier to understand, the following explanation of control deals almost exclusively with synchronous control.

3.1.3 Coding

The ability to follow a list of instructions, such as

a computer program, requires first of all, a method of communicating the instructions in the list to the computer. This is done by means of a special operation or instruction code, which each computer is designed to follow. This operation code may be in any of several forms. It may be for instance, that the entire program is built into the machine; this is the case in special purpose computers meant to do only one specific job. It also may be that the code will be in the form of numbers or instruction words, which can be put into the machine just like data. Such a machine may be able to execute 50 different instructions. If this is so, 50 numbers will be assigned to stand for these instructions, and the control element will be built to respond to these numbers. This last is the case with which this manual is concerned. This is called a stored program computer.

When the operation code consists of a series of numbers, each of the numbers (instructions) has the same basic format. In general, instructions will consist of two parts as illustrated in figure 4–2. One part, called the operation part, specifies the operation to be performed. The other, called the address part, specifies the location in memory where the operand to be operated upon may be found. There are some exceptions to this general rule, however. Sometimes there is an instruction for which no operand must be specified. For instance, the instruction may tell the computer to stop computing. In this case the address portion of the instruction is automatically ignored.

3.1.4 Basic Control Element Functions

Two fundamental abilities are necessary if the control element is to be able to follow a list of instructions made up of code numbers as described above: The control element must be able to pick out the correct instruction to perform at all times; the control element must be able to interpret each instruction and to command the action from the other elements which will execute the instruction.

3.1.5 Program Time: Operate Time

Each instruction execution may be divided into two phases. During the first, which is commonly called "program time," the control element receives the instruction and decodes it. During the second phase, commonly called operate-time, the other elements are caused to perform the indicated operation.

During program time, the computer generally does three things. It selects the instruction to be executed; it will cause this instruction to be read into the control element; and, finally, it decodes the instruction. This decoding includes conditioning the circuits in other elements to perform the operation called for, as well as selecting the memory location of the operand.

During operate time two things generally occur: The selected operand is read out of memory (or in some instructions read into memory), and the operation is performed.

In any computer control function, two elements are used to perform the control. One element is the logic of the circuits. The other is the timing of the circuits. For instance, a 2-way AND circuit may have an output only if both inputs are up at the same time. If the inputs go up and down at different times, no outputs are produced. In the sample computer, this principle of timing is used to differentiate between program-time operations and operate-time operations by using diffrent timing devices for the two types of cycles. During program time, time pulses come from the program-time pulse distributor; during operate cycles they come from the operate-time pulse distributor. These two time-pulse distributors never distribute their pulses simultaneously. (See fig. 4–9.) A time pulse distributor is a circuit which distributes pulses entering it on a single line into cycles of pulses leaving it on several lines.

3.2 OPERATION OF SAMPLE COMPUTER CONTROL

3.2.1 General

Both instructions and operands are stored in memory. Most instructions require two memory cycles for execution. One memory cycle is required to obtain the instruction from memory; another is required to obtain the operand from memory. This is in accordance with the idea of program time and operate time mentioned in the previous section.

3.2.2 Program Time

3.2.2.1 Program Sequencing

The method of determining the sequence of instructions during execution of a program is based upon the method of storing the instructions. The instructions are always stored in the proper sequence in consecutive memory locations. Usually the instructions are located at the beginning of the memory addresses (lower numbered addresses). For instance, the first instruction might be stored at location 000, the second at location 001, the third at location 002, and so forth. With such a system of program storage, it is simple to keep track of the location of the instruction to be executed. This is done by the program counter, which counts the number of instructions executed. Since the counter starts at 0 and the address of the first instruction is 0, the contents of the counter always equal the address of the instruction to be executed. Whenever an instruction is to be read from memory, the contents of the counter are transferred to the memory-selection circuits to indicate the address of the instruction to be selected.

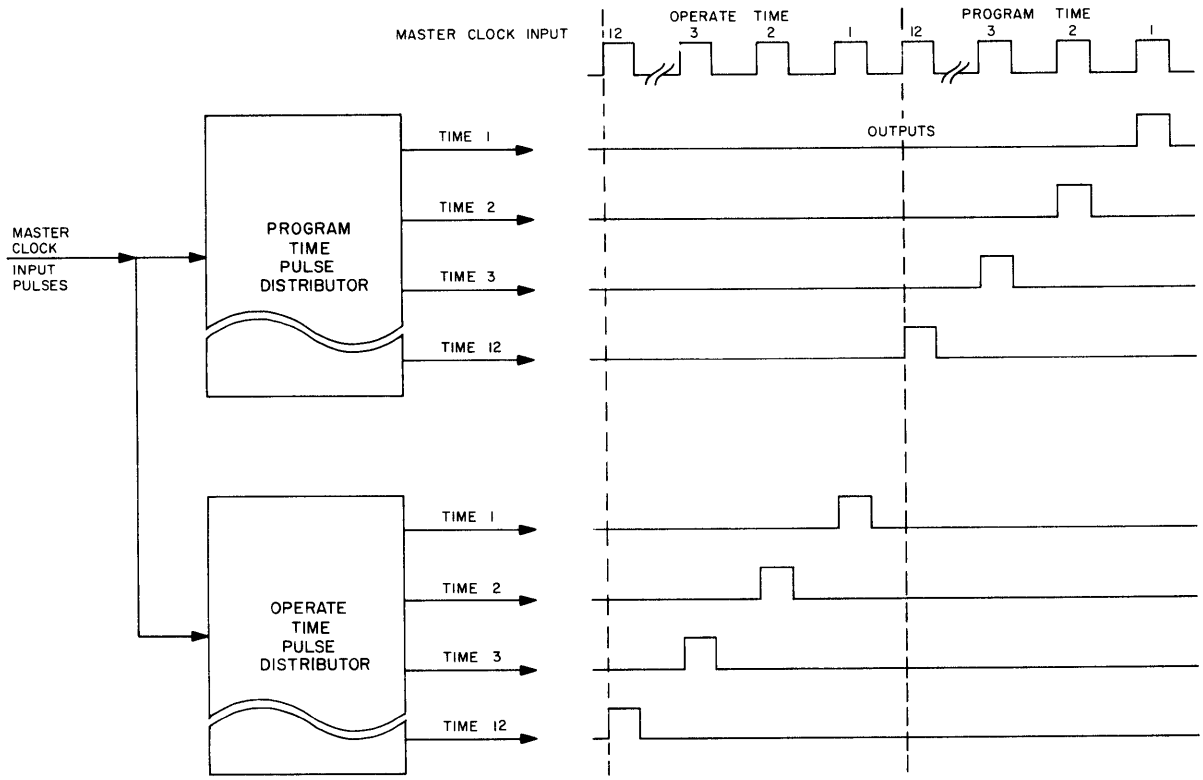


Figure 4–9. Program Time: Operate Time, Pulse Distributor Operation

Once the instruction has been selected, it is read into the decoding register where it is temporarily stored while the decoding is accomplished.

Figure 4–10 shows the process; the contents of memory is to be four instructions at addresses 0000 through 0003 and two data words in locations 0004 and 0005. At the beginning of the program, the program counter is set to zero. At the beginning of the first program time, the contents of the program counter are gated into the memory-slection circuits. The memory-selection circuits, therefore, are conditioned to select address 0000, the contents of the program counter, and also the location of the first instruction. A memory read operation is started, and the instructions are read into the memory buffer register. From here, the instruction word is gated into the instruction register by gates which are conditioned during program time only.

At the end of program time, the program counter is stepped by 1 so that it contains the address of the next instruction to be executed (0001 in this case). When the next program time occurs, the contents of the program counter can again be used to select the proper instruction address. It can be seen, then, that the instruction sequencing is an automatic function of the control element. In Part 5, 3.3, a method of altering this sequence by programming is illustrated.

In figure 4–10 the memory location immediately following the last instruction (*Halt*) is a data word. Since data and instruction words are both in the same form (binary numbers), it is conceivable that a data word could be mistaken for an instruction word. If the program should continue beyond program step 4 (instruction in location 0003), this mistake would occur. To prevent it, the *Halt* instruction is put between the block of instructions and the data. This instruction stops the computer before it can mistake data for instructions. The only means of the computer's distinguishing between data and instruction words is by its determining where the word is stored and separating the two storage areas.

3.2.2.2 Instruction Decoding

As soon as the instruction is transferred from memory to the operation and address registers, decoding of the instructions can start. The decoding process depends upon the fact that all instructions words have fundamentally the same format.

An instruction word, in general, specifies an operation and the address of an operand. Therefore, part of the word is decoded to carry out the operation and another part is decoded to select the proper word from storage for transfer to the arthmetic element. This 2-part decoding implies that the two parts of the word must be decoded by separate matrices. In order to facilitate separate handling, an instruction word is split into two parts when it is transferred from storage to

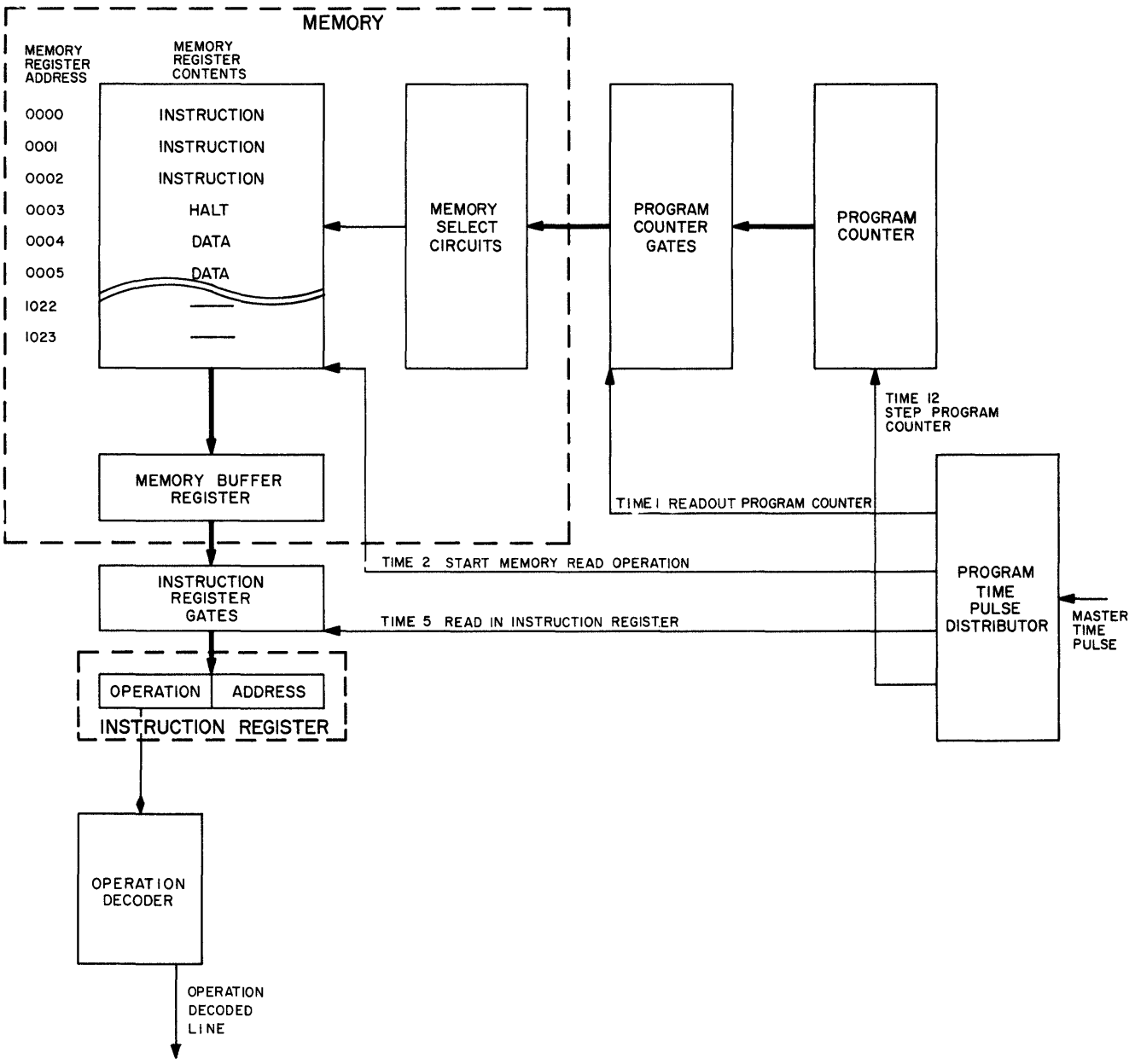


Figure 4–10. Instruction Selection, Readout, and Decoding (Program Time)

the control element. One part is entered into an operation register and the other into an address register.

The general method for decoding the instruction and selecting the operand is shown in figure 4–10. The binary number which forms the operation part of the instruction is decoded in a decoding matrix (operation decoder). This decoder conditions a separate output line (or set of lines) for each instruction. This line (or lines) condition(s) circuits in the rest of the control element (such as the transfer address gate in the figure) so that they will be ready to perform the operation specified when the operand becomes available or when the proper time comes.

Once the operation part of the instruction has been decoded, the address part may be transferred to the memory selection circuits. This is accomplished later, during operate time.

In summary, then, the control element accomplishes two things during program time: It selects the instruction from memory; it decodes the operation part of the instruction to prepare for execution of the instruction.

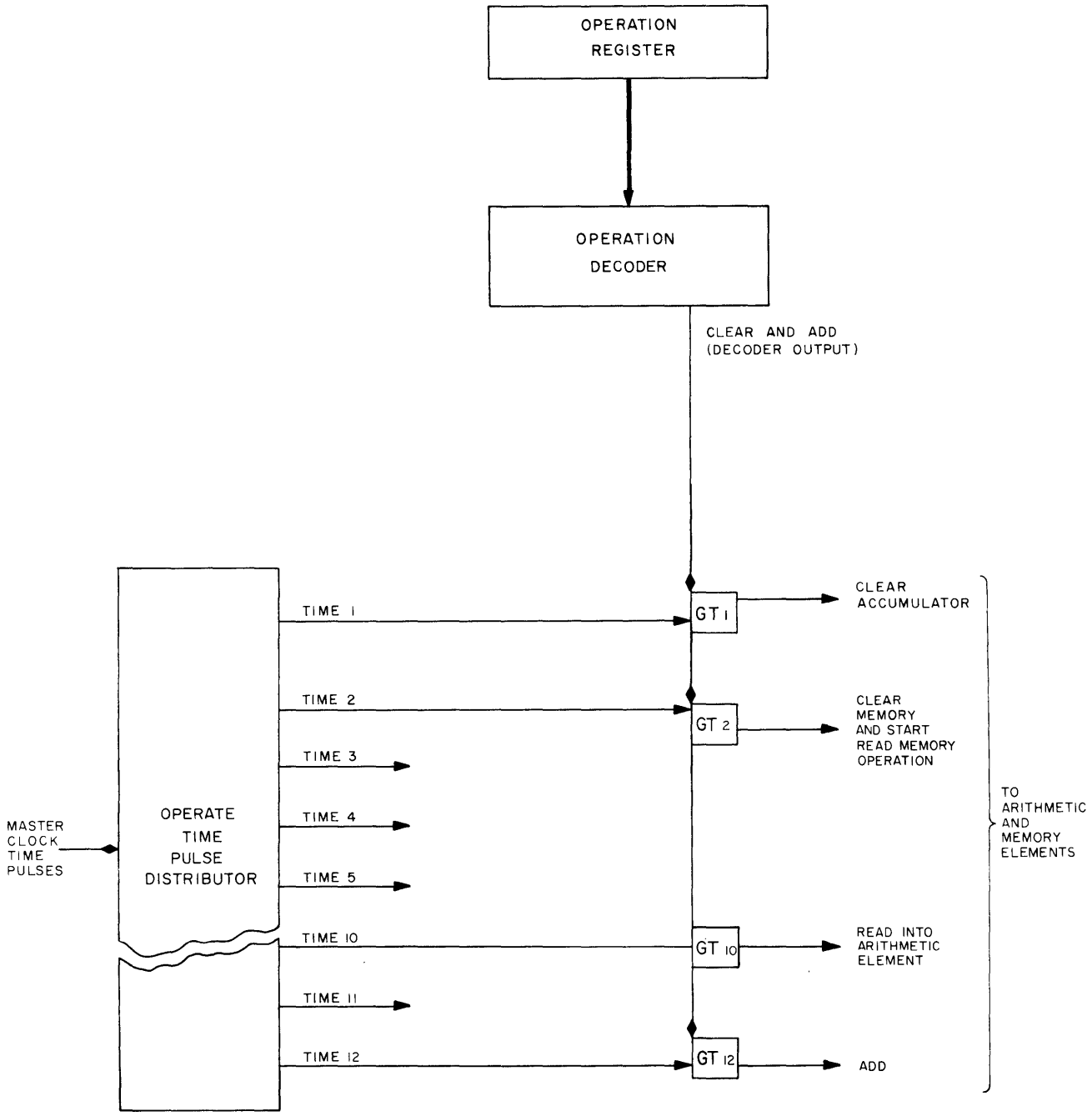


Figure 4—11. Instruction Decoding (Operate Time)

3.2.3 Operate Time

As soon as the control element has obtained and decoded the instruction, the computer starts an operate cycle. During this cycle, the operand is transferred to or from memory, and the pulses necessary to accomplish the operation specified are generated. In other words, the control circuits which were set up or conditioned during program time will be used during operate time to control the actual execution of the instruction.

Figure 4—11 illustrates how this is done for a *Clear and Add* instruction. It shows the operate-time pulse distributor feeding into gates which have been previously conditioned by the instruction decoder outputs. Since the operate-time pulse distributor produces an output during the operate cycle, the control pulses are transmitted to the arithmetic and memory elements as shown, only during operate time.

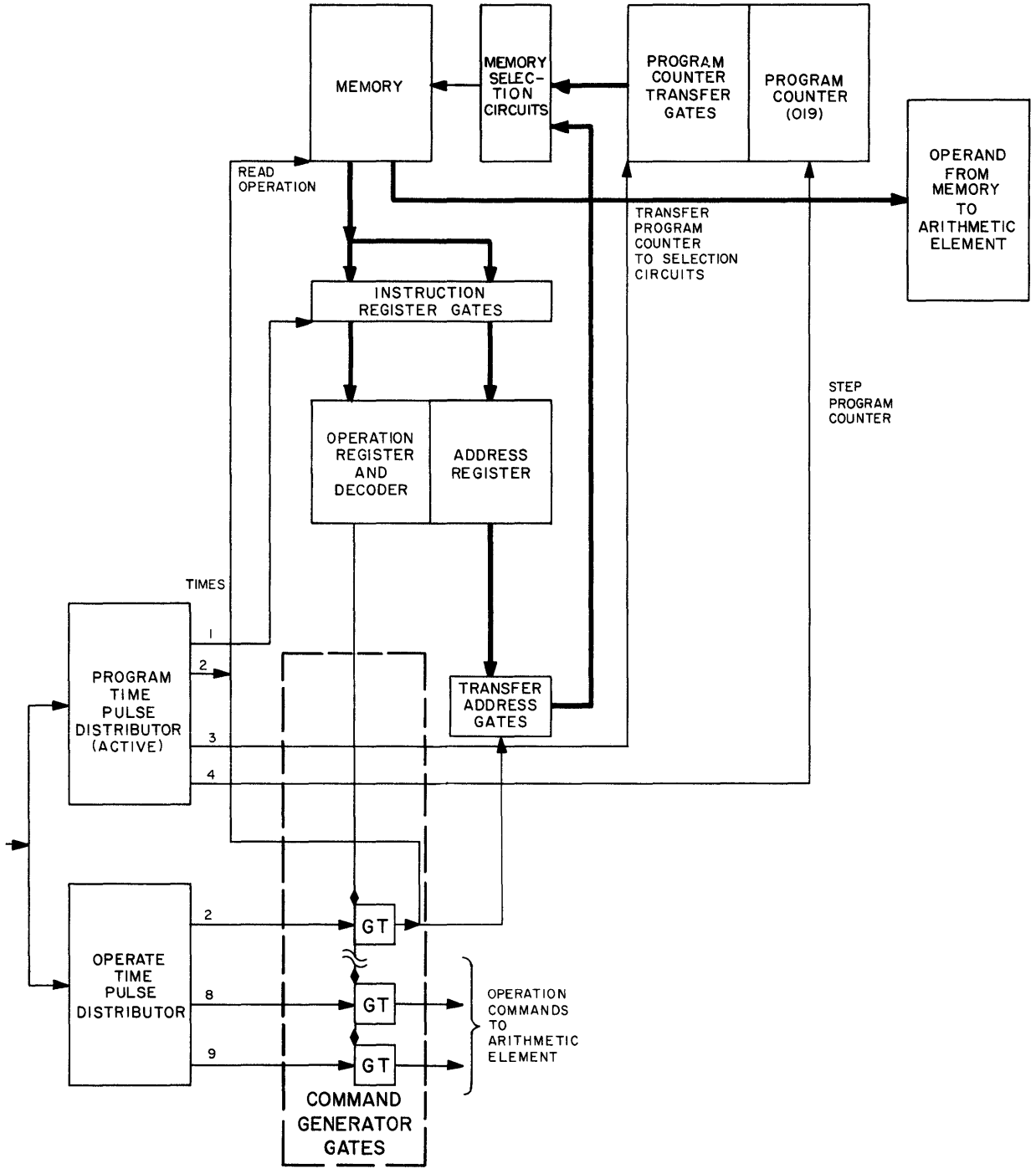


Figure 4–12. Control Operations for ADD Instruction

3.3 CONTROL ELEMENT OPERATION

As an example of the operation of the control element when executing an instruction, assume that the instruction *ADD 36* occurs in a program. This instruction means that the number in memory location 36 should be added to the number which is already in the accumulator as a result of previous arithmetic operations. Assume also that this instruction occurs on step 20 (instruction at address 19) of the program. At the time considered, step 19 has just been finished. The following listing is a summary of what happens in the execution of the instruction. Reference should be made to figure 4–12.

a. At the completion of the previous instruction, the computer automatically went into program

time. Therefore, the program-time pulse distributor is active.

b. Since 19 instructions have been executed, the contents of the program counter is 019, which is the address of the 20th instruction. The contents of the counter have been transferred to the memory selection circuits.

c. A memory operation is commanded, and the instruction register gates are pulsed to transfer the instruction word from memory to the operation and address registers.

d. The instruction word is then decoded. The operation part conditions the command generator gates.

e. Program counter is stepped by 1 so that it contains the address of the next instruction to be executed.

f. Computer goes into operate-time so that operate-time pulse distributor will be active. The address part is transferred to the memory selection circuits to select the proper operand.

g. A memory read operation causes the operand to be read into the arithmetic element.

h. Command lines, to cause the arithmetic element to add, are pulsed in their turn.

i. The computer goes back into program time in order to start execution of next instruction in program.

