3–112–0

THEORY OF PROGRAMMING

FOR

AN/FSQ-7
COMBAT DIRECTION CENTRAL
AND
AN/FSQ-8
COMBAT CONTROL CENTRAL

1 APRIL 1959

MILITARY PRODUCTS DIVISION
INTERNATIONAL BUSINESS MACHINES CORPORATION
KINGSTON, NEW YORK

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 DESTROY SUPERSEDED PAGES.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 changed, 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)

CONTENTS (cont'd)

CONTENTS (cont'd)

LIST OF ILLUSTRATIONS

LIST OF ILLUSTRATIONS (cont'd)

LIST OF ILLUSTRATIONS (cont'd)

LIST OF TABLES

LIST OF TABLES (cont'd)

LIST OF TABLES (cont'd)

PART 1
INTRODUCTION

CHAPTER 1
PURPOSE AND SCOPE

1.1 PURPOSE

This manual explains the program instructions used in the AN/FSQ-7 and AN/FSQ-8. Although the manual is designed to be used primarily in the training of IBM field engineers, it also serves as a reference manual for IBM field engineers and other personnel assigned to the various air defense sites.

1.2 SCOPE

The manual was written to serve as an introduction to programming digital computers in general and the AN/FSQ-7 and AN/FSQ-8 in particular. Some chapters apply to all the courses of instruction conducted by Field Engineering Education; others pertain only to one of the courses. A detailed description of each part of the manual follows.

Part 1 covers the history of computing devices from the abacus to the modern digital computer. A description of the binary number system, which is used in many digital machines (including the AN/FSQ-7 and -8), is presented and examples of arithmetic operations using this system are given. This part also outlines the general organization and operation of the SAGE System employed in air defense. A brief introduction to the AN/FSQ-7 and AN/FSQ-8, the computing elements of the SAGE System, is presented.

Part 2 contains the information necessary for an understanding of the programmed operation of the AN/FSQ-7 and AN/FSQ-8. The various systems which make up the AN/FSQ-7 and AN/FSQ-8 are described and their overall relationship is discussed. Some of the basic instructions of these two machines are described, and program examples of their use are given. In addition, techniques, such as indexing, which are useful in programming operations, are introduced.

The third part of the manual presents a detailed description of the instructions which are applicable to the Central Computer System of the AN/FSQ-7 and AN/FSQ-8. Wherever possible, examples of the application of these instructions are given. In addition, Part 3 explains some of the programming options which are available for the AN/FSQ-7 as a result of the installation of an expanded memory unit.

Part 4 describes the programming of equipment and systems which are external to the Central Computer System. The various instructions which are used to program these external systems are given, as are program examples. Each chapter within this part is concerned with the programmed operation of one of the logical groups of external equipment.

Advanced programming methods, such as the content and operation of utility systems and the scaling of fixed point numbers, are contained in Part 5. This part enables the reader to understand all of the operations necessary, in addition to the coding of a program to prepare it for entry into the AN/FSQ-7 and AN/FSQ-8.

Appendix A lists those codes which are referred to as illegal instructions within the AN/FSQ-7 and AN/FSQ-8. The effect that the execution of these illegal instructions has on the Central Computer System is described, and the use of some of them is presented.

CHAPTER 2
DIGITAL COMPUTERS

2.1 HISTORY OF DIGITAL COMPUTERS

The word "computer" comes from the latin verb "computare" which means to reckon or think. Thus, a digital computer is a machine that reckons (or calculates) with digits. The following text contains a brief history of computing devices which brought about the modern digital machines.

Before the advent of large-scale digital computers, man had to rely on manual methods and slow mechanical calculators to perform arithmetic operations. No doubt the first devices used by man as an aid in computation were fingers, sticks, stones, and similar objects. These items served only as reminders or indicators of a particular quantity and could not themselves do any work; the actual computation was carried out in the mind of the individual. The abacus, originated sometime before 1200 A.D., evolved from the use of pebbles as counters, and has been highly touted as a computer; however, it is really nothing more than an indicating device.

After the abacus and the soroban (a refinement of the abacus) came into use, little progress was made in the field of computing devices for several years. Then, in 1642, Pascal, the French mathematician, invented what was perhaps the first actual accounting machine. This machine was used to figure currency in a customs house. It was basically a hand-operated, gear-driven counter, with addition performed by turning a wheel a distance equal to the currency to be added. Pascal's machine did not attract much attention, possibly because he was only 19 years old at the time. However it was noteworthy because it was the first instrument with provision for an automatic carry into the next higher order column when the sum of a column exceeded 9. In 1673, a German mathematician named Leibnitz developed a similar machine, which could multiply as well as add. Unfortunately, it did not work well enough to be of much value in computation.

After Leibnitz's machine, there was a period of inactivity until 1801, when another Frenchman named Joseph Jacquard came upon the idea of punched cards. Jacquard used a chain of perforated cards to control weaving of figured fabrics on a loom. This mechanism functioned quite successfully, and the Jacquard loom proved to be the basis for some remarkable developments, as we shall see later.

During the 19th century, several developments took place which furthered the use of automatic accounting machines. First, an American named Thomas invented a desk calculator in 1820 which utilized the same principles as Leibnitz's machinery and was also reliable. Improved models of Thomas' calculator were used in business and industry for the next 100 years. Two years later, Charles Babbage, a brilliant but highly unpopular English mathematician started work on what he called a "difference engine." This device was to be used to calculate mathematical tables to 20 places. Babbage's design was fundamentally sound, but production techniques at that time were not, and after many unsuccessful attempts to construct a working machine, the British government stopped supplying funds for this project. Despite the furor caused by his failure to produce the difference engine, Babbage was elected to a chair in mathematics at an English University. He distinguished himself in this position by refusing to deliver even one lecture in the 11 years he held the position. Then in 1833, the remarkable Mr. Babbage again received a grant from the British government to work on another computing machine, called the Babbage Analytical Engine. Elaborate plans were prepared for the construction of this computer, which differed largely from Babbage's first machine because it was controlled by punched cards and was capable of making logical decisions. Parts of it were completed and made to work, but the project failed as a whole because Babbage's design was still too advanced for the engineering techniques then in existence. Nevertheless, Babbage continued to work on his computer for another 10 years or so, until government funds were exhausted. Detailed descriptions of his machine were preserved, as were substantial parts of it, but Babbage's idea was soon forgotten.

In 1886, a very important milestone in computer history was reached when Dr. Herman Hollerith invented a machine using punched cards. Hollerith was head of the U. S. Bureau of Census at the time, and he discovered that the 1880 census was not yet completed, due largely to the fact that all calculations were being made by hand. He set to work to find a way by which all recording, tabulating, and analyzing of facts could be done by machine. His solution, which was based upon Jacquard's idea of punched cards, was to record the facts of any situation by punching holes in a definite

code in a piece of paper. Hollerith had originally planned to use strips of paper, but he found it necessary to rearrange the information, so he cut the strips into a standard size and shape and thus had a card for each situation. Once the card was developed, Dr. Hollerith developed a sorting device, using these cards, and opened up a whole new field of computing aids.

As business and industry grew during the first part of the 20th century, the demand for accounting machines grew steadily. In 1914, a mechanical key punch, a gang-punch, a vertical sorter, and a tabulator were available to meet the accounting needs of the nation. As time went on, more and more equipment was introduced which eased the task of calculations. However, all of this equipment was electromechanical in nature, and each machine could perform only one or two basic operations. What was needed was a machine that could perform a multitude of tasks at a high rate of speed.

The man who set about to design such a general-purpose machine was Dr. H. H. Aiken of Harvard University. He directed a project which started in 1939 and was climaxed by the construction of the automatic Sequence Controlled Computer in 1944. This computer, commonly referred to as the Mark I, was built by IBM for Harvard from components already in use in IBM's electromechanical business machines. It is believed that the Mark I is an outgrowth of Babbage's analytical engine, since his work was reviewed in great detail before the actual construction began. Although the Mark I was electromechanical in nature, and therefore was still quite slow, it marked the appearance of the first of a long line of large digital computers.

The first electronic computer was the Electronic Numerical Integrator and Calculator, or ENIAC, built in 1946 by Dr. J. W. Mauchley and Mr. J. P. Eckert of the Moore School of Engineering of the University of Pennsylvania. The ENIAC utilized 18,000 vacuum tubes as storage elements instead of the relays and switches used in the Mark I. The fact that vacuum tubes were used at all represented a considerable venture in computing techniques, since the performance of tubes at that time was not very reliable. However, ENIAC proved to be a highly successful digital computer, and is still in operation today, although it has been obsoleted for some time. As an example of the improvement in arithmetic speed between Mark I and ENIAC, let us consider the addition of two numbers. Mark I required 300 ms to perform this task, whereas ENIAC could do the same thing in two-tenths of a millisecond. or, roughly, 1,500 times faster. From this comparison, it is possible to see the tremendous boost given the computer field with the advent of vacuum tubes and electronic circuitry.

After ENIAC, the next big computer of significance was the Selective Sequence Electronic Calculator, or SSEC, built by IBM in 1948. This machine, installed at IBM World Headquarters, proved beyond a doubt that large-scale digital computers had commercial applications. The government, which was doing research in the atomic energy field, utilized the SSEC as an aid in solving some of its large problems. Other customers soon realized that the SSEC could solve problems which were never before attempted. The success of this computer prompted many manufacturers of electronic equipment to develop digital computers for commercial use.

About the same time, a group at Cambridge University in England built the first stored-program computer. The man in charge of the project was M. V. Wilkes, and he called the computer EDSAC. EDSAC was modest in size and capability, but Wilkes and his associates made significant contributions to the computer field by refining programming techniques and procedures. They were the first to make extensive use of program subroutines with an assembly system for making new programs.

In 1946, the designers of ENIAC, Dr. Mauchly and Mr. Eckert, resigned from the University of Pennsylvania and set up their own firm, called the Electronics Control Company. They began to develop an electronic computer which could handle alphabetic as well as numeric data, and which would be more versatile than any machine then in existence. This machine was called the Universal Automatic Computer, or UNIVAC, and was first produced in 1951 by the Remington-Rand corporation, which had made the Mauchly-Eckert firm one of their divisions. UNIVAC proved to be the nucleus for a new computer field; namely, the large-scale, general-purpose digital machine.

In addition to the UNIVAC, several other computers were developed at this time, most of them by the government and leading universities. For example, the MANIAC, built by Princeton; the SEAC, built by the Bureau of Standards; and the WHIRLWIND, built by the Massachusetts Institute of Technology. All of these machines aided the advancement of computers and computing techniques. At the same time, they confirmed the belief, started with the introduction of the SSEC, that large digital machines could prove a powerful tool to both science and industry.

The vast experience gained in the punched-card field enabled IBM to develop an entire "family" of large digital computers, known as the 700 series. First came the 701, a scientific computer with a memory composed of electrostatic storage tubes. This computer was extremely fast and reliable, and was used extensively

in the aircraft industry to perform design calculations. In 1955, the first 702 was installed at the Monsanto Chemical Company in St. Louis. This model was designed to handle mostly commercial data processing such as billing, stock inventory, payrolls, etc. These two machines were capable of handling any type of problem that could be reduced to some combination of the basic arithmetic processes. However, improvements were still being made in the computer field, mainly in the type of memory to be utilized with the computer. IBM introduced two computers in 1955 which contained magnetic cores as the high-speed memory device. The 705 was designed to replace the 702, and the 704 was the successor to the 701. Use of the new memory and other improvements enabled both machines to compute much faster than their predecessors. As an example, the 704 can select two numbers from core memory, add them together, and place the sum in a temporary storage register in 24 μsec. Newest of the 700 series is the 709, also a scientific computer, which has the added capability of handling up to six input and output mediums at one time. Previously, computers had been limited in their speed of calculation because only one input or output unit at a time could be used to feed in or receive information, a disadvantage overcome by the 709.

Of particular interest are the AN/FSQ-7 and AN/FSQ-8 digital computers, also manufactured by IBM. These machines are the world's largest digital computers and form the center of a huge air defense network. The remainder of this manual is devoted primarily to a study of the techniques involved in programming these two computers.

This history of computers and computing devices is, of necessity, incomplete. Only those events which have been milestones in the development of today's digital computers have been included. Future developments in the field may radically change the operation of digital computers. These developments will probably occur in the use of bigger and faster memories, reduction in physical size of computers, and increased flexibility in programming. However, the basic concepts of Pascal, Babbage, and others who have contributed to this art will remain the same; that is, using machines to ease the task of computing and calculating.

2.2 DIGITAL COMPUTER OPERATION

As was explained previously, a digital computer works with digits or numbers. To illustrate the method by which a digital computer handles a quantity, a comparison will be made with the other type of computer presently in use; namely, the analog machine. In contrast to the digital computer, a quantity in an analog computer is represented by a direct relation, or analogy, to some machine function such as voltage variations. As an example, suppose we wish to represent a quantity of 12 units in both types of machines. Figure 1–1, part A, shows a possible representation of 12 in a digital computer; part B shows the same quantity as it might be represented in an analog machine. Now, if we multiply 12 by a factor of 2, the resulting product (24) is shown in part C of figure 1–1 for the digital computer and in part D for the analog computer. In this example, pulses were used to represent digits, but this is only one of the many techniques available for digital representation.

In order for a digital computer to operate, certain elements are required for the proper handling and manipulation of data, just as a man needs certain tools to perform arithmetic tasks. This comparison is easily supported by describing the elements of the computer that correspond to a man working at a desk. Assume that the man is a clerk working in a payroll office, computing the net pay of various individuals. The "in" box on his desk contains the pay rates of the personnel involved, plus miscellaneous data, such as the initiation of bond deductions, etc. A digital computer has an input element which is capable of accepting various types of data and of presenting it to the computing portion of the equipment. The clerk has several tables he refers to, such as tax deduction tables, standard weekly deductions, etc.; in addition, he has a pad and paper on which he notes the deductions applicable to each employee. In a digital computer, the memory element would serve as the temporary storage device for all

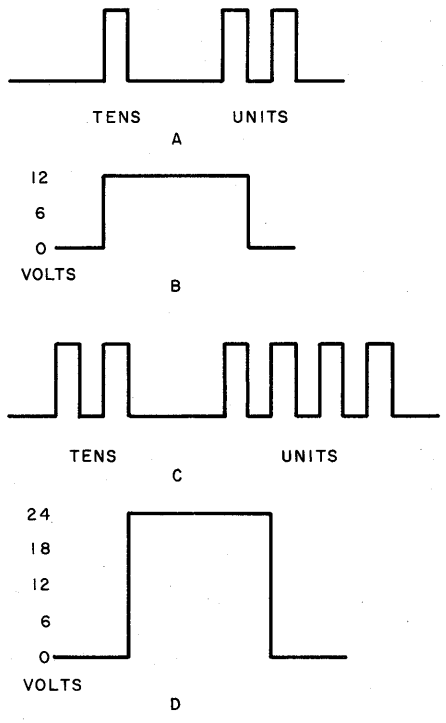


Figure 1–1. Representation of Numbers in Digital and Analog Computers

these facts. The actual computation of an individual salary is done in the payroll clerk's head, or, perhaps with a desk calculator; in either case, this function is the same as that performed by the arithmetic element of a digital machine. Once the net pay of each person has been calculated, the clerk fills out a standard form which contains the employee's name and the amount due. He then places all these forms in the "out" box on his desk, thus completing his job. The output element of a digital computer accepts the results of computation by the arithmetic element and presents the results in a form recognizable by the user. Of course, all the actions of the payroll clerk are controlled and co-ordinated by his nervous system. The control element co-ordinates the actions of a digital computer and is connected to all the other elements. From this discussion, we can see that a digital computer is essentially composed of the following elements:

a. Input
b. Output
c. Memory
d. Arithmetic
e. Control

Each of these elements is discussed in detail below, and reference is made to the general organization and operation of each.

The input element is capable of accepting data in a variety of forms and converting it to a standard format which is used by the computing portion of the equipment. The type of inputs which constitute the input element for different machines vary a great deal; therefore, it is not possible to say that any one combination of units makes an input element. However, some of the more common types of input units are punched card readers, magnetic tapes, paper tapes, typewriters, and telephone lines which transmit data from remote locations. The input element provides 1-way communication between the computing elements and external sources: information is received through the input element, but no information or data is returned.

As might be expected, the output element is almost the opposite of the input element. The results of computer operation are fed through the element to designated locations, but no information is given back to the computing elements. Some examples of output units that might constitute a typical output element are line printers, card punches, magnetic tapes, and visual indicators. As with the input element, the output element makeup is flexible and varies widely from one computer to another.

The memory element comprises a large number of storage locations in which information can be stored until it is needed by one of the other elements. As a rule, memory elements have absolute addresses assigned to each location, and such an address is specified when information is needed. For instance, a typical computer instruction might be to "add the quantity which is stored in location 1000." The instruction which stated that address 1000 contained the desired operand is also stored in the memory element; this makes the memory element a "shared" device. Sometimes the instructions for performing a given operation are always contained in the same physical portion of the memory element; sometimes they may be located any place within the memory. Memory elements may be constructed of several types of mechanisms; however, some of these are rarely used because of new developments which have taken place. At present, magnetic cores are the most popular device, primarily because of their high speed and stability. Other devices which have been used include acoustic delay lines, electrostatic tubes, and magnetic drums. The AN/FSQ-7 and AN/FSQ-8 both utilize magnetic cores in the memory element.

The arithmetic element of a digital computer is basically a device which performs addition only; the other arithmetic functions are simply variations of the addition function. For example, multiplication in an arithmetic element is simply a repetitive addition. Arithmetic elements are, for the most part, made up of various vacuum-tube circuits and switches. The basic circuit in an arithmetic element is the flip-flop, which is used to store or transfer the results of computation. There are about 100 such flip-flop (FF) circuits in the arithmetic element of the AN/FSQ-7 and AN/FSQ-8.

The function of the control element is to generate the proper signals at the proper time which will cause a desired action to take place within the computer. Several decoders are used in the control elements, along with flip-flops and other switching devices. The control element keeps track of what instruction is to be decoded and performs part of the decoding. In addition, this element provides timing pulses which synchronize all elements of the computer.

Figure 1–2 shows the five elements we have been discussing arranged as a typical computer configuration. Information, which includes computer instructions as well as data, enters the computer through the input element, where it may be converted to a common form and placed in a buffer storage device. The memory element accepts this information at specific intervals and places it in the proper storage location. Notice that there are two paths leading out of the memory element; one to the control element and the other to the arithmetic element. Earlier, we spoke of the memory element as a shared device, and this is true because it serves as the memory for two other elements. The instructions are transferred from memory to the control element,

where they are decoded, and certain commands are set up by the control element. One of these commands is to go into memory again and transfer out the designated operand to the arithmetic element. Following the calculation in the arithmetic element, the results of a desired operation are usually programmed to be returned to memory and then transferred to the output element during an alloted time interval.

From this, we can see that a digital computer follows a basic flow pattern during its operation. The time when the instructions are decoded is known as program time (PT); the time when the calculation is performed is known as operate time (OT). Most of the operations that can be carried out by the AN/FSQ-7 and AN/FSQ-8 follow this PT-OT pattern, usually referred to as cycle configuration. The cycle configuration varies slightly among the various operations because more time is sometimes required for one operation than for another. In addition, the AN/FSQ-7 and AN/FSQ-8 are capable of performing a few operations which do not require any OT time; however, these are of a specialized nature and are explained in detail in the appropriate parts of the manual.

In figure 1–2, you will notice that the memory, control, and arithmetic elements are enclosed in a box, and referred to as the central computer. This has been done to clarify the use of the terms "computer" and "central computer." A computer comprises all the elements necessary for proper operations, including the input and output elements. The central computer refers only to that portion of the equipment which does the actual calculations. This distinction is necessary because the two terms are used frequently throughout the manual, and it is important to know whether all or only a portion of the equipment is being referred to at any one time.

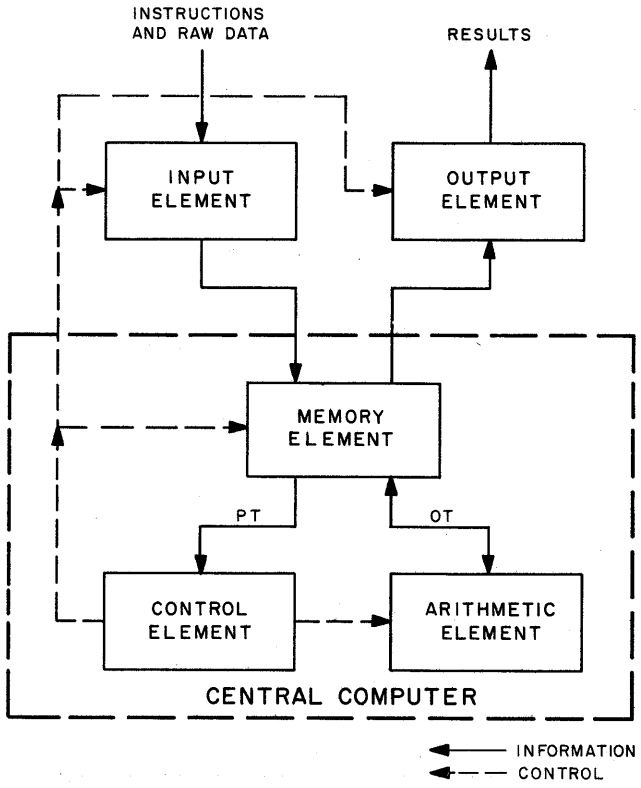


Figure 1–2. Digital Computer, Block Diagram

CHAPTER 3
NUMBER SYSTEMS

3.1 GENERAL DESCRIPTION

When we speak of a "number," it usually brings to mind some combination of the decimal digits 0 through 9. This number may represent several different things, although in each case only the digits mentioned above are used. For instance, we might think of a 10-day course of instruction. In this case, the number 10 is used to represent a quantity of items namely, days. Therefore, one use of a number is as a quantity designator, or a number that specifies an amount. However, numbers do not always indicate an amount or a quantity. Consider a post office mail box. The number on the mail box does not indicate how many mail boxes are located in the post office nor does it tell how many letters are in the mail box. What the number does tell is the location of the mail box with respect to other mail boxes, or the address of the box. All incoming mail with that particular box number on it will be placed in that box. In this case, the number has told us only the location of the mail box, and nothing else. There is one more general use to which numbers are put; i.e., as an identification code. For instance, a telephone number does not indicate quantity or a location but merely tells the automatic dialing equipment what phone to ring when someone dials a particular number. In the above examples, the decimal system has been used simply because it is the one we are most familiar with. However, the fact that numbers can act as quantity designators, addresses, or identification codes holds true for all number systems.

All number systems have many properties in common, and these properties are explained, using the decimal systems as an example. The use of the decimal system will permit us to see exactly what effect each property has without giving us trouble with the mathematics involved, since we are all used to doing arithmetic in this system. Later on, each of the properties discussed will be used with the two other number systems covered in this chapter, and we will see that the same rules still apply.