3.4 VARIATION OF PROGRAM BY CONTROL ELEMENT

3.4.1 Changing Program Sequence

3.4.1.1 General

The control element described so far is capable of executing a program which has only one possible sequence once it is entered into the computer; that is, once a program is stored in memory, there is no way to change it except by replacing it with another or part of another program. Since this is true, it would be difficult for the computer to make logical decisions; for instance, "If the number in the accumulator is negative add a constant to it, but if the number is zero or positive subtract the constant from it."

The ability to make logical decisions is very important in a mathematical process. This ability must, therefore, be included in any computer which is to solve problems other than the most simple kind.

What is required in the execution of the logical issue posed above? The results of a previous calculation must be studied. (Is the answer positive or negative?) If one result was obtained one process will be executed; but, if the other result was obtained, a different process will be executed. In other words, the sequence of operations depends upon the results of a previous operation. This, in general, is the way that a computer accomplishes a logical choice. When the program requires a logical choice to be made, a special instruction is inserted which allows the program sequence to be changed according to the results of a previous calculation.

3.4.1.2 Conditional Branch

The special instruction used is called a *Conditional Branch*. It specifies from what location in memory the next instruction in the program will be taken if the condition is met. The operation part of the *Branch* instruction gives the conditions of the branch of control; e.g., it could specify that a branch occur if the accumulator is negative. The address part of the instruction specifies the location of the next instruction to be executed. If the conditions of the branch are not met, no change of program sequence will be made; consequently, the next instruction executed will be the one which follows the *Branch* instruction in the normal sequence.

A brief illustration may help to make this clear. Assume that the fifth instruction (in location 004) in a program is, "branch, if the accumulator is minus, to location 020." At the time of the instruction, if the accumulator is minus, the next instruction executed will be that in memory location 20. If, on the other hand, the accumulator sign is positive, the branch operation is ignored and the next instruction is obtained in the usual way from location 005.

The execution of the *Branch on Accumulator Minus* instruction is done by examining (electrically sensing) the sign of the accumulator and transferring the contents of the address register into the program counter if the sign is minus.

Figure 4–13 illustrates the execution of the *Branch on Full Minus* instruction. During program time, the instruction is transferred to the operation and address register just as in any other instruction. The program counter is stepped, and the command circuits are set up. During operate time, the sign bit of the accumulator is examined. If this sign is negative, the contents of the address register are transferred to the program counter. Otherwise, nothing happens to the program counter; in which case, the next instruction is selected as it would be during normal sequential operation. (Remember that the program counter has already been stepped.)

3.4.1.3 Unconditional Branch

Another feature which is included to make the computer more flexible is the *Unconditional Branch*. This instruction directs the control element to take its next instruction from the location specified in the address part of the branch instruction.

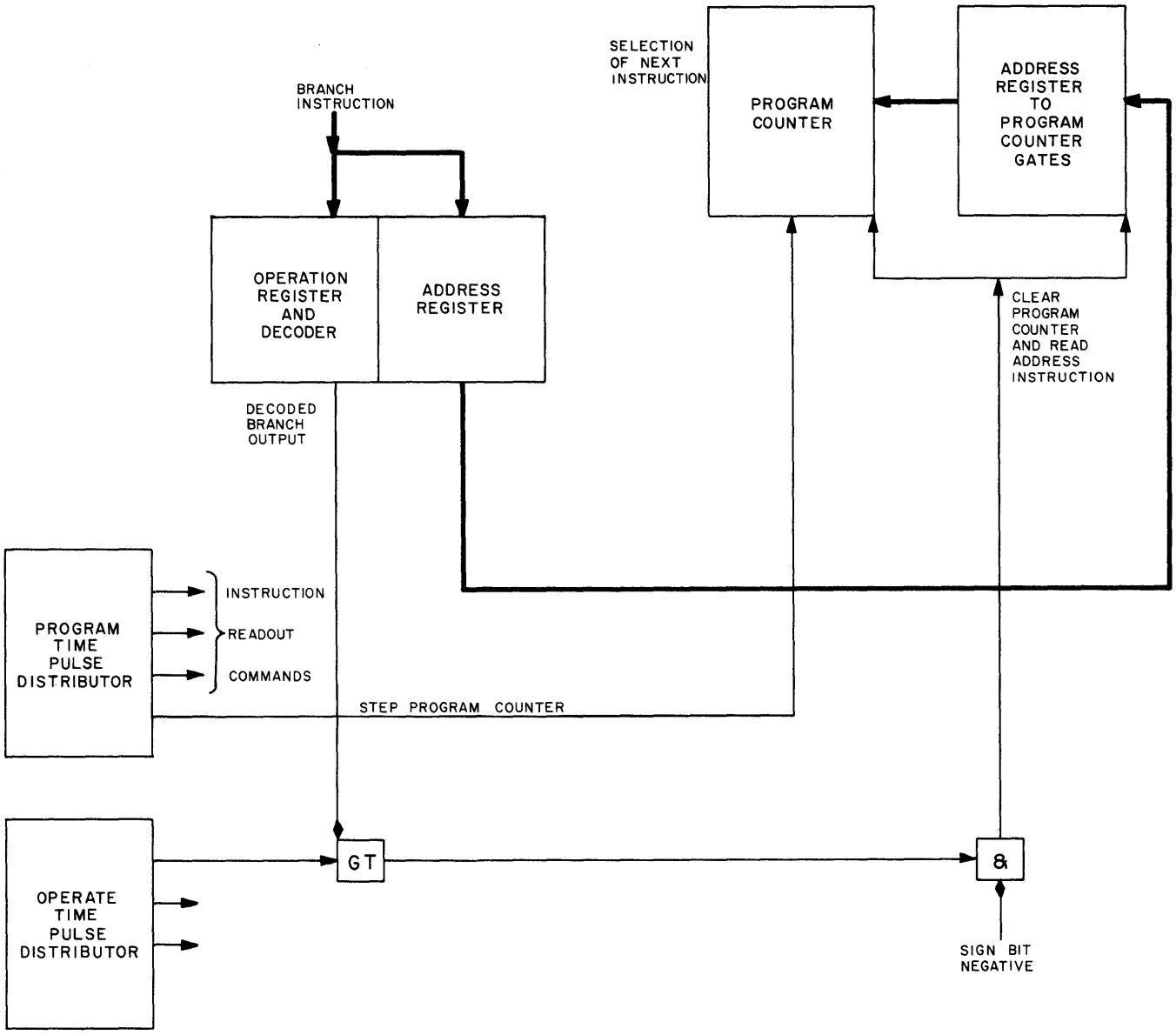


Figure 4–13. Conditional Branch Instruction Execution (Branch on Full Minus)

The *Unconditional Branch* instruction is accomplished by transferring the contents of the address register into the program counter. The only difference between the execution of this and of the *Conditional Branch* is that there is no sign bit-sensing required.

3.4.2 Alteration of Instructions

3.4.2.1 General

A very common feature of computer programs is their repetitive nature; for example a program to add 20 numbers together would contain a *Clear and Add* instruction, 19 add instructions, and a *Halt* instruction. Such a program would require about 41 memory locations to hold the 21 instructions and 20 operands. But, in this program, the 19 *Add* instructions are all the same except in their address parts. This implies that a single *Add* instruction (stored in one memory location) might be used to accomplish what 19 *Add* instructions do in this program. This can be done by changing the computer program sequence so that the same program step is repeated 19 times. However, each time the step is repeated the address specified will be modified by circuits in the control element. The 19 modifications to the address of the instruction will make the repetitive operation equivalent to a program where 19 different add instructions are used. A program to add 20 numbers together using an address modification scheme would have the following sequence of events, assuming the 20 numbers are stored in memory locations 60 to 79:

1. Clear the accumulator and add in the number contained in location 60.

2. Add the number in location 61 to the contents of the accumulator.
3. Repeat step two 18 times. Just prior to each execution, add a one to the address portion of the instruction of step 2.
4. Stop computing after the 18th repetition is completed.

In general, there are two ways to accomplish the address modification required in this type of program. One method is by a special programming trick explained in Part 5, 3.4. Another way is by means of special program control circuitry called index registers.

3.4.2.2 Index Registers

In a repetitive routine an index register is used to modify the addresses in the routine, as well as to count the number of repetitions. The address is changed by adding the contents of the index register to the address of the instruction being executed just prior to selection of the operand (see fig. 4–14). The process, therefore, results in the selection of an operand whose location address is the sum of the original address specified by the instructions and the contents of the index registers.

A repetitive program is accomplished with the indexing feature as follows:

a. The index register is loaded with the number of repetitions which are required.
b. During each repetition, any instruction which requires address modification specifies so as a part of the instruction. (This is done when the program is written.) In this case, the contents of the index register are added to the contents of the address register between the time when the address specified by the instruction enters the address register and the time the (modified) address is sent to the memory selection circuits. (See fig. 4–14.)
c. Each time a new repetition is specified, the contents of the index register are automatically decreased by 1. (1 is called the index interval.)
d. A new repetition is specified by a *Branch and Index* instruction, which branches the control back to the first instruction in the repetitive routine.
e. The *Branch and Index* instruction is a conditional branch. The branch occurs only as long as the index register is positive. When the index register becomes negative the iterations are over and the non-repetitive portion of the program continues.

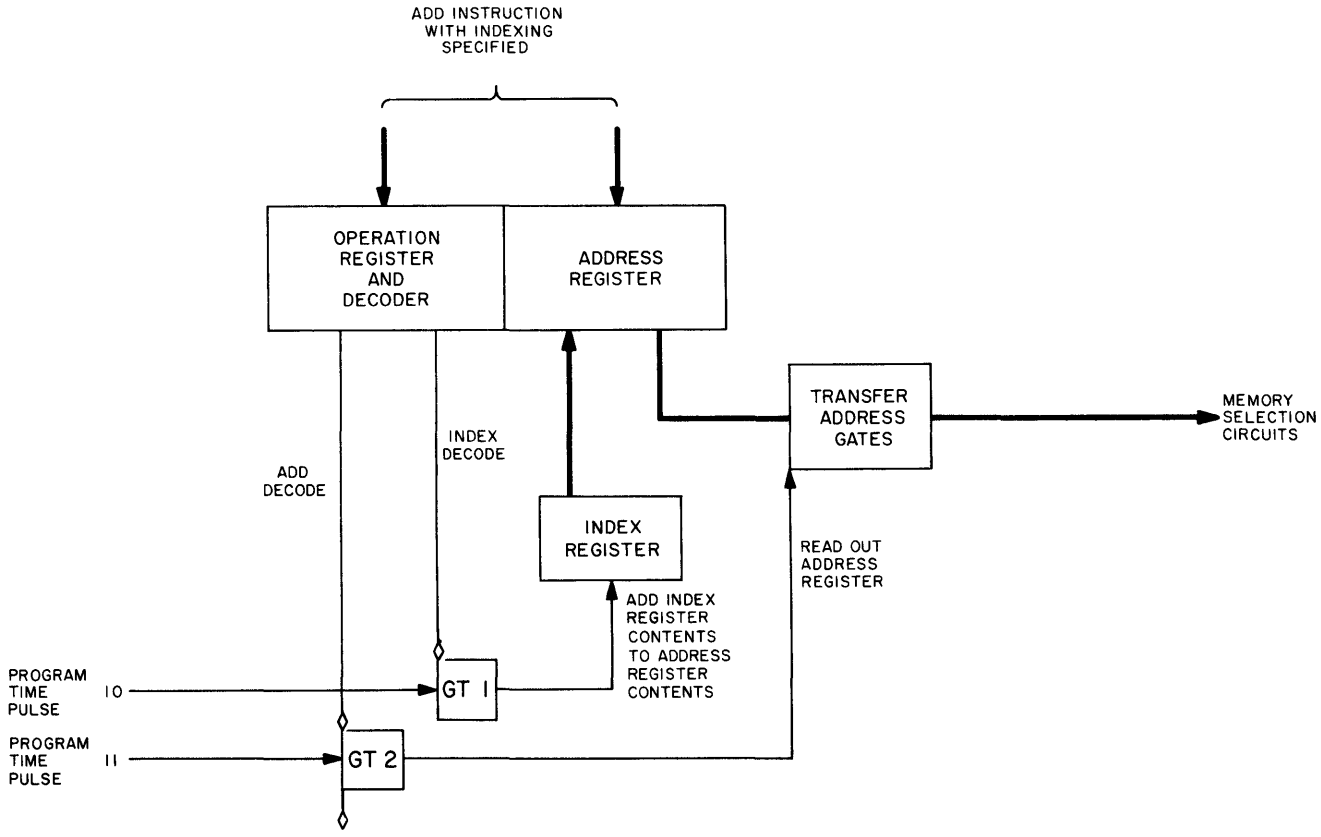


Figure 4–14. Address Modification by Index Register

To provide this repetitive program facility in the computer, the following new circuits must be added to the control element: an index register and adder, which can add the contents (positive) of the index register to the contents of the address register; controls to initiate this addition at the proper time; and circuitry for an extra bit in the instruction word to indicate that indexing is necessary. Also, there must be a way of clearing from, and reading into, the index register through programming. And finally, there must be a way of stepping the index register down one on the *Branch and Index* instruction.

Figure 4—14 shows the operation of the index register when an *ADD* instruction, with indexing specified, is given. The instruction is read out of memory in the usual way during the first part of program time. The operation part of the instruction is decoded to give an add-decode output level as in a normal *ADD* instruction. The decoder also puts out an index-decode level which conditions the indexing command generator (GT 1). Just before the transfer of the address to the memory selection circuits, the contents of the index register are added to those of the address register.

CHAPTER 4
ARITHMETIC ELEMENT

4.1 GENERAL

Data processing is accomplished in a computer by a controlled transfer of information between storage devices. The transfer is made through a data-manipulating or -switching device. It is this device which does the actual changing of data which is necessary in the process. The switching device changes the data by performing simple arithmetic operations upon it. For this reason the device is called the arithmetic element.

4.2 ARITHMETIC ELEMENT PURPOSE

The arithmetic element of a computer has two main functions: It operates on data, and it operates on program instructions. The data operated upon comprises the actual numbers of the problem to be solved. The arithmetic element does the addition, subtraction, etc., for solution of the problem. As stated in 3.4, it is often necessary to change the instructions of a program as the solution is carried out. Because the instructions is by programming. In this case, the arithmetic element can be used to change them while the program is in progress.

Operations on the program are accomplished in two general ways. Sometimes the arithmetic element is used as a part of the control element circuitry; for instance, the accumulator of the arithmetic element might be used as an index register. A more obvious way of using the arithmetic element to change instructions is by programming. In this case, the arithmetic element is used merely as an arithmetic device. The instructions to be changed are taken as operands in an instruction-changing subprogram. Such a program changes an instruction by calling it out of its location in memory, adding or subtracting a previously stored constant to the instruction, and, then, restoring the (modified) instruction to its original location in memory.

4.3 REQUIREMENTS OF AN ARITHMETIC ELEMENT

It would be possible to perform arithmetic by using a purely logical network in combination with the memory of the computer. It would also be possible to perform arithmetic purely by means of counting registers and the memory of the computer. A much more efficient system for doing arithmetic, however, is to use an arithmetic element containing both logic and limited, short-term storage facilities which work together with a large computer memory and a central control element.

In Chapter 2 of Part 3 it is shown that all arithmetic operations in such an arithmetic element are a combination of a few simple operations, which the arithmetic element registers and logic must be able to do. These are:

a. Read in a number. There must be a means of reading the operand into the arithmetic element in one or more ways.
b. Read out a number. There must be a means of reading the results out of the element.
c. Clear. There must be a way of clearing the element.
d. Add. There must be some means of adding two numbers together.
e. Complement. There must be some means of subtracting a number. Usually, this implies a means of complementing a number.
f. Sense. There must be a means of sensing the contents of one or all bit positions of a register.
g. Shift. Finally, there must be a means of shifting a number either right or left with respect to the radix point.

In order to perform arithmetic by a combination of all of these functions, the arithmetic element must have not only registers and switching circuits but also a sequence-controlling device. This device must control the sequence of the various functions so that the proper mathematical operation is carried out; for instance, to clear the accumulator and add a number, a sequence such as the following is necessary:

1. Clear the accumulator.
2. Read the number into the accumulator.

This operation would be unsuccessful if the wrong basic operations were called for or if the correct operations were called out in reverse order. Therefore, an arithmetic control and timing section must be provided.

Not only must the internal functions be properly sequenced, but also the interaction of the arithmetic element with the other elements of the system must be sychronized. This synchronizing function is performed by the control element of the computer. The internal functions of the arithmetic element could be controlled by an autonomous control and timing device such as that used in the memory. However, in most computers both the internal and external operations of the arithmetic element are controlled by a single, centralized control element (Ch. 3).

4.4 OPERATION OF SAMPLE SYSTEM ARITHMETIC ELEMENT

4.4.1 Introduction

Now that the general properties of an arithmetic element have been reviewed, an element designed for operation with the direct-access memory and control elements already described will be developed.

4.4.2 Arithmetic Element Description

4.4.2.1 General

In Chapter 1 it was stated that the arithmetic element of the sample computer would be of the parallel binary type. The computer uses a word length of 15 magnitude bits plus a sign bit. This, of course, corresponds to the word length used in the other elements of the computer. However, in 3.4.2.4 examples of multiplication are worked out using a word of four magnitude bits plus a sign bit. The smaller number of bits is used to simplify the description of the process.

The arithmetic element operates on numbers by performing the four fundamental arithmetic operations: addition, subtraction, multiplication, and division. An arithmetic element capable of performing all four of the arithmetic operations while satisfying the need for high-speed operation can be built with three flip-flop registers, together with a logical network that is essentially a set of full adders. Each of these registers and adders will contain 15 bits plus a sign bit.

The general layout of the registers and adders of such a system is shown in figure 4–15. In this arithmetic element, all arithmetic is performed by an addition process. Addition itself is an add process. Subtraction is accomplished by addition of a complement number. Multiplication and division are, respectively, repetitive add and subtract processes. The basic function of each of the units in the figure can be explained in terms of these fundamental operations. The A register, accumulator, adders, and carry-accumulator gates perform the addition and subtraction.

When numbers are contained in the A register and the accumulator, addition is initiated by conditioning the adder gates. The B register is an auxiliary register which is used in combination with the accumulator in the multiply and divide processes. The other gates shown are included to allow information transfers from or to the memory buffer. The read-in gates determine where a word from the memory buffer enters the arithmetic element. The readout gates determine when the accumulator contents are read to the memory buffer.

Figure 4–15 indicates that it is possible to read from the memory buffer directly into the A register or into the accumulator. It is also possible to read information into the accumulator from memory via the A register and adders. To insert information in the B register, it must be first inserted in the accumulator and then shifted into the B register. Although this is relatively slow, it saves equipment and decreases the number of instructions the computer must execute. The only way to get information from the arithmetic element to the memory buffer (and then to memory) is to read it out of the accumulator via the accumulator readout gates.

4.4.2.2 Addition

Addition can be performed by an element which consists only of two flip-flop registers and a set of full adders. The routine in the element of figure 4–15 is as follows:

a. The augend is entered into the accumulator and the addend is read out of memory and into the A-register.
b. Outputs of the two flip-flop registers are fed to the full adders.
c. The sum developed by the adders is gated to the accumulator register where it replaces the augend. The carry-accumulator gates propagate the carry and gate the sum into the accumulator register.