3.2 DECIMAL NUMBER SYSTEM

The decimal number system derives its name from the fact that it is composed of 10 different symbols. These symbols are, of course, 0, 1, 2, 3, 4, 5, 6, 7, 8, and 9. Any one of these symbols may appear in any position of a number. When we count in the decimal system, adding a 1 to any number 0 through 8 gives the next higher number (e.g., $7 + 1 = 8$, $3 + 1 = 4$, etc.). However, adding a 1 to 9 does not give a higher number but, instead, causes a return to the lowest symbol and causes a 1 to be added to the next position to the left ($9 + 1 = 0$ plus a carry of 1 to the left, or 10). From this, we can see that 10 steps are required to use all the symbols in the decimal system and cause a return to the lowest order symbol. Also, there are 10 different symbols in the decimal system. These two facts can be stated by saying that the decimal number system has a radix or base of 10. The radix of any number system thus indicates the number of symbols in that system and the number of steps in the counting cycle.

Numbers have different values in the decimal number system according to the symbol being used, and the position of that symbol. Of course the number 5 has a higher magnitude than 4, but the number 23 is larger than 9, even though the digits used are not. Adding a digit to the left of a number increases its value by the power of the radix times the digit added. Using 3 as an example, adding a 2 to the left and multiplying by 10, we increase the value of 3 by 20, or 2 x 10. The fact that the value of a number is determined by the position of its digits is known as positional notation of magnitude. In the case of the number 23, this is actually an abbreviation of the positions and their magnitude; namely, $(2 \times 10^1) + (3 \times 10^0)$. Similarly, the number 4735 actually represents the sum of the terms $(4 \times 10^3) + (7 \times 10^2) + (3 \times 10^1) + (5 \times 10^0)$. It is important to keep in mind that the power of the radix increases by 1 for each position to the left of the preceding digit.

When dealing with the decimal system, it is not necessary to show numbers in the above form, since it is generally understood by everyone what each digit position indicates; namely, some power of the radix. The same thing holds true for the other number systems we are going to study; the value of the number is determined by the position of its digits.

There is one more property of the decimal system that we wish to examine; that is the concept of modulus. Modulus refers to the number of numbers that can be contained in a restricted number of digit positions. If there were no limitations on the size of numbers, there would be no need to discuss the modulus. However, most devices used in counting, computing, etc., have definite limits as to the size of number that can be manipulated. A mileage indicator such as that used in an

automobile is a good example of a limited position device. If a particular mileage indicator had five digit positions it would be capable of showing 100,000 different numbers (0 through 99,999). Such a counter would have a modulus of 100,000. The importance of modulus becomes apparent when we deal with numbers that exceed the capacity of a particular device. For example, a distance of 105,000 miles on the indicator mentioned above would appear the same as a distance of 5,000 miles. Once the modulus is exceeded, the counter resets itself to 0, and the count starts over again. Later on, it will be shown how the AN/FSQ-7 and AN/FSQ-8 are capable of handling a number that exceeds the modulus of their storage and arithmetic registers.

3.3 BINARY NUMBER SYSTEM

Several different number systems are currently in use with digital computers. Each of these systems has its particular advantages and disadvantages, and no attempt will be made to compare one system against another. However, one number system that is becoming increasingly popular with manufacturers of digital machines is the binary system. As indicated by the prefix "bi," this system utilizes only two symbols, 0 and 1, in its counting cycle. Therefore, the binary number system has a radix of 2, and all the numbers in this system are combinations of various powers of 2. This number system lends itself very well to electromechanical and electronic circuits, since the majority of these circuits are bistable devices. Table 1–1 lists the various devices used in digital computers, and shows how they are employed to represent a binary digit, in the AN/FSQ-7 and AN/FSQ-8 computers.

An illustration of a binary number in a computer can be given by using an electron tube circuit, since this is one of the most common binary devices used within the AN/FSQ-7 and AN/FSQ-8. Keep in mind that if a tube is conducting, it represents a 1; if it is cut off, it represents a 0. If we have three tubes side by side in a circuit, and they are all conducting simultaneously, the output of that circuit is 111. This is read as one-one-one, not one hundred and eleven, since we are no longer dealing with the decimal system. Remembering that the radix is 2, and that positional notation of magnitude is used, the output 111 is actually a representation of the series $(\underline{1} \times 2^2) + (\underline{1} \times 2^1) + (\underline{1} \times 2^0)$ or $(\underline{1} \times 4) + (\underline{1} \times 2) + (\underline{1} \times 1)$ or $4 + 2 + 1 = 7$. Thus, binary number 111 is equivalent to decimal 7. All of the numbers in the binary system are represented in the same manner, and it is very easy to find out the decimal equivalent of any binary number by using the same method. Table 1–2 gives some of the numbers in the binary system and their decimal equivalents.

To prove the validity of table 1–2, assume that it is desired to find the decimal equivalent of the binary number 1010. From inspection, it is possible to see that this number is an abbreviation for

$$(\underline{1} \times 2^3) + (\underline{0} \times 2^2) + (\underline{1} \times 2^1) + (\underline{0} \times 2^0) \text{ or}$$

$$(\underline{1} \times 8) + (\underline{0} \times 4) + (\underline{1} \times 2) + (\underline{0} \times 1) \text{ or}$$

$$8 + 0 + 2 + 0 = 10.$$

Referring back to table 1–2, we see that binary 1010 does indeed equal decimal 10. More examples of conversion from the binary system to the equivalent number in the decimal system follow.

TABLE 1–1. REPRESENTATION OF BINARY DIGITS

DEVICE	BINARY 1	BINARY 0
Electron tube	Tube conducting	Tube cutoff
Punched card	Hole present	Hole missing
Switch	Active (on)	Inactive (off)
Neon indicator	On	Off
Relay	Closed	Open
Pushbutton	Depressed	Not depressed
Plugboard	Pin plugged	Pin missing
Magnetic core	Flux in one direction	Flux in opposite direction
Magnetic drum	Flux in one direction	Flux in opposite direction
Magnetic tape	Change in flux	No change in flux

Binary $11110100111 = (\underline{1} \times 2^{10}) + (\underline{1} \times 2^{9}) + (\underline{1} \times 2^{8}) + (\underline{1} \times 2^{7}) + (\underline{0} \times 2^{6}) + (\underline{1} \times 2^{5}) + (\underline{0} \times 2^{4}) + (\underline{0} \times 2^{3}) + (\underline{1} \times 2^{2}) + (\underline{1} \times 2^{1}) + (\underline{1} \times 2^{0})$

This reduces to the following:

$(\underline{1} \times 1024) + (\underline{1} \times 512) + (\underline{1} \times 256) + (\underline{1} \times 128) + (\underline{0} \times 64) + (\underline{1} \times 32) + (\underline{0} \times 16) + (\underline{0} \times 8) + (\underline{1} \times 4) + (\underline{1} \times 2) + (\underline{1} \times 1)$ or $1024 + 512 + 256 + 128 + 32 + 4 + 2 + 1 = 1959$ decimal.

Binary $10000010 = (\underline{1} \times 2^{7}) + (\underline{0} \times 2^{6}) + (\underline{0} \times 2^{5}) + (\underline{0} \times 2^{4}) + (\underline{0} \times 2^{3}) + (\underline{0} \times 2^{2}) + (\underline{1} \times 2^{1}) + (\underline{0} \times 2^{0})$.

This series is equivalent to:

$(\underline{1} \times 128) + (\underline{0} \times 64) + (\underline{0} \times 32) + (\underline{0} \times 16) + (\underline{0} \times 8) + (\underline{0} \times 4) + (\underline{1} \times 2) + (\underline{0} \times 1)$ or 130 decimal.

From the above examples, we can see that it is useful to know the various powers of 2 when converting from binary to decimal. Table 1–3 lists the powers of 2 up to 2^{17}. The limit of 2^{17} was chosen because this is the highest power of 2 that is represented in the AN/FSQ-7 and AN/FSQ-8.

Conversion from the decimal system to the binary system is accomplished by reversing the process used for binary to decimal conversion. A binary number tells at a glance what powers of 2 are present or absent by the inclusion of a 1 or 0 in each digit position. However, a decimal number does not tell this, and some method must be used to determine what powers of 2 are present, and then place the corresponding 1's in the proper position to form the binary number. One method that can be used is to refer to table 1–3 and pick out what powers of 2 are present. The largest power of 2 that is less than the decimal number is located and then subtracted from the number. This process is repeated with the remainder, and all succeeding remainders, until the remainder is 0. In this manner, the powers of 2 that make up the number will have been found. Then a 1 is placed in the digit position, commonly called a "bit" position in binary numbers, for each power of 2 that is present, and 0's are placed in the other bit positions.

TABLE 1–2. BINARY AND DECIMAL EQUIVALENTS

BINARY	DECIMAL
000	0
001	1
010	2
011	3
100	4
101	5
110	6
111	7
1000	8
1001	9
1010	10

TABLE 1–3. POWERS OF TWO

POWER	EQUIVALENT
2^{0}	1
2^{1}	2
2^{2}	4
2^{3}	8
2^{4}	16
2^{5}	32
2^{6}	64
2^{7}	128
2^{8}	256
2^{9}	512
2^{10}	1024
2^{11}	2048
2^{12}	4096
2^{13}	8192
2^{14}	16384
2^{15}	32768
2^{16}	65536
2^{17}	131072

For example, to find out the binary number that is equivalent to decimal 100, we refer to table 1–3 and see that $2^6(64)$ is the largest power of 2 that is contained in decimal 100. Therefore:

$$\begin{array}{rr} & 100 \\ & -64\ (2^6) \\ \hline \text{partial remainder of} & 36 \\ & -32\ (2^5) \\ \hline \text{partial remainder of} & 4 \\ & -4\ (2^2) \\ \hline & 0 \end{array}$$

This shows that $100 = 2^6 + 2^5 + 2^2$. Substituting 1's for these powers and 0's for the absent powers, we have:

$$100 = (\underline{1} \times 2^6) + (\underline{1} \times 2^5) + (\underline{0} \times 2^4) + (\underline{0} \times 2^3) + (\underline{1} \times 2^2) + (\underline{0} \times 2^1) + (\underline{0} \times 2^0) \text{ or}$$

decimal 100 = 1100100 in binary.

Another system of decimal to binary conversion is to divide the decimal number by 2 and record the partial quotient and remainder. The partial quotient is again divided by 2, and so is each succeeding partial quotient until the quotient is 0. Each division will leave a remainder of 1 or 0; this remainder represents a bit position in the corresponding binary number. As an example, consider the number 100 again, and let us find out the binary equivalent by division. We proceed as follows:

	2 ⟌ 100		
1st partial quotient	2 ⟌ 50	0	1st remainder
2nd partial quotient	2 ⟌ 25	0	2nd remainder
3rd partial quotient	2 ⟌ 12	1	3rd remainder
4th partial quotient	2 ⟌ 6	0	4th remainder
5th partial quotient	2 ⟌ 3	0	5th remainder
6th partial quotient	2 ⟌ 1	1	6th remainder
final quotient	0	1	final remainder

Reading from bottom to top we have 1100100 in binary, which is the same result as that achieved by the process of finding the highest power of 2 from a table. A further example of this method is to find the binary equivalent of the number 7221.

TABLE 1–4. RULES OF BINARY ADDITION

ADDEND	+	AUGEND	=	SUM	WITH CARRY OF
0	+	0	=	0	0
1	+	0	=	1	0
0	+	1	=	1	0
1	+	1	=	0	1

2 ⟌ 7221		
2 ⟌ 3610	1	
2 ⟌ 1805	0	
2 ⟌ 902	1	
2 ⟌ 451	0	
2 ⟌ 225	1	
2 ⟌ 112	1	
2 ⟌ 56	0	
2 ⟌ 28	0	
2 ⟌ 14	0	
2 ⟌ 7	0	
2 ⟌ 3	1	
2 ⟌ 1	1	
0	1	

Thus, decimal 7221 = 1110000110101 in binary.

The important thing to remember when dealing with this method of conversion is that it is not merely some "short cut" that happens to work, but that it is based on powers of 2, just as the other method shown. Division by 2 will determine what powers of 2 are present in the number. The only actual difference between the two systems is that division by 2 will determine the lowest power of 2 first, rather than the highest power. Both methods have distinct advantages, but it is completely arbitrary which method is used; both will yield the same result.

Since binary numbers do resemble decimal numbers, or numbers in any system, for that matter, the usual practice is to place a subscript by the number being used. The subscript gives the radix of the number system, employed for a particular number. For example, 1010_2 is read as one-zero-one-zero, indicating binary representation. On the other hand, 1010_{10} is read as one thousand and ten, indicating decimal representation. Subscripts are used with all numbers in the remainder of this chapter.

Now that it is possible to convert both to and from the decimal system, we can start to learn the arithmetic associated with the binary system. The basic arithmetic function performed by the AN/FSQ-7 and AN/FSQ-8 computers is that of addition, so addition is considered first. Essentially, the addition process in binary is the same as that in the decimal system; adding a 1 to a digit (or bit in the binary system) gives the next higher order digit, until the highest digit is reached. A 1 added to the highest digit gives a return to the lowest order

digit, with a carry of 1 to the next position to the left. However, in the binary system, there are only two bits, 0 and 1; so adding to a bit will always give the opposite bit. This may be seen by examining the low order bit position of the binary numbers listed in table 1–2. Notice that there is an alternating pattern of 1's and 0's as the binary numbers go higher in magnitude. Because we have only two bits to deal with, the rules of binary addition are simple. These rules are shown in table 1–4.

This gives us the four basic rules from which any binary addition can be performed. However, in the above case, it was assumed that there was no carry of 1 into the bit positions being added. If there had been a carry of 1, the table could be expanded slightly to show the effect. The results of binary addition with a carry of 1 are shown in table 1–5.

As an example of binary addition, consider $101011010_2 + 0010111_2$. A carry of 1 from one bit position into the next is indicated by an arrow($\overleftarrow{c=1}$).

$$\begin{array}{cccccccccc} & & & & \overleftarrow{c=1} & \overleftarrow{c=1} & \overleftarrow{c=1} & \overleftarrow{c=1} & & \\ 1 & 0 & 1 & 0 & 1 & 1 & 0 & 1 & 0 & \\ + & & & 0 & 0 & 1 & 0 & 1 & 1 & 1 \\ \hline 1 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & 1 & \end{array}$$

$$\begin{array}{cccccccccc} & \overleftarrow{c=1} & \overleftarrow{c=1} & \overleftarrow{c=1} & \overleftarrow{c=1} & \overleftarrow{c=1} & & & & \\ & 1 & 0 & 1 & 0 & 1 & 1 & 0 & 0 & \\ + & 1 & 1 & 1 & 1 & 1 & 0 & 0 & 1 & \\ \hline 1 & 1 & 0 & 1 & 0 & 0 & 1 & 0 & 1 & \end{array}$$

To check on our binary addition, we can convert out of and into the decimal system, using one of the methods previously described. For example, convert and add $63_{10} + 12_{10}$.

2 | 63	
2 | 31	1
2 | 15	1
2 | 7	1
2 | 3	1
2 | 1	1
0	1

2 | 12	
2 | 6	0
2 | 3	0
2 | 1	1
0	1

Therefore:

$$\begin{array}{cccccccc} & \overleftarrow{c=1} & \overleftarrow{c=1} & \overleftarrow{c=1} & \overleftarrow{c=1} & & & \\ & 1 & 1 & 1 & 1 & 1 & 1 = 63_{10} \\ + & & & 1 & 1 & 0 & 0 = 12_{10} \\ \hline 1 & 0 & 0 & 1 & 0 & 1 & 1 & \end{array}$$

Converting 1001011_2 to decimal, we get $(\underline{1} \times 2^6) + (\underline{1} \times 2^3) + (\underline{1} \times 2^1) + (\underline{1} \times 2^0)$ or $64 + 8 + 2 + 1$ or 75_{10}.

Of course, 75_{10} is the correct sum of $63_{10} + 12_{10}$, so we are assured that our binary addition is correct.

It is not necessary to consider addition of more than two binary numbers, since the AN/FSQ-7 and AN/FSQ-8 can not add more than two numbers at one time.

In the foregoing examples, we assumed that it was always possible to make a carry-out from the highest order bit position into the next position, if necessary. However, since the AN/FSQ-7 and AN/FSQ-8 have arithmetic elements with a specific limit or modulus, it is not always possible to do this. When two numbers that are added together produce a sum that is greater than the capacity of the machine, a condition known as "overflow" exists. For ease of explanation, assume that we are dealing with an arithmetic element that has only six bit positions. Computer overflow would result from this addition:

$$\begin{array}{cccccc} \overleftarrow{c=1} & \overleftarrow{c=1} & \overleftarrow{c=1} & & & \\ 1 & 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 1 & 0 \\ \hline 0 & 0 & 0 & 0 & 1 & 0 \end{array}$$

The carry of 1 from bit position 6 is added back into the low order bit position, giving us a result that is incorrect numerically. Also, an alarm is generated, informing operating personnel of the AN/FSQ-7 and AN/FSQ-8 that a problem which exceeds the capacity of the arithmetic element has been attempted. Care should be taken by the programmer to avoid computer overflow if possible.

Previously, it was mentioned that the AN/FSQ-7 and AN/FSQ-8 computers are capable of addition only. The philosophy behind this fact is easily explained. By

TABLE 1–5. BINARY ADDITION WITH CARRY OF 1

ADDEND	+	AUGEND	+	CARRY OF 1	=	SUM WITH CARRY OF	
0	+	0	+	1	=	1	0
1	+	0	+	1	=	0	1
0	+	1	+	1	=	0	1
1	+	1	+	1	=	1	1

limiting the arithmetic element to one process, it is possible to use the same circuits for all the operations in the computer. This, in turn, results in the saving of space, and easier maintenance. This restriction would seem to pose a problem when considering the other arithmetic functions; however, this is not the case. Consider the subtraction of two numbers, for example. We usually think of subtraction in the decimal system as a process whereby we "borrow" digits from the minuend. This is known as direct subtraction, and it is possible to do the same thing in the binary system; however, it is not necessary. In the AN/FSQ-7 and AN/FSQ-8 computers, subtraction is first converted to addition, and then the normal addition process takes place. This is possible by means of complementing the subtrahend, or number to be subtracted. From trigonometry we know that the complement of an angle is that angle which is added together with the original angle to form a right angle. In other words, it is the missing part which makes up the whole. The same definition is valid for the complement of a binary number; it is that part which makes the whole. The "whole" in this instance is the largest possible binary number that can be expressed. Of course this means a number containing all 1 bits. In the AN/FSQ-7 and AN/FSQ-8, this number is 2^{17}, or 17 1 bits, but since this is a rather cumbersome upper limit to work with, we shall again use an upper limit of 2^6 or 111111_2. To obtain the complement of a binary number, we can perform a direct subtraction from this number. For instance, the complement of 010100_2 is:

```
  111111
 -010100  original number
  101011  complement
```

Also, the complement of 110111_2 is:

```
  111111
 -110111  original number
  001000  complement
```

We can generate the complement of any binary number in this manner, but closer inspection of the above examples will show a short-cut method. Notice that the complement of the number is simply a reversal of each bit position; that is, if a bit position contains a 1, the complement contains a 0, and vice versa. Although it is not necessary to actually perform direct subtraction to obtain a complement, it should be remembered that this is what is effectively taking place. A complement formed by the reversal of bit positions is known as the 1's complement and is used almost exclusively in arithmetic operations in the AN/FSQ-7 and AN/FSQ-8.

The application of complements in the case of the subtraction process can be best explained by giving an example, such as 101101_2 minus 011000_2. We proceed as follows:

a. Complement the subtrahend 011000_2, which gives us 100111_2

b. Perform a normal addition 101101 minuend

```
          +100111  (complement)
C = 1 ←   010100  difference
```

Now we are faced with a situation which we encountered before: there is a carry of 1 out of bit 6; but the limit or modulus of the number has been placed at 6, so we can not create a seventh bit. Examination of the binary numbers being subtracted will show the difference to be in error by 1, so the carry-out of the sixth bit is added back into the low order bit to correct the error, as follows:

```
          101101  minuend
         +100111  (complement)
          010100  difference (uncorrected)
C = 1  ┌──→ 1
          010101  difference (corrected)
```

Another example of binary subtraction is:

```
  111111
 -111110
```

This becomes:

```
          111111
         +000001  complement
          000000  difference (uncorrected)
C = 1  ┌──→ 1
          000001  difference (corrected)
```

Adding the carry-out of the most significant bit back into the low order bit is called the "end-around carry," or more commonly "end carry." (This same condition may result from pure addition also, with a different significance, as previously explained.)

Sometimes a binary subtraction will yield a result that can not be interrupted directly; for instance:

```
  101111
 -111110
```

which is complemented to become an addition, or

```
  101111
 +000001  (complement)
  110000  (difference)
```

Notice that there was no end carry from this problem, and examination would seem to show the difference as being incorrect. However, inspection of the original problem reveals that we were subtracting a large subtrahend from a smaller minuend; therefore, the difference will be a negative number. The arithmetic element of the AN/FSQ-7 and AN/FSQ-8 is not capable of distinguishing between positive and negative numbers as such, so the answer appears to be wrong, if we attempt to interpret it directly. To avoid this error, it is neces-

sary to remember that negative quantities are contained in the arithmetic element in complemented form. Thus, our answer in the above problem is not 110000_2 but -001111_2. Obviously, in calculations where the results are determined as rapidly as they are in the AN/FSQ-7 and AN/FSQ-8, it would be impossible for us to keep track of what quantities were in true form and those which were in complement form. To tell the signs of the various magnitudes, an additional bit is added to the left of the magnitude bits. A 1 in this position (called the sign bit) tells us that the number is negative and in complemented form; a 0 indicates that the number is positive and in true form. The sign bit is processed by the arithmetic element just as if it were a magnitude bit. For instance, if sign bits were added to the magnitude bits in the foregoing example, we would have:

$$\begin{array}{r} 0.101111 \\ -0.111110 \\ \hline 0.101111 \\ +1.000001 \\ \hline 1.110000 \end{array} \quad \text{or} \quad \text{(complement)}$$

Now we can see at a glance that the result is negative and in complement form by the presence of the 1 in the sign bit position.

Because the AN/FSQ-7 and AN/FSQ-8 work with the complement system, an interesting possibility arises in the case of 0. The number 0.000000_2 is read as "positive" 0 $(+0)$ whereas its complement 1.111111_2 is read as "negative" 0 (-0). Both numbers have the same numerical value, and it is unimportant which 0 is used in arithmetic operations. However, in cases where logical decisions are involved, there is a significant difference between the two representations.

Multiplication in the binary system is accomplished exactly as it is in the decimal system. However, since the only digits that can be combined in a binary multiplication are 1 and 0, the products can only be 1 or 0 also. The binary multiplication table showing all possible products is given in table 1–6.

TABLE 1–6. RULES OF BINARY MULTIPLICATION

MULTIPLIER	×	MULTIPLICAND	=	PRODUCT
1		1		1
1		0		0
0		1		0
0		0		0

As an example, consider multiplication of the numbers 110101_2 and 100100_2.

$$\begin{array}{r} 0.110101 \\ 0.100100 \\ \hline 000000 \\ 000000 \\ 110101 \\ 000000 \\ 000000 \\ 110101 \\ \hline 0.011101110100_2 \end{array} \quad \begin{array}{l} \\ \\ \text{1st partial product} \\ \text{2nd partial product} \\ \text{3rd partial product} \\ \text{4th partial product} \\ \text{5th partial product} \\ \text{6th partial product} \\ \\ \text{final product} \end{array}$$

The method shown is direct binary multiplication, and although it is not exactly how the AN/FSQ-7 and AN/FSQ-8 perform multiplications, it will serve as the basis for an explanation of the machine method. Notice that all the partial products are 0's except when a 1 bit is encountered in the multiplier, in which case the partial product is the multiplicand. In the above example, there were two 1 bits in the multiplier; therefore, the multiplicand appeared twice as a partial product (products 3 and 6). To obtain the final product, the partial products are added together, as shown above. Thus, multiplication also becomes a function of addition. Examination of the partial products shows that each time a new product is formed it is shifted one position to the left before being placed in the column. In the AN/FSQ-7 and AN/FSQ-8, the need to shift left is overcome by shifting the previous partial product one place to the right and performing a normal addition when a 1 bit is sensed in the multiplier. When a 0 bit is encountered, no addition of the multiplicand occurs, but the entire partial product is still shifted one place to the right. This process is called the "inherent shift right" and, as its name implies, is performed automatically during multiplication. (The inherent shift right also occurs during pure addition and subtraction problems but is compensated for at the end of each addition or subtraction by a correctional shift left.) Now we can follow our original example of 0.110101_2 multiplied by 0.100100_2 as it occurrs in the AN/FSQ-7 and AN/FSQ-8. Since multiplication of two 6-bit numbers will result in an 11- or 12-bit product, it is necessary to join two registers of the arithmetic element together during multiplication. These registers are the accumulator register, where addition is normally performed, and the B register, which is an auxiliary register. At the start of the multiplication, the multiplier is placed in the B register; the multiplicand is contained in the A register (a temporary storage register); and the accumulator is cleared. Figure 1–3 shows this arrangement. First, the low order bit of the multiplier is sensed. If it is a 1, the multiplicand and accumulator are added together. If it is a 0, no action is taken. Then the partial product is shifted one place to the right, and the action is repeated until

all the multiplier bits (except sign bit) have been sensed. Table 1–7 shows the execution of this problem.

TABLE 1–7. MULTIPLICATION BY ADDITION AND SHIFTING

		PARTIAL AND FINAL PRODUCT	
STEP	MULTIPLICAND	ACCUMULATOR	MULTIPLIER
Start	0.110101	0.000000	0.100100
1		0.000000	0.010010
2		0.000000	0.001001
3	0.110101	0.011010	1.000100
4		0.001101	0.100010
5		0.000110	1.010001
6		0.011101	1.101000

During steps 1 and 2, only 0 bits were sensed in the multiplier, so the partial product was merely shifted to the right two places. This left a 1 bit in the low order position where it was sensed and caused an addition and shift at step 3. Steps 4 and 5 sensed 0 bits and caused two more shifts to the right. The last addition and shift occurred at step 6, and upon completion of the step, the final product is contained in the accumulator and B register (step 6). Examination will show this answer to be the same as that obtained by direct multiplication, except for the inclusion of an extra 0 at the right of the product. This 0 is the sign bit which has been shifted to the right and, therefore, is not a significant digit. Excluding this last digit will produce the correct answer.

The binary division process is perhaps the most difficult of all the binary arithmetic processes to understand. However, it is actually nothing more than the reverse of multiplication, just as in the decimal system. To multiply in binary, we added; to divide in binary, we subtract. But binary subtraction consists of the addition of complements, so, again, the addition function is employed. One slight difference exists when subtracting for a division problem, however; instead of using the 1's complement, the 2's complement of the number is used. This complement is generated by first obtaining the 1's complement (reversing bit positions) and then adding a 1 to the low order bit position. Thus, binary 0.101_2 complemented becomes 1.010_2, and the addition of 1 gives us 1.011_2, which is the 2's complement of 0.101_2. It will be recalled that the use of the 1's complement left the final remainder in error by 1, so the carry-out from the addition had to be added back into the low order bit position to compensate for this deficiency. Since a 1 bit is already added into the 2's complement, no correction of the remainder is necessary, and the carry-out may be ignored. Use of the 2's complement for the division process is based on the fact that the execution time for each trial subtraction is less than it would be with the 1's complement. This is true because it is not necessary to correct the remainder after each trial subtraction. Let us follow an example of direct binary division, often called the "restoring" method. Assume that we wish to divide 0.001101011101_2 by 0.010010_2. Notice that the first six bits of the dividend are smaller in magnitude than the corresponding divisor bits. This condition will result in a fractional answer, since we are dividing a small number by a larger number. The AN/FSQ-7 and AN/FSQ-8 treat all numbers as fractions; therefore, it is desirable that our quotient also be a fraction. The use of fractions rather than whole numbers is unimportant at this point and will be explained later. First, it is necessary to generate the 2's complement of the divisor. This is accomplished in the following manner:

$$\begin{array}{ll} & 0.010010 = \text{divisor} \\ \text{(reverse bits)} & 1.101101 = \text{1's complement} \\ \text{(add 1)} & \underline{\qquad\quad 1} \\ & 1.101110 = \text{2's complement} \end{array}$$

Common sense will tell us that it is useless to try to divide this number into the first six bits of the dividend because the dividend is too small. So a shift one place to the left is performed on the dividend before the first division step. Remember that when we say "divide" we are actually adding the complement of the divisor. Our first step in division then looks like this:

$$\begin{array}{lrl} & & \quad\quad 1 \\ & 0.010010 \;/ & \overline{0.001101011101} \\ \text{2's complement} & (1.101110) & \quad 1101110 \\ & C=1 & \quad 0001000 \end{array}$$

Notice that there was a 0 bit in the most significant position of the current remainder; this indicates that

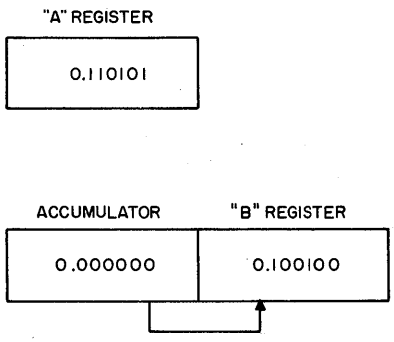


Figure 1–3. Registers Used in Multiplication

the first trial subtraction was successful, so a 1 bit is entered in the quotient. The next step is to shift the remainder one place left, bring down the next bit from the dividend, and attempt another trial subtraction. Therefore:

```
                    10
0.010010  /0.001101011101
(1.101110)   1101110        1st subtraction
             --------
             00010001       1st remainder
              1101110       2nd subtraction
             --------
             1111111        current remainder
```

This time the remainder has a 1 as its most significant bit; this means that the subtraction was not successful since it resulted in a negative number, so a 0 bit is entered in the quotient. Now it is necessary to restore the remainder to its original shifted magnitude (1st remainder) and shift another position to the left. This is accomplished by adding the uncomplemented value of the divisor to the current remainder to offset the 2nd subtraction. The restoration step is shown below:

```
                    10
0.010010  /0.001101011101
(1.01110)    1101110        1st subtraction
             --------
             00010001       1st remainder
              1101110       2nd subtraction
             --------
             11111111       2nd remainder
              0010010       restoration
             --------
             00100011       current remainder
```

Notice that the current remainder now has the same value as the 1st remainder, except that it is shifted one place to the left. The rest of the problem is completed in the manner shown. The entire problem is illustrated below:

```
                   101111
0.010010  /0.001101011101
(1.101110)   1101110        1st subtraction
             --------
             00010001       1st remainder
              1101110       2nd subtraction
             --------
             11111111       2nd remainder
              0010010       restoration
             --------
             00100011       3rd remainder
              1101110       3rd subtraction
             --------
             00100010       4th remainder
              1101110       4th subtraction
             --------
             00100001       5th remainder
              1101110       5th subtraction
             --------
             0001111        final remainder
```

Thus, the final quotient is 0.101111_2 plus a remainder of 1111_2.

The method of binary division explained above is fine for paper and pencil problems, but it is not efficient enough for actual use within the AN/FSQ-7 and AN/FSQ-8 because it is too time-consuming. Each time a trial subtraction results in a negative remainder, the restoration step is necessary. This step can be eliminated first by shifting the negative remainder one place

TABLE 1–8. DIVISION BY NONRESTORATION

STEP	ACTION	DIVISOR	CURRENT REMAINDER	QUOTIENT
1		0.010010	0.001101011101 (dividend)	—
2	Complement	1.101110	0.00110101110	—
3	Subtract	1.101110	1101110	
			0.0010001	1
4	Subtract		1101110	
			1.111111	10
5	Double		1.1111111	
			1111111	10
6	Subtract	0.010010	0010010	
			00100011	101
7	Subtract	1.101110	1101110	
			00100010	1011
8	Subtract	1.101110	1101110	
			00100001	10111
9	Subtract	1.101110	11101110	101111
			0001111 (final remainder)	0.101111

to the left and then performing another trial subtraction with the uncomplemented divisor. This method of division is referred to as the "nonrestoring" method. The rest of the division process is the same. In the division process, the shift of one place to the left effectively multiplies the number by the radix (2) or doubles it. For this reason, the shift left is commonly referred to as the "double" step. Table 1–8 shows the execution of the same division problem by the nonrestoring method. Naturally, division by this method gives the same result as direct or restoring division.

3.4 OCTAL NUMBER SYSTEM

Now that we have covered all the phases of the binary system, it is easy to see that the exclusive use of binary notation is cumbersome. For this reason, common usage is made of the octal number system, which serves as a type of shorthand notation for the binary system. By referring to table 1–2, it can be seen that it is possible to represent a maximum of eight numbers with three binary bits. Thus, if we devise a number system with a radix of 8, each digit in that system will correspond to three binary bits. This is exactly what the octal system does. Table 1–9 gives a comparison of the decimal, octal, and binary equivalents.

TABLE 1–9. DECIMAL, OCTAL, AND BINARY EQUIVALENTS

DECIMAL	OCTAL	BINARY
0	0	000
1	1	001
2	2	010
3	3	011
4	4	100
5	5	101
6	6	110
7	7	111
8	10	001 000
9	11	001 001
10	12	001 010
20	24	010 100
50	62	110 010
75	113	001 001 011
100	144	001 100 100

Notice that the binary numbers are grouped by threes; this is done to provide for easier inspection of the number. Since it has been stated that any octal number may be represented by three binary bits, conversion from octal to binary and back again may be made by inspection. For example, octal $12(12_8)$ may be represented by the binary equivalents for 1 and 2, or 001_2 and 010. Thus, $12_8 = 001\ 010$. Table 1–9 proves this to be true. To convert any binary number to octal, or vice versa, it is necessary only to memorize the binary equivalents for 0_8–7_8. This is easily accomplished, and a little practice will enable one to become quite rapid when making binary-octal conversions. Some examples of such conversions are:

$$1010100010101110101 0_2 = ?_8$$

First, separate the binary numbers in groups of three, starting from the low order bit position. This gives us:

$$010\ 101\ 000\ 101\ 011\ 101\ 010_2$$

Since there are only two binary bits at the high order position (10_2), we assume that there is a leading 0 and append it as shown. Now, each group of three bits is converted to its octal equivalent:

$$010\ 101\ 000\ 101\ 011\ 101\ 010_2$$

$$2\quad 5\quad 0\quad 5\quad 3\quad 5\quad 2_8$$

Thus, $1010100010101110101 0_2 = 2505352_8$.

Converting from octal to binary is just as easy. For instance, to convert 7341076_8 to binary:

$$7\quad 3\quad 4\quad 1\quad 0\quad 7\quad 6_8$$

$$111\ 011\ 100\ 001\ 000\ 111\ 110_2 \quad \text{or}$$

$$7341076_8 = 111011100001000111110_2.$$

Not only does the octal system serve as an abbreviated notation for the binary system, but it can also be used as an intermediate step when converting to decimal notation from binary. This does not imply that it is always easier to employ the octal system for this purpose, but it may be used if desired. In addition, by first converting to the octal system from binary or decimal, and then to the desired system, a check on the direct conversion may be made. For example:

$$7221_{10} = 1110000110101_2$$

by direct conversion, as shown in 3.3. However, if 7221_{10} was first converted to octal, and the resulting octal number was converted to binary by inspection, we could check the result. To convert 7221_{10} to octal, we proceed exactly as in a decimal-to-binary conversion, except that we are now dealing with powers of 8 rather than 2. Table 1–10 gives the values of various powers of 8.

TABLE 1–10. POWERS OF EIGHT

POWER	EQUIVALENT
8^0	1
8^1	8
8^2	64
8^3	512
8^4	4096
8^5	32768
8^6	262144

By referring to this table, we can convert 7221_{10} as follows:

$$\begin{array}{rl} 7221 & \\ -4096 & (1 \times 8^4) \\ \hline 3125 & \\ -3072 & (6 \times 8^3) \\ \hline 53 & \\ -48 & (6 \times 8^1) \\ \hline 5 & \\ -5 & (5 \times 8^0) \\ \hline 0 & \end{array}$$

Remember that 8^2 was not used, so a 0 must be included for this power. Therefore, $7221_{10} = (1 \times 8^4) + (6 \times 8^3) + (0 \times 8^2) + (6 \times 8^2) + (6 \times 8^1) + (5 \times 8^0)$ or $7221_{10} = 16065_8$. Converting this octal number to binary by inspection gives us:

$1\ 6\ 0\ 6\ 5_8$

$1\ 110\ 000\ 110\ 101_2$ or 1110000110101_2.

This result is the same as that obtained by the direct decimal-binary conversion. The other conversion system employed in decimal-binary conversion is also applicable to the octal system. Dividing the decimal number by the radix (8) and recording the partial quotient and remainder gives us:

	8 ⟌ 7221		
1st partial quotient	8 ⟌ 902	5	1st remainder
2nd partial quotient	8 ⟌ 112	6	2nd remainder
3rd partial quotient	8 ⟌ 14	0	3rd remainder
4th partial quotient	8 ⟌ 1	6	4th remainder
final quotient	0	1	final remainder

Reading from the bottom up, we have 16065_8, exactly the same result obtained by the other method.

To convert from octal to decimal, two methods are again available for our use. One method is based on powers of 8, as listed in table 1–10. The position of each octal number determines what power of 8 is contained in the decimal equivalent. For instance, $3247_8 = ?_{10}$. Of course, this means $(3 \times 8^3) + (2 \times 8^2) + (4 \times 8^1) + (7 \times 8^0)$ or $1536 + 128 + 32 + 7$. Thus, $3247_8 = 1703_{10}$. Another example of this method of conversion is $16505_8 = ?_{10}$. Expanding the number gives us $(1 \times 8^4) + (6 \times 8^3) + (5 \times 8^2) + (0 \times 8^1) + (5 \times 8^0)$, or $4096 + 3072 + 320 + 0 + 5$. Adding these numbers will produce a sum of 7493_{10}, the decimal equivalent of 16505_8. The other method of octal-decimal conversion uses a combination of multiplication and addition, just as the first method, but it is not necessary to know the powers of 8 for this conversion. The rules of conversion are as follows:

1. Multiply the most significant digit by the radix (8 in this case).
2. Add the next most significant digit to this product and again multiply by the radix.
3. Repeat step 2 for every significant digit except the low order digit. Simply add this digit and do not multiply.

Using the number 16505_8 again, let us perform a conversion by this method:

Step 1. $1 \times 8 = 8$

Step 2. $8 + 6 = 14$

(repeat) Step 2. $14 \times 8 = 112$

(repeat) Step 2. $112 + 5 = 117$

(repeat) Step 2. $117 \times 8 = 936$

(repeat) Step 2. $936 + 0 = 936$

(repeat) Step 2. $936 \times 8 = 7488$

Step 3. $7488 + 5 = 7493_{10}$

It should be noted that this method can be used to convert a number from any system to the decimal system and is not limited to the octal system alone.

The rules of octal arithmetic are no different than the rules for any number system; it is only necessary to remember that we are dealing with a radix of 8. However, since the octal system does have several symbols which are used in the decimal system, the results of octal arithmetic may sound odd at first. It is surprising to hear someone say that "seven plus seven equals sixteen" or "five times five equals thirty-one." But examination will show that $7_8 + 7_8 = 16_8$ and $5_8 \times 5_8 = 31_8$ are true statements. Octal arithmetic must be learned the same way we learned decimal arithmetic: by memorization of tables. For convenience, table 1–11 shows octal addition and table 1–12 shows octal multiplication.

TABLE 1–11. OCTAL ADDITION

AUGEND	ADDEND										
	0	1	2	3	4	5	6	7	10	11	12
0	0	1	2	3	4	5	6	7	10	11	12
1	1	2	3	4	5	6	7	10	11	12	13
2	2	3	4	5	6	7	10	11	12	13	14
3	3	4	5	6	7	10	11	12	13	14	15
4	4	5	6	7	10	11	12	13	14	15	16
5	5	6	7	10	11	12	13	14	15	16	17
6	6	7	10	11	12	13	14	15	16	17	20
7	7	10	11	12	13	14	15	16	17	20	21
10	10	11	12	13	14	15	16	17	20	21	22
11	11	12	13	14	15	16	17	20	21	22	23
12	12	13	14	15	16	17	20	21	22	23	24

Some examples of octal arithmetic (and the decimal equivalents performed as a check) are as follows:

$$\begin{array}{r} 74_8 = 60_{10} \\ +32_8 = 26_{10} \\ \hline 126_8 = 86_{10} \end{array}$$

$126_8 = (\underline{1} \times 8^2) + (\underline{2} \times 8^1) + (\underline{6} \times 8^0) = 64 + 16 + 6 = 86_{10}$

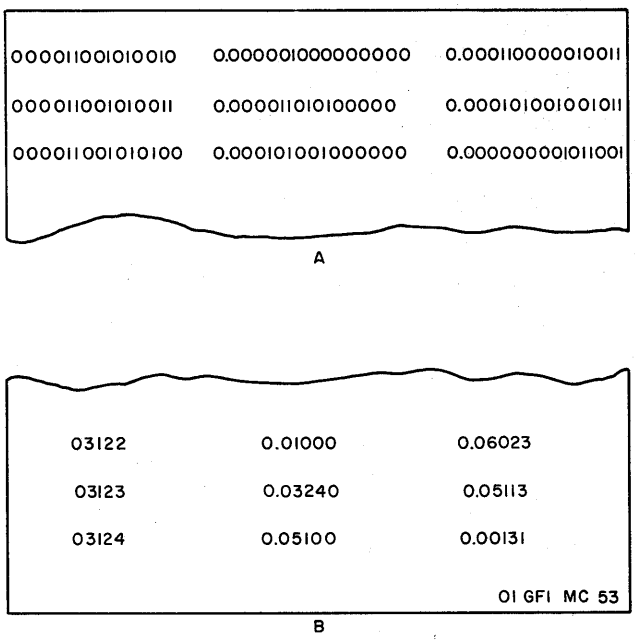


Figure 1–4. Comparison of Binary and Octal Notation

$$\begin{array}{r} 156_8 = 110_{10} \\ -71_8 = 57_{10} \\ \hline 65_8 = 53_{10} \end{array}$$

$65_8 = (\underline{6} \times 8^1) + (\underline{5} \times 8^0) = 48 + 5 = 53_{10}$

$$\begin{array}{r} 13_8 = 11_{10} \\ \times 13_8 = 11_{10} \\ \hline 41 = 11 \\ 13 \quad 11 \\ 171_8 = 121_{10} \end{array}$$

$171_8 = (\underline{1} \times 8^2 + (\underline{7} \times 8^1) + (\underline{1} \times 8^0) = 64 + 56 + 1 = 121_{10}$

$$\begin{array}{r} 176_8 = 126_{10} \\ 420_8 = 272_{10} \\ \hline 000 \quad 252 \\ 374 \quad 882 \\ 770 \quad 252 \\ \hline 102740_8 = 34272_{10} \end{array}$$

$102740_8 =$ (1) $1 \times 8 = 8$

(2) $8 + 0 = 8$

(3) $8 \times 8 = 64$

(4) $64 + 2 = 66$

(5) $66 \times 8 = 528$

(6) $528 + 7 = 535$

(7) $535 \times 8 = 4280$

(8) $4280 \times 8 = 4284$

(9) $4284 \times 8 = 34272$

(10) $34272 + 0 = 34272_{10}$

TABLE 1–12. OCTAL MULTIPLICATION

MULTIPLIER	MULTIPLICAND										
	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
3	0	3	6	11	14	17	22	25	30	33	36
4	0	4	10	14	20	24	30	34	40	44	50
5	0	5	12	17	24	31	36	43	50	55	62
6	0	6	14	22	30	36	44	52	60	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

Obviously, octal division and subtraction can be performed simply by referring to tables 1–11 and 1–12. As mentioned previously, octal arithmetic may seem confusing at first, but, in reality, it is simply a straightforward application of the laws concerning radix and positioned notation of magnitude.

As a rule, it is not necessary to perform a great number of arithmetic operations while in the octal system, since the main use of octal notation occurs when it is substituted for long binary expressions, and when converting between the binary and decimal system. Figure 1–4 (part A) shows the binary notation of an actual maintenance program used in the AN/FSQ-7; part B of this figure shows the equivalent octal notation. It can readily be seen that the latter notation is much easier to interpret than the equivalent binary notation.

CHAPTER 4
THE SAGE SYSTEM

4.1 THE AIR DEFENSE PROBLEM

Several years ago, the military authorities realized that the system of air defense then in use was outmoded and could not maintain adequate security for our nation. The problem of many planes flying at high speeds was more than a manual control center could handle, and it became apparent that a system was needed which could keep track of all the planes in the air, together with any and all data collected about each plane. At the same time, this system would have to be flexible enough to allow human monitors to make important decisions regarding weapons assignment, target identification, etc.

After considerable research and study, the Air Force decided that the large-scale digital computers being developed at the time were the best instrument to bring our air defense system up to its best operating capability. These computers would handle all the data which could be gathered concerning the air movements in a given sector, and would be capable of updating this information and presenting it to human monitors. However, the manual system could not be abandoned immediately, because it would take some time for these computers to be installed, if they proved to be feasible. In addition, the problem of integrating the radar sets, etc., associated with the manual system, was still unsolved. The need for an improvement of the air defense system was very real, but the task of bringing about such an improvement was a problem of equal weight.