In a single-address machine, an instruction to add means add a number to the number which is already in the accumulator. If it is assumed that the augend is already in the accumulator, the following is a summary of the controls to cause addition in the arithmetic element (fig. 4–15) when used in conjunction with the core memory and control elements discussed in the previous chapters.

1. Read from memory location specified in the instruction. (This gets addend into memory buffer.)
2. Read into the A register. (This transfers information from memory buffer to A register.)
3. Pulse the carry-accumulator gates. (This causes the sum to enter the accumulator register.)

From this summary it can be seen that the only controls needed will be one to start the readout of the specified memory location, one to condition the A register read-in gates, and one to condition the carry-accumulator gates. Circuits of the control element explained in Chapter 3 will be used to accomplish the required functions.

The operation of the control element during operate time (execution time) to accomplish this addition process is shown in figure 4–16. It should be recalled that during program time (not shown) the operation and address parts of the instruction word have been decoded. The operation decoder output in this case is a single level (add decoded) which remains conditioned until the end of operate time. This add-decoded level conditions certain gates of the control element. At the proper time, then, the control element gates (command generators) pass pulses, which control the action of the

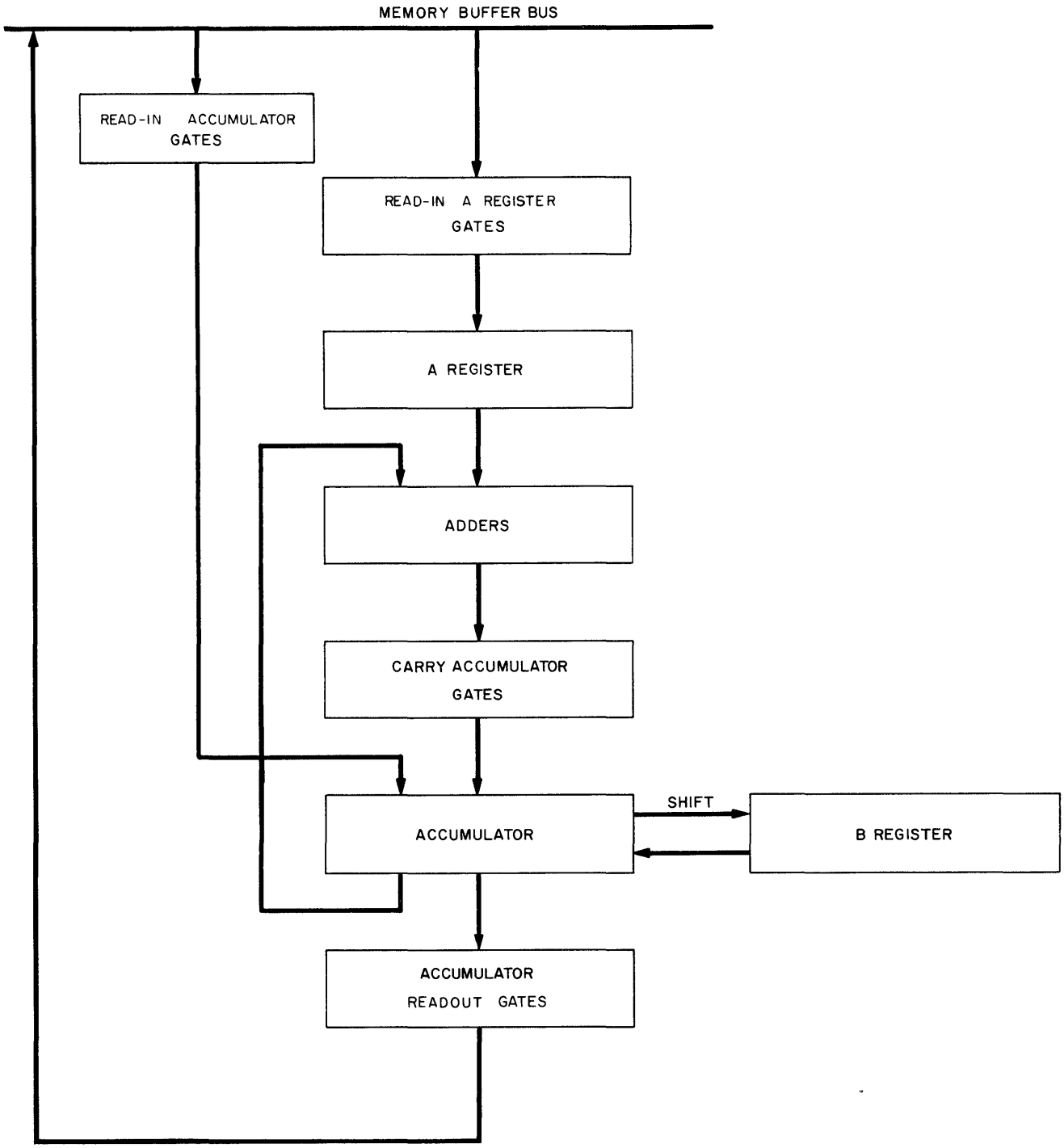


Figure 4–15. Arithmetic Element Information Flow

arithmetic registers and gates, to perform the required addition.

The sequences of the commands to the arithmetic element are determined by the operate time pulse distributor. The control element, therefore, controls the sequence of the fundamental operations of which the arithmetic is capable. It also controls the start and stop of the arithmetic element, along with the start of the memory element. Notice the time lag between the start of the memory read cycle (when the operand is to be read from memory) and the actual start of the addition process. This time lag is necessary because it takes the memory several time pulse intervals to transfer a word from a memory location to the memory buffer.

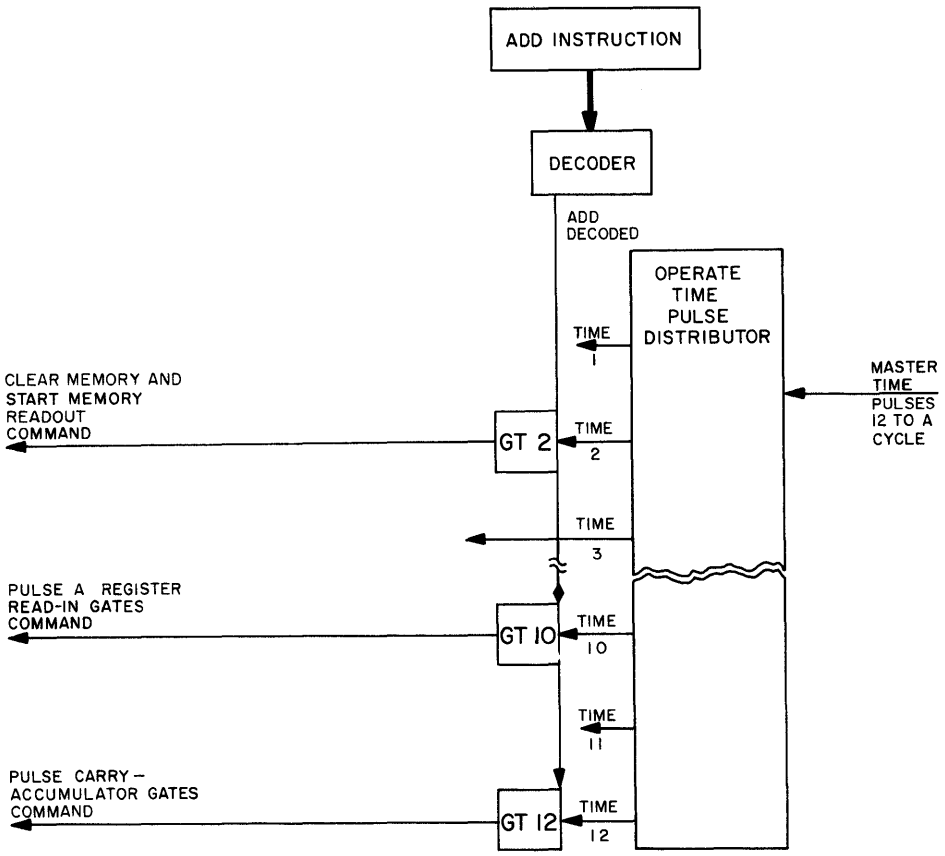


Figure 4–16. Add Instruction Arithmetic Control (Operate Time)

4.4.2.3 Subtraction

Subtraction can be performed using the same components and by almost the same routine as addition. The only additional requirement is that the A register be able to form the complement of the number it holds. The 1's complements can be obtained readily in a flip-flop register by pulsing all the complement inputs simultaneously. In this element, the subtraction routine consists of entering the minuend in the accumulator, entering the subtrahend in the A register, complementing the subtrahend, and performing the addition operation. That this is equivalent to subtraction by the usual pencil and paper method is shown in 4.2 of Part 2.

In a single-address machine, the sequence of operations in the subtraction routine is based upon the assumption that the minuend is already in the accumulator when the instruction is started. The subtrahend must be obtained from memory just before the start of the actual subtraction process. Actually, as explained in Chapter 3, it is obtained as a part of the subtract instruction just as it was in the add instruction. Knowing the whereabouts of the subtrahend and minuend, the following summary of operation can be written for the subtraction process:

1. Start memory readout operation.
2. Read into the A register.
3. Complement A register.
4. Pulse carry-accumulator gates.

As in the add instruction execution, the process, thus, can be broken down into a series of simple sequential processes. The control element described in Chapter 3 can be used to control the action of the arithmetic element.

Figure 4–17 shows the operation of the control element during operate time of a subtract instruction. Notice that the only difference between this and the add instruction decoder previously described is a new output level from the decoder and a new A register complement pulse gate. Since the figure shows only operate time of the instruction cycle, it can be assumed that the instruction has already been decoded. This time the decoder has two output lines. One of these lines is used to condition the same add command generators as were conditioned for the add instruction. The other line conditions a command generator which complements the A register just prior to the start of the actual addition. Thus, the add operation is converted to a subtract operation by a difference in decoder outputs.

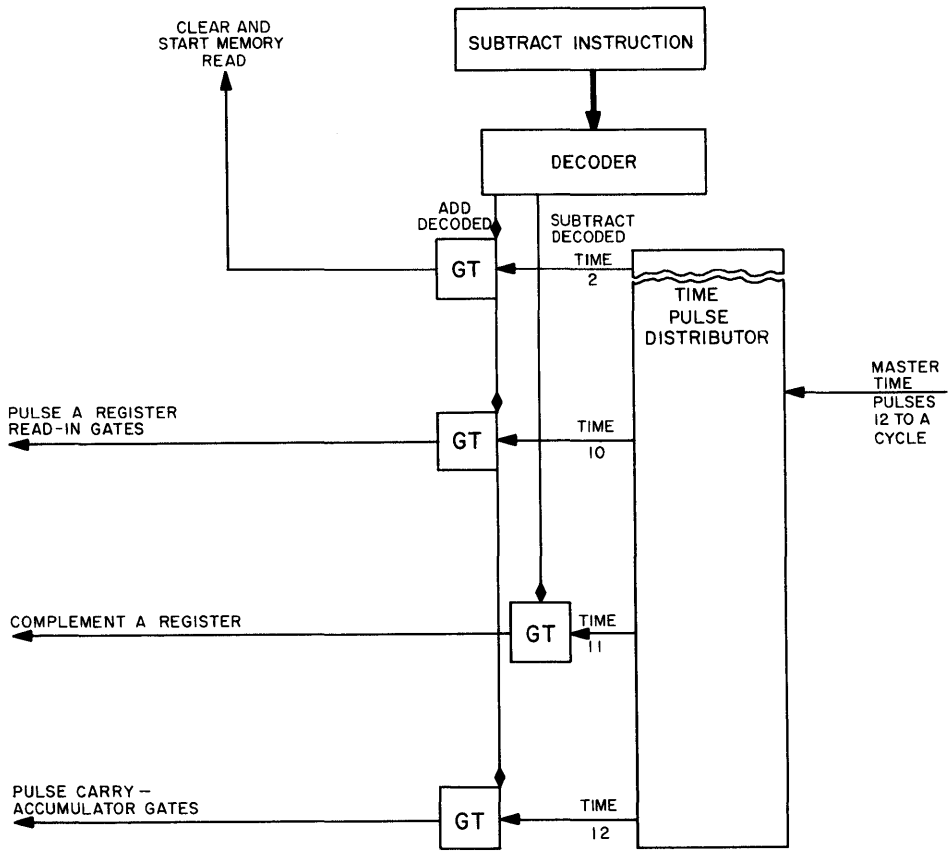


Figure 4–17. Subtract Instruction Arithmetic Control (Operate Time)

As in the addition process, the operation of the memory is synchronized with the arithmetic element by proper timing of the start of the operation of the two elements. After the memory readout operation is started, sufficient time is allowed for the word to get into the memory buffer. Once in the buffer, the word is available to the arithmetic element, and the arithmetic process is started.

4.4.2.4 Multiplication

High-speed multiplication can be performed by an add and shift routine, which employs a third flip-flop register (called the B register in addition to the accumulator, the A register, and the adders. The multiplication process is more complicated than are the add and subtract processes. Nevertheless, the example given below as a brief review of the process (which is thoroughly explained in 4.3.2 of Part 2 and 3.4 of Part 3) shows that the operation can be reduced to a series of simply controlled basic operations, just as the add and subtract processes were.

The example is illustrated in figure 4–18. The numbers held in the three registers after each step of a multiplication routine are shown. The multiplicand is assumed to be .1010 (decimal .625) and the multiplier is assumed to be .1101 (decimal .8125).

The multiplication is always performed on positive numbers. This means that, if any negative number is to be multiplied, it must first be made positive. Furthermore, the sign of the product must be determined before multiplication is started.

In the illustration, as in the arithmetic element, a sign bit is included as a bit position of each register. The accumulator and A register sign bits are used when predetermining the sign of the product. In the actual multiplication process the accumulator sign bit position is used not as a sign indicator but as an extra place to temporarily hold carries. (See step 5.) During multiplication the sign bits of the A and B registers are both unused. Notice therefore, that the shift right from the accumulator to the B register shifts the right most bit of the accumulator into the left most bit excluding the sign bit of the B register.

The details of the multiplication process are as follows:

a. The multiplicand (.1010) is placed in the A register and the multiplier (.1101) in the accumulator. If necessary, both numbers are made posi-

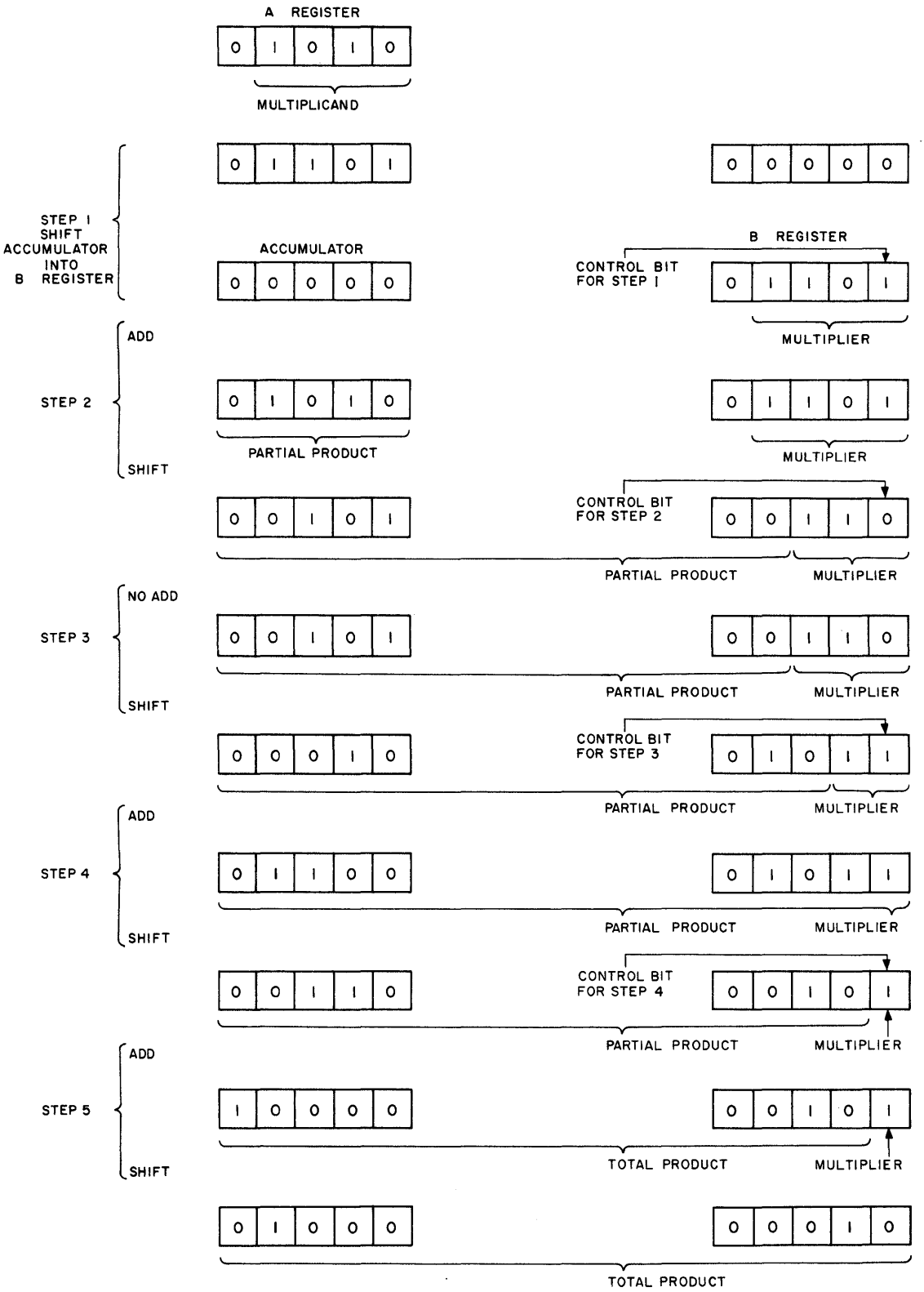


Figure 4–18. Contents of A-Register, B-Register, and Accumulator During Multiplication

tive and the predicted sign of the product is stored (in a special sign storage flip-flop). The first step of multiplication is to shift the multiplicand out of the accumulator into the B register.

b. The least significant bit of the multiplier is examined. If this bit is 1, an operation is initiated, placing the multiplicand in the accumulator (without clearing it from the A register). If the least significant bit is 0, no addition is initiated and the accumulator remains cleared. In either case, multiplication by the least significant bit of the multiplier has been carried out, and the accumulator contains the first partial product.

c. The contents of the accumulator-B-register, treated as a single register, is now shifted one place to the right. This moves the least significant bit of the first partial product from the accumulator to first data bit of the B register. At the same time, the least significant bit of the multiplier is lost. However, this is of no consequence because the bit has already completed its part in the multiplication routine. (In example of fig. 4–18, the product bit which is moved from the accumulator to the B register is 0, and the multiplier bit which is dropped is 1.)

d. The second least significant bit of the multiplier is now examined. If this bit is 1, the multiplicand is added to the shifted first partial product in the accumulator. If the bit is 0 (as in example of fig. 4–18), no addition is initiated. In either case, the number occupying the accumulator and the first bit position of the B register is now the sum of the first and second partial products.

e. The contents of the accumulator-B-register, treated as a single register, are now shifted right a second time, preparatory to the addition of the third partial product. After this shift, two bits of the sum of the partial products occupy flip-flops of the B register, and the second least significant bit of the multiplier has been dropped.

f. The routine continues in the same way, providing a step for each multiplier bit. At each step, the multiplier bit is examined. If it is 1, an add command is generated; if the multiplier bit is 0, no add command is generated. For each step, there is a shift right of the partial product in the accumulator-B-register combination until, after the final step, the product has completely replaced the multiplier and occupies the entire combined register.

g. The final step is to correct the sign of the accumulator, if necessary. The correction has been determined by the sign storage flip-flop which counted the number of complement operations that were necessary to make both the A and B registers positive at the beginning of the routine. If the number of complement operations was even, the result is a product of two negative numbers and is, therefore, positive. In this case no sign correction will be necessary. If the number of complement operations was odd, the result is a product of a negative and a positive number, and, therefore, the sign of the product should be negative. In this case, a complement of the product would be necessary. In the example no complement operation is necessary.

A summary of the routine is given below. This summary is made under the assumption that a previous set of instructions has put the multiplier into the accumulator.

1. Start memory read operation.
2. Read into the A register.
3. Sense sign of A register and make positive.
4. Sense sign of accumulator and make positive.
5. Shift accumulator-B-register 5 places right.
6. Sense least significant bit of B register.
   a. If this bit is a 1, pulse the carry-accumulator gate. (Add.)
   b. If this bit is a zero, do not pulse the carry-accumulator gates. (Do not add.)
7. Shift contents of accumulator- and B-register one place to the right.
8. Repeat 6 and 7 until as many add-and-shift steps have been accomplished as there are bits in the B register (multiplier). In a 15-bit-word machine, a total of 15 add-and-shift steps will be required.
9. Correct sign of accumulator (product).
10. Use a round-off instruction to correct the product.