In 1950, the Air Force contracted for civilian assistance in solving the air defense problem. At the time, the goal of this project was to find the most efficient application of a digital computer to meet the ever-growing problem. The first of many systems to evolve as a result of these studies was the Cape Cod System, established in 1953. The Whirlwind 1 computer, built by MIT, was the nucleus of the system, and it utilized several small radar sets in the New England area. This system provided valuable data concerning the size and speed of the computer that was necessary to do the job the Air Force wanted and proved the feasibility of using digital computers. In 1954, the system was expanded to employ more radar sets as inputs, and additional improvements were made in the accuracy of the detection programs being run in the computer. Satisfied that digital computers could handle the air defense situation, the Air Force set up what was known as the semi-automatic ground environment system (SAGE), a huge network of computers and radar sites. The radar sites would supply information about an area to a SAGE computer, which could then decide what action to take, depending on the situation. The first experimental SAGE subsector was installed in 1955, using a huge digital computer, designated AN/FSQ-7 (XD-1), which was built by IBM. This computer was located at Lexington, Mass., and its objectives were to provide data on computer reliability, operating procedures, and training of personnel for the newly formed SAGE system.

4.2 ELEMENTS OF SAGE

As mentioned earlier, radar sites provide much of the information which is used as raw data for the computer. However, there are many other elements in the SAGE System which provide raw data to the computer's input system; among these are status reports from surrounding air bases and weather stations, Ground Observer Corps reports, and relay information from adjacent computers. The computer outputs go to airborne aircraft, other computers, and various defense centers employing teletype receivers. Each SAGE computer is connected to some or all of these devices; together, they form what is known as a SAGE sector. As each sector is concerned only with the geographic area in which it is located, a higher echelon of command, which surveys the overall air defense situation, has been established. This command level is referred to as a division, and is controlled by a computer similar to those used at the sector level. Instead of raw data as inputs, however, a SAGE division level computer receives data already processed by the other computers. In turn, the division summary of its air defense situation is forwarded to the headquarters of the Continental Air Command. The computers used at the sector level are designated AN/FSQ-7 Combat Direction Central; those used at the division level are designated AN/FSQ-8 Combat Control Central. Figure 1–5 is a simplified drawing showing how the various elements described are integrated into the SAGE System.

4.3 LOGICAL ELEMENTS OF AN/FSQ-7 AND AN/FSQ-8

It can be seen from figure 1–5 that the basic building block of the SAGE System is the AN/FSQ-7. This

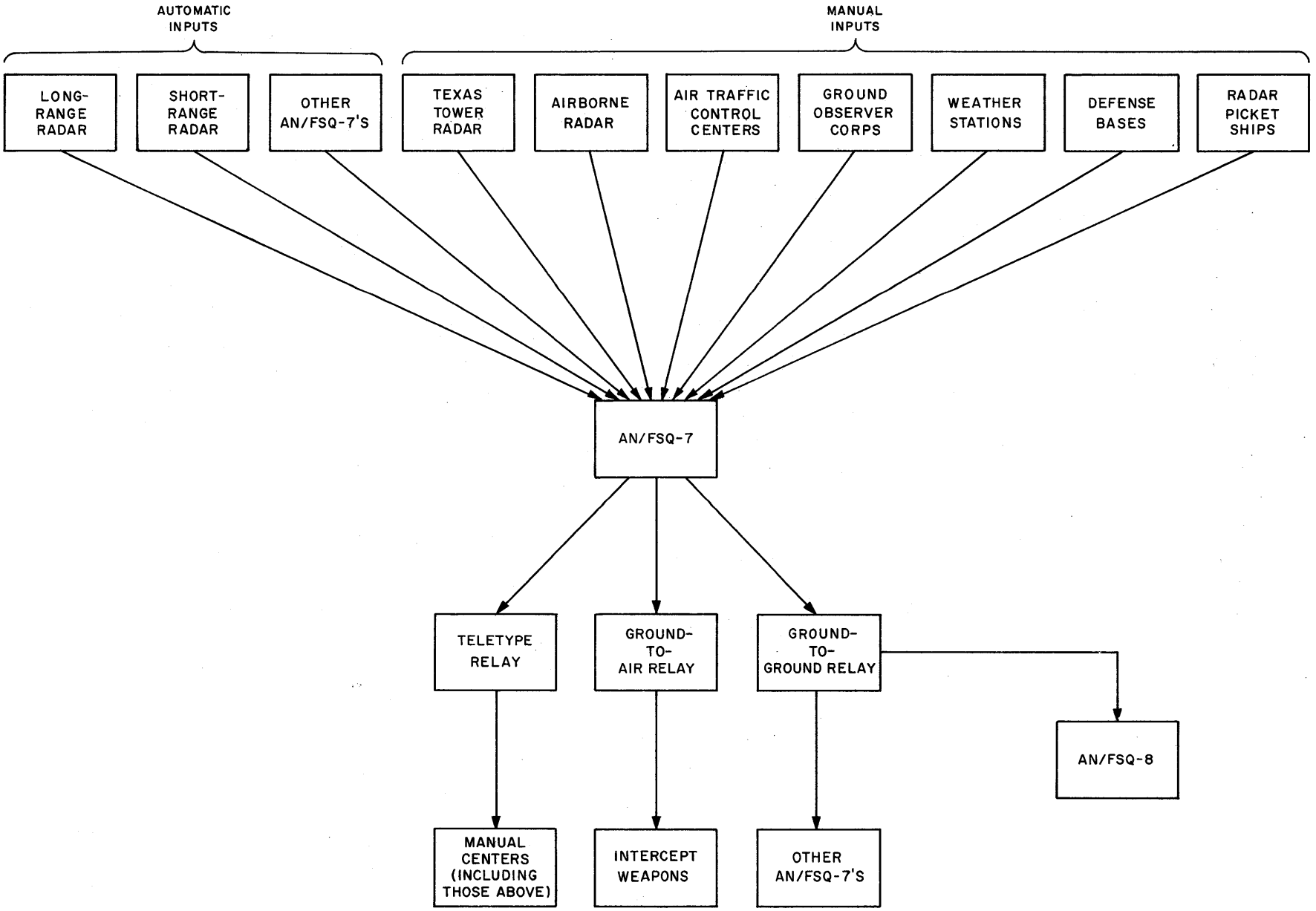


Figure 1–5. SAGE System, Simplified Diagram

computer is a high-speed, general-purpose machine which is used in a real time situation. (Real time means that the data being processed will provide information about an event that is in progress at the time of processing.) Because the AN/FSQ-7 is such a large computer, and its operating characteristics are unique, it includes seven separate systems. These systems, together with their proper logic numbers, are listed below:

a. Central Computer System
b. Drum System
c. Input System
d. Output System
e. Display System
f. Power Supply and Marginal Checking System
g. Warning Light System

The function of the Central Computer System is to process the data delivered to the AN/FSQ-7. The actual arithmetic operations are performed by this system; in addition, the Central Computer also synchronizes many of the operations of the other systems. The Drum System has a dual capacity within the AN/FSQ-7; it serves as a buffer, or time delay, between the high-speed Central Computer and the other systems, and is also used as an auxiliary storage device. Raw data from the sources shown in figure 1–5 is delivered to the Input System in a variety of forms and is decoded and rearranged into a standard form for use within the AN/FSQ-7. The processed data is handled by the Output System where it is sorted into the proper categories and transmitted to relay stations. Human monitoring and tactical decisions are made through the use of the Display System which presents processed data in visual form. The Power Supply and Marginal Checking System supplies the necessary voltages to the AN/FSQ-7 and, in addition, provides for a special testing facility used by maintenance personnel. Figure 1–6 shows the logical arrangement of these systems within the AN/FSQ-7.

Due to the fact that the proper operation of the AN/FSQ-7 is so vital to the air defense system, provisions are made for two identical Central Computer Systems to be included in each AN/FSQ-7. Some portions of the other systems are also duplicated, or duplexed. The basis for duplexing some equipment but not all of it was determined by the overall effect that equipment would have on the operating capability of the AN/FSQ-7 if it were to fail. That is, equipment that would render the entire AN/FSQ-7 useless if it failed was duplexed, but equipment that would merely lower the performance standards slightly if it failed was not duplexed. The identical Central Computer Systems are referred to simply as "computer A" and "computer B."

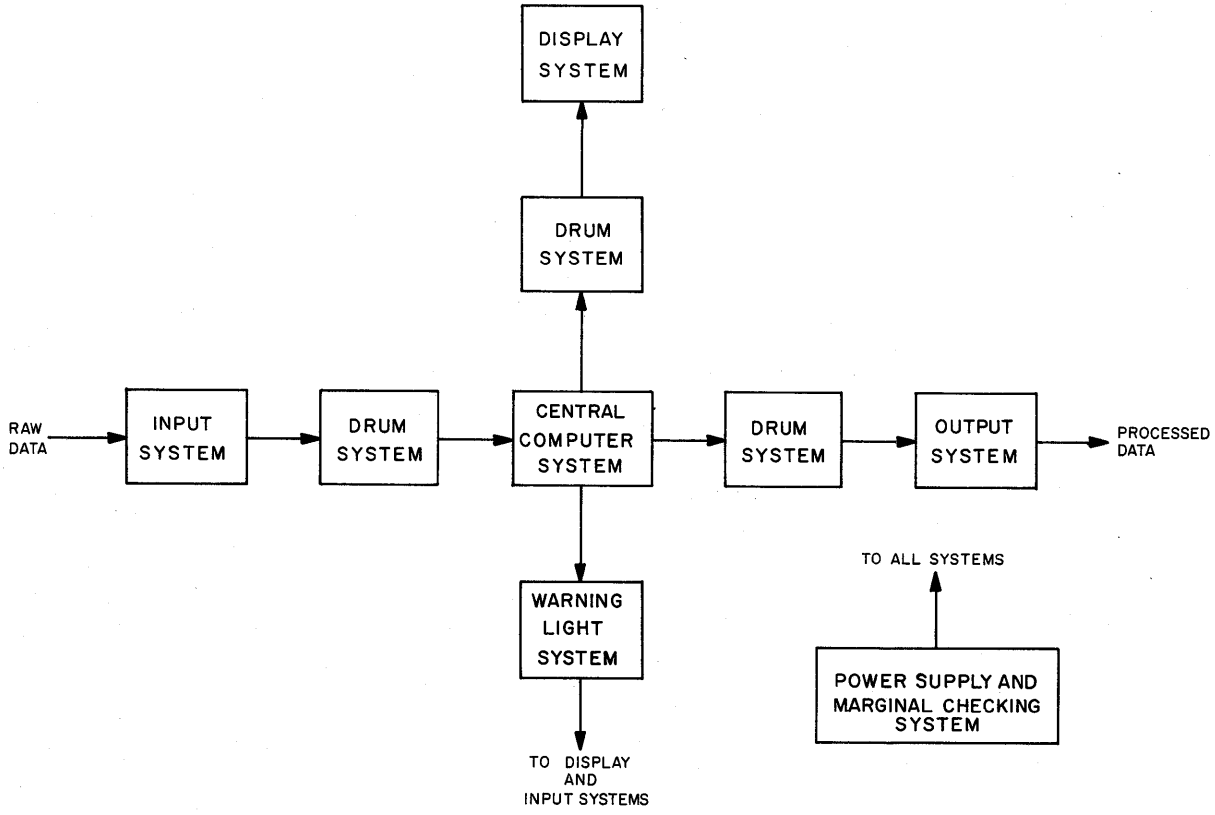


Figure 1–6. AN/FSQ-7, Simplified Block Diagram

One or the other of these computers is in active status 24 hours daily, ensuring continuous functioning of the air defense network. The other computer is said to be on standby status, receiving preventive maintenance or running data reduction programs.

The AN/FSQ-8 Combat Control Central is a modified AN/FSQ-7. Its Input System is not as extensive as that of the AN/FSQ-7, since the data it receives has already been processed by one of the AN/FSQ-7's under its control. Similarly, the Output System does not have to transmit data to as many places as the AN/FSQ-7 and is proportionately smaller. However, in general operation, both the AN/FSQ-7 and AN/FSQ-8 are alike, and are discussed together throughout this manual. Where differences in equipment that may affect programmed operation occur, the AN/FSQ-8 will be discussed separately.

PART 2
BASIC PROGRAMMING

CHAPTER 1
INTRODUCTION

1.1 GENERAL

For proper operation, a digital computer must be able to execute a problem or series of problems in strict accordance with a known method of solution. Obviously, the computer cannot devise its own method of solution; therefore, all computer operations must be controlled by a series of instructions to the computer. This series of instructions is referred to as a program, or computer program. This part of the manual presents some of the basic instructions associated with the AN/FSQ-7 and AN/FSQ-8 and the various ways of combining such instructions to produce a desired program.

1.2 COMPUTER WORD DESCRIPTION

A digital computer must be able to recognize the instructions it receives. Therefore, these instructions must be presented in a combination of digits, since this is the only "language" the computer can interpret. The AN/FSQ-7 and AN/FSQ-8 utilize a system of pure binary numbers for all operations; therefore, it is mandatory that computer instructions are coded in binary. Data must also be in binary form. To allow air defense data and instructions to be processed by the same circuits, a standard form or layout is necessary for both types of numerical information. This layout is referred to as a computer "word" and is composed of 32 bits in the AN/FSQ-7 and AN/FSQ-8. The computer word is divided into two half-words, simply referred to as the "left half-word" and the "right half-word." The reason for this division is that the Central Computer System has dual arithmetic elements, allowing two operations to take place simultaneously. The arithmetic elements are also designated as left and right, with the left arithmetic element processing the left half of a computer word and the right arithmetic element processing the right half. Figure 2–1 shows the word layout for the AN/FSQ-7 and AN/FSQ-8. Each half-word consists of 15 magnitude bits, plus one bit for sign. In addition, an extra bit is included at the extreme left of the word. The bit is generated by the Central Computer and is used to check the information transfer to and from the Central Computer. This bit, called the parity bit, is not used in arithmetic computations and does not have to be assigned a position for instructions or data. It is included to show that a word stored in core memory actually consists of 33 bits – the two half-words plus the parity bit. Often, it is necessary to refer to single bits within a computer word; for this reason, the letter L or R is placed before the bit position to designate the proper half-word. For instance, a reference to L9 means that we are concerned with the ninth magnitude bit of the left-half word.

1.2.1 Instruction Words

As previously mentioned, the computer cannot function by itself, but must be controlled by programs. The instructions within the program direct the computer to add, subtract, print results of computations, accept more input data, and perform various other operations. The instruction word is composed of several parts, each of which gives part of the information regarding the overall function of the instruction. Figure 2–2 shows the instruction word layout, with the various portions indicated.

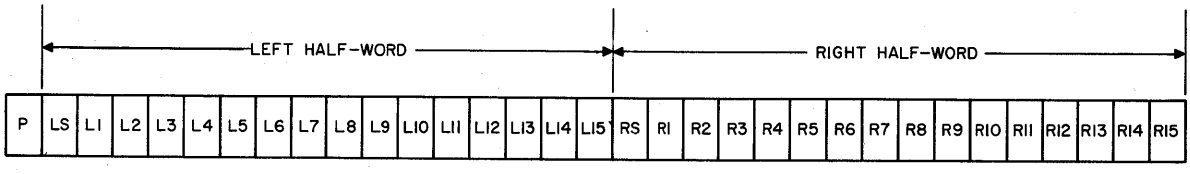


Figure 2–1. Computer Word Layout

The left half-word is commonly referred to as the operation portion of the instruction because this half-word tells what operation is to be performed. Analysis of the various bits in the operation portion is as follows:

LS – In the AN/FSQ-7, this bit is logically connected to the right half-word (address). This resulted from the installation of an expanded memory which has a greater number of addresses than can be specified by the right half-word alone. The AN/FSQ-8 does not have an expanded memory so bit LS is not used as part of the address.

L1 through L3 – These bits specify what index register, if any, is to be used for address modification.

L4 through L6 – Any one of eight classes of instructions are indicated by these bits.

L7 through L10 – Any one of several variations possible within each particular class of instructions is indicated by these bits.

L10 through L15 – These bits are referred to as auxiliary bits because they serve various purposes, depending on the instruction. In several instances, only some of the available bits are used. A list of the various uses of the auxiliary bits, together with the particular bits employed, is given below:

a. Index interval (Bits L10-L15. Determines the amount the index register selected by L1-L3 is to be reduced.)
b. Sense code (Bits L10-L15. Determines what device is to be sensed for a particular condition.)
c. Select code (Bits L10-L15. Selects an IO device.)
d. Operate code (Bits L10-L15. Starts or stops an internal operation.)
e. Select drum code (Bits L10-L15 plus R1. Selects a drum field for reading or writing.)
f. Interleave code (Bits L13-L15. Specifies the increment between drum registers to be used in reading or writing.)
g. Overflow suppression (Bit L13 only. Suppresses all overflow alarms.)
h. Left overflow control (Bit L14 only. Determines whether action is to be taken when overflow occurs in left arithmetic element.)
i. Right overflow control (Bit L15 only. Determines whether action is to be taken when overflow occurs in right arithmetic element.)
j. Compare variation code (Bits L10-L12. Used in conjunction with bits L4-L9 to determine which one of eight compare instructions is to be executed.)
k. Test variation code (Bits L10-L15. Bit L10 determines which one of two test instructions is to be executed; bits L11-L15, which one of 32 bits is to be tested.)
l. Seventeen bit option code (Bit L12 only. Used to determine whether 16-bit operation or 17-bit operation is used in conjunction with certain instructions.)

It can readily be seen that bit L10 has an overlapping function. Generally, it is used to specify the last bit of the class variation; however, it is sometimes used as an auxiliary bit. No difficulty is encountered, however, because the codes for the different instructions are so arranged that when bit L10 is used as an auxiliary bit it is not needed as a class variation bit.

The right half-word of an instruction word is referred to as the address portion. The term "address" does not necessarily mean that an absolute location will always be found in this half-word, although this is a common use of the right half-word. When explaining the layout of the address portion, it is necessary to remember that bit LS is logically part of the right half-word in the AN/FSQ-7. As mentioned before, bit LS is not needed in the AN/FSQ-8 as part of the address because of the smaller memory it contains. Bit RS is not needed either. Thus, these two bits may be disregarded when speaking of the address portion of the AN/FSQ-8.

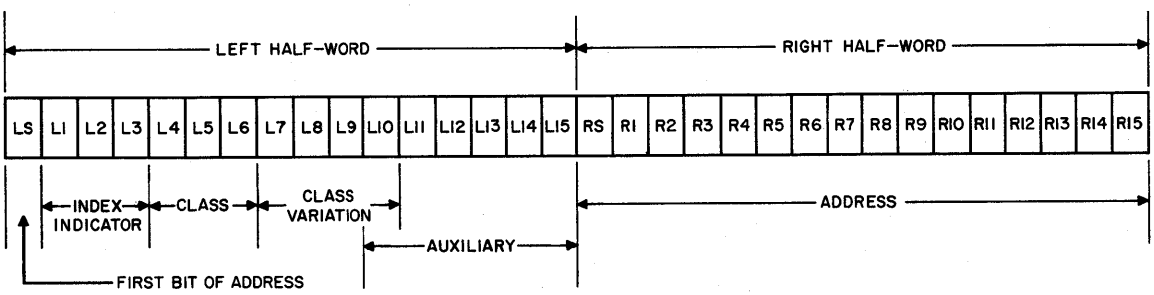


Figure 2–2. Instruction Word Layout

A list of the uses of the right half-word is given below. The LS and the RS through R15 bits are used except where noted:

a. Actual address of an operand (memory location).
b. Initial address of an addressable drum.
c. Number of words to be transferred to or from internal memory during an input-output (IO).
d. Number of steps in a shift or cycle instruction.
e. Drum group selection (Bit RS only. Determines whether the main drum or auxiliary drum group is selected for reading or writing.)
f. Identity code (Five codes, consisting of either R5 through R10, R7 through R15, R11 through R15, R12 through R15, or R14 through R15. These codes tell the source or type of information desired by the Central Computer.)
g. Value to be loaded into an index register.
h. A constant.

1.2.2 Data Words

Just as with instruction words, a data word consists of two half-words which are processed simultaneously by the dual arithmetic elements of the Central Computer System. Data received from various sources will have various forms, so the most general layout of a data word is what we shall consider. This layout conforms with the word layout shown in figure 2–1. The sign bit of a data word actually indicates polarity, whereas it served only as a code in an instruction word. Therefore, we have two half-words of 15 bits plus magnitude. In the discussion on binary division, it was stated that all numbers in the AN/FSQ-7 and AN/FSQ-8 are treated as fractions. This restriction is placed on data primarily so that the multiplication of two numbers will always result in a product smaller than either of the numbers, thus positively avoiding overflow. Another consideration is that the results of a multiplication may be stored from one register without a loss of bit significance. As a result of this restriction, all numerical data within the AN/FSQ-7 and AN/FSQ-8 lies somewhere between the limits of $+1$ and -1. Data which has an actual magnitude of more than unity must therefore be scaled (factored) so that it appears in fractional form.

1.3 CENTRAL COMPUTER TIMING

In the discussion of digital computers (Part 1, Ch 2), it was stated that most computers follow a definite cycle configuration during operation. For a better understanding of how certain combinations of cycles are employed in the execution of various instructions, we must first examine the basic timing of the AN/FSQ-7 and AN/FSQ-8.

1.3.1 Machine Timing

The AN/FSQ-7 and AN/FSQ-8 operate on a pulse frequency of 2 mc per second. In other words, all internal operations are controlled by a timing generator that issues pulses every $\frac{1}{2}$ μsec. These pulses are referred to as timing pulses (TP's), and they constitute the primary component in machine timing. Twelve consecutive TP's make up a machine cycle, a period 6 μsec long. For ease of identity, each pulse is numbered from 0 through 11, and we usually consider a machine cycle to start at TP 0. Figure 2–3 illustrates a typical machine cycle for the AN/FSQ-7 and AN/FSQ-8. Five types of machine cycles are used during operation of the Central Computer System. The type of cycle depends on the control signals being generated at any one time. Since two of these machine cycles are used for IO transfers, they will be discussed subsequently. The three remaining cycles are PT (program time) cycles, OTA (operate time A) cycles, and OTB (operate time B) cycles.

A PT cycle is used to obtain an instruction from memory and decode it. In some cases, the actual execution of an instruction may also be performed at this time because of the simplicity of the operation and because an operand from memory is not required. However, the majority of instructions available for use in the AN/FSQ-7 and AN/FSQ-8 require operands from memory, so the main purpose of the PT cycle is to select and decode an instruction. One significant point to remember is that while a PT cycle is usually thought of in terms of instruction decoding, arithmetic operations may be going on at the same time. The two functions are completely separate, but they may occur within the same PT cycle. This will be demonstrated in the discussion on instruction timing (1.3.3).

Two types of operate time cycles are necessary because it is not possible to select an operand from memory, perform an operation on it, and return the operand

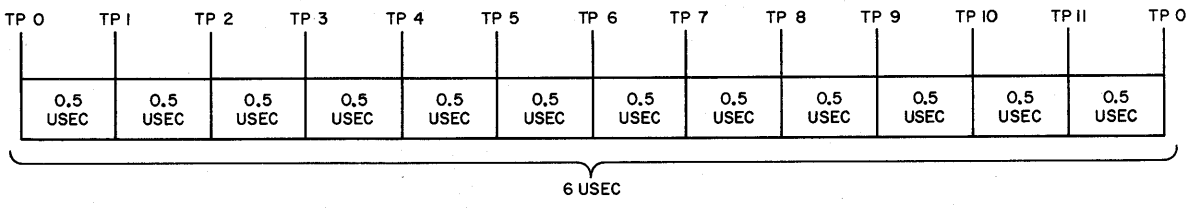


Figure 2–3. Basic Machine Cycle

to memory within the 6-μsec period. Therefore, an OTA cycle is used to select an operand from memory and transfer it to the arithmetic element, and an OTB cycle is used to select a memory location and store an operand from the arithmetic element in that location.