It can be seen that the process can be reduced to a set routine. This same routine gives correct results in the multiplication of any two numbers up to the capacity of the machine. Without going into the details, the reader should be able to see that the control element described previously could be made to perform the sequencing of the operations. The general operation of the element is similar to that for addition and subtraction. The first part of the instruction is devoted to obtaining the multiplicand from memory. The second phase of the execution process is devoted to the actual multiplication process described. The only new kind of equipment required in the control element is a counter to keep track of the number of repetitions of the partial

product steps. This counter is a common feature of computers called a "step-counter."

Of course, the multiplication routine is more complicated than either the addition or subtraction routines. Therefore, it takes a longer time to perform. This time is made available; the multiply instruction automatically stops the progression of the program until completion of the instruction. This is called a "pause".

4.4.2.5 Division

A brief review of a division process (explained in 4.4.3.3 of Part 2 and 3.5 of Part 3) will show that it is easily mechanized with the arithmetic element registers shown in figure 4–15. The usual method of dividing is by a repetitive subtraction process. The number of times that the divisor can be subtracted from the dividend is counted to determine the quotient. What is left of the dividend after all the subtractions have been completed is the remainder.

No new components are required to perform division by a subtract and shift routine. The divisor is entered in the A register. As in multiplication, the accumulator and B register are used to form a single register of double capacity. At the outset of the routine, as the result of previous program instructions, the dividend is placed in this combination register. As the routine progresses, the dividend is shifted left, making room for quotient bits in the B register and dropping remainder (dividend) bits as one step follows another. At the end of the routine, the quotient is in the B register and the remainder occupies the accumulator.

Figure 4–19 illustrates the general process of division. The process shown is a non-restoring subtract and shift routine. The divisor is plus .1110 (decimal .875) and the dividend is plus .1000 1100 (decimal .546875). The quotient is, therefore, four bits excluding the sign bit.

Division is only performed upon positive numbers. As in multiplication, therefore, the signs of the dividend and divisor must be corrected if either is negative when the process starts.

In division the sign bit position of both the A register and the accumulator are used as sign indications throughout the process. The sign bit of the B register is used as a sign indicator only at the end of the process, when the signs of the remainder and quotient are finally corrected to their predetermined value. During the actual division process, the B register sign bit position is used as an extra position to store one bit of the dividend.

If it is assumed that the dividend, which can be a double length word, is contained in the accumulator-B-register as a result of previous instructions, the division routine (fig. 4–19) proceeds as follows:

a. The divisor is read out of memory into the A register.

b. The signs of the dividend and the divisor are examined. If either is negative, it is complemented to its positive form. (In the example, both numbers are positive as indicated by 0's in the sign bit positions of the A register and accumulator. Thus, no conversion is necessary.) The number of conversions necessary is counted; the result of the count, which indicates the sign to be given the quotient, is stored in the sign storage flip-flop.

c. The divisor is now complemented and added to the portion of the dividend in the accumulator; i.e., the divisor is lined up left with the dividend and is subtracted from it. The remainder obtained is negative; consequently, a "0" is inserted in the least significant position of the B register, and the combined accumulator- and B-register is shifted one place to the left.
This "0" will eventually be contained in the B register sign-bit position. However, it has no meaning in this process. The step to generate the zero is merely included to make all the subtract and shift routines as much alike as possible. The zero obtained is called a dummy sign bit in the text below. When the dividend is shifted left, by shifting the contents of the combined accumulator-B-register, the dividend sign bit is lost, and the most significant magnitude bit of the remainder occupies the sign bit position of the accumulator.

d. The divisor, in true form, is now added to the contents of the accumulator. Notice that the sign bit of the divisor (which is 0) is lined up with the most significant magnitude bit of the remainder as a result of the shift operation. If the result of the addition is a positive current remainder, half the divisor has been successfully subtracted from the full dividend; that is, in the first step, the full divisor has been subtracted from the dividend leaving a negative remainder, while, in the second step, half of the divisor has been replaced. Thus, the net amount removed is one-half the divisor. Therefore, a 1 can be placed in the quotient position that is now the least significant position of the B register. After this 1 is inserted another shift left occurs. Now, a dummy sign bit of the quotient and a one bit of the quotient have been generated by two subtract-and-shift operations. Note that a 0 was generated whenever the addition resulted in a negative remainder and that a 1 was generated whenever the addition resulted in a positive remainder. It should also be recalled that in this non-restoring division method, the true form of the division is added to the negative remainder,

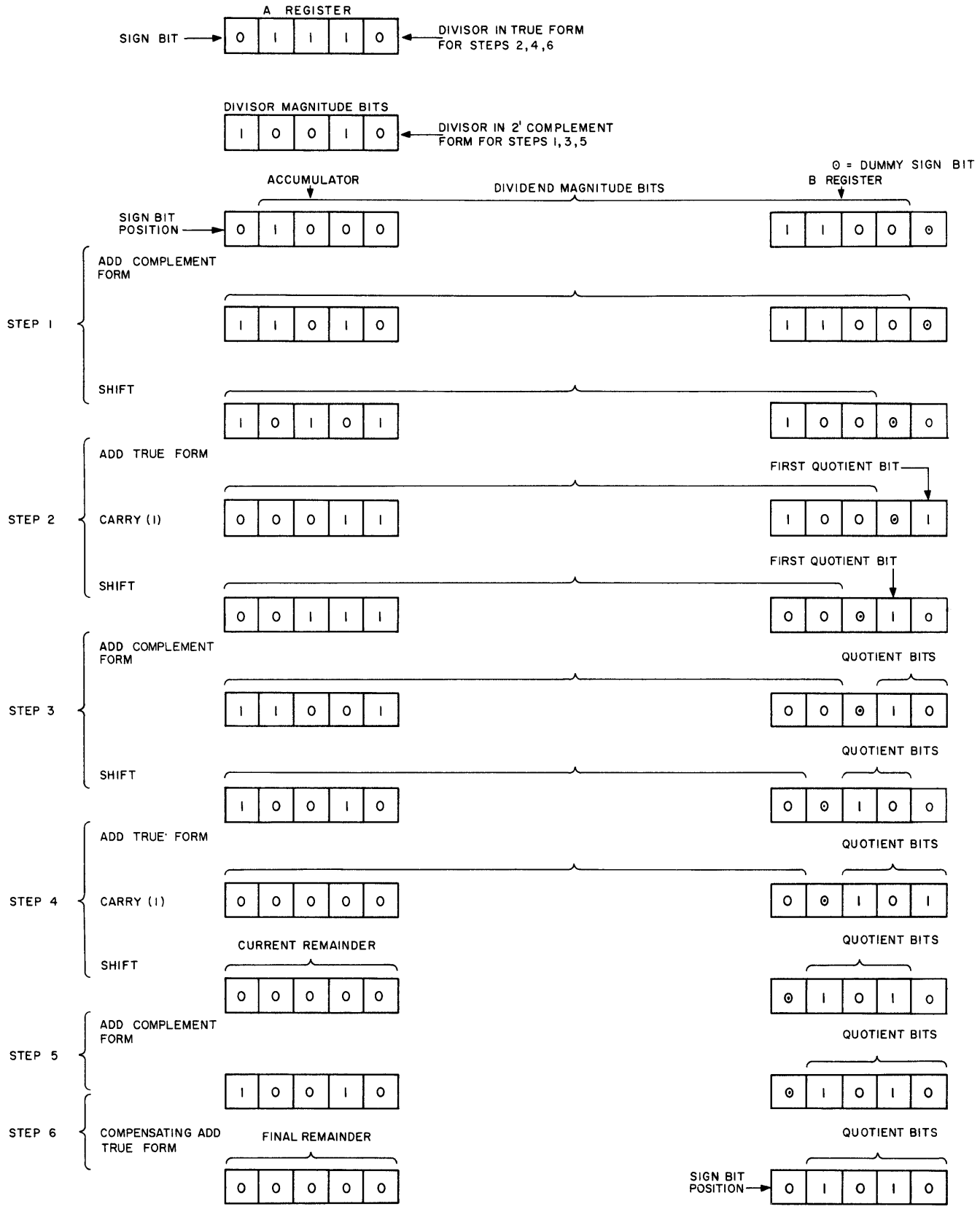


Figure 4–19. Contents of A-Register, B-Register, and Accumulator During Division

while the complement of the divisor is added when the remainder is positive.

The routine continues in this same way, providing one subtract and shift step for each quotient data bit (excluding dummy sign bit) that is generated. In this case, four subtract-and-shift steps are required. In the case of a 15-bit-word machine, 15 subtract-and-shift steps are required.

If the last quotient bit is a 0, then an extra compensating addition must be performed so that the final remainder will be positive. This is the case in the illustration. However, the final remainder turns out to be zero.

At the end of the routine, the quotient is contained in the last four bits of the B register; the final remainder is contained in the last four bits of the accumulator; and the signs of both quotient and remainder are positive. At this time, the signs of the accumulator and the B register are converted to the sign which was predicted when the signs of the dividend and divisor were made positive. (In this case, no conversion is required.)

A summary of the division routine is given below. It is assumed that the dividend is already contained in the combined accumulator-B-register when the divide instruction is given. Again, it should be noted that the routine below does not include the program-time operations of the control elment. Only the actions required in execution of the decoded instruction are listed.

1. Start memory read operation.
2. Read divisor into A register from memory.
3. Sense sign of A register complement, if negative.
4. Sense sign of accumulator, complement, if negative.
5. Sense sign of accumulator.
   a. If positive, complement A register and add to accumulator.
   b. If negative, add true A register contents to accumulator.
6. Sense sign of accumulator.
   a. If negative, insert 0 in least significant position of B register.
   b. If positive, insert 1 in least significant position of B register.
7. Shift combined accumulator-B-register one place to left.
8. Repeat steps 5, 6, and 7 until one add-and-shift routine has been completed for each bit of the quotient to be generated, excluding the dummy sign bit. In the illustration a total of four add-and-shift steps are required. If a 15-bit-plus-sign-bit word were to be generated (as in the example machine) 15 add-and-shift steps would be required.
9. Take one extra step 4 and 5.
10. Sense the sign of the accumulator.
    a. If negative, add true form of A register to accumulator (compensate add).
    b. If positive, do not compensate add.
11. Correct signs of accumulator and B register.

The example shows that the division process may also be divided into a set routine. This same routine gives correct results in the division of any two numbers if the dividend is smaller than the divisor. Without going into the details of mechanization, the reader should be able to see, therefore, that it is possible to incorporate the controls necessary to perform division in the control element described in the previous chapter. As in the multiplication routine, the control element requires a step counter to keep track of the number of repetitions made.

4.4.2.6 Shifting

Some instructions which the arithmetic element is called upon to perform do not require a reference to memory during execution time. In other words, the instruction calls for an operation to be performed upon a number which is already in the arithmetic element. An example of such an instruction is one to shift the combination accumulator-B-register contents left five places.

This instruction comes from memory to the computer control element in the usual instruction word form, with an operation part and an address part. The operation part of the instruction has the usual significance. It specifies the operation to be performed (shift accumulator-B-register left). The address part of the instruction (5) does not have the usual meaning. In this instruction the address part specifies the number of places the registers are to be shifted. Thus, the instruction itself contains all the information necessary to perform the operation; no further reference to memory is required.

The operation is carried out as follows:

a. The shift left is ordered, and the number starts to shift.
b. The number of single shifts is counted by the control element (step counter).
c. When the number of shifts is equal to the address part of the instruction, the shifting is stopped.

The control element devised to control the other arithmetic operations could be made to sequence the arithmetic element during a shift operation. All that would be necessary would be a shift-demand level and a step counter to count the number of shifts. The step counter is the same counter as that used to count the number of add-and-shift operations in the multiplication process or the subtract-and-shift operations of the divide process.

CHAPTER 5
INPUTS AND OUTPUTS

5.1 GENERAL

The computing section of the computer has been discussed in the last three chapters. This chapter discusses the way results obtained from the computing section are entered into, and extracted from, the computer; that is, the Input and Output Systems.

In general, the input-output facilities (IO) provide means of communication between the computer and its environment. This communication is characterized by translation between the symbols used in the environment (which could include both men and other machines) and those used within the computer. For instance, man uses a set of written decimal symbols to convey numerical information. In a binary computer whose input information originates from a human operator, the Input System translates from the written decimal to the binary expressed in electrical terms.

Usually there is a great difference between the symbols used by the computer and those used by its environment, and between the comparative speeds of writing and reading the symbols. If the computer demands information from an input device and then has to wait until the input device can deliver it, the computer also has to wait. Since the input device is invariably much slower than the computer, this is not a desirable condition. The Input System, therefore, includes a buffer device which greatly decreases the time required to effect the transfer of information between input and computer. The input buffer is a storage device which can be written upon by the input device at the pace of the input device and can be read from by the computer at the computer's pace. The output buffer performs the same function in reverse between the output devices and the computer. It matches the high speed of the computer to the slow speed of the output device.

The IO equipment of a computer determines its speed, capacity and versatility to a great extent. The speed and capacity of the computer is affected by the IO speed in three general ways. If individual transfers to or from the computer are slow, the computer is delayed during these transfers. If, on the other hand, individual transfers are fast but not frequent enough, the computing section may be delayed for lack of data to work on. The effect of the IO system upon computer speed is also determined by the amount of translation which is accomplished within the IO. If the IO system does not do a complete job of translation from the symbols of the outside world to those of the computer, the computing section wastes program time on this function. For instance, information is often entered into the Input System of a binary computer in decimal symbols. The Input System may convert these decimal symbols to electrical impulses in binary form, or it may convert them to electrical impulses in a decimally coded form. The first method requires more equipment in the Input System, but the second method requires that the computing section do part of the translation by programming. If much decimal to binary conversion is necessary, the computer can compute at a higher speed if all the conversion is done by the IO system. On the other hand, if very little decimal-to-binary conversion were required, the average computing speed would not be greatly affected by the extra time required to do this conversion within the computing section. In the sample computer, all necessary translation is performed in the IO equipment.

The versatility of the computer is also greatly affected by the IO system. The IO system (particularly input) more nearly defines the use to which a computer can be put than does any other element.

Computing machine uses can be divided into two general classes. A computer may be used in data processing to process a great deal of data at a time, or it can be used, in the solution of mathematical problems, to process a little data by a great many operations. In general, the data processing machine requires a high-capacity IO system; a computer used in mathematics requires only a small capacity IO system.

Computing machines are sometimes used for real-time control systems. Unless the IO system properly matches the computer to the control process, this use is not possible. In the SAGE computer the data comes in to the computer site from the various radar and other sources via telephone line. If the necessary speed of operation is to be obtained, the IO system must be able to enter this information directly from the phone lines to the computer without human intervention. Communication from operator to computer is limited to low-volume information transfer, except during program loading. In this case, a low capacity manual input may be efficiently used. The air defense control process requires that the output of the computer present a great

deal of information to the operator in a form which can be quickly assimilated. A display output makes this possible.

A computer which is to be used for business purposes requires still different types of IO devices. The input and output should be compatible with the data reduction system used in the outside world. For instance some offices use punched cards for their data processing. A computer in such an office should certainly be equipped with punched card IO equipment.

5.2 SAMPLE SYSTEM CONSIDERATIONS

The requirements for the sample system were stated in Chapter 1. It was stated that the system would need an automatic input and output to transfer binary information to and from remote machines. It was also stated that the system would require a manual input for such functions as program insertion, and a display Output System which would present the entire air situation. The comparative speeds of the IO devices and the computer also require that each device feed into, or be fed from, a buffer device. This is a temporary storage device which allows the asynchronous transfer of large amounts of information between the IO equipment and the computer. Drums which are accessible to both the computer and the IO equipment are used for this purpose.

5.3 IO BUFFER DRUM

5.3.1 Purpose

The purpose of an IO buffer drum is to match the speed of the computer to that of the IO devices. It uses an asynchronous demand system of reading and writing. This means that the device writing into it or reading from it governs the speed of writing or reading. The drums also serves as an assembly memory where information to be read or to be written can be assembled into blocks of information, thus allowing the computer to read or to write a large number of words at once whenever an input or output operation is required.

5.3.2 Drum Operation

5.3.2.1 General

All of the buffer drums are used to transfer information only one way; that is, the input drums always transmit their information from the input device to the computer. Similarly, the output drums only transfer information from the computer to the output device. Consequently, each drum must have a set of reading heads connected to, and controlled by, the reading element and a set of writing heads connected to, and controlled by, the writing element. This arrangement allows both reading and writing elements to transfer information at their own pace.

Figure 4–20 illustrates the general operation of the drum in either an output or an input system. If the system described is an output device, the writer represents the computer and the reader represents the output. If, the system described is an input device, the writer of the illustration is the input while the reader is the computer. The general operation is for the writer to write a word on the drum when an empty drum register (does not contain useful information) is found. The reader then reads all words which are in full registers (contain useful information) as soon as it can; i.e., when the word comes under the read heads and at the same time the reader has someplace to put the word. This method of controlling the drum is known as status control operation. The operation of the drum is controlled by an outside device (reader or writer) according to the full or empty status of the drum register.

5.3.2.2 Status Control Operation

When a drum is controlled by status it must have two extra status control channels (fig. 4–20). These channels have bits recorded in them to correspond with each register in the drum. These bits may be read or written by the writer- and reader-control circuits to indicate the status of each corresponding register; that is, to indicate whether the information in the register has been read before (register empty) or is yet to be read (register full). The bits in the write status control channel cause a write operation if the register under the heads is empty (has been previously read) and if a word is available. The bits in the read status control channel cause a read operation if the register under the heads is full (has not been previously read) and if a word is demanded.

The status of a register is indicated by the bits of the status control channel which controls the particular operation being determined. If a read operation is being determined, a 1 bit in the read control channel indicates that the register is full; a 0 bit here indicates that the register is empty. If a write operation is being determined, a 0 bit in the write control channel indicates that the register is empty and that it can be written upon; a 1 bit here indicates a previously filled register.

The operation of the status-control circuit, then, is as shown in figure 4–20. When the writer writes a word on the drum, it also writes a 1 bit in the read control channel. This channel is then monitored by the reader-control circuit. The 1 bit being read by these read control circuits indicates that the corresponding drum register is full and, so, may be read. If the reader needs the information, it can read this full register. As the reader reads the register it also causes a 0 to be written in the write control channel. When this register arrives back at the write heads, the 0 indicates to the write-control circuits that the register is again empty.

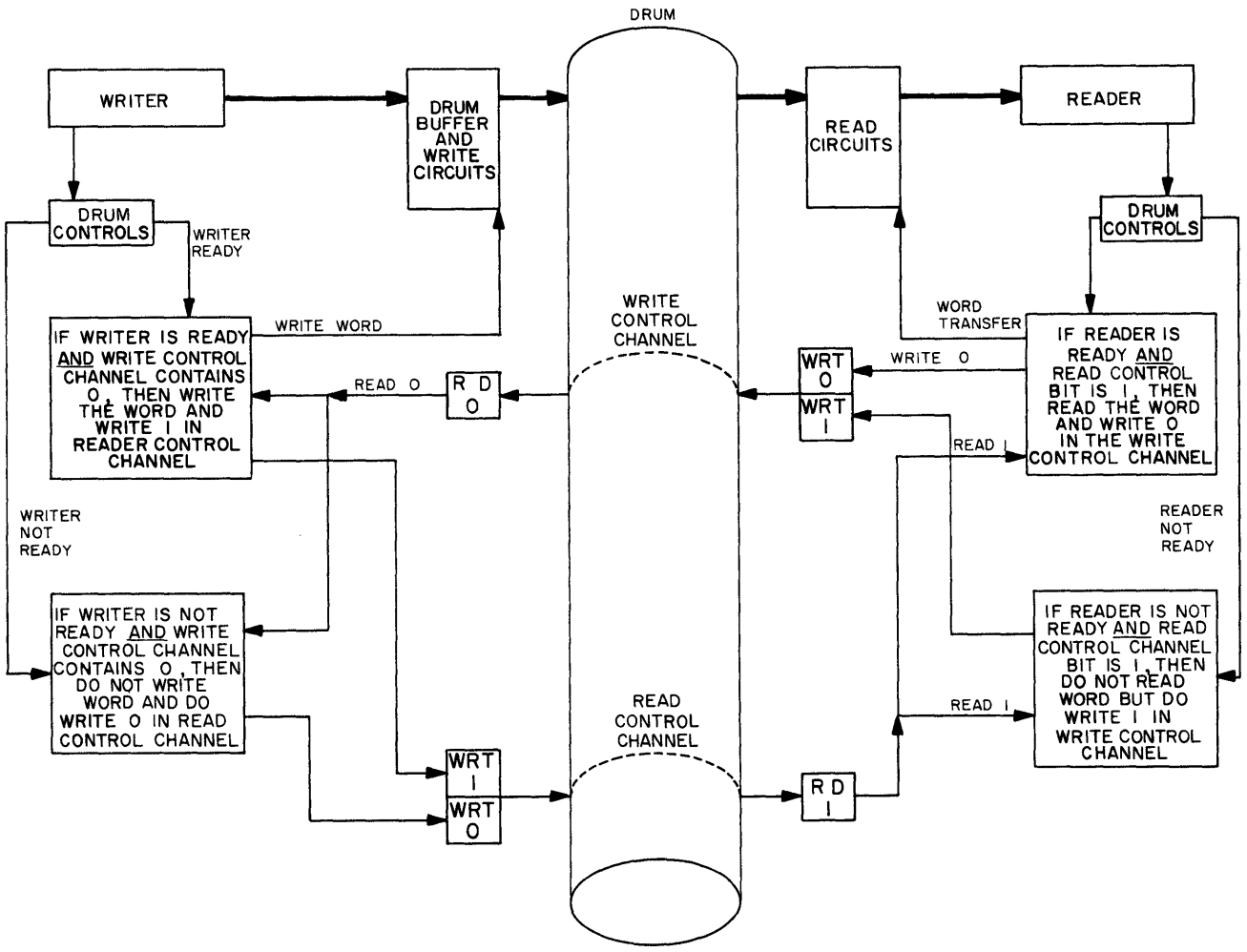


Figure 4–20. Status Control of Drum

Now consider the alternate operations from the writer's point of view. If the writer reads a 0 in the write control channel but for some reason cannot write a word (no word to write), it will write a 0 in the read control channel. When this register gets to the read heads, the 0 indicates that the drum status is empty and, therefore, the register will not be read. If the write control channel contains a 1 bit, a 1 bit automatically is rewritten into the read control channel.

If the alternate operations are considered from the reader's point of view, the operation is very similar to the operation above. If the reader reads a 1 in the read control channel but is unable to read the word at that time, the reader-status-control circuits write a 1 in the write control channel. This 1 protects the information in the register when the register gets to the write heads. If the reader reads a 0 in the read control channel, a 0 is written into the write control channel.

5.3.2.3 Program Operation

The sample computer reads an input word as the result of a single instruction. The operation part of the instruction specifies the read operation and which input to select. The address part of the instruction specifies where in memory the word is to be stored. The drum-control circuits themselves specify which words are to be read from the drum. They specify that the first word which comes along after an instruction and which has not been read before is to be read.

This system requires a special type of instruction which is typical of all drum operations in the sample computer. When the instruction is first given, it may be some time before the first full register comes along. During execution of the usual (not drum) instruction, the program control comes to the instruction and attempts to execute it whether the operand is available or not. It then goes right on to the next instruction in the program. Some provision must be made, therefore, to stop the progression of the program until the drum operation has been completed. Such a provision in this computer is included in all drum-reading and -writing instructions. These IO instructions cause the

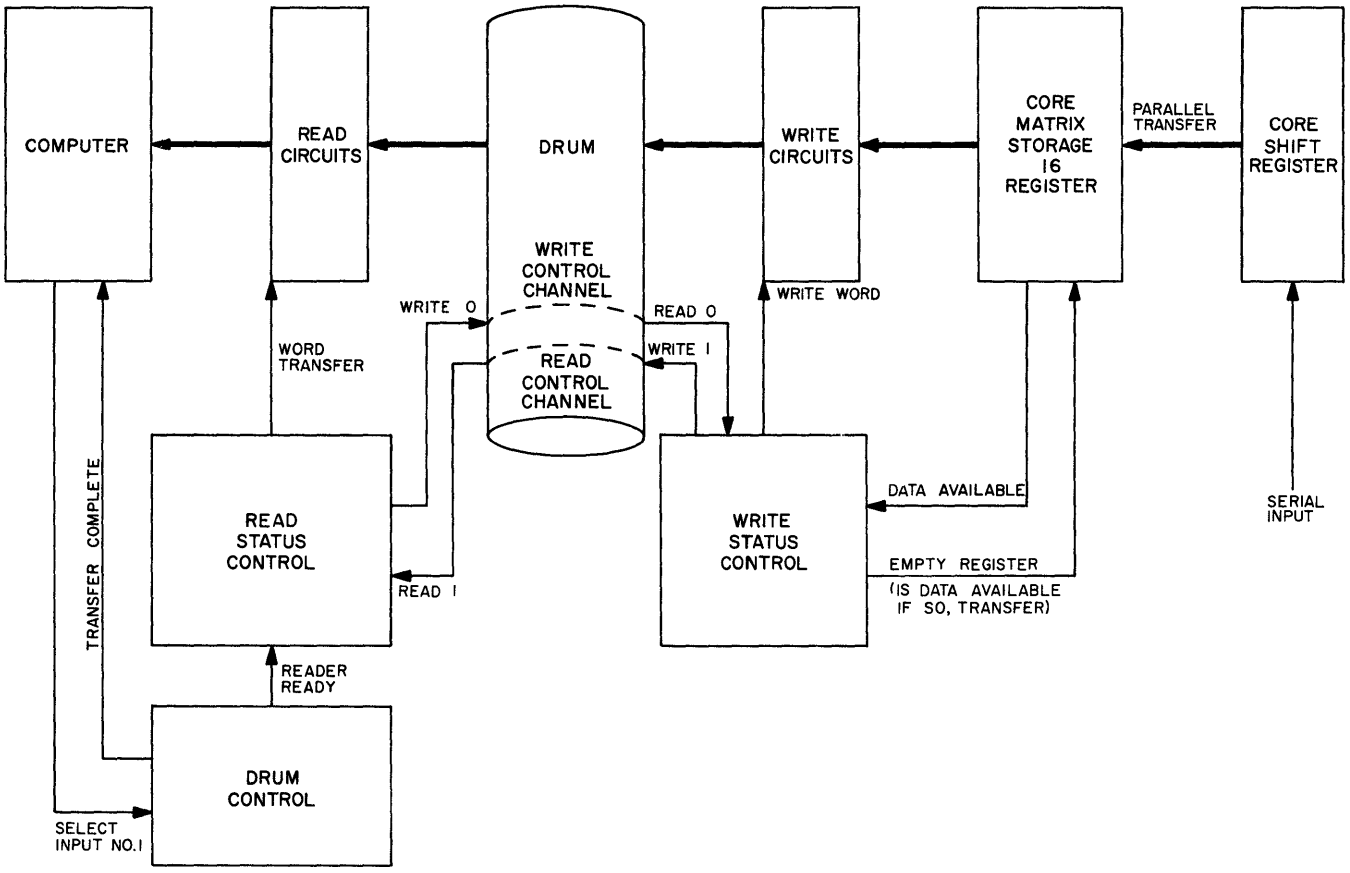


Figure 4–21. Input System

computer to go into a pause (stop program progression) until the IO operation is finished.

5.4 AUTOMATIC INPUTS

5.4.1 General

The input information comes over telephone lines in a continuous stream of serial, binary words. Each of the words has the same format and contains the same type of information. The job of the input device is merely to take in each serial word, translate it to parallel, and put it on the input buffer drum. Once the word is on the input buffer drum, it is accessible to the computer.

5.4.2 Operation

Figure 4–21 illustrates the operation of the automatic input system. Serial information from the telephone lines is shifted into the core shift register a bit at a time. This core shift register is provided with a serial input and a parallel output. (See 2.2.3.2 of Part 3.) As soon as the core shift register contains a full word, it is transferred to the core matrix storage. This is a 16-word matrix storage used to store the input words until an empty drum register becomes available. When the write status control indicates that an empty drum register is under the write heads, this core matrix storage is examined. If data is available, the status control causes the oldest word in core matrix storage to be written on the drum.

The drum serves as an assembly point to collect information from the Input System. The computer program is set up so that it only asks for an input reading operation at intervals. These intervals are of sufficient length so that the drum nearly fills up before a *Read* instruction is given. Then, a *Read* instruction causes the drum to read out a large block of information. (In this computer, this can be accomplished by an indexed repetitive program).

When the computer is instructed to read a word from the input, a pulse is sent to the drum selection controls. These controls condition the read-status-control circuits. When a full register is found by the status circuits, a word is transferred from the drum to the computer. When this transfer is complete, a pulse is sent to the computer to indicate that the transfer is complete. This pulse is used to condition the computer control so that it may continue with the program.

5.5 TYPEWRITER INPUT

The operator communicates with the computer by

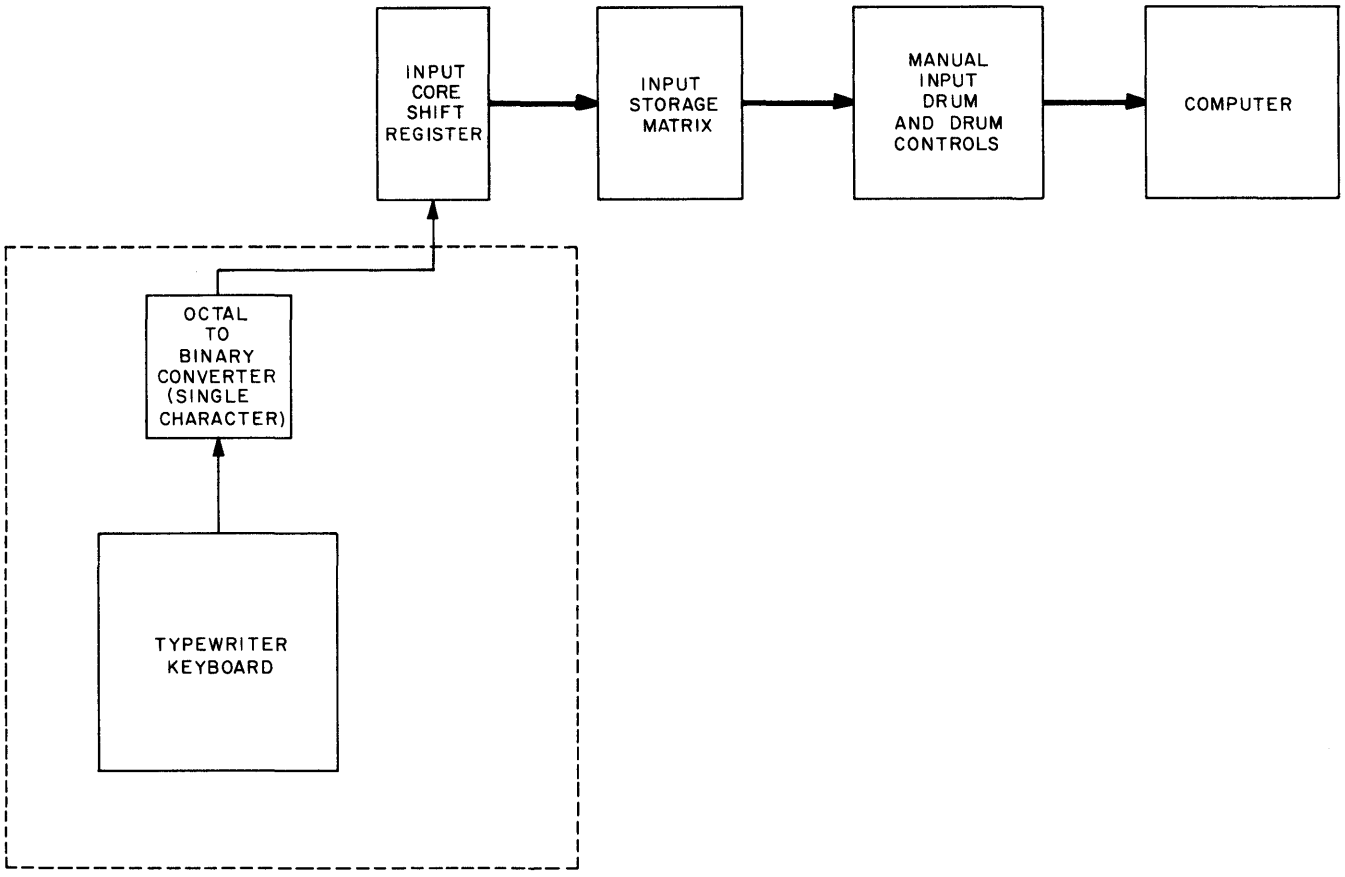


Figure 4–22. Manual Input

means of a modified typewriter. Since this typewriter can only type octal numbers, the operator must code any information he wishes to insert in the computer into an octal number code before it is entered. This is common practice in all programming work.

The typewriter has a converter which converts each octal number to binary as it is typed. This binary input from the typewriter enters the machine in serial form and is treated in exactly the same manner as was th input information entered in the automatic inputs.

Figure 4–22 illustrates the manual input system described. The output of the typewriter in octal-coded characters is fed into the octal-to-binary converter. The output of this converter (binary) is shifted into the input system core shift register. From there it reaches the computer via a status-controlled drum, as in the automatic input system.

5.6 AUTOMATIC OUTPUTS

5.6.1 General

The use of the computer in a real-time control process such as air defense requires on-line automatic outputs. These outputs feed binary information serially to telephone line transmitter. This output can then be transferred to remote users (such as missile directors) on telephone lines.

The information on telephone lines is a series of serial, binary words. Each word has the same format and length. The job of the output system, therefore, is to obtain the word from the computer, translate it from parallel to serial and then transfer it to a modulator which will transmit it over telephone lines.

5.6.2 Operation

5.6.2.1 Program Operation

When it is necessary to read a word from the computer, the instruction *Write Automatic Output* from memory address X is given. This instruction stops further computing and causes the word in memory location X to be translated into the output buffer register. It remains there until the first empty drum register comes under the write heads. At this time, the status control circuits cause the word to be transferred. When the transfer is complete, the drum control circuits produce a pulse which allows the computer to start on the next instruction. Again, this type of operation is employed to allow for drum-search time necessary during the transfer operation. Normally, the outputs are read out of the computer a drum at a time by a series of

repeated *Write* instructions. When this is done the drum will start off empty so that the search time lost is not great.

5.6.2.2 System Operation

Figure 4–23 illustrates how a word is read out of the computer by automatic outputs. When the instruction is given, the word is transferred from memory to the drum buffer. From there the status control circuits cause it to transfer to the drum and, eventually, to the core matrix storage. From the core matrix storage it goes into a core shift register where it is translated into a serial form. In serial form it is fed out of the computer and into a telephone line modulator, which sends it out.

5.7 DISPLAY OUTPUT

5.7.1 General

One of the best ways to present a great deal of information to a person in a easily assimilated form is by means of a picture. This is the purpose of the cathode-ray-tube display output. The information is processed in the computer so that the various binary-number results are proportional to the deflection voltages necessary to generate a picture output. The information is read out of the computer into the display system where the binary information is converted into analog voltages proportional to the quantity expressed by the binary numbers. These analog voltages are then used as the X and Y direction deflection voltages of a cathode-ray-tube. Thus, the picture desired is formed.

5.7.2 Program Operation

Th program operation is the same as that of the automatic output, except that the instruction used is *Write Display* address X. First, the data is prepared in memory so that a block of information gives the desired picture; then, this block of information is transferred to the display system.

5.7.3 System Operation

The system operation of the display is, as shown in figure 4–24, similar to that of the automatic outputs, except in the translation and output device sections. The information is transferred through the drum by status. When it gets to the output read heads, the reading is under control of the display controls and the drum status controls. The display controls call periodically for a transfer of all information on the drum. This information is converted to analog deflection voltages by the converter, and the output appears as a picture on the cathode-ray tube.

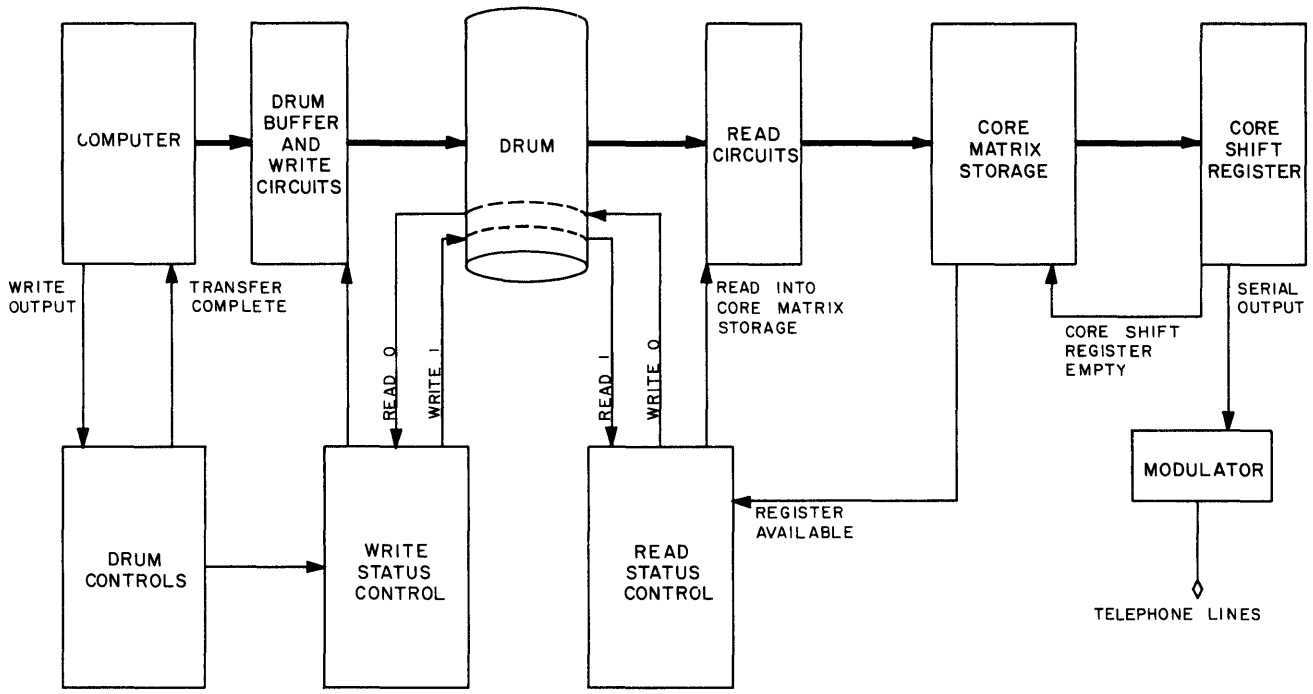


Figure 4–23. Automatic Output System

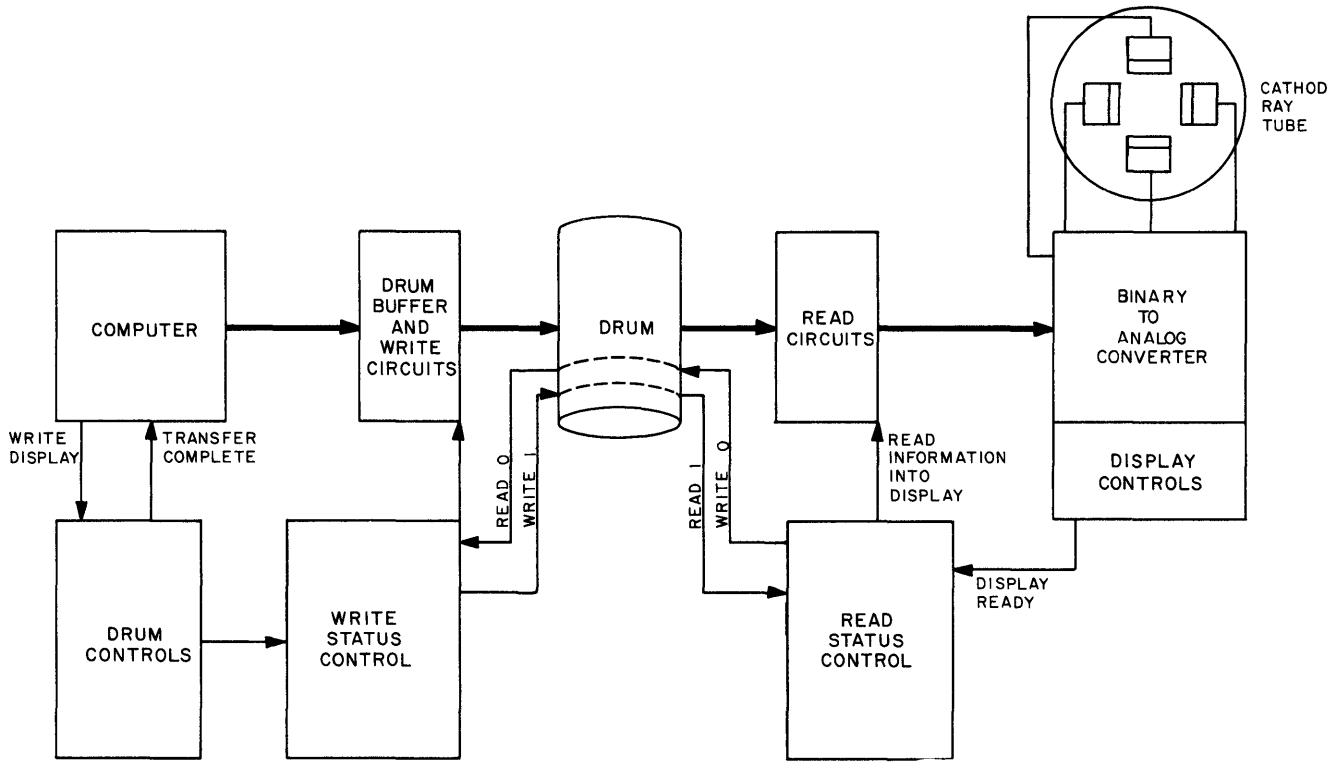


Figure 4–24. Display Output

CHAPTER 6
OPERATION OF THE SAMPLE SYSTEM

6.1 GENERAL

The sample computer was developed to be able to process data for an air defense control activity. The description below is given with this purpose in mind. It should be remembered that the computer could easily be used for different purposes, particularly if the inputs and outputs were adapted to a different use. For instance, if the inputs and outputs were changed from automatic to card machines, the computer could easily be used as a business data processor.