1.3.2 Memory Timing

When we execute an instruction that necessitates obtaining an operand from or placing a number in core memory, it is necessary to energize certain circuits that will accomplish the desired transfer to or from the core memory location. The same thing is true for obtaining the instructions themselves, since they are stored in core memory along with the operands. The length of time that is necessary to accomplish a core memory transfer is known as a memory cycle. The memory cycle for the AN/FSQ-7 and AN/FSQ-8 is 6 μsec in duration, enabling internal memory to operate in the same length of time as a machine cycle. Figure 2–4 shows the relationship of the machine cycle to the memory cycle.

At approximately TP 2, the selection of the register is completed and its contents are read out of the magnetic cores. Then the word is rewritten back into the same location or a new word is substituted, depending on the instruction. The delay between the read and write portions of the memory cycle is to allow the circuitry associated with rewriting to stabilize.

1.3.3 Instruction Timing

As mentioned in 1.3.1, above, most instructions are composed of PT and OT machine cycles. Therefore, most instructions require some multiple of 6 μsec for their execution. Certain instructions, such as multiplication and division, require more execution time than is available in the OT cycles because of the formation of partial products and quotients. Referring to figure 2–4, we can see that an instruction is completely read out of memory by TP 6; therefore, it is available for decoding at TP 7. From TP 7 on, signals are generated which are necessary for the arithmetic element to become prepared for an operation (if the next cycle is an OT cycle) or for the next instruction to be read (if the next machine cycle is a PT cycle). For purposes of convenience, the time pulses are referred to by the name of the machine cycle in which they occur. Therefore, we say that an instruction is ready for decoding at program time PT 7. It is important to remember that there is no PT 7 pulse as such; it is simply the eighth timing pulse issued during the PT cycle. Similarly, the first timing pulse that occurs during an OTA cycle is referred to as OTA 0. Because an instruction is available for decoding at PT 7, we consider this time to be the start of an instruction. Some justification for this may be necessary, since it would appear that the logical starting point for instruction execution would be with the selection of that instruction. If the latter were true, a typical operation (such as addition) would require an entire PT cycle to select and decode the instruction and an entire OTA cycle to start the addition process. However, the addition is not completed at the end of the OTA cycle; the process will continue approximately another 3 μsec. Thus, we have on the order of 15 μsec for an addition to be performed. However, if the selection of the addition instruction was considered part of the previous instruction (regardless of what it was), we could start the addition instruction of PT 7. From PT 7 through PT 11, the decoding would be performed, and the next machine cycle would be an OTA cycle, enabling us to start the addition process. Just as with the first example, the addition is not completed at the end of the OTA cycle; however, instead of waiting for completion, we can start another PT machine cycle to select the next instruction. This will occur at PT 0 and carry on through PT 6. This is a period of 3 μsec, which is enough time to complete the addition. Thus, during PT 0 through PT 6, two operations are going on simultaneously within the Central Computer System: the execution of the current instruction is being completed and the next instruction is being selected from memory. With this type of operation, we have also effectively reduced the time required for an addition to 12 μsec.

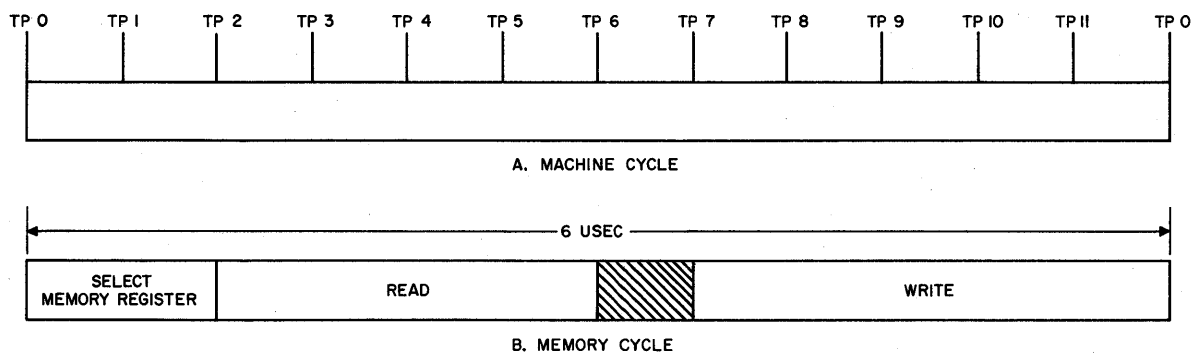


Figure 2–4. Relationship of Machine Cycle to Memory Cycle

All instructions used in the AN/FSQ-7 and AN/FSQ-8 operate on this principle of starting an instruction at PT 7. The addition instruction described above is known as a 2-cycle instruction, since it requires one complete PT machine cycle and one complete OTA machine cycle. This cycle configuration is shown in figure 2–5.

A few instructions available for the AN/FSQ-7 and AN/FSQ-8 are so simple that their execution can be carried out during one PT machine cycle. These 1-cycle instructions take place from PT 7 through PT 6. Figure 2–6 (part A) shows the configuration of this type of instruction.

Many of the instructions associated with the AN/FSQ-7 and AN/FSQ-8 require only one operand to be transferred to or from memory. These are the 2-cycles instructions, and their configuration is shown in figure 2–5. However, some instructions require that an operand be read out of memory and then returned. In this case, both an OTA and an OTB machine cycle are required in addition to the PT cycle; therefore, these instructions are known as 3-cycle instructions (fig. 2–6, part B).

Some arithmetic instructions do not fall into any of the above cycle configurations because they require a variable amount of time to execute. Instructions dealing with operands from memory (such as multiplication and division) require a PT cycle, an OTA cycle, and what is known as an arithmetic pause. This type of cycle configuration is shown in figure 2–6, part C. Still another type of instruction does not utilize an operand from memory (such as round-off of a number already in the arithmetic element). These instructions have a cycle configuration comprising a PT cycle and an arithmetic pause, as illustrated in figure 2–6, part D. During these pauses, machine operation is controlled by the 2-mc pulses issued by the master oscillator.

1.4 CENTRAL COMPUTER SYSTEM ANALYSIS

Before beginning a study of the actual instructions used in the AN/FSQ-7 and AN/FSQ-8, it is useful to become acquainted with a general description of the Central Computer System and the various registers involved in arithmetic operations. The Central Computer System comprises several elements which are listed below with their proper logic numbers:

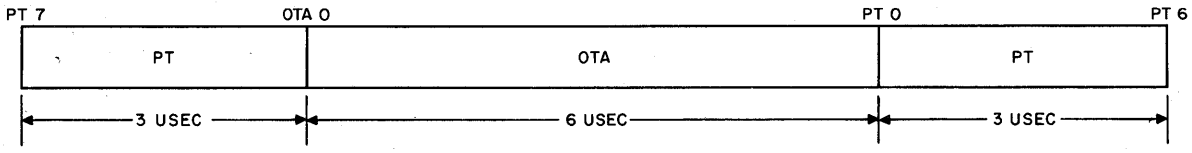


Figure 2–5. Cycle Configuration for 2-Cycle Instruction

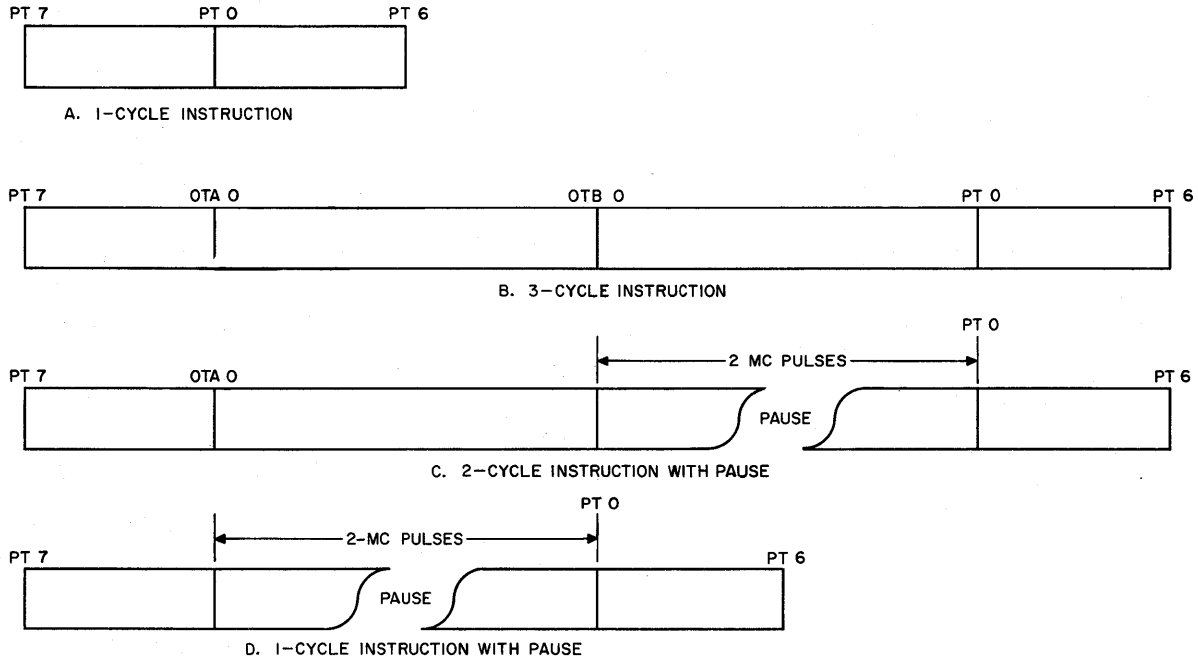


Figure 2–6. Various Cycle Configurations for AN/FSQ-7 and AN/FSQ-8

0.1 Memory element

0.2, 0.3 Instruction control element

0.4 Program element

0.5 Arithmetic element

0.6, 0.7 Selection element

0.7 IO element

In addition, the manual controls associated with the Central Computer System are usually included in any discussion of the system. However, these controls are not considered an integral element of the Central Computer since they are located on consoles which contain controls for all systems within the AN/FSQ-7 and AN/FSQ-8. The logic number for the manual controls is (7.). Figure 2–7 shows the elements of the Central Computer System as they are related in the AN/FSQ-7 and AN/FSQ-8.

1.4.1 Memory Element

1.4.1.1 General

The memory element in the AN/FSQ-7 and AN/FSQ-8 is a high-speed, large-capacity device. As can be seen by referring to figure 2–7, it stores both data and instructions, making it a shared memory, similar to the typical digital computer described in Part 1, Chapter 2. The main component of the memory element consists of two magnetic core storage units. In addition, a test memory facility is provided for maintenance purposes. Also logically associated with the memory devices are four registers: the left and right memory buffer registers, the memory address register, and the clock register. Each of these components is discussed below.

1.4.1.2 Core Memory

In the AN/FSQ-7, the physical core memory units are referred to as memory 1 and memory 2. Memory 1 contains $65{,}536_{10}$ or $200{,}000_8$ storage registers; memory 2 contains $4{,}096_{10}$ or $10{,}000_8$ storage registers. In the AN/FSQ-8, both memory 1 and 2 contain $4{,}096_{10}$ or $10{,}000_8$ registers. The core memories are nonvolatile, meaning that they retain the information which is stored in them, even when power is not applied to the units. We also consider the memories to have random access, meaning that any memory location may be selected and read out in the same amount of time. This time is referred to as random access time (or memory cycle) and is 6 μsec in the AN/FSQ-7 and AN/FSQ-8. Thus, a minimum of 6 μsec must elapse between successive word transfers. One more important point to consider is that readout from core memory is nondestructive. If we transfer the contents of location 100_8 to the arithmetic element, the word is automatically rewritten into memory location 100_8 and can be used again. However, when we write a word into core memory, the contents of the selected register are destroyed, and replaced by the new word. Figure 2–8 shows memory 1 and figure 2–9 shows memory 2 for the AN/FSQ-7. Figure 2–9 also represents both memory units for the AN/FSQ-8.

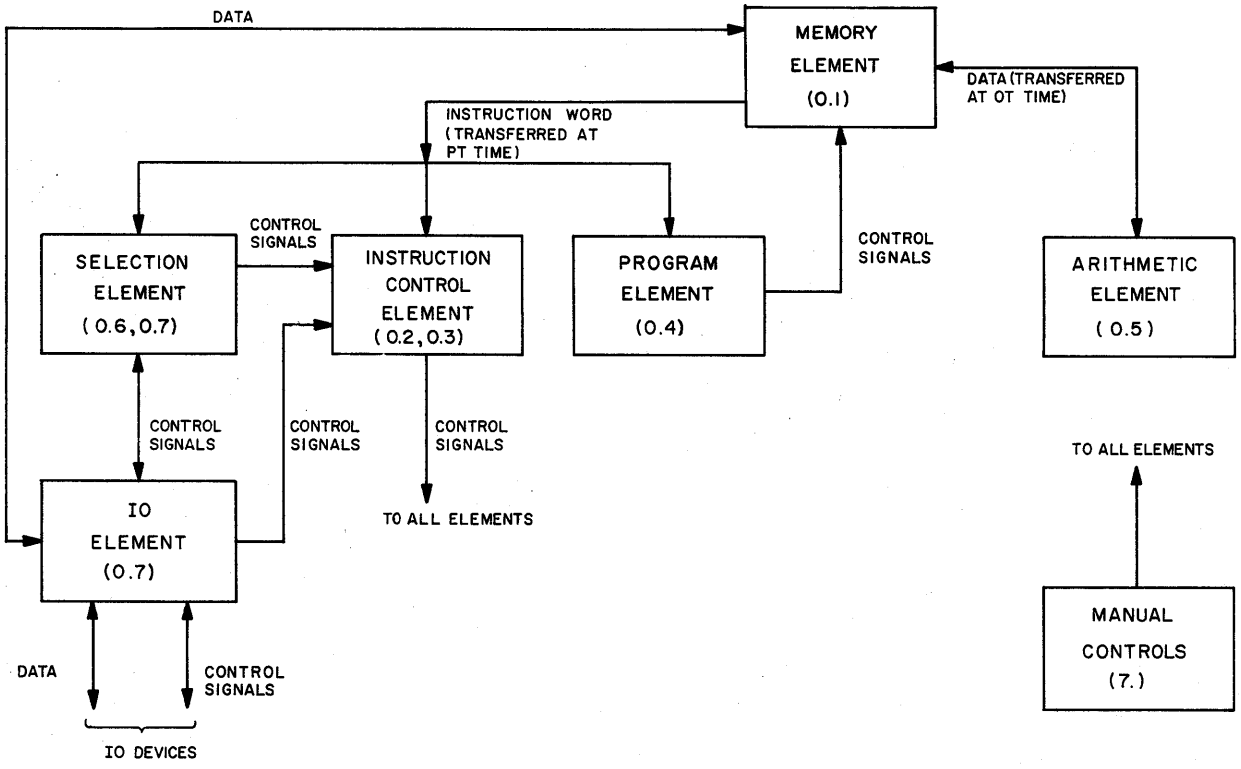


Figure 2–7. Central Computer System

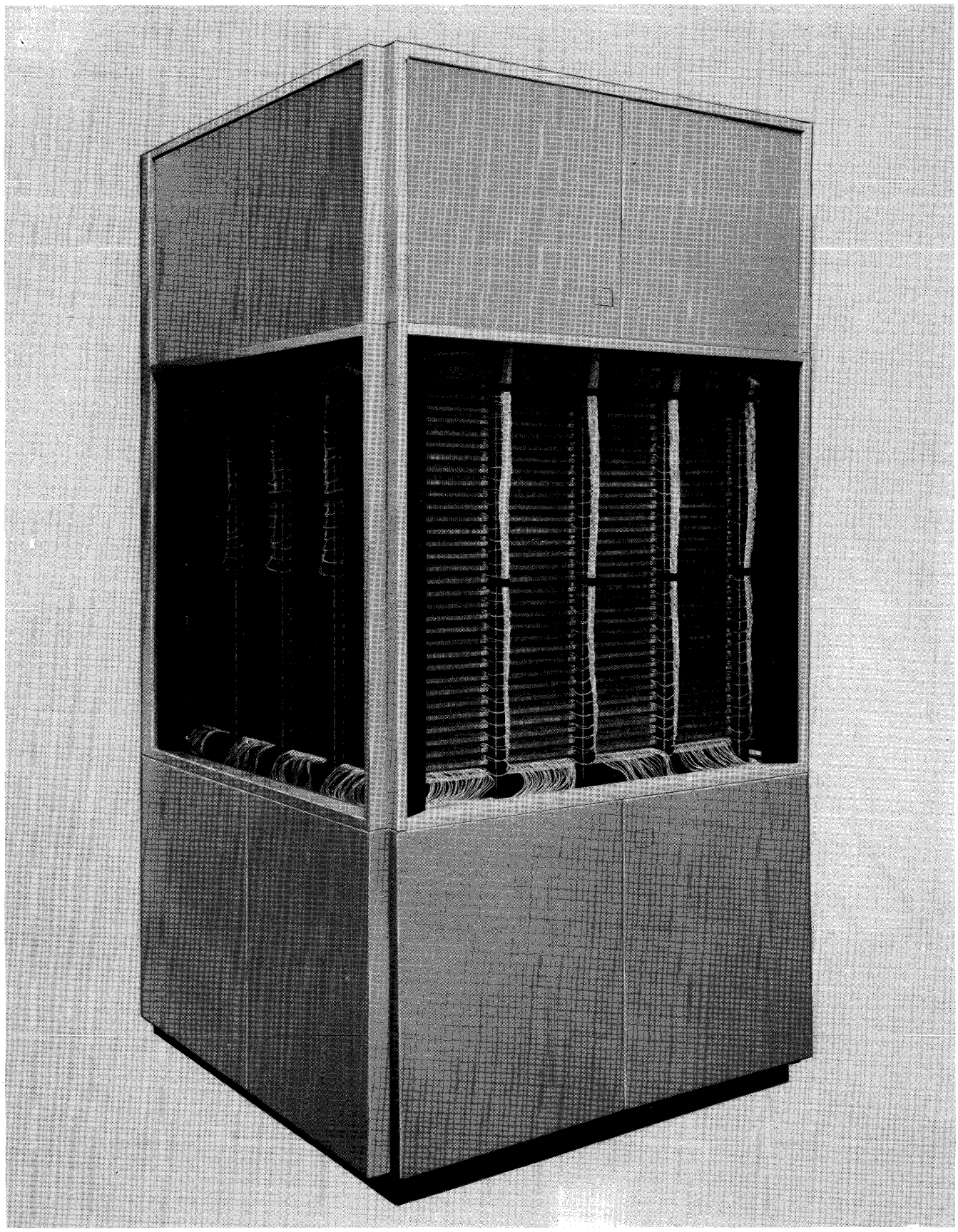
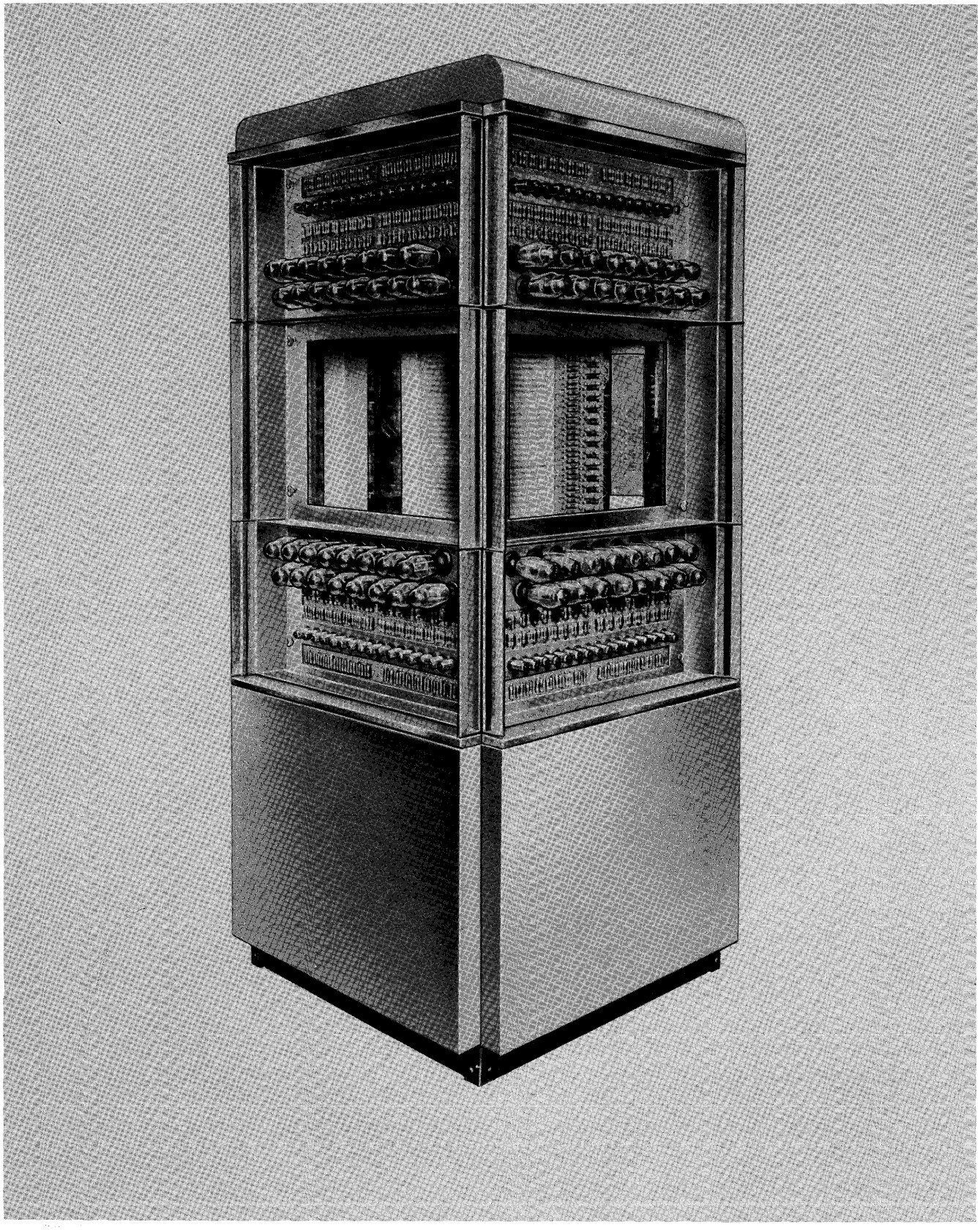


Figure 2–8. 256² Memory Unit

65,536 x 36 bits
64K memory



4096x36 bit memory
Figure 2–9. 64² Memory Unit

1.4.1.3 Test Memory

In addition to the core memories described above, the AN/FSQ-7 and AN/FSQ-8 each contain another storage device referred to as test memory. Test memory consists of 16 plugboard registers and two toggle switch registers located on the duplex maintenance console, and a flip-flop register located in the arithmetic element. Thus, there are 19 test memory registers that may be used. However, only 16 addresses have been reserved for test memory, so 16 is the maximum number of registers that can be used at any one time. The main purpose of test memory is to allow information to be entered directly into the memory element without resorting to punched cards, etc. Naturally, since only a limited number of addresses are available, most information entered in this manner is for maintenance purposes. The Central Computer System can read information out of any of the test memory registers at the normal rate of 6 μsec. When writing into test memory, the flip-flop register, commonly called the "live register," is always selected, regardless of which of the 16 available addresses is specified.