The operation of this or any other computer can be considered as consisting of four phases. The program must be entered first because all operation depends upon it. Next, the data is entered under the control of the program. Once enough data has entered the machine, the actual processing starts. Finally, of course, the output must be presented to the user.

Once the program is finished one of two things may be done. In some cases the computer stops and a new program and data must be inserted for solution of a new problem; in a real-time application, however, the computer repeats the same program over and over again, each time using a new block of input information.

6.2 PROGRAM LOADING

The method of loading the program depends upon the complexity of the program storage requirements. In this sample computer, a manually operated control is furnished which causes the computer to read words from the input drum into sequential addresses of the memory, starting with address 000. This control is operated. Then, the coded instructions are inserted via the typewriter input. This system works well when there is no complication in the program's storage requirements. However, when a complicated storage pattern exists (e.g., some of the program being stored in auxiliary memory), a loading program must be used.

A loading program is one used to control the loading of the operational program. When the loading program is used, it is inserted first under manual control. Then, the operational program is stored according to the instructions of the loading program. After the operational program is stored, the computer is prepared to start computing. This it does according to the last instruction of the loading program, which is a branch to the first instruction of the operational program.

6.3 LOADING AND PROCESSING DATA

The first step of the operational program is to store the data to be used. This data is constantly entering the machine from the automatic input. It is changed into standard computer words by the input system and then stored on the input buffer drum. At certain intervals of time, when enough information is on the drum, the computer calls for a block transfer of the input information from the input drum to the computer memory. It may be recalled that this transfer is accomplished by means of a small, repetitive program.

Enough data is now in the memory so that the computer can process it. In general, the processing can be split into three phases: The information is prepared for the operator's use; the operator sees the processed information and decides what action can be taken. The action decided upon is inserted in the machine and the machine further processes the information to produce outputs.

The air defense problem requires the compilation of information from input sources, processing this information into an easily used form and then presenting this information to operating personnel to enable them to take the appropriate action on the basis of the information. The operations involved in the air defense problem and in most other data processing applications can be classified as data simplification operations; that is, the operations tend to reduce a large number of discrete items of information into a smaller, more comprehensible body of information. The simplified data is presented to the operator in the form of a display.

The main purpose for including the operator in the system is in the type of decisions he is required to make. He must make decisions based upon the best information the computer can give; but his decisions are also based upon information which is not in the computer. For instance, his decisions as to the disposition of a supposedly hostile aircraft will depend not only upon the fact that the plane is not identified but upon whether it is likely that a war has started. In other words the decisions are based upon an overall picture of the strategy to be followed at the moment. This is the type of decision which requires the imagination, intuition, and judgment which only a man posseses.

The human operator obtains the essential informa-

tion in its condensed and easily assimilated form from the display. He combines this information with what he knows and decides what, if anything, should be done. He then notifies the computer of his decision by way of the manual input.

The computer then takes the new information and processes it further. It is combined with the originally processed information to be shown to the operator on the display (if needed). The computer also processes the results of the whole operation into the form of an automatic output message. This message then is sent to the output drum and, eventually, is converted to the proper form for transmission to the eventual users (e.g., the weapons bases).

PART 5
PRINCIPLES OF PROGRAMMING

CHAPTER 1
INTRODUCTION

1.1 GENERAL

This part presents the information necessary for an understanding of the principles of programming. In Chapter 2, the basic techniques involved in programming the computer will be explained and will include the knowledge required of a programmer, problems encountered in program preparation, and aids available to a programmer. Several sample problems will be analyzed utilizing standard techniques. Chapter 3 presents a summary of the capabilities and limitations of a computer. Various methods for extending the usefulness of large-scale computers will also be discussed.

1.2 PROGRAM DEFINITION

A program may be defined as a series of instructions, coded in a form recognized by the digital computer, calling for the operations to be performed by the digital computer in the order necessary to solve a given problem. For example, even the solution of a simple arithmetic problem requires a program, whether solved by a digital computer or by a man with pencil and paper. Although the man can recognize the necessary steps in a program, the digital computer must be given step-by-step directions for the solution of any problem.

1.3 NECESSITY FOR PROGRAMMING

Without this series of instructions the computer would not be capable of performing any type of operation. The necessity for programming becomes apparent; it must be used to initiate and exercise control over the operations of the computer. This control may be predetermined through the use of a specific instruction, or it may depend on the value of the numbers being manipulated at any particular point. In addition to controlling arithmetical operations in the computer, programs are used for various other functions, such as maintenance routines, monitoring, etc. The different types of programs will be discussed in Chapter 4.

CHAPTER 2
TECHNIQUES OF PROGRAMMING

2.1 GENERAL

The program is designed after obtaining a statement of the problem to be analyzed. With this initial requirement satisfied, four subsequent phases listed below are required to produce a finished program.

a. Problem analysis
b. Program organization
c. Program coding
d. Program testing

In general practice, the first phase, problem analysis, is handled by mathematicians, and the last three phases are handled by programmers. However, problem analysis often determines the organization of the program; therefore, problem analysis is usually done either by a mathematician-programmer or by a mathematician and a programmer working as a team.

2.2 PROGRAM PREPARATION

2.2.1 Problem Analysis

After a statement of the problem is obtained, all of the factors that may be encountered have to be examined and arranged in a mathematical expression. This expression must represent the problem expressed as simply as possible. It is usually quite complex at this point and must be reduced to even simpler terms (addition, subtraction, etc.) by a mathematical technique known as numerical analysis.

Numerical analysis, involves the reduction of complex mathematical operations to arithmetic operations within the capabilities of the computer being programmed. The most common reductions are: calculus operations to simpler arithmetic operations such as changing integration to an approximate summation operation, and changing differentiation to an approximate difference-quotient operation. These changes result in approximations to the more complex methods, but approximations which can be made as exact as desired. Given a method of approximation by the mathematician, the programmer must then determine the program to obtain the result.

2.2.2 Organization

When the programmer receives a problem and its numerical analysis, the first step is that of organizing a program to solve the problem using the arithmetic methods outlined in the numerical analysis. Program organization involves the sequencing of the operations to be performed into an order which will simplify coding, minimize execution time, and, if possible, minimize the number of storage registers required. At this point, a flow chart which is a pictorial representation of the structure of the program (see fig. 5–1) is useful, both to keep the entire program in view and to develop the sequence of operations into the proper order. Very few programmers can take a complex problem and draw a flow chart which exactly fits the problem on the first try. Therefore, the flow chart will start out in rough form and become finalized only after some thought and some reworking has gone into it.

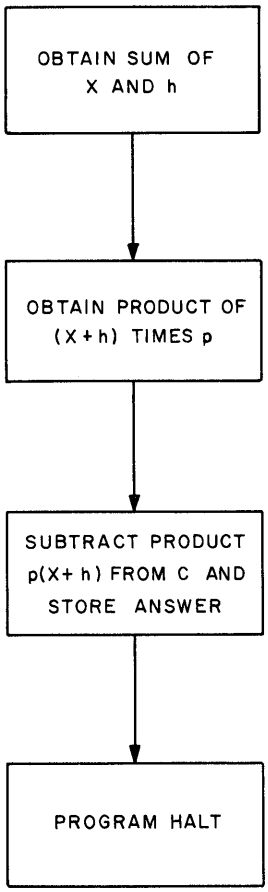


Figure 5–1. Flow Chart for Straight-Line Program

2.2.3 Coding

Once a tentative flow chart has been prepared, the

program can be coded. The coding operation can be performed block by block from the flow chart. Consideration must be given to the data provided by the preceding block and the data required by the following block of the program. While the coding is being done, the techniques of selecting library routines (universal routines already written and catalogued for use), of determining the precision required, and of scaling are brought into play. The product of the coding phase is a mnemonically coded program, ready for testing.

2.2.4 Testing

Once a program has been completely coded, it must be tested to insure its proper operation in solving the given problem. This program testing period is similar to the shakedown period for a new piece of equipment; the logical design of the program is tested and revised until it correctly performs its intended function. In the course of program testing, modifications of both the program organization and the coding may be required to get the program into proper operation. Program organization and program coding should be done with utmost care to avoid having program testing become excessively time-consuming and difficult.

CHAPTER 3
PROGRAM EXAMPLES

3.1 GENERAL

This chapter consists primarily of sample programs together with their respective flow diagrams and explanations. Each program is considered to be designed for use with the machine described in Part 4. As the examples become increasingly complex, new instructions and techniques used in preparation of programs will be introduced.

3.2 STRAIGHT-LINE PROGRAM

3.2.1 General

A straight-line program is a series of instructions which fall into numerical order. In this type of program, step 1 is always followed by step 2, step 2 is always followed by step 3, etc. This is the easiest type of program to understand from the operator's point of view. However, it is not necessarily the easiest type of program for the computer to execute because of the time required and the space in core memory that is needed for data storage.

3.2.2 Statement of Problem

In the following case, the problem is to evaluate the imaginary expression for the amount of fuel remaining in the tank of a jet interceptor. The factors necessary for the derivation of such an expression are:

C = capacity of tank

p = pounds consumed per mile at average speed of aircraft

x = distance flown

h = altitude factor

The computer instructions which are available to the programmer for the solution of this problem are given in table 5–1. The first column lists the instruction name, and the last column gives the operation performed. The CODE column gives the mnenonic code for the instruction. The capital letters represent the operation part of the instruction, and the lower case x represents the address of the register in core memory from which the operand is to be obtained. For instance, *CAD 10* means to clear the accumulator and place the contents of memory register 10 in the accumulator.

3.2.3 Problem Analysis

It is apparent that the resultant expression will be the capacity of the tank less some combinations of the other factors. To find the fuel consumed, all that is necessary is to multiply the miles-per-pound factor by the distance traveled plus the altitude factor. The alti-

TABLE 5–1. BASIC COMPUTER INSTRUCTIONS

INSTRUCTION NAME	MNENONIC CODE	OPERATION PERFORMED
Clear and Add	*CAD* x	Clears the accumulator of any value remaining from another operation, then places contents of x in accumulator.
Add	*ADD* x	Adds contents of x to the contents of the accumulator. At the end of the operation, the contents of location x are unchanged.
Subtract	*SUB* x	Subtract contents of x from the contents of the accumulator. At the end of the operation, the contents of location x are unchanged.
Multiply	*MUL* x	Multiplies contents of x by the contents of the accumulator. At the end of the operation, the contents of location x are unchanged.
Full Store	*FST*	Places the contents of the accumulator into memory location x. Contents of the accumulator also remain there.
Program Stop (or *Halt*)	*HLT*	Stops computer operation (address portion of this instruction is meaningless).

tude factor must be added to the distance traveled since the height above sea level at which a jet aircraft flies determines the rate of fuel consumption. This may be expressed as $C - p(x+h)$.

No actual numerical analysis is required on this expression since it does not contain any higher mathematics, only addition, subtraction, and multiplication. All of these may be handled by the computer; therefore, the expression is in its simplest form as it now stands.

3.2.4 Organization

Because this problem involves straight-line programming, the sequence of instructions will be fairly simple. At this point, the general structure of the program may be formed by utilizing a flow chart. Looking at the expression $C - p(x + h)$, one can see that x and h must be added, the resultant sum multiplied by p, and, finally, the product subtracted from C. The flow chart in figure 5–1 shows what the program is going to do, and in what sequence, but does not actually list the steps involved.

3.2.5 Coding

The blocks in the flow chart may now be broken down into individual instructions with the appropriate operation and address parts specified for each instruction. The memory locations for the data are arbitrarily selected by the programmer and do not affect the program in any way. Once the coded instructions have been selected they are numbered sequentially, and the program is now completed and ready for testing. (Refer to table 5–1 for an explanation of the instruction codes.) In the coded program shown in figure 5–2, the number to the left of the mnenonic code for each instruction is the address of the memory location containing the instruction. An explanation of the coded program showing the results of each operation is given in table 5–2. In this table, as well as in figure 5–2, locations 1 through 8 refer to the location of the instruction words which must be in numerical sequence. No testing is involved in this program because of its simplicity, but in actual practice the testing phase is the next logical step of program preparation since many programs contain several thousand instructions.

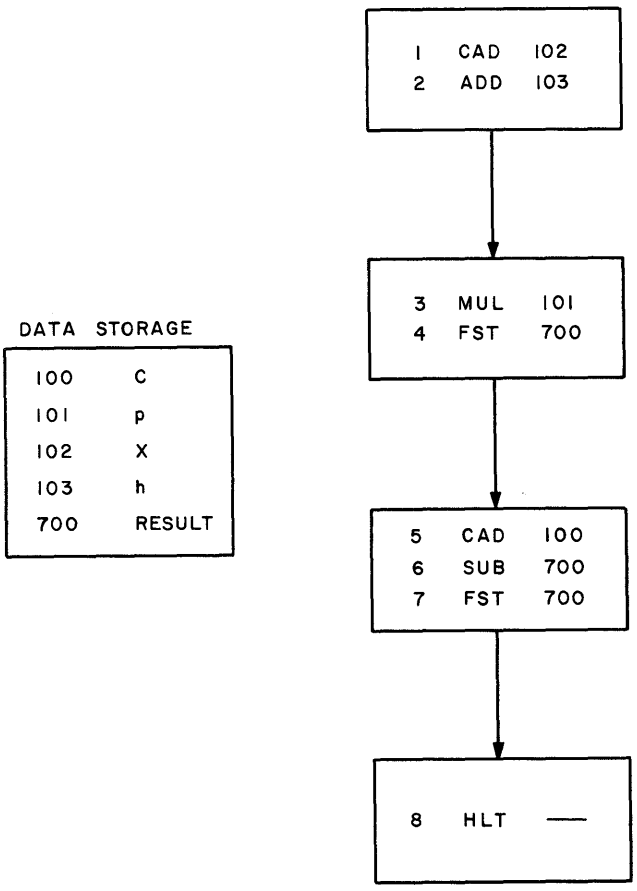


Figure 5–2. Flow Chart, Coded Straight-Line Program

3.3 LOGICAL PROGRAM

3.3.1 General

A logical program is one in which the steps involved constitute some type of operation where the final result will be logical rather than arithmetical. A human being can use the logical process for such things as sorting numbers, comparing sizes, etc., but the computer cannot think for itself; consequently, any type of operation that is considered logical must first be designed by a programmer to fall within the capabilities of the computer, which are strictly arithmetical. This type of program does not find the value of an arithmetic expression, as did the straight-line program in the first example given, but merely uses arithmetic functions to arrive at some sort of logical decision.

3.3.2 Statement of Problem

The logical operation involved in this program is to determine the largest of 3 numbers. The numbers represent three flights of enemy planes, with the largest number posing the greatest threat. Only the three numbers, N_1, N_2, and N_3, are given.

All the instructions listed in table 5–1 are available, plus two new instructions given in table 5–3.

3.3.3 Problem Analysis

The problem, in the case cited, is to select the largest of three numbers, rather than to set up an arithmetical expression. The analysis may begin by comparing any two numbers since the program is not confined to any specific starting point. However, it is known that to find the largest of three numbers, two comparisons must be made regardless of which number is selected first.

TABLE 5–2. STRAIGHT-LINE PROGRAM

LOCATION	INSTRUCTION OPERATION	INSTRUCTION ADDRESS	COMMENTS	CONTENTS OF ACCUMULATOR
1	*CAD*	102	Places x in accumulator.	x
2	*ADD*	103	Adds h to x	(x + h)
3	*MUL*	101	Multiplies (x + h) by p.	p (x + h)
4	*FST*	700	Places p (x + h) in location 700.	p (x + h)
5	*CAD*	100	Places C in accumulator.	C
6	*SUB*	700	Subtracts p (x + h) from C.	C—p (x + h)
7	*FST*	700	Places desired result in location 700.	C—p (x + h)
8	*HLT*	—	Stops computer operation.	C—p (x + h)

Data Storage Location	Contents	Comments
100	C	Capacity of fuel tank
101	p	Consumption rate
102	x	Distance flown
103	h	Altitude factor
700	—	Initial and final result storage

3.3.4 Organiaztion

At this point, a flow chart, with arbitrary selection of numbers, can be started. As shown in figure 5–3, the first two numbers compared are N_1 and N_2. Since there are two possible alternatives after this comparison a decision block must be included in the flow chart. This block is shown with the two possible paths leading away from two of the points. A decision block must follow each comparison, because the value of the numbers being compared is not known at any comparison point. When completed, the chart will enable the programmer to design a program which will cover all the possible selections and still determine the largest number by the process of elimination. Notice that in the actual execution of the selection, only one path will be followed depending upon which number is largest.

3.3.5 Coding

A straight transfer from flow chart to the coded layout is shown in figure 5–4. The *BFM* instruction determines the path the program takes. This program is correct in every respect and will find the largest number, if used as shown. However, as shown in figure

TABLE 5–3. BRANCHING INSTRUCTIONS

INSTRUCTION NAME	MNEMONIC CODE	OPERATION PERFORMED
Unconditional Branch	*(0) BPX x*	This instruction changes the sequence of instructions followed in a program. The next instruction executed will be obtained from location x. The zero (0) specifies that the branch is unconditional; that is, it does not depend on any special circuit polarities.
Branch on Full Minus	*BFM x*	The contents of the accumulator are examined, and if negative, the next instruction executed will be obtained from location x. If the contents of the accumulator are positive, the next sequential step in the program is taken.