1.4.1.4 Clock Register

Although it is not actually used as a storage register, the clock register is considered an active memory device. It is located in the right arithmetic element and consists of 16 flip-flops which form a counting circuit. The clock register is pulsed every 1/32 of a second, and thus maintains accurate track of real time. The contents of this register are used when it is desired to use real time increments in various calculations.

1.4.1.5 Memory Buffer Register

The memory buffer register consists of 33 flip-flops which accommodate the left- and right-half portions of a computer word, plus the parity bit. (Refer to 1.2.) The function of this register is to provide temporary storage for a word being transferred into and out of a memory location. Temporary storage is necessary because the memory cycle is such that word transfers may be made faster than the arithmetic element can perform operations. Therefore, after a location is specified by a memory address, the contents of that location are transferred to the memory buffer register and remain there until ready for use. The parity check is also performed by the circuitry associated with this register and the proper bit placed in the P (parity) portion of the computer word.

1.4.1.6 Memory Address Register

The memory address register contains the circuitry that actually performs the memory register selection. This register receives information from one of three other registers which contain the address of the desired location. The memory address register then drives decoding matrices which, in turn, select the proper core windings and, finally, the proper register in core memory. Only one address may be handled at any one time by the memory address register; however, it is used to select both instructions and operands from memory.

1.4.1.7 Memory Addresses

All of the storage locations within core memory and test memory are assigned absolute addresses. This is necessary because we must know the location of both the instructions in a program and the operands to be used before the program can be correctly executed. Since the AN/FSQ-7 and AN/FSQ-8 have different size memories, the addresses assigned to them are necessarily different. Table 2–1 gives the octal addresses used in the AN/FSQ-7.

TABLE 2–1. AN/FSQ-7 MEMORY ADDRESSES

0.00000 – 1.77777	Memory 1
2.00000 – 2.07777	Memory 2
2.10000 – 3.77757	Illegal addresses
3.77760 – 3.77777	Test memory

The addressing system used in the AN/FSQ-7 provides for the inclusion of another memory the same size as memory 1. Therefore, the addresses listed above which are now illegal are reserved for the additional locations that will be available if another expanded memory is installed. In the meantime, if an illegal address is specified, the memory register actually selected will be the register in memory 2 corresponding to the last four octal digits in the original address. For example, if the illegal address 2.76355 was specified, the register selected would be 2.06355.

The AN/FSQ-8, which has no expanded memory at this time, utilizes a slightly different address designation. Table 2–2 lists the octal addresses which may be specified in the AN/FSQ-8.

TABLE 2–2. AN/FSQ-8 MEMORY ADDRESSES

0.00000 – 0.07777	Memory 1
0.10000 – 0.17777	Memory 2
0.00000 – 0.07777	Memory 1
0.60000	Clock register

We reserve an address for the clock register in the AN/FSQ-8 because originally there was no instruction that could specify the clock register. Although both the AN/FSQ-7 and AN/FSQ-8 can now select the clock

register through the use of a clock instruction, only the AN/FSQ-8 can select the clock register using a different instruction containing an address of 0.60000.

1.4.2 Instruction Control Element

The instruction control element accepts a portion of the instruction word from the memory element and decodes it. The decoder is capable of recognizing each of the various binary codes that indicate what operation is to be performed. More specifically, bits L1-L12 of the instruction word are transferred to a register, known as the operation register, which determines the class of instruction and instruction variation to be executed. The output of the operation register and its associated circuitry constitutes control signals which are sent to all the other elements of the Central Computer at specific times (depending on the instruction) and cause the instruction to be executed.

1.4.3 Program Element

1.4.3.1 General

The overall purpose of the program element is to supply the correct memory address to the memory address register so that the proper location in core memory may be selected. The program element is composed of the program counter, the address register, and four index registers.

1.4.3.2 Program Counter

The program counter is a flip-flop register which is responsible for keeping track of the location of instructions within a program. During normal operation of a program, the program counter will contain the address of the next instruction to be executed. Thus, during PT time, the contents of the program counter will be transferred to the memory address register, and the program counter will be stepped once. The contents of the program counter may also be changed when we wish to "jump" or branch to an instruction which is not in sequence within the program. In this case, the contents of the address register (which contains the location we wish to reach) are transferred to the program counter and then to the memory address register.

1.4.3.3 Address Register

The address register accepts the right-half instruction word during PT time and does the necessary decoding to provide the memory address register with signals. In the case of memory 1 for the AN/FSQ-7, the LS bit is also transferred to the address register.

1.4.3.4 Index Registers

The Central Computer System can perform many routine functions such as sorting, tabulating, table makeup, etc., by the use of programs which will repeat a certain number of instructions as often as necessary. When a program such as this, called an iterative program, requires data stored in sequential locations, the data can be obtained through the use of a single instruction and an associated index register. The index register may be loaded with any value up to 2^{16}, and each time the instruction is executed, the amount remaining in the index register is added to the contents of the address register, causing an address modification. For example, if we wished to add 100 numbers stored in sequential locations, it would not be necessary to use 100 instructions to add, but only one instruction specifying the first number. Then the index register (which is reduced for every repetition) is added to the address register each time another operand is required, enabling us to select all 100 numbers. The AN/FSQ-7 and AN/FSQ-8 each contain four index registers, plus the right accumulator of the arithmetic element which is sometimes used as a type of index register. Indexing is discussed in greater detail in Chapter 2 of this part.

1.4.4 Arithmetic Element

1.4.4.1 General

The actual computations which are specified by the instructions within a program are carried out by the arithmetic element. The arithmetic element in the AN/FSQ-7 and AN/FSQ-8 is a dual element, and the circuitry in each is identical. The left arithmetic element handles the left-half data word; the right arithmetic element handles the right-half data word. When an instruction is executed, the same action occurs in the two elements, enabling data processing to proceed at a higher rate of speed. It should be noted, also, that the arithmetic element will perform arithmetic operations on instruction words as well as data. If a memory register containing an instruction is to be added to a number already in the arithmetic element, the addition will take place in the normal manner, because the arithmetic element cannot distinguish between types of words. The main units in the arithmetic element are the A registers, adders, accumulators, and and registers.

1.4.4.2 A Registers

The A registers contain one of the operands used during arithmetic operation. The memory buffer register contents are transferred to the A register, and then the arithmetic process actually begins. Since all operations consist basically of addition, the A registers condition the adders.

1.4.4.3 Adders

Basically, the adders consist of various logical circuits which are capable of producing a sum and a carry for each bit added. The accumulator register supplies one of the bits to be added for each position; the A register supplies the other. The sum produced by each adder is transferred to the corresponding flip-flop of the accumulator registers.

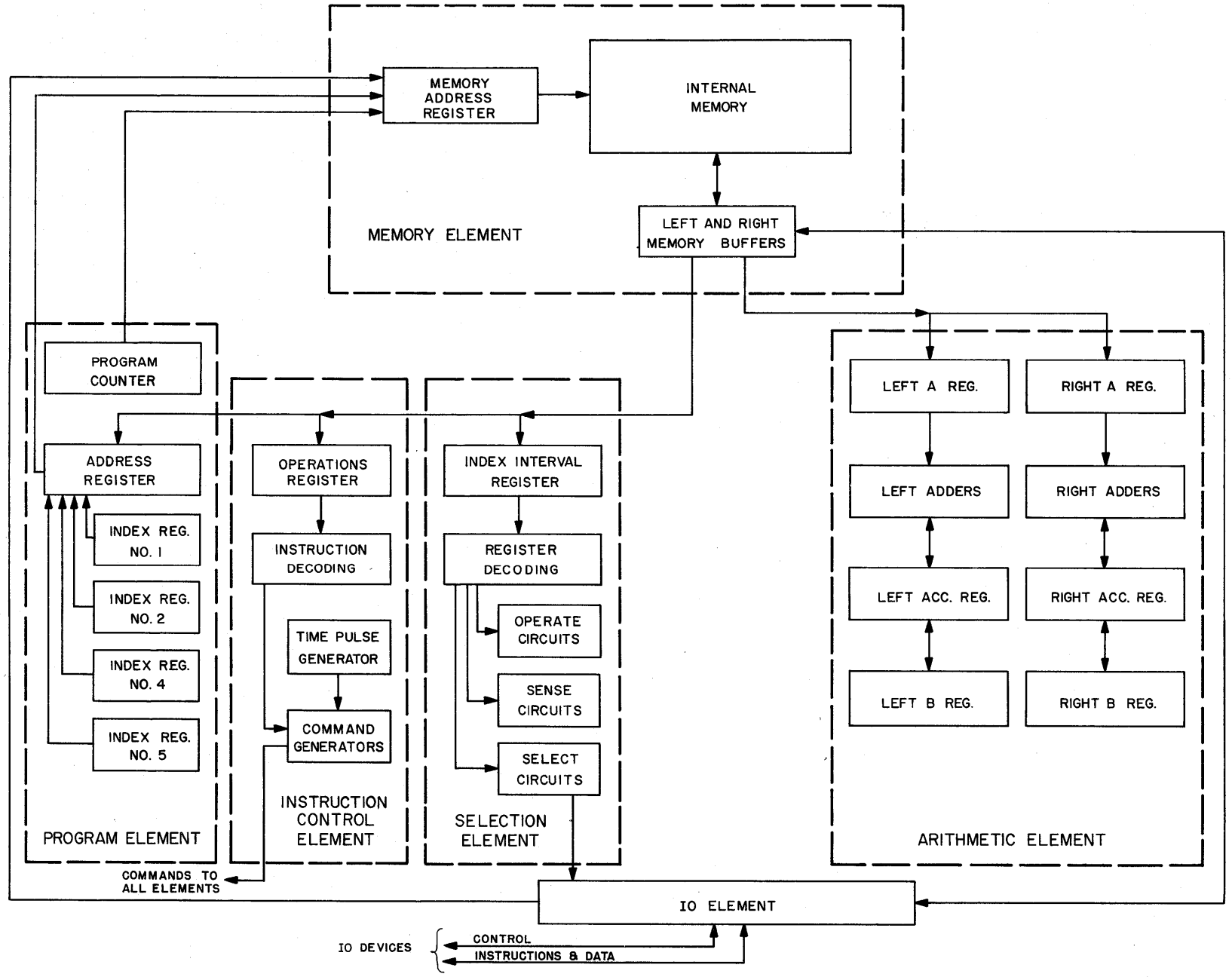


Figure 2–10. Overall Information Flow, Central Computer

1.4.4.4 Accumulators

As mentioned above, the accumulator registers contain the result of operations performed by the adders. Thus, the accumulators may contain a number which represents a sum of an addition, the difference after a subtraction, the most significant bits of a product (fig. 1–3), or the remainder after a division. The accumulators may also be used in the execution of several instructions that shift numbers already in the arithmetic element (such as rounding off a product).

1.4.4.5 B Registers

The B registers act as extensions of the accumulators during execution of instructions such as multiplication and division (fig. 1–3). These registers are made up of flip-flop circuits, just as the accumulators are; however, they are not associated with any adder circuitry. In addition to containing the least significant bits of a product or the magnitude of a quotient, the B registers can also perform shifting operations on their contents.

1.4.5 Selection Element

The selection element is composed of various groups of logical circuits which enable us to perform a number of operations on all the systems in the AN/FSQ-7 and AN/FSQ-8. For instance, we can start an input test pattern generator or cause the tape units to rewind. These are primarily unconditional commands; in each case, we are directing (or "operating") the computer to take some definite action. However, there are occasions when we wish to examine the status of various parts of the computer and take some sort of action, depending on the status. For example, if we want to print something with the printer, we must first check to see whether the printer is ready for use. This is known as "sensing" the status of the printer. If the printer is not ready, we can take corrective action, depending on what options we have made available to the computer through the program. The third operation that the selection element performs is to determine whether an IO unit has been selected for operation and, if such is the case, to condition the IO element.

1.4.6 IO Element

Once this element receives a signal from the selection element indicating that we wish to read or write using an IO unit, it takes over the task of performing this operation. Several of the functions of this element are as follows:

a. Controlling the amount of information that is transferred.
b. Determining where in memory we read or write the information.
c. Determining what IO unit is involved.
d. Acting as a buffer storage device between the IO unit and internal memory.

1.4.7 Overall System Information Flow

The main transfer paths for both data and instructions within the Central Computer System are shown in figure 2–10. Not all the paths have been shown; however, those which connect the registers and circuitry primarily responsible for decoding and the execution of instructions are indicated. The IO element has not been broken down to show any of its operational registers because it is not involved in any of the basic instructions we are about to study. This element will be discussed more fully when IO programming and IO instructions are considered.

CHAPTER 2
BASIC INSTRUCTIONS

2.1 GENERAL

As previously stated, a program is a series of instructions which control the operations of a computer. Each instruction is used to cause some action which is a part of the overall task we wish to perform. Therefore, we say that an instruction is the basic building block of a computer program.

An efficient program makes full use of the instructions which are available to accomplish the task in the shortest possible time and uses the least number of instructions. In most cases, one criterion, either time or the number of instructions, has to be chosen over the other, and the program is developed along this line. If time is important, we try to write a program which uses instructions of short duration but may use quite a few memory locations for storage. On the other hand, if time is relatively unimportant, but only a few restricted locations are available, we must then choose instructions which do a number of things or will cause the computer program to run through the same routine more than once. Later, we shall see how two different programs can be written to perform the same task, one being fast in execution time but the other requiring less memory space.

From the above discussion, it is apparent that to write a satisfactory program it is necessary to have a thorough knowledge of the instructions we can use. This includes execution time, the overall purpose of the instruction, when the instruction may be used, and the state of the computer after the instruction has been carried out. In addition, we should know whether the instruction can be indexed and what internal conditions must be satisfied before it can be executed. The following text describes 17 basic instructions that are used in the AN/FSQ-8. Each description contains the information listed above, and program examples of the instructions are given. Since most problems have several possible solutions, the program given for a particular problem may not represent the most efficient way of arriving at an answer. Rather, the programs are designed to show the application of individual instructions within a program.

2.2 *HALT* INSTRUCTION

The *Halt (HLT)* instruction causes the computer to stop executing instructions under program control. However, any operation which is in progress at the time the *HLT* instruction is decoded will be completed first. For example, if we are reading information into memory from a deck of 150 punched cards, all 150 cards will be read before the computer halts, even though the *HLT* instruction may have been issued just after the reading operation began. This instruction requires 12 μsec to execute and is designated by an octal code of 000 (bits L4–L10). The address portion of the *HLT* instruction is not used; therefore, indexing is not possible. When the computer is halted by this instruction, the program counter contains the address of the instruction immediately following, so that restarting the computer will cause this next instruction to be executed.

2.3 *CLEAR AND ADD* INSTRUCTION

The *Clear and Add (CAD)* instruction is used to enter a quantity into the accumulators from memory without changing the sign or magnitude of the words. This instruction is usually used when it is desired to begin a type of addition problem. The accumulators are first cleared, and then the location specified by the address portion of the *CAD* instruction is transferred to the A registers. Then an actual addition between the A registers and accumulators is started; however, since the accumulators are cleared to +0, this addition has the overall effect of transferring the word from memory into the accumulators unchanged. The memory location used is unchanged, and the A registers are cleared to +0 after execution of the *CAD* instruction. An octal code of 100 is used to designate a *CAD* instruction, and it may be indexed. Execution time for this instruction is 12 μsec.

2.4 *ADD* INSTRUCTION

This instruction is similar to the *CAD* instruction except that it does not provide for clearing the accumulators before the addition process begins. Thus, the *ADD* instruction will generate the sum of the word contained in the specified memory address and anything that may be in the accumulators. This sum is placed in the accumulators, and the A registers are cleared to +0. The *ADD* instruction requires 12 μsec for execution and may be indexed. The octal code for this instruction is 104. It should be noted that the *ADD* instruction can cause an overflow if the numbers added together are sufficiently large. If this happens, the result

in the accumulator is meaningless. Because the arithmetic elements are dual, an overflow may occur in one accumulator and not the other; however, overflow in both accumulators may occur as a result of the same *ADD* instruction. Later on, we shall see how this condition (overflow) can be dealt with by a computer program.

2.5 *FULL STORE* INSTRUCTION

The *Full Store (FST)* instruction is used to transfer words from the accumulators into a memory location specified by the address portion of the instruction. The left accumulator is stored in the left half-word and the right accumulator is stored in the right half-word. Thus, this instruction enables us to place the results of any operation performed by the arithmetic element into memory for future use. The contents of the specified register are first cleared, and then the contents of the accumulators are stored in via the memory buffers. However, the accumulators remain unchanged by this instruction. Execution time for this instruction is 12 μsec, and it may be indexed. The *FST* instruction is designated by an octal code of 324.

2.6 SAMPLE PROGRAMS INVOLVING ADDITION

Now we have learned enough instructions *(HLT, CAD, ADD,* and *FST)* to solve a basic and simple problem involving addition only. However, the programming principles presented in this problem are the same as those outlined previously; i.e., to choose the proper instruction and place it in the proper sequence in our program. Assume that the problem is to add several sets of quantities together and store the results for future use. Before we can write a program, we must know where in memory those quantities are stored initially; we must also know where to place their sum. In addition, we must have memory locations available for storage of the computer program itself. For ease of explanation, assume that memory locations 0_8– 10_8 are available for the program; locations $100_8 - 200_8$ are available for data, including the result. Further assume that the quantities we wish to add are contained in memory locations 100_8, 125_8, and 135_8. Let the quantities be designated A, B, C, D, E, and F, and have them located as follows:

a. Memory location 100_8 contains A in the left half-word and B in the right half-word.

b. Memory location 125_8 contains C in the left half-word and D in the right half-word.

c. Memory location 135_8 contains E in the left half-word and F in the right half-word.

Now we are ready to solve the problem. The program used is given in table 2–3.

TABLE 2–3. BASIC ADDITION PROGRAM

LOCATION	OPERATION	ADDRESS
0.00000	*CAD*	0.00100
0.00001	*ADD*	0.00125
0.00002	*ADD*	0.00135
0.00003	*FST*	0.00150
0.00004	*HLT*	–

Notice that the octal notation is used when referring to memory locations; this method simplifies the reading of the program, as previously explained. Now let us take the first instruction in the program and see what it does.

The instruction itself is located in memory location 0.0000, and it says to *CAD* location 0.00100. This will cause the accumulator to be cleared to positive zero and the contents of 0.00100 to be added in. Since we specified that 0.00100 contains A, B, the left accumulator will contain A and the right accumulator will contain B after execution of the *CAD* instruction. It should now be obvious that the *CAD* instruction is the best instruction with which to start our program, since our problem is to add numbers together, and the *ADD* instruction would not suffice since it is possible (and probable) that the accumulators would not be cleared at the start of the program. Thus, using an *ADD* instruction might lead to a result we do not want. The next instruction, which is located in 0.0001, tells us to *ADD* 0.00125. After execution of this instruction, the left accumulator contains A + C and the right accumulator contains B + D. Remember that simultaneous addition is carried on in both arithmetic elements. The third instruction, *ADD* 0.00135, will add in E and F so that the left accumulator contains A + C + E and the right accumulator contains B + D + F. Now the result we wished to obtain is in the accumulators. However, to be of any real value, the result must be stored in memory, so the fourth instruction tells the computer to *FST* these sums in 0.00150. Now location 0.00150 contains A + C + E in the left half-word and B + D + F in the right half-word, regardless of what quantity may have been there previously. Of course, these sums are also still contained in the accumulators. The last instruction in our program is *HLT*, which causes the computer to stop operation.

This problem has been simple, and no doubt the solution was obvious from the beginning. However, most problems do not offer such straight forward methods of solution. In some cases, it is advantageous to add one quantity before another, although in the above example it made no difference which quantities were initially placed in the accumulators, since the end result

would have been the same. In addition, the choice of memory location 0.00150 for the result was arbitrary, the only restriction being that it had to be somewhere between locations 0.00100 - 0.0200, as mentioned before. In many cases, however, the results of computation have to be in a particular location or locations because another program or portion of the same program may refer to that location on the assumption that the proper quantities have been placed there. The following example shows how a program may refer to a location that has just been used for storage. Assume that we wish to obtain the sums 2A + 2C, 2B + 2D, using the same data locations as those given in the first program. The program to obtain these sums is given in table 2–4.

TABLE 2–4. MEMORY REFERENCE PROGRAM

LOCATION	OPERATION	ADDRESS
0.00000	*CAD*	0.00100
0.00001	*ADD*	0.00125
0.00002	*FST*	0.00150
0.00003	*ADD*	0.00150
0.00004	*FST*	0.00150
0.00005	*HLT*	—

Of course, this is not the only method of solution, merely one of the possible ones. However, it does show how one memory location may be used several times in the same program. The first instruction places A, B in the accumulator. The second instruction, *ADD,* will leave the sum A + C, B + D in the accumulators. Now we *FST* these sums in 0.00150, with the result that A + C, B + D is now in both the accumulators and the specified memory location. The fourth instruction adds the sums we have just stored, leaving the desired result 2A + 2C, 2B + 2D in the accumulators. Then we *FST* in 0.00150 again and *HLT.*

2.7 *CLEAR AND SUBTRACT* INSTRUCTION

The *Clear and Subtract (CSU)* instruction is used to enter a quantity into the accumulators in complemented form. This is accomplished in much the same manner as the *CAD* instruction. The accumulators are cleared to +0, and the contents of the register specified in the address portion of the instructions are transferred to the A registers. However, the A registers are then complemented, which results in having the original contents of the selected memory register in negative form. Then a normal addition takes place between the A registers and the accumulators, placing the complemented number in the accumulators. The *CSU* instruction requires 12 μsec to execute and will not cause a computer overflow. An octal code of 130 designates the *CSU* instruction, which may be indexed.

2.8 *SUBTRACT* INSTRUCTION

A *subtract (SUB)* instruction is used to subtract the contents of the selected memory register from the accumulators. Again, we employ the normal addition process between the A registers and the accumulators after first complementing the A registers, which contain the contents of the specified register. The accumulators are not cleared, however, so that the result will be the difference between whatever is in the accumulators and the A registers. Execution time of the *SUB* instruction is 12 μsec, and it may be indexed. The octal operation code is 134, and it should be noted that the *SUB* instruction may cause a computer overflow.