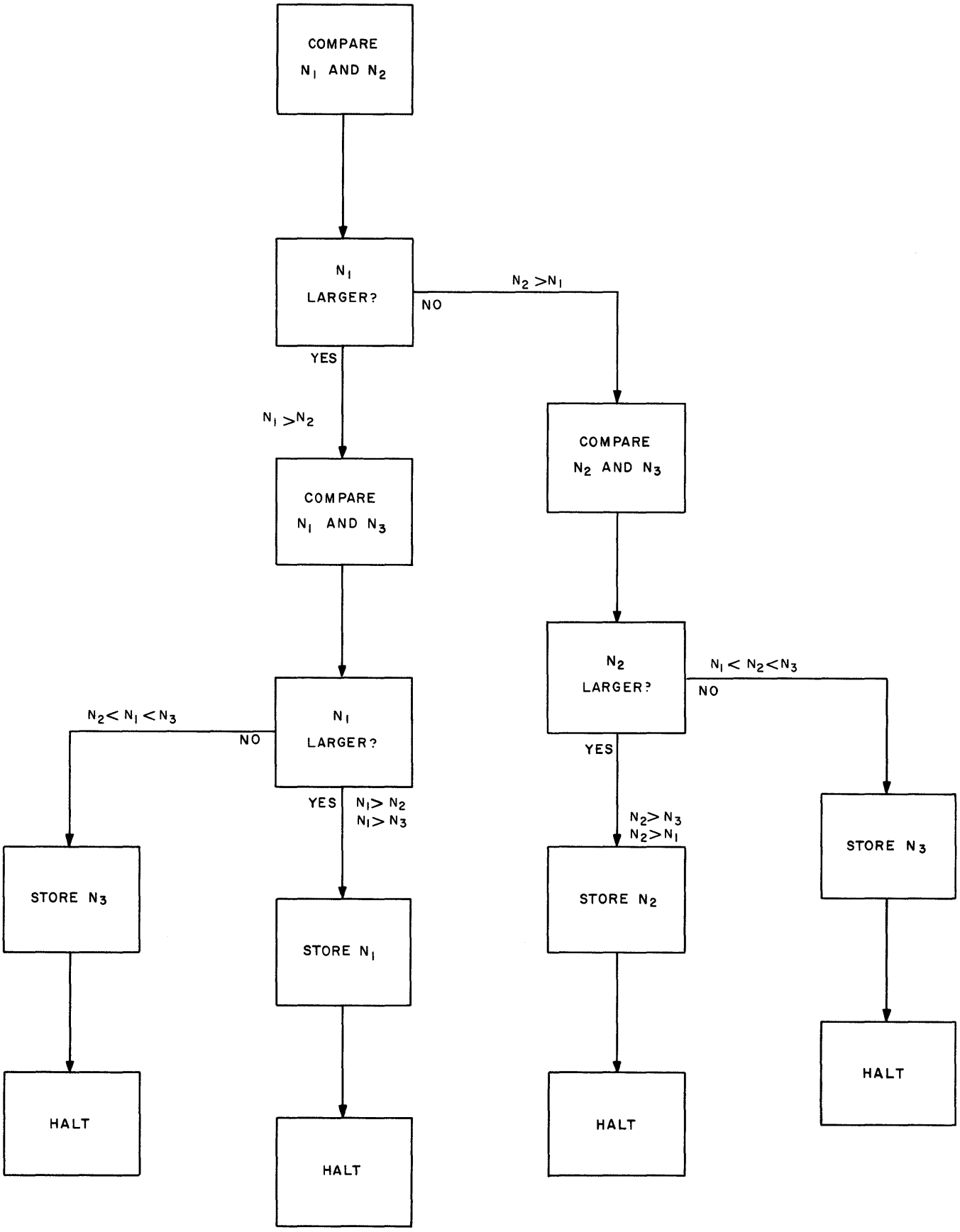
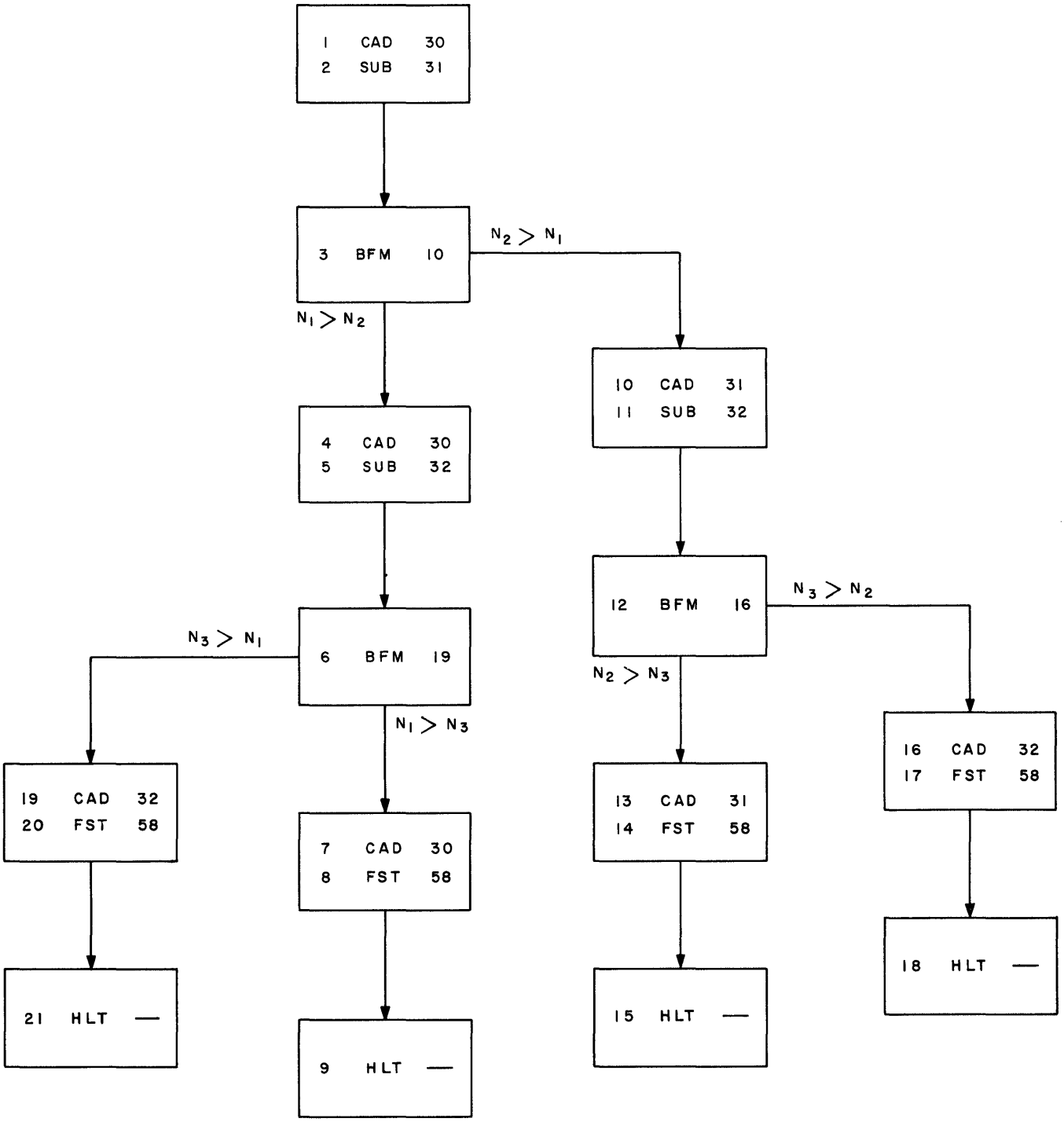


Figure 5–3. Flow Chart for Logical Program



DATA STORAGE

30	N_1
31	N_2
32	N_3
58	RESULT

Figure 5–4. Flow Chart, Coded Logical Program, Preliminary Layout

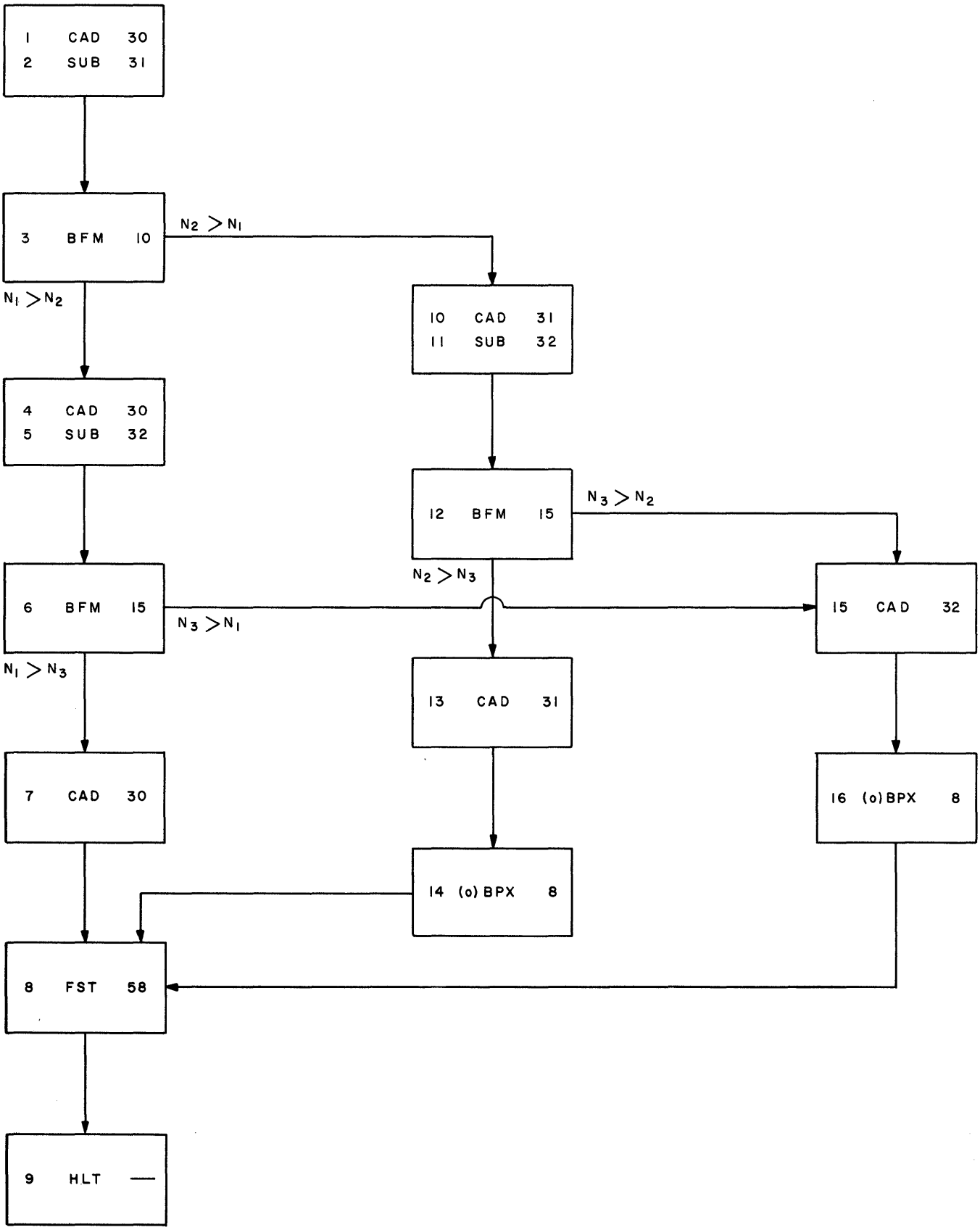


Figure 5–5. Flow Chart, Coded Logical Program, Final Layout

5–5, the program could be written using fewer instructions and, consequently, using fewer memory locations. Upon close examination, it can be seen that the last two steps (storing the number found to be the largest and halting the computer) in each of the four paths are the same. Also, in two cases, N_3 has been found to be the largest; so a *(0) BPX* instructions can be used to combine the last 3 steps of the two cases into one 3-step sequence. When this is done, there remain 3 paths which have two identical, final instructions. Therefore, by use of a *(0) BPX* instruction the final two instructions of the 3 remaining paths can also be combined into a single 2-step sequence. If N_1 is the largest number, the store process directly follows the comparison of N_1 and N_3. If N_2 or N_3 are larger a *(0) BPX* will direct these numbers to the *FST* instruction. Thus, the program can be reduced from 21 steps to 16 by the use of the *(0) BPX* instruction. The *(0) BPX* instruction is extremely useful in computer operations since it reduces the amount of space required for storage of the program and gives the computer added versatility. A full explanation of the finalized program is given in table 5–4. Again, no testing phase will be discussed.

TABLE 5–4. LOGICAL PROGRAM

LOCATION	INSTRUCTION OPERATION	INSTRUCTION ADDRESS	COMMENTS	CONTENTS OF ACCUMULATOR
1	*CAD*	30	Places N_1 in accumulator.	N_1
2	*SUB*	31	Subtracts N_2 from N_1.	$N_1 - N_2$
3	*BFM*	10	Examines contents of accumulator; if negative, branches program to location 10; if positive, go on to instruction 4.	
4	*CAD*	30	Places N_1 in accumulator.	N_1
5	*SUB*	32	Subtracts N_3 from N_1	$N_1 - N_3$
6	*BFM*	15	Examines contents of accumulator; if negative, branches program to location 15; if positive, go on to instruction 7.	$N_1 - N_3$
7	*CAD*	30	Places N_1 in accumulator.	N_1
8	*FST*	58	Stores largest number in location 58.	N_1
9	*HLT*	–	Stops computer operation.	N_1
10	*CAD*	31	Places N_2 in accumulator.	N_2
11	*SUB*	32	Subtracts N_3 from N_2	$N_2 - N_3$
12	*BFM*	15	Examines contents of accumulator; if negative, branches program to location 15; if positive, go on the instruction 13.	$N_2 - N_3$
13	*CAD*	31	Places N_2 in accumulator.	N_2
14	*(0) BPX*	8	Branches program to location 8.	N_2
15	*CAD*	32	Places N_3 in accumulator.	N_3
16	*(0) BPX*	8	Branches program to location 8.	N_3

Data Storage Location	Contents	Comments
1-16	Program	
30	N_1	
31	N_2	
32	N_3	
58	–	Storage for largest number

3.4 ITERATIVE PROGRAM

3.4.1 General

An iterative program is one in which the same group of instructions is used several times by modifying the address portion of the instruction. Such a repetition of instructions is called an iterative loop and is employed frequently in computer operations.

3.4.2 Statement of Problem

In this example, the problem is to find the total number of guided missiles available at a particular time, for air defense of a sector which has 20 guided missile bases in the area. The number of missiles available at each of the 20 bases is known, but the grand total is not. The number of missiles at any one base is stored in one memory location, and the twenty memory locations storing the missile quantities are in a block of sequential addresses reserved for missile status reports.

All the instructions listed in table 5–1 and table 5–3 are available for the solution of this problem.

3.4.3 Problem Analysis

No actual analysis of this problem is needed. This example is intended primarily to show how twenty numbers can be added employing an iterative program without the use of index registers and not merely how to add 20 numbers. The general operation will be to add the 20 numbers using the same instruction over and over again with an address which is modified to specify new data each time the instruction is repeated.

3.4.4 Organization

In this type of problem there must be at least two branch instructions, one to get into the iterative loop and another one to leave it. It will be shown that, if there were no provision for leaving the loop, the program would become hung up and could not complete the problem. However, there must be some provision for telling the computer when to stop making passes through the loop. This is done by comparing the results of a step with a constant, and branching back for another run through the loop if the result is negative. The flow chart for this problem is shown in figure 5–6. The general flow of the program is to add one of the numbers to the accumulated total, modify the address, determine if the required number of iterations have been completed, and, if they have, not to repeat the process. The branch in step 2 is necessary to prevent adding the contents of memory location 150, prior to the start of the program, into the total being sought (see table 5–5).

3.4.5 Coding

The flow chart may now be replaced by the coded program which is shown in figure 5–7. By examination, it can be seen that this type of program has a

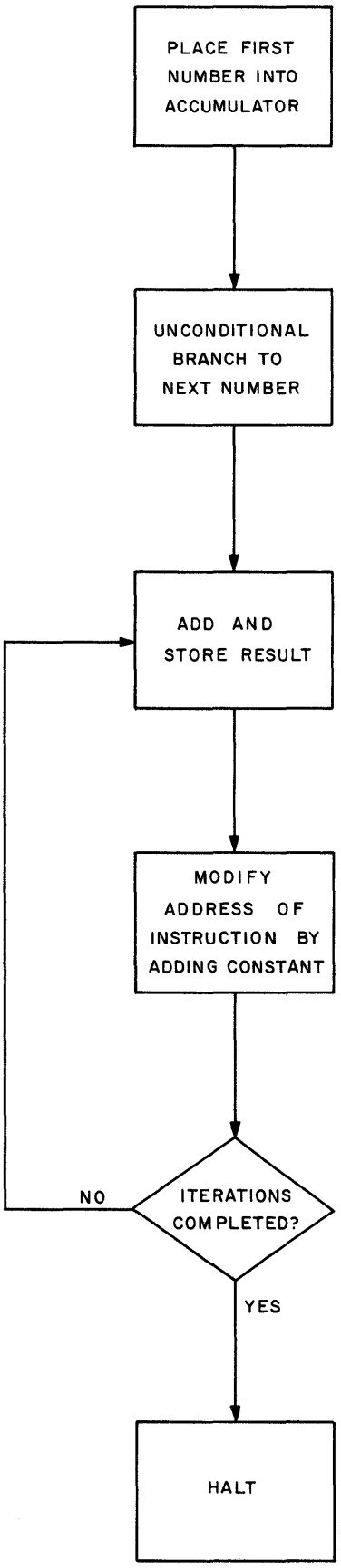
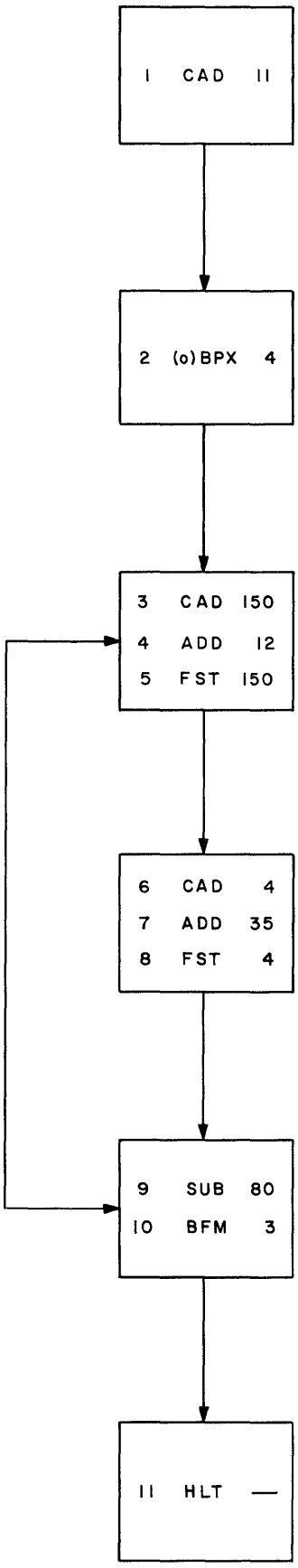


Figure 5–6. Flow Chart for Iterative Program



LOCATION	CONTENT
11-30	INITIAL DATA STORAGE
35	CONSTANT OF 1
80	30 (VALUE OF LAST ADDRESS)
150	TEMPORY AND FINAL RESULT

Figure 5–7. Coded Iterative Program

distinct advantage over the straight-line addition of twenty numbers, because much less memory space is required for the storage of the program. A full explanation of the program is given in table 5–5.

3.5 INDEXED ITERATIVE PROGRAM

3.5.1 General

An indexed iterative program is basically the same as the nonindexed type, the only difference being that the indexed type has a specific index register which modifies the instruction. The *Branch and Index* instruction (Unconditional Branch), together with the one which loads the index register, is explained in table 5–6. The index registers change the address portion of an instruction by adding the contents of the register to it. However, the original address specified in the instruction is not destroyed; it is only modified for that one particular step.

3.5.2 Statement of Problem

Since the problem to be analyzed is the same one used in 3.4, the flow chart can be drawn immediately.

3.5.3 Organization

The first thing to be done in the organization of a program of this type is to determine what value should be placed in the index register. In this problem, twenty numbers are to be added together, so the index register is loaded with the value 18. The justification of this lies in the fact that 19 repetitions (numbered 0 through 18) are to be made, because the first run through the program is not a repetition. The index register will modify an address as long as the value contained in it is positive. Therefore, on the next to last pass through the program, the index register contains 1. On the last pass, the index register contains positive zero, so the instruction is not modified (zero added to it) and the original address is selected. Then the register steps negative, and the computer stores the sum of the 20 numbers. The flow chart for this program is shown in figure 5–8.

TABLE 5–5. ITERATIVE PROGRAM

LOCATION	INSTRUCTION OPERATION	INSTRUCTION ADDRESS	COMMENTS
1	*CAD*	12	Places first number in accumulator.
2	*(0) BPX*	4	Unconditionally branches to first addition; this prevents the contents of location 150 from being added to the sum in place of the first number.
3	*CAD*	150	Places last partial sum in accumulator.
4	*ADD*	13	Adds next number to partial sum (or first number).
5	*FST*	150	Places partial sum (or last addition) into storage.
6	*CAD*	4	Places instruction in location 4 in accumulator.
7	*ADD*	35	Modifies address of instruction 4 by 1.
8	*FST*	4	Places modified instruction in location 4.
9	*SUB*	80	Subtracts value of last address to be added.
10	*BFM*	3	Examines accumulator for polarity, if negative branches to step 3.
11	*HLT*	—	Stops computer operation.

LOCATION	CONTENTS
1-11	Location of program
12-30	Location of data
30	Constant of 1.
80	Temporary and final result storage.
150	Constant of 30 (value of last address to be added).

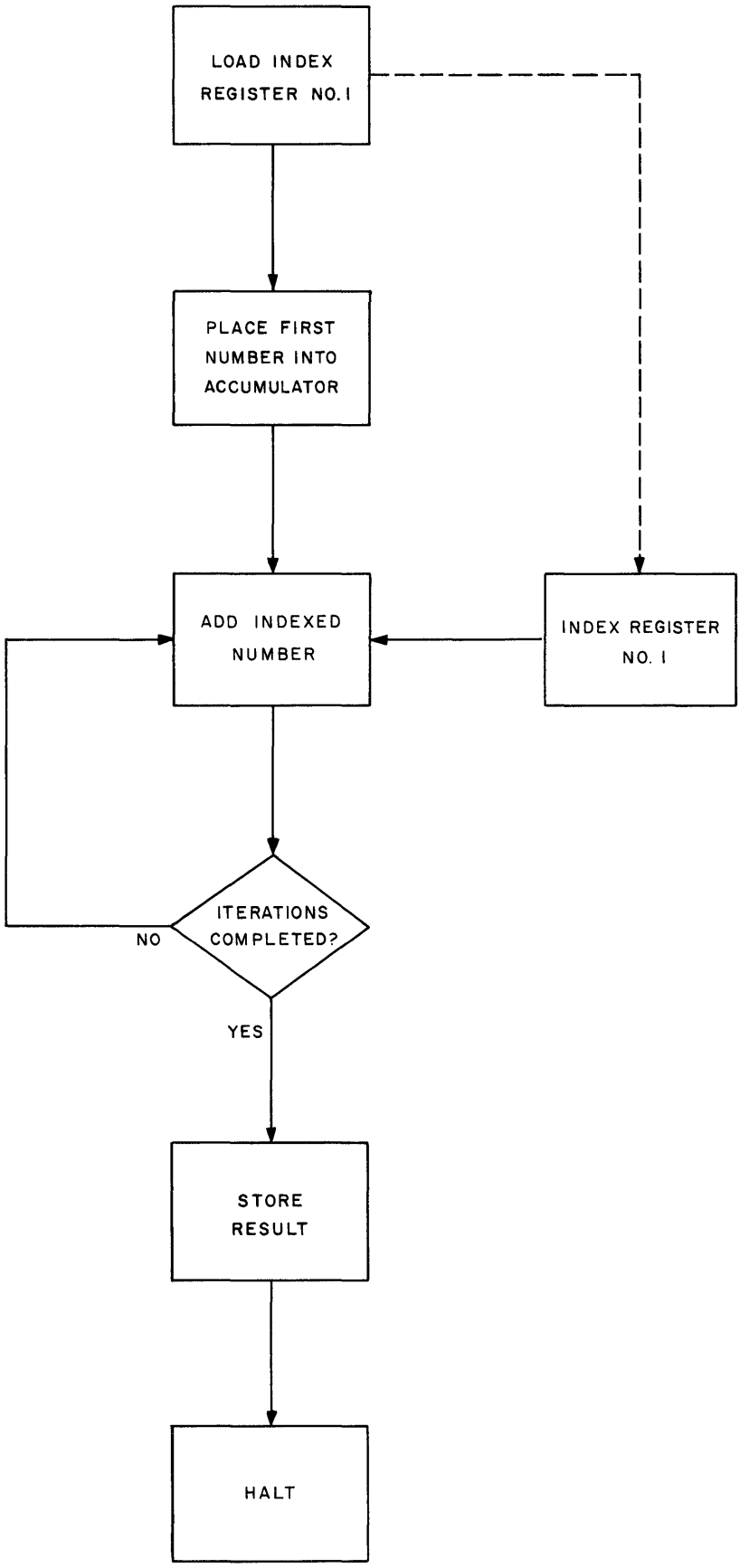


Figure 5–8. Flow Chart for Indexed Iterative Program

TABLE 5–6. INDEXING INSTRUCTIONS

INSTRUCTION NAME	MNEMONIC CODE	OPERATION PERFORMED
Branch if Positive Index	*y BPX (01) x*	Steps the content of index register y down by the amount 1 then, if the contents of the index register is positive, the computer branches to instruction at location x; if contents are negative, go on to next instruction.
Load Index Register	*y XIN (x)*	Loads index register y with the value x.
Indexed *ADD*	*y ADD x*	The contents of index register y are added to the address of the instruction x (in the control elements), then the operand at the modified location is added into the accumulator.