2.9 *TWIN AND ADD* INSTRUCTION

The *Twin and Add (TAD)* instruction causes the left half portion of the contents of the specified memory register to be added to both the left and right accumulators. The right half of the data word is not used at all, otherwise this instruction is identical in execution to the *ADD* instruction. The octal operation code for the *TAD* instruction is 110, and it may be indexed. Since the same value is added to both accumulators, overflow may occur in either or both accumulators, depending on their original contents. The *TAD* instruction requires 12 μsec to execute.

2.10 *TWIN AND SUBTRACT* INSTRUCTION

The *Twin and Subtract (TSU)* instruction is employed to subtract the left half portion of the specified memory register contents from both the left and the right accumulators. This instruction is similar in execution to the *SUB* instruction. The left half word is transferred to both A registers where it is complemented and then added to the accumulators. The difference appears in the accumulators, and just as with the *TAD* instruction, overflow may occur in either or both accumulators as a result of the *TSU* instruction. The octal operation code that designates a *TSU* instruction is 140, and requires 12 μsec to execute. This instruction may also be indexed.

2.11 SAMPLE PROGRAMS INVOLVING SUBTRACTION AND TWINNING

Now that we have learned enough instructions to cover two of the four basic arithmetic processes, it is possible to write programs which will solve problems that are more complex than those given previously. In the following examples, the locations of data are listed at the end of the tables containing the program. Assume that our programs will always start at location 0.00000 and that we have up to 0.00050 available for the instructions. The first problem is to compute two sets of values;

namely, A — C + 2E, B — D + E + F and A — C + 2E, 2A — B — 2C + D + 3E — F. The program is executed as shown in table 2–5.

TABLE 2–5. SAMPLE SUBTRACTION PROGRAM

LOCATION	OPERATION	ADDRESS	
0.00000	*CSU*	0.00151	
0.00001	*ADD*	0.00150	
0.00002	*ADD*	0.00152	
0.00003	*TAD*	0.00152	
0.00004	*FST*	0.00200	
0.00005	*CSU*	0.00200	
0.00006	*FST*	0.00250	
0.00007	*TAD*	0.00200	
0.00010	*TSU*	0.00250	
0.00011	*FST*	0.00250	
0.00012	*HLT*		
0.00150	A	B	Data
0.00151	C	D	
0.00152	E	F	
0.00200	First result		
0.00250	Second result		

2.12 *LEFT STORE* INSTRUCTION

The *Left Store (LST)* instruction will place the contents of the left accumulator into the left half portion of the specified memory register. The right half portion of this register is not changed, nor are the accumulators. The octal operation code of the *LST* instruction is 330, and it may be indexed. Execution time is 18 μsec. At this point, the reader may wonder why the *LST* instruction should require 18 μsec for completion when the *FST* instruction requires only 12 μsec. The reason is simply this: During execution of an *FST* instruction, it is not necessary to preserve any part of the register involved, so only one OT cycle is required to read the contents out and place the new contents into memory. An *LST* instruction must preserve the right half-word of the selected register, so two OT cycles are required, one to read out the contents of the selected register, another to replace the left half portion of the register and then write both this changed portion and the original contents of the right half-word back into memory. This is accomplished by reading the original contents of the memory register into the A registers during the OTA cycle. The left accumulator is transferred to the left memory buffer register and the right A register is transferred to the right memory buffer register, so that during the OTB cycle, these quantities are stored in memory.

2.13 *RIGHT STORE* INSTRUCTION

The *Right Store (RST)* instruction is similar to the *LST* instruction except that it involves the opposite half of the dual arithmetic element. Its function is to replace the right half of the selected register with the contents of the right accumulator. This is accomplished in the same manner as the *LST* instruction just described. An octal code of 334 is used to designate an *RST* instruction, and it may be indexed. The execution time for the *RST* instruction is 18 μsec.

2.14 SAMPLE PROGRAMS INVOLVING HALF-WORD STORAGE

The use of the *LST* and *RST* instructions can best be illustrated through the use of some program samples. Remember that the solutions given for a particular problem do not represent the only method of solution, but merely one of the possible ones. Our first problem is to compute the following: A — C + E, B — C + E + 2F, 3E — C, E — C + 2F, 6E — 2C, E — C + 2F. The program to compute these values is given in table 2–6.

TABLE 2–6. SAMPLE PROGRAM USING HALF-WORD STORE INSTRUCTIONS

LOCATION	OPERATION	ADDRESS	
0.00000	*CAD*	0.00075	
0.00001	*ADD*	0.00077	
0.00002	*TSU*	0.00076	
0.00003	*LST*	0.00430	
0.00004	*TAD*	0.00077	
0.00005	*ADD*	0.00077	
0.00006	*RST*	0.00430	
0.00007	*SUB*	0.00075	
0.00010	*FST*	0.00440	
0.00011	*RST*	0.00450	
0.00012	*ADD*	0.00440	
0.00013	*LST*	0.00450	
0.00014	*HLT*		
0.00075	A	B	Constants
0.00076	C	D	
0.00077	E	F	

TABLE 2–6. SAMPLE PROGRAM USING HALF-WORD STORE INSTRUCTIONS (cont'd)

LOCATION	OPERATION	ADDRESS
0.00430	First result	
0.00440	Second result	
0.00450	Third result	

Here, we can see the advantage of using the *LST* and *RST* instructions. For instance, at program step 0.00003, we used the *LST* instruction to store the value A — C + E. However, this in no way affected the contents of the accumulators, so we were able to proceed to step 0.00006 where the proper value for the right half-word was arrived at and stored. The last two results called for the same value in their right half-words, so after an *FST* instruction at 0.00010, an *RST* instruction was given to place the desired value in another location. Notice that the third result required a value in the left half-word that was exactly twice the value of the left half-word in the second result. This could have been computed by a series of *ADD* and *SUB* instructions; however, it was much more convenient to merely add the second result to itself, which was done at step 0.00012. This is a valid procedure and in no way affects the second result; we are merely using it as another constant in this case. The sum in the right accumulator is of no importance either, since the proper value for the right half-word is already stored; all that is necessary is the *LST* instruction given in program step 0.00013.

Another example involving the use of *LST* and *RST* instructions is given below. We wish to compute the following values: — A, C — B + D and + 0, 2D. The program to compute these values is given in table 2–7. Notice that in program step 0.00005 we used a *CAD* instruction, giving the address of the *HLT* instruction as the desired memory location. Since we know that the octal code of an *HLT* instruction is 000, this means that the left half portion of the *HLT* instruction is +0. Therefore, we can use the *HLT* instruction as a constant of 0, as has been done here. It is a valid programming technique to perform arithmetic operations on instructions since they are read out of memory during an OT cycle, in this case, and are not decoded, but treated as straight binary numbers. As a rule, it is not necessary to use instructions in this manner; however, it should be remembered that they are simply numbers in internal memory and do not have any significance to the Central Computer unless they are decoded during a PT cycle.

TABLE 2–7. ADDITIONAL EXAMPLE OF HALF-WORD STORAGE

LOCATION	OPERATION	ADDRESS
0.00000	*CSU*	0.10000
0.00001	*LST*	0.22050
0.00002	*ADD*	0.20000
0.00003	*TAD*	0.20000
0.00004	*RST*	0.22050
0.00005	*CAD*	0.00012
0.00006	*LST*	0.22051
0.00007	*CAD*	0.20000
0.00010	*ADD*	0.20000
0.00011	*RST*	0.22051
0.00012	*HLT*	0.00000
0.10000	A	B } Constants
0.20000	C	D } Constants
0.22051		First result
0.22050		Second result

2.15 BRANCH INSTRUCTIONS

2.15.1 General

Up to this point we have discussed only those instructions which specify that definite action is to take place, such as *ADD, FST, HLT,* etc. However, if we limited the AN/FSQ-7 and AN/FSQ-8 to execution of this type of instruction only, it would seriously limit the capabilities of the machine. What we need are instructions that can examine the computer and decide what to do, depending on the condition we are looking for. These instructions are known as Branch instructions, and the first ones we will study involve checking the accumulators for one condition or another. Branching simply involves transferring the program control to the address specified in the right half-word of the Branch instruction if the condition we are checking for is met. Branch instructions constitute some of the most powerful instructions that can be placed in a computer program, and they allow us to put a great deal of flexibility into various computer programs.

2.15.2 *Branch on Left Minus* Instruction

The *Branch on Left Minus (BLM)* instruction examines the sign bit of the left accumulator and branches to the location specified in the address portion of the instruction if the sign bit contains a 1 bit. When the

branching condition is met, the program counter, which has already been stepped, is transferred to the right A register, and the address register is transferred to the program counter. Thus, at the completion of a *BLM* instruction (assuming the branch condition is met), the program counter will contain the location of the next instruction we desire to execute. If the sign bit of the left accumulator is positive, the branch condition has not been met, and the next instruction executed will be the one immediately following the *BLM* instruction.

The accumulators are not affected by the execution of this instruction. Execution time of the *BLM* instruction is 6 μsec, and it cannot be indexed. The octal operation code for this instruction is 550.

2.15.3 *Branch on Right Minus* Instruction

The *Branch on Right Minus (BRM)* instruction checks the sign bit of the right accumulator and branches to the memory location designated by the address portion of the instruction if the right sign bit is negative. As with the *BLM* instruction, the program counter is transferred to the right A register, the address register is transferred to the program counter, and the accumulators are left unchanged. If the sign of the right accumulator is positive, the program executes the next sequential instruction. An octal code of 554 designates a *BRM* instruction which requires 6 μsec to execute. This instruction cannot be indexed.

2.15.4 Table Construction Program

Following is a program which illustrates the use of the *BLM* and *BRM* instructions. Assume that the problem is to examine both halves of a data word stored in memory and to construct a table of all negative values, using right half-words, beginning at location 0.00100. We do not know the values of the left and right portions of the data word. The solution to this problem may be rather simple; however, when writing programs of the type that involves a number of decisions, it is useful to construct a program flow chart. This chart is actually a graphic representation of what is desired and depicts a method of arriving at a solution. The technique is to separate the problem into a number of blocks, each of which contributes to the solution. Then the actual programmed instructions that can be used to perform the function of each block are determined, and the end result is the finished program. The problem given above will be laid out using the flow chart method.

2.15.4.1 Preliminary Flow Chart

The problem stated that we are to examine a data word for negative quantities and construct a table from these quantities, if any are present. In addition, we know that this table is to start at location 0.00100 and that we must use the right half-word for storage. With this much information, we can lay out a preliminary flow chart. This chart should resemble the one shown in figure 2–11.

2.15.4.2 Final Flow Chart

From the examination of figure 2–11, it should now be obvious that we can use the *BLM* and *BRM* instructions to arrive at a solution to this program. There are four possible arrangements of the signs within a data word, and this flow chart will handle all of them. Now we can proceed to fill in each of the blocks as done in figure 2–12. For ease of explanation, assume that the data word we are interested in is located at address 0.00050 and that it contains the quantities x and y in the left and right-half words, respectively. A listing of this program is also given in table 2–8.

TABLE 2–8. TABLE CONSTRUCTION PROGRAM

LOCATION	OPERATION	ADDRESS
0.01000	*CAD*	0.00050
0.01001	*BLM*	0.01004
0.01002	*BRM*	0.01011
0.01003	*HLT*	0.00000
0.01004	*CAD*	0.01003
0.01005	*TAD*	0.00050
0.01006	*RST*	0.00100
0.01007	*CAD*	0.00050
0.01010	*BLM*	0.01002
0.01011	*RST*	0.00101
0.01012	*CAD*	0.00100
0.01013	*BRM*	0.01003
0.01014	*CAD*	0.00101
0.01015	*RST*	0.00100
0.01016	*BRM*	0.01003
0.00050	x	y
0.00100		First value
0.00101		Second value

Note that figure 2–12 specifies that we are checking x and y to see if they are smaller than +0. This is necessary because −0 will contain a sign bit of 1 and will satisfy the branching conditions. Therefore, it is possible that one or both of the values could be −0 and not actually numbers with an absolute magnitude of less than 0. Later, we shall see how another branching instruction can be used to determine whether the quantities being tested are really negative numbers.

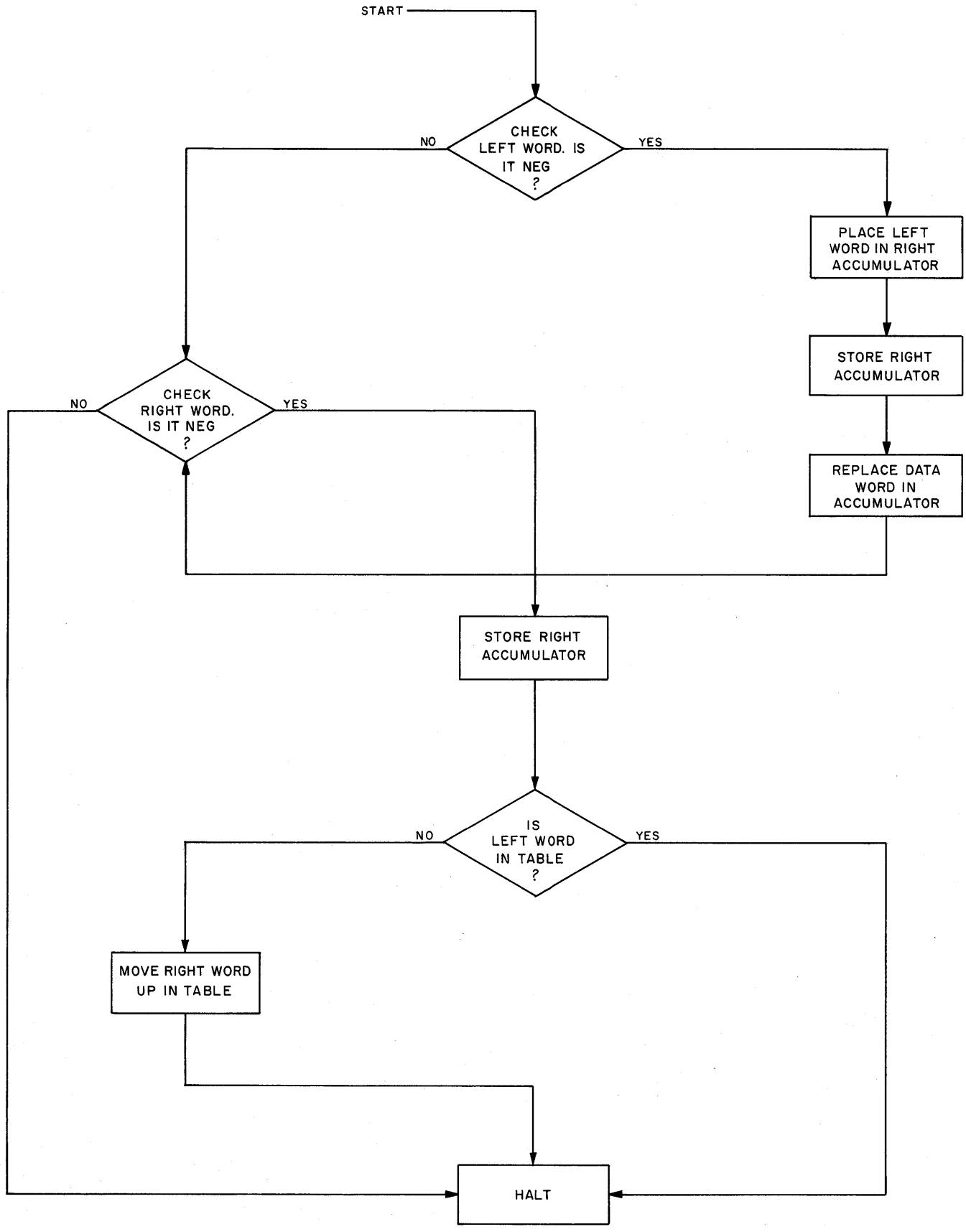


Figure 2–11. Preliminary Flow Chart, Table Construction Program

2.15.5 Number-Sorting Program

As another problem involving the use of branching instructions, let us consider three numbers of unknown magnitude. We will assume that the numbers are all positive integers and that they are stored in the left half portions of the data words concerned. We want to write a program which will sort through these three numbers, select the largest, and store it in a specified location. Here, again, the best solution to this problem is to use the flow chart method previously illustrated.

2.15.5.1 Preliminary Flow Chart

We know that our problem is to sort through three numbers and select the largest. Since we do not know the magnitude of the numbers, it is necessary to compare all three numbers against each other. This involves making two comparisons, determining which number is larger at each comparison, and finally storing the correct value. The flow which graphs this solution is shown in figure 2–13.

2.15.5.2 Final Flow Chart

The final flow chart for this program is shown in figure 2–14. It can be seen that the use of the *BLM* instruction has enabled us to determine quite rapidly what is the largest number and to store it. Notice that there are two separate ways in which the third number compared can be found to be the largest number. Since