3.5.4 Coding

After the flow chart for this problem is blocked out, the transition to a coded program is made (fig. 5–9). The only instruction which may require clarification is the one in memory location 3. The 1 before the *ADD* step merely indicates to the computer that indexing is specified and which index register is to be combined with the memory location in the address portion of the instruction. Thus, the first number added to the value in memory location 11 would be the value in memory location 30 (12 + 18). The second number added (during the next repetition) would be 29 (12 + 17) and so on until the index register becomes negative and no more repetitions are performed.

It is worth noting that this type of program is even more efficient than the iterative program since the number of instructions required to perform the addition is less. Also, if there were additional data in memory locations 31 through 50, the only change necessary in the indexed iterative program would be to set the index register to a value of 38. However, the straight-line addition program would require an additional 20 instructions, one *ADD* instruction for each number. The indexed iterative program is employed to great advantage in computers because of its great flexibility, as shown in the case of adding 40 numbers with a program which was originally written for 20 numbers. An explanation of the given problem, using the indexed iterative method is given in table 5–7.

TABLE 5–7. INDEXED ITERATIVE PROGRAM

LOCATION	INSTRUCTION OPERATION	INSTRUCTION ADDRESS	COMMENT
1	*1 XIN 18*	–	Loads index register 1 with decimal value 18.
2	*CAD*	11	Places operand in location 11 into accumulator.
3	*1 ADD*	12	On first iterative step, adds operand in location 30 (12 + 18, original address plus contents of index register) into accumulator. On second iteration adds operand in location 29 into accumulator and so on.
4	*1 BPX (01)*	3	Branches back to location 3 if register is positive, steps index register down by 1.
5	*FST*	150	Stores final sum.
6	*HLT*	–	Stops computer operation.

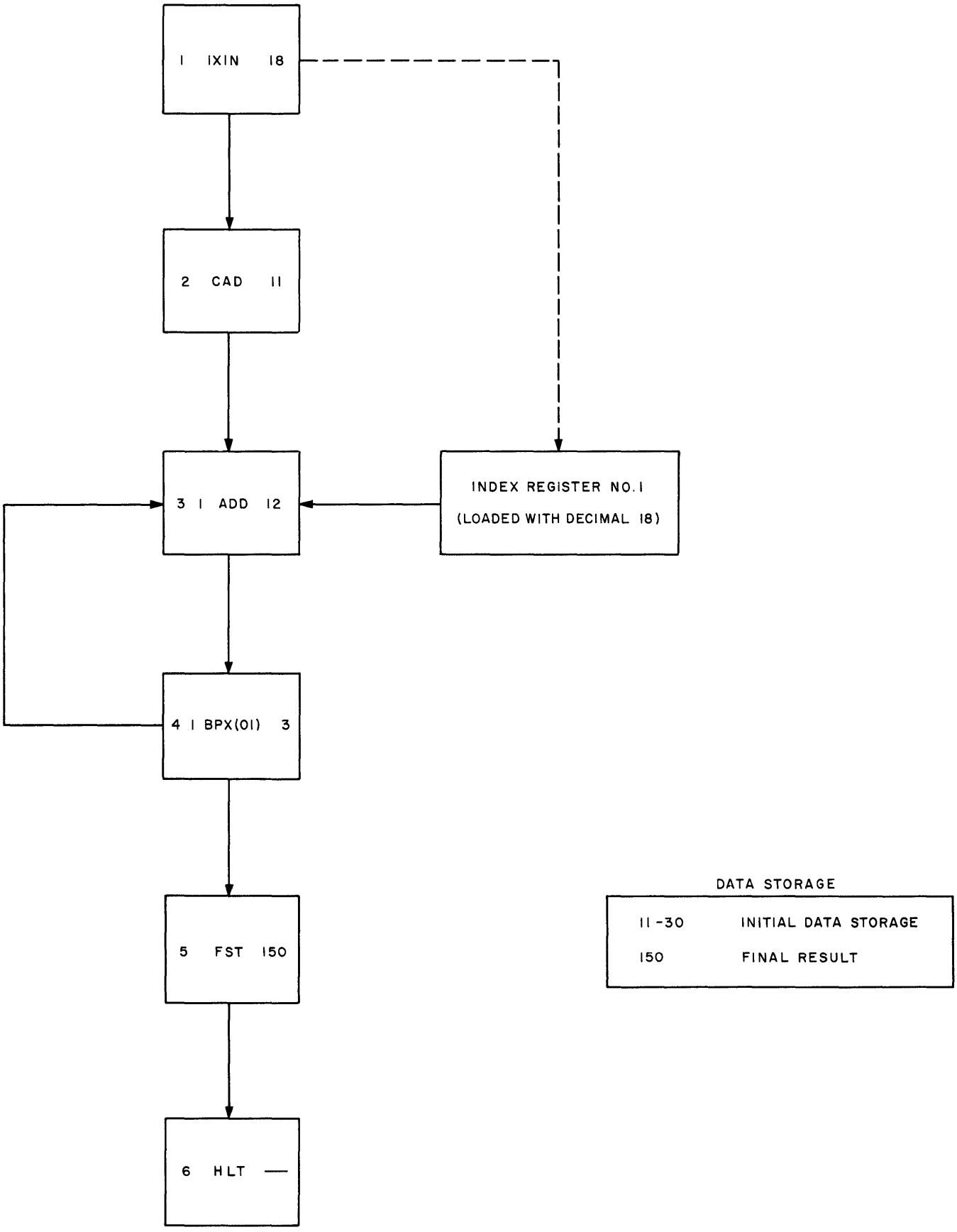


Figure 5–9. Coded Indexed Iterative Program

CHAPTER 4
TYPES OF PROGRAMS

4.1 GENERAL

In this section the different kinds of programs which are handled by computers will be discussed briefly. Some computers can perform more programs than those mentioned.

4.2 EXAMPLES

4.2.1 Master Program

A large program may be made up of several smaller programs, each performing a portion of the overall function of the large one. In some cases, the large program, which does not properly contain these smaller programs, selects each smaller program in the proper order to perform necessary data processing without performing any data processing itself. A program of this type, which controls other programs without itself doing any processing, is called a master program, an executive routine, or a sequence-selection program. In effect, a master program can do no processing except through the use of other programs which, in turn, cannot perform the entire processing task without having the master program to control the sequence in which they are performed.

The smaller processing programs may themselves contain blocks of instructions which can be called subprograms or subroutines. For example, if a given program requires the performance of a particular operation, a routine may be written to perform that operation and placed in core memory with the processing program. The processing program may then be written to refer to the routine whenever the operation is to be performed.

4.2.2 Subroutines

A subroutine is a group of instructions which perform a distinct function and may be written in one of two ways, called open or closed subroutines.

An open routine is designed as a group of instructions which are incorporated into the main body of the program. These instructions do a specific job at the particular point of the program where they are inserted. The routine is *open* with respect to the rest of the program; i.e., it falls into a logical order.

A closed subroutine also performs a certain function; however, this function may need to be repeated several times in one program. A *Branch* instruction is usually required to get into a closed subroutine, and another type of *Branch* is required to leave it. Therefore, this routine is closed with respect to the main program and requires specific instructions to utilize the result of its operation.

4.2.3 Utility Programs

4.2.3.1 General

Utility programs, sometimes called auxiliary programs, perform nonchecking functions such as loading programs, assembly programs, tracing programs, and simulation programs. Loading programs take programmed information from punched cards, tapes, or drums and transfer them into the core memory element. Assembly programs are used to prepare other programs for transfer to punch cards. Tracing programs provide printed records of registers to aid in the following of program operations. Simulation programs are used to pretest programs on other computers known to be operating without error. Simulation programs thus provide a means of detecting program errors (errors which are not due to equipment failure but are mistakes in programmed routines).

Utility programs can also be used to generate a particular pattern which is used to calibrate portions of the equipment and to exercise (operate) a particular group of circuits so that normal waveforms can be observed or design data obtained.

4.2.3.2 Symbolic Program

This type of program is utilized when a long overall program is being drawn up. The final addresses of the instructions may not be known until the program is completed, so some sort of symbols (such as a series starting with 00.00.00) are used to designate memory locations. Although not strictly necessary, consecutive instructions are usually assigned consecutive symbolic locations to simplify sorting of the address steps into their proper order. Symbolic addresses need be assigned only to the location of the first instruction in a sequence and to those instructions whose locations are referred to in the address halves of other instructions. Further, not all addresses within the program need be indicated in symbolic form. If an actual address is known for a particular operand or operation, it can be written into the program and is usually indicated by enclosing the actual address in parentheses. Similarly, those instruc-

tions whose address halves should contain numerical constants can be written with the absolute value (indicated as absolute by their inclusion in parentheses) in octonary form.

4.2.3.3 Assembly Programs

Once a program is written in symbolic form, the task of translating it into absolute form can be performed under control of a utility program called an assembly program. The assembly program accepts a symbolic program presented on instruction cards. An absolute location is assigned to each instruction in the same order in which the instructions of the program are presented. The assembly program then makes another pass through the program being assembled to complete the assignment of absolute equivalents for all symbolic addresses and to translate all information in the program itself into binary form. (Since symbolic addresses are not translated, they can be written decimally without introducing any difficulties. The assembly program simply provides a one-to-one correspondence of symbolic address to absolute address.) The assembly program will also punch a deck of binary cards from which the assembled program may be inserted and executed, together with a printout of the program, listing all comments, symbolic locations, constants, assigned storage locations and their contents in octonary form, and the initial and final drum storage locations, if any.

4.2.4 Operational Program

When speaking of programs, the operational type is the one most usually discussed. It is the overall program for the computer and is written to suit a particular need, whether it be inventory, business accounting, or air defense. There are many things to be considered when writing an operational program, and its suitability is determined to a large extent by the programmer designing it. Good operational programs use the minimum amount of storage space to obtain the best result in the shortest possible time. However, it is almost impossible to achieve the optimum values for these three things in one program, and something must be sacrificed. Operational programs may or may not contain subroutines, but as a rule they do. The program examples given in Chapter 3 could be called operational programs, but it must be remembered they are extremely short when compared to actual useful operational programs.

4.2.5 Maintenance Programs

4.2.5.1 Reliability Programs

Reliability programs are used to check the operation of specific portions of the computer equipment.

Performance of these checks enables errors caused by circuit failure to be detected rapidly. Included in the error detection performed by reliability programs is the discovery of failures that may occur only under particular operating conditions, such as failures that appear at specific repetition rates and for certain combinations of bits. In order to discover these and other types of errors, reliability programs check logical operation, paths of information flow, timing, ability of equipment to function in all states, execution of instructions, etc. Because of their ability to perform these varied checks, equipment reliability programs can be used for both preventive and corrective maintenance.

4.2.5.2 Diagnostic Programs

Diagnostic programs are corrective maintenance programs which localize malfunctions to small area of the computer. They isolate errors to a specific pluggable unit or a small number of pluggable units. In general, diagnostic programs are designed to isolate known failures, unlike reliability programs which are designed to discover these failures. However, there is no clearly defined distinction between the two maintenance program types. Reliability programs can provide indications of the nature and location of the failure and may actually be used as diagnostic programs. By the same token, diagnostic programs may indicate that a given portion of the equipment is operating reliably. The characteristics of reliability and diagnostic programs fall somewhere between the two extremes of overall check without failure localization and diagnosis with failure isolation.

CHAPTER 5
CAPABILITIES AND LIMITATIONS OF COMPUTER

This chapter discusses some of the capabilities and limitations of digital computers, explaining how their performance is affected by different characteristics of their construction and organization, and by the procedures employed for operation and maintenance.

Different types of problems impose widely different requirements for computer performance. For example; a computer controlling a process in real time needs faster IO equipment and may need faster computing speeds than the ordinary scientific or business computer. A computer processing business data often needs a larger external memory capacity than a scientific computer. A computer used for real-time control of military operations needs a high degree of reliability. And, finally, there are innumerable problems for which present day computers are unsuitable or where they cannot handle the problem at all.

5.1 PROBLEM-SOLVING CAPABILITY

What sort of problems can a computer solve, and what is it that determines whether one computer is better than another?

If a digital computer is capable of executing only a few instructions, then the only things that limit the problem solving ability of the computer are the size of its memory and the allowable time for solution. Suppose that a sequentially programmed digital computer of the internally stored program type is capable of performing only the operations, *Add, Branch if Minus, Clear and Add, Halt, Multiply, Read, Shift, Store, Subtract, and Write.* It has been proven by the English mathematician Turing that such a computer can be programmed to solve any problem that any other computer can solve if its memory capacity is large enough and if enough time is available.

In practical applications, however, other characteristics in addition to memory capacity affect the problem solving ability. Practical limitations on the problem solving capability of a digital computer include: (1) the speed of the computer, (2) its ease of programming and operation, (3) its reliability, (4) the nature of the IO equipment, and (5) the ability of the programmers. For scientific problem solving, the most important of these limitations are usually memory capacity, speed, ease of programming, and ability of the programmers. For business data processing, the most important limitations are usually memory capacity and speed. And for most real time process control applications, the important limitations are speed, reliability, ease of operation, and the type of IO equipment.

5.2 SPEED

The development of computers has been a continuous quest for faster methods of computing. In the last decade comparatively few new mathematical techniques have been discovered. Yet, a great number of problems formerly thought to be insoluble have been solved through the use of digital computer techniques. These problems have been solved because for the first time the speed of the computer has made old methods of solution practical. Before the advent of computers, the methods of solving the equations used in the design of aircraft wings were known; however, such equations require so much calculation to solve that it used to be impractical to do so. The wings were, therefore, designed by crude mathematical approximation and experiments. The speed of the digital computer is now great enough to make mathematical solutions practical.

The solution of some problems requires even greater speed than can be provided by present day computers. For instance, there are problems that occur in physics, which it is estimated would require six months to solve with the fastest of present day computers. It appears, therefore, that the speed of a computer is its major capability and an increase in speed the area in which the greatest improvement can be expected on future computers.

5.3 EASE OF PROGRAMMING AND OPERATION

The ease with which a computer can be programmed is one of the most important characteristics determining its practical usefulness. In a typical application, a group of men may spend two months writing out a program, two weeks checking and correcting it, and then the computer will take an hour to perform the computations. It can be seen, then, that a computer which is easy to program will generally be much more useful than one which is difficult to program, because a problem which cannot be programmed economically cannot be solved economically on a computer.

Another characteristic which helps to make a computer more useful is the ease of operation by personnel when, in some applications, human intervention is necessary. Often, in the SAGE computers for example, this

intervention must be accomplished rapidly if it is to be useful, and the ease of entering information determines whether the particular operation required can be performed or not. For instance, a Display System was used in the sample computer of Part 4 because it is the only method yet found to present a mass of data to the operator in an understandable form and at high speed.

5.4 RELIABILITY

Reliability is a special problem in digital computers because of the complexity of their design and operation. A large digital computer may have thousands of vacuum tubes, and a breakdown in any of them may cause an error in any single step which may throw off a scientific computation involving 10 million steps.

One way to improve reliability is to use selected components in very conservatively designed circuits with controlled temperatures and carefully regulated supply voltages. Tube types are chosen that have low failure rates, and then the particular tubes to be used are individually tested. Capacitors are operated at far below the working voltages recommended by their manufacturers. Forced air cooling is provided, and usually a large computer is operated in an air conditioned enclosure.

In addition, to further reduce component failures, a preventive maintenance routine is employed at regularly scheduled intervals. One procedure in preventive maintenance is marginal checking which involves the varying of operating voltages on a few circuits at a time, in order that any circuits close to failure level will show their defects at this time. Another procedure in preventive maintenance is to have the computer check itself by means of reliability and diagnostic programs (Ch 4).

No matter how much is done in the way of component selection, careful design, and preventive maintenance; errors are still likely to occur from time to time. Therefore, many computers are designed with special provisions for automatic detection of errors. One method often employed is to include in a computer word one or two extra bits for error checking. For example, in the 33-bit words used in the AN/FSQ-7, 32 bits are for data or instructions, and the 33rd bit is for parity check. The total number of 1 bits in an AN/FSQ-7 control computer word is always supposed to be an odd number. Therefore if the 32 information bits have an odd number of 1's, a 0 is used as the 33rd, or parity bit; but if the number is even, then, at the time the word is originated, the computer writes a 1 in the parity position. Several parts of the Central Computer System contain parity checking circuits which count the number of 1 bits in a word and give an alarm if a wrong number is found.

A parity check is a useful safeguard because most errors due to malfunction of a component or circuit will affect only a single bit position, and errors are so infrequent that it is unlikely that two will occur at the same time. A single error will change the total number of 1 bits from an odd to an even number. This system will not detect an even number of bit-change errors.

Even at best, when full use is made of all the resources that have been described for increasing reliability, no large computer has yet been developed which can operate without error more than about 95 to 99% of the time. Therefore, in a system such as SAGE where it may be disastrous for a computer to be out of operation for even a few minutes, it is necessary to increase reliability by duplexing many parts of the computer, so that if one part is out of order or is shut down for preventive maintenance, the duplicate part can take over.

5.5 CONCLUSION

Digital computers do many types of work exceedingly well, but there are many other types of problems at which they are quite slow and incompetent in comparison with man. For almost any type of numerical calculation, a high speed computer is roughly ten thousand to a hundred thousand times faster and far more accurate than the average experienced human calculator equipped with a desk calculating machine. On the other hand, a competent human translator is incomparably superior to the best digital computer in translating from one language to another.

Computers are better than human beings at numerical calculations because of their greater speed and accuracy, and because most numerical calculations do not require tremendously complex programs or vast amounts of data storage. On the other hand, an expert translator or chess player draws upon years of experience involving tens or hundreds of millions of bits of memory. In fact, any man during a few minutes of reasoning, will whether he realizes it or not, ordinarily make use of much of what he has learned throughout his life. In other words, a man can make use of a tremendous memory capacity to solve very complex problems.

But man has one other advantage which so far has not been built into a computer. He has the ability to think creatively. The average man can look at a complex situation and arrive at a conclusion which may never have been thought of before. A computer on the other hand can only do what it is told to do; i.e., it can only come to conclusions which are built into a specific written program.

The way in which computers and operating personnel are employed in the SAGE System makes excellent use of the different abilities of man and machines. Thus the routine mathematical calculations are performed swiftly and accurately by the computers, while

the difficult and critical decisions are made by men. If at some time in the future, an AN/FSQ-7 computer displays a group of unidentified planes approaching the United States in a suspicious way, then a man must within a few seconds decide whether to attack immediately or wait for additional information. Within those few seconds he must make use of what he knows about planes and attack strategy, the effects of nuclear weapons, the effectiveness of the air defense system, possible intentions of likely enemies, and so forth, involving thousands of times as much information as is stored in the AN/FSQ-7's memory.

INDEX

INDEX (cont'd)

INDEX (cont'd)

INDEX (cont'd)

INDEX (cont'd)

INDEX (cont'd)

INDEX (cont'd)

INDEX (cont'd)

INDEX (cont'd)

INDEX (cont'd)

INDEX (cont'd)

INDEX (cont'd)

INDEX (cont'd)

INDEX (cont'd)

INDEX (cont'd)

INDEX (cont'd)

INDEX (cont'd)

INDEX (cont'd)

INDEX (cont'd)

INDEX (cont'd)

INDEX (cont'd)

INDEX (cont'd)

INDEX (cont'd)

INDEX (cont'd)