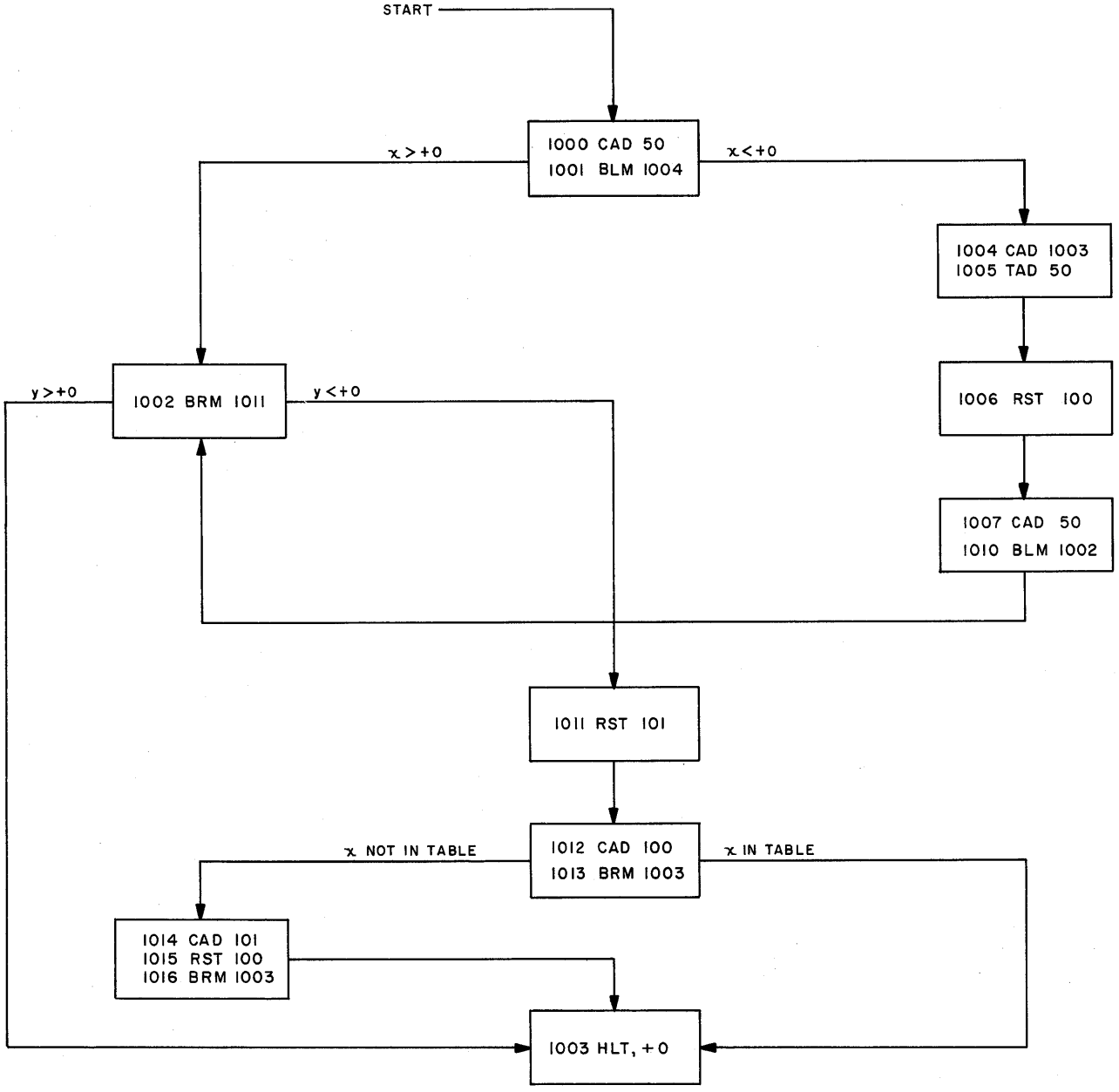


Figure 2–12. Final Flow Chart, Table Construction Program

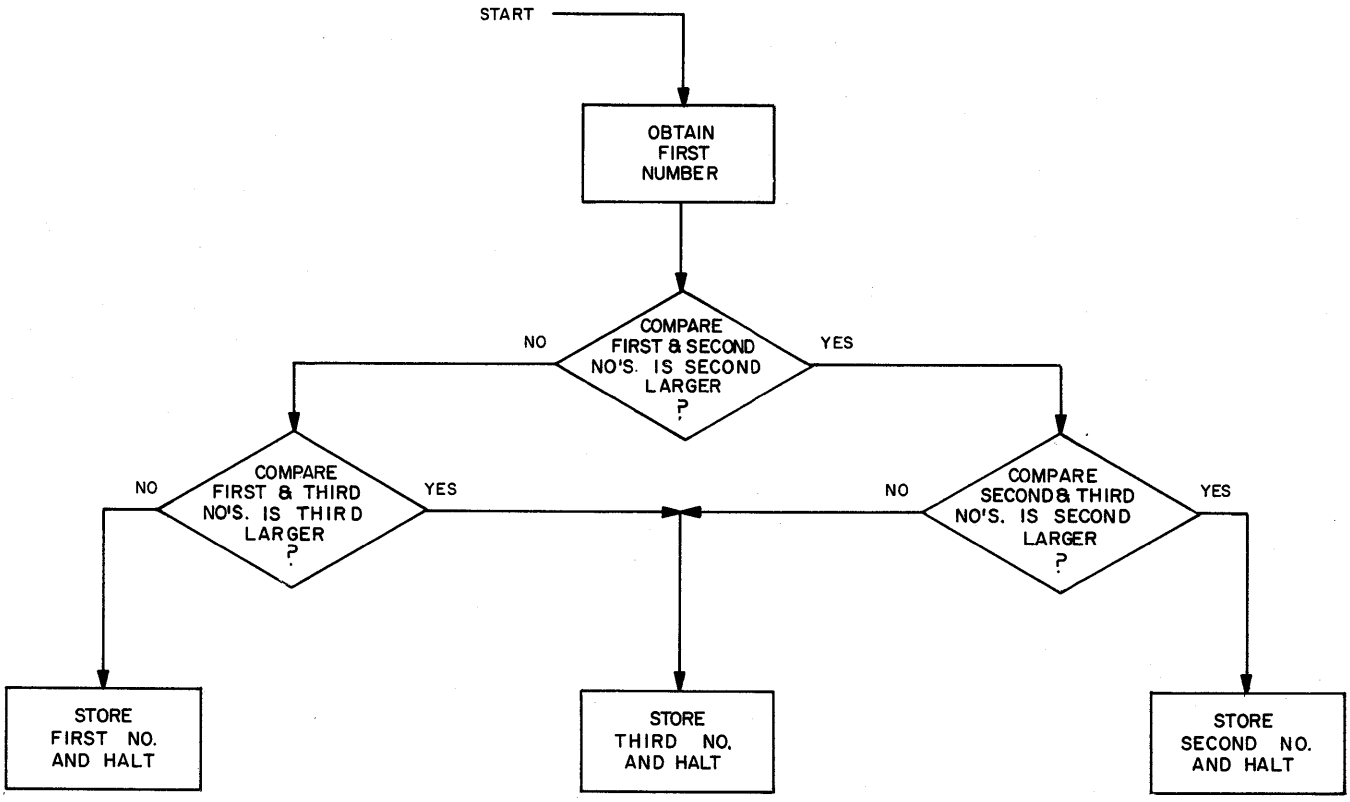


Figure 2–13. Preliminary Flow Chart, Number-Sorting Program

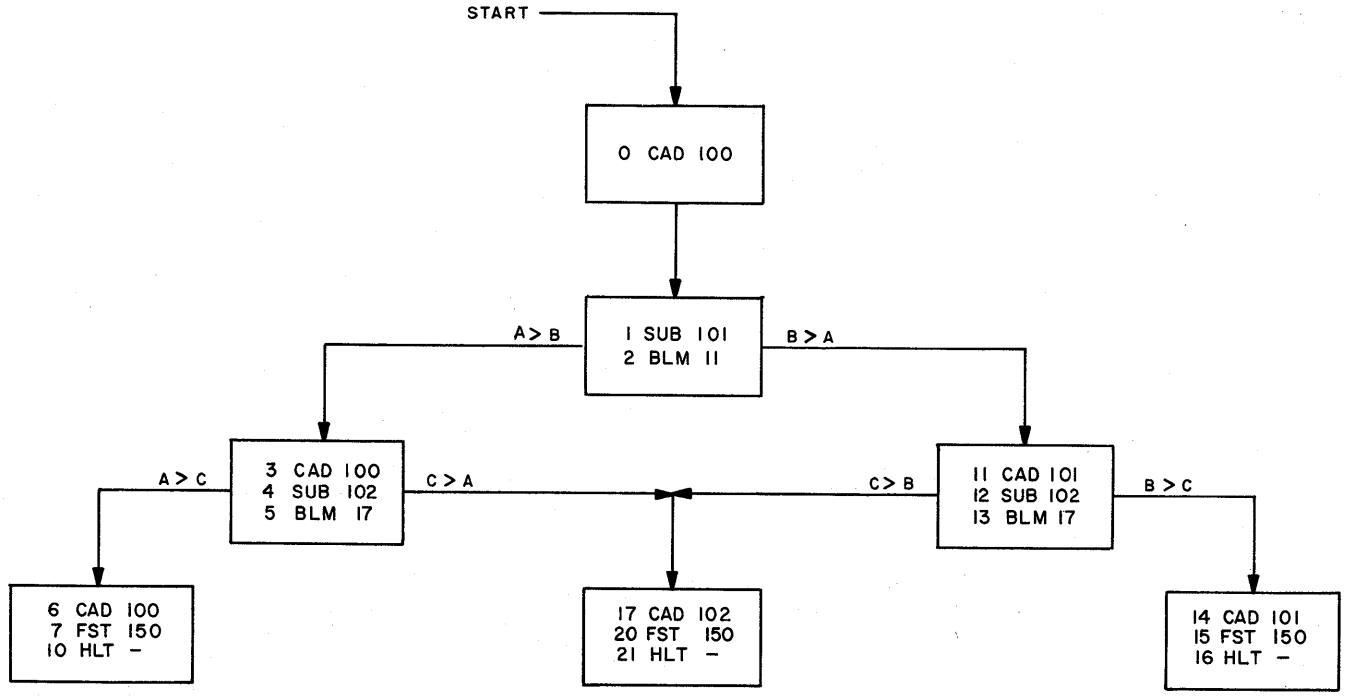


Figure 2–14. Final Flow Chart, Number-Sorting Program

this is the case, we can use one group of instructions to place this number in the desired memory location, and allow entry to this group from two separate points. For purposes of explanation, we have assumed that the numbers to be compared are A, B, and C and that they are in memory locations 0.00100, 0.00101, and 0.00102, respectively. Location 0.00150 is used to store the result of the comparison. This program is listed in table 2–9.

TABLE 2–9. NUMBER-SORTING PROGRAM

LOCATION	OPERATION		ADDRESS
0.00100	*CAD*		0.00100
0.00001	*SUB*		0.00101
0.00002	*BLM*		0.00011
0.00003	*CAD*		0.00100
0.00004	*SUB*		0.00102
0.00005	*BLM*		0.00017
0.00006	*CAD*		0.00100
0.00007	*FST*		0.00150
0.00010	*HLT*		—
0.00011	*CAD*		0.00101
0.00012	*SUB*		0.00102
0.00013	*BLM*		0.00017
0.00014	*CAD*		0.00101
0.00015	*FST*		0.00150
0.00016	*HLT*		—
0.00017	*CAD*		0.00102
0.00020	*FST*		0.00150
0.00021	*HLT*		—
0.00100	A		—
0.00101	B	Data	
0.00102	C		—
0.00150			Result

This type of program is commonly referred to as a logical program because, strictly speaking, no arithmetic operations are performed on the data. We use arithmetic processes to determine which of the three numbers is the largest; however, no computation is made with this number once we know what it is. By comparison, a program which utilizes data to evaluate a function is referred to as an arithmetic program, since the data itself does not represent an answer but merely part of the information needed to compute an answer.

2.15.6 *Branch on Full Minus* Instruction

The *Branch on Full Minus (BFM)* instruction checks the signs of both the left and right accumulator and branches to the memory location specified by the address of the instruction if both accumulators are negative. Thus, it can be seen that this instruction is a combination of the *BLM* and *BRM* instructions. As with these other instructions, only the sign bits of the accumulators are checked, and if they both contain a 1, the branching condition is met. When this is the case, the contents of the program counter are transferred to the right A register, and the address portion of the *BFM* instruction is transferred to the program counter. If the branching condition is not met, the program counter will select the next instruction in sequence. Execution time of the *BFM* instruction is 6 μsec, and it is not indexable. This instruction is designated by an octal operation code of 544.

2.15.7 *Branch on Full Zero* Instruction

The *Branch on Full Zero (BFZ)* instruction checks the contents of both accumulators to see if they are 0. Since the AN/FSQ-7 and AN/FSQ-8 utilize both a positive and a negative zero, there are four combinations of positive and negative zero which will satisfy the branching conditions. These combinations are given in table 2–10.

TABLE 2–10. COMBINATIONS SATISFYING THE *BRANCH ON FULL ZERO (BFZ)* CONDITIONS

LEFT ACCUMULATOR	RIGHT ACCUMULATOR
0.00000	0.00000
1.77777	1.77777
0.00000	1.77777
1.77777	0.00000

The execution of this instruction is carried out in the following manner. First, the contents of both accumulators are made positive if they are not already so. Then the accumulators are complemented and made negative. A 1 is added to both accumulators by initiating a carry of 1 at bit 15, and the carry-out of the most significant bit is disregarded. If the accumulators contained either positive or negative zero, complementing and then adding 1 without the end carry would have rippled through the accumulator bits and cleared them to 0. At this point, the accumulators are complemented again, and the sign bits are checked. If the sign bits contain 1 bits, we know that the original values were

TABLE 2–11. EXECUTION OF *BRANCH ON FULL ZERO (BFZ)* INSTRUCTION

LEFT ACCUMULATOR	RIGHT ACCUMULATOR	ACTION
1.77777	0.00000	Initial starting point
*0.00000	*0.00000	Make accumulators positive
1.77777	1.77777	Complement accumulator
0.00000	0.00000	Add "1;" no end carry
1.77777	1.77777	Complement accumulator
1.77777	1.77777	Test, branch conditions met
0.00000	0.00000	Add "1;" no end carry
*1.77777	*0.00000	Restore accumulators

**These steps conditional, depending on sign of accumulator.*

either positive or negative zeros, and the branching condition has been satisfied. As with all branching instructions that find their conditions satisfied, the *BFZ* instruction will transfer the program counter to the right A register and the address portion of the instruction itself to the program counter if both sign bits are 1. Due to the number of operations involved, an OT cycle is required for the *BFZ* instruction steps described thus far. After the test of the sign bits has been made, the "carry 1" line is again pulsed, and the accumulators may be complemented, to restore them to their original values. Execution time for this instruction is 12 μsec; the instruction is designated by a code of 540. The *BFZ* instruction cannot be indexed. Table 2–11 shows the various steps encountered during the execution of a *BFZ* instruction. In this case, we have assumed that both accumulators are zero but have different algebraic signs.

By performing the above steps on a number not positive or negative zero, the reader may easily see that the sign of the accumulator is always positive at the time of test. Therefore, we can always be sure of the contents of the accumulators with respect to zero by the *BFZ* instruction.

2.15.8 Sample Program Using *BFM* and *BFZ* Instructions

The following example illustrates how the *BFM* and *BFZ* instructions can be used to aid in the solution of a problem. Suppose we wish to sort through a group of three data words and make up a table of those that have both half-words equal to or greater than zero. Because −0 and +0 both give the same results when used in arithmetic computation, it is necessary for us to make provisions in our program to check for −0. The preliminary flow chart for this problem is shown in figure 2–15. Here, the diamond-shaped "decision" blocks clearly indicate the use of the *BFM* instruction when checking for positive values. However, the possibility of a −0 satisfying the branching conditions of the *BFM* instruction has to be dealt with by a second check on the word, this time using a *BFZ* instruction. If the check discloses that the word is not −0, we know definitely that it is a true negative number and proceed to the next word. This program is also a logical program, since the

TABLE 2–12. PROGRAM USING FULL BRANCH INSTRUCTIONS

LOCATION	OPERATION	ADDRESS	
0.00000	*CAD*	0.01000	
0.00001	*BFM*	0.00012	
0.00002	*FST*	0.67770	
0.00003	*CAD*	0.01001	
0.00004	*BFM*	0.00014	
0.00005	*FST*	0.67771	
0.00006	*CAD*	0.01002	
0.00007	*BFM*	0.01016	
0.00010	*FST*	0.67772	
0.00011	*HLT*	—	
0.00012	*BFZ*	0.00002	
0.00013	*BFM*	0.00003	
0.00014	*BFZ*	0.00005	
0.00015	*BFM*	0.00006	
0.00016	*BFZ*	0.00010	
0.00017	*HLT*		
0.01000	Word one		Data
0.01001	Word two		Data
0.01002	Word three		Data
0.67770	Storage		
0.67771	Storage		
0.67772	Storage		

data is left unchanged. We will assume that the program starts at location 0.00000 and the data is stored at 0.01000 through 0.01002. The table is to start at location 0.67770 and run in sequence. A finalized form of this program is shown in figure 2–16 and listed in table 2–12.

2.16 *ADD ONE RIGHT* INSTRUCTION

The *Add One Right (AOR)* instruction is used to add one to bit R15 of the memory location specified by the address portion of the instruction. The execution of the *AOR* instruction takes places in the following manner. The right accumulator is cleared, and the contents of the selected memory register are transferred to the A registers. Then the carry 1 line into the bit R15 adder is pulsed, thus adding 1 to the address portion of the word. It should be remembered that the AN/FSQ-7 can have an address of 17 bits; therefore, if 17-bit operation is specified the LS bit is also used in the addition process. Also, an *AOR* instruction can cause computer overflow. After the address modification has taken place, the right accumulator is transferred to the right memory buffer (and bit LS of the left accumulator is transferred to the left memory buffer sign during 17 bit operation). The entire word (including the left half portion which has been stored in the left A register) is then replaced in its original memory loca-

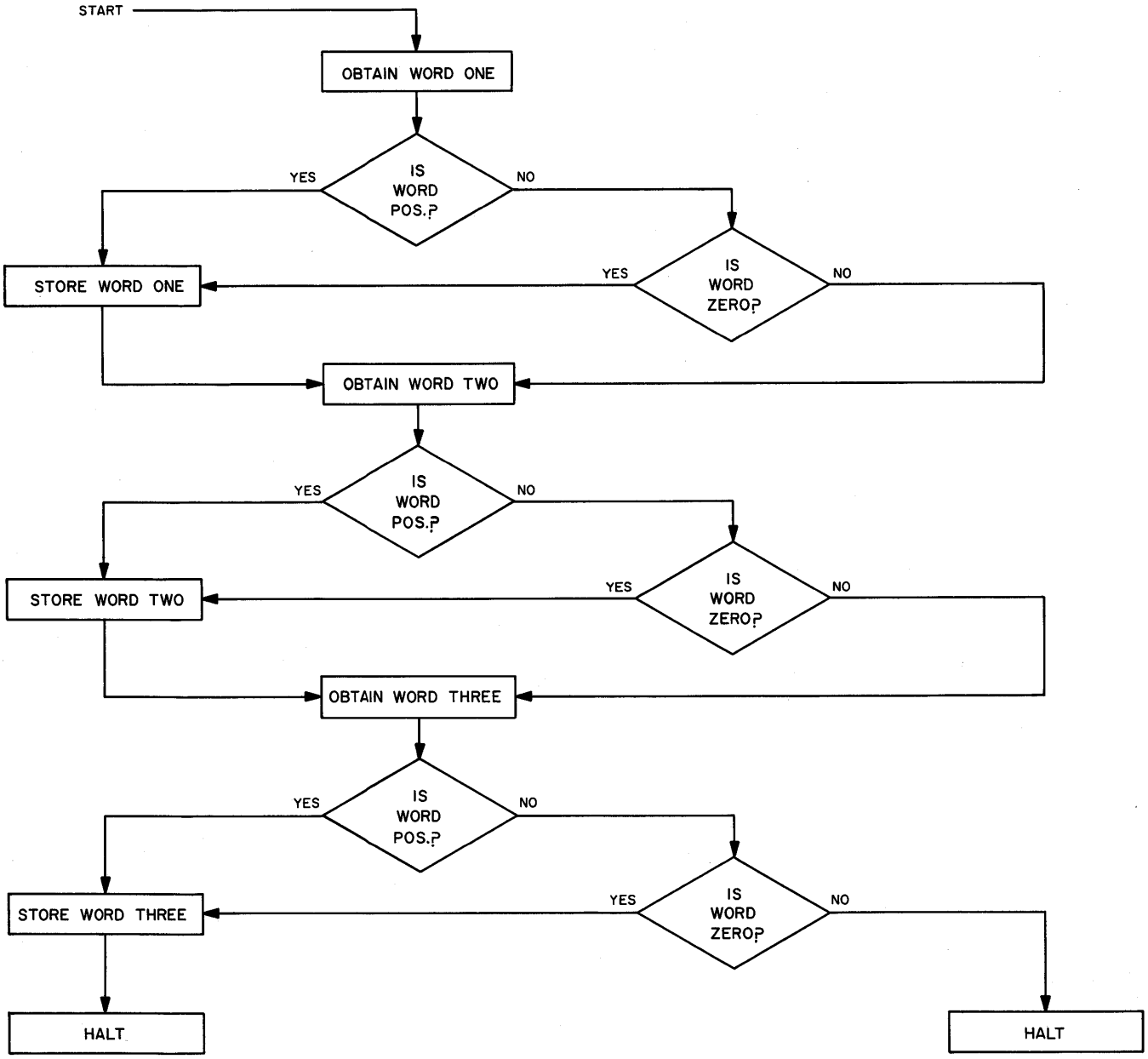


Figure 2–15. Preliminary Flow Chart Using Full Branch Instructions

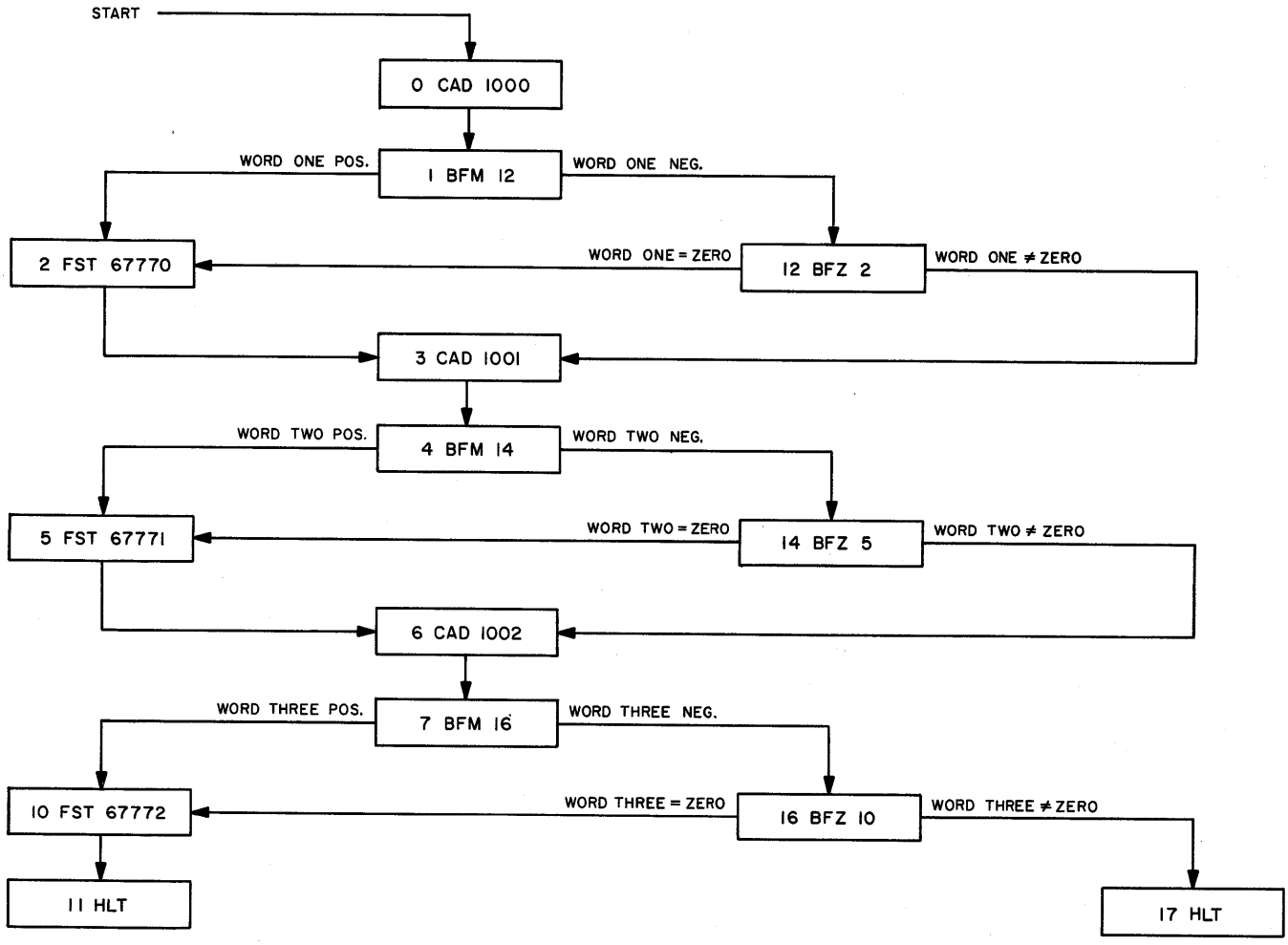


Figure 2–16. Final Flow Chart Using Full Branch Instructions

tion. The *AOR* instruction is similar to the *FST* instruction in that it takes an entire word from memory and replaces the same register with a different word. Because of this, an OTA and an OTB cycle are required in addition to the PT cycle, making execution time of the *AOR* instruction 18 μsec. The octal operation code for this instruction, which is indexable, is 344.

2.16.1 Uses of the AOR Instruction

The *AOR* instruction has two primary uses in most programs written for the AN/FSQ-7 and AN/FSQ-8. One use involves modifying the address portion of an instruction so that the same instruction can be used more than one time but with a different address portion. This technique is called indexing and can be performed in other ways that are more efficient than using the *AOR* instruction; however, the general purpose of indexing can be shown by the use of this instruction.

The second use for the *AOR* instruction is to step (increase) the value of a register by one each time a desired action has taken place, thus keeping the count of the number of occurrences. Such an occurrence might be the successful completion of a program which is written to continuously repeat itself. If we used an *AOR* instruction that specified the same register each time the program was completed, that register would contain the number of successful completions (or "passes") through the program. A register used in this manner is commonly referred to as a pass counter.

2.16.2 Address Modification Using the AOR Instruction

Assume that we have 11 numbers which are located in memory address 0.01100 through 0.01112. It is desired to find the sum of these numbers and store it in location 0.00150, using the least number of instructions possible. Of course, it is possible to use a series of *ADD* instructions, but this is quite lengthy. By use of the *AOR* instruction, we can use the same *ADD* instruction as many times as we need it. However, it is also necessary to have some sort of control that tells us when we have added all 10 numbers. For this, we can use a constant which represents the last address we wish to use, and compare it against the address we are using.

When the two match, we know we are using the last address we are interested in. We will assume that the sum of the 11 numbers will not cause an overflow and that the memory location 0.00150, which is to be used for final result storage, is cleared at the start of the program. The preliminary flow chart for this program is shown in figure 2–17, part A; the final chart is shown in part B of this figure.

It can be seen from figure 2–17 that we are modifying the address of the instruction in location 0.00001 by the use of the *AOR* instruction at program step 3. Then the *HLT* instruction is subtracted from the new memory address which is left in the right accumulator after execution of the *AOR* instruction. The right half of the *HLT* instruction is being used as a constant; it contains the address of the last number to be added. Each time the address in the accumulator is smaller than the final address, we will get a negative result. The *BRM* condition will be satisfied and will cause the program to select the partial sum and add another number to it. When the address being used reaches 0.01112, the *AOR* instruction will step it to 0.01113. The subtraction will leave 0.00001 in the right accumulator; the branch condition will not be met, and the program will fall through to the *HLT* instruction.

The program described above is listed in table 2–13. In table 2–14, a straight-line program that achieves the same result without modification is listed for comparison.

TABLE 2–13. ADDRESS MODIFICATION USING THE AOR INSTRUCTION

LOCATION	OPERATION	ADDRESS
0.00000	*CAD*	0.00150
0.00001	*ADD*	0.01100
0.00002	*FST*	0.00150
0.00003	*AOR*	0.00001
0.00004	*SUB*	0.00006
0.00005	*BRM*	0.00000
0.00006	*HLT*	0.01112
0.01110 - 0.01112	Data	
0.00150	Result	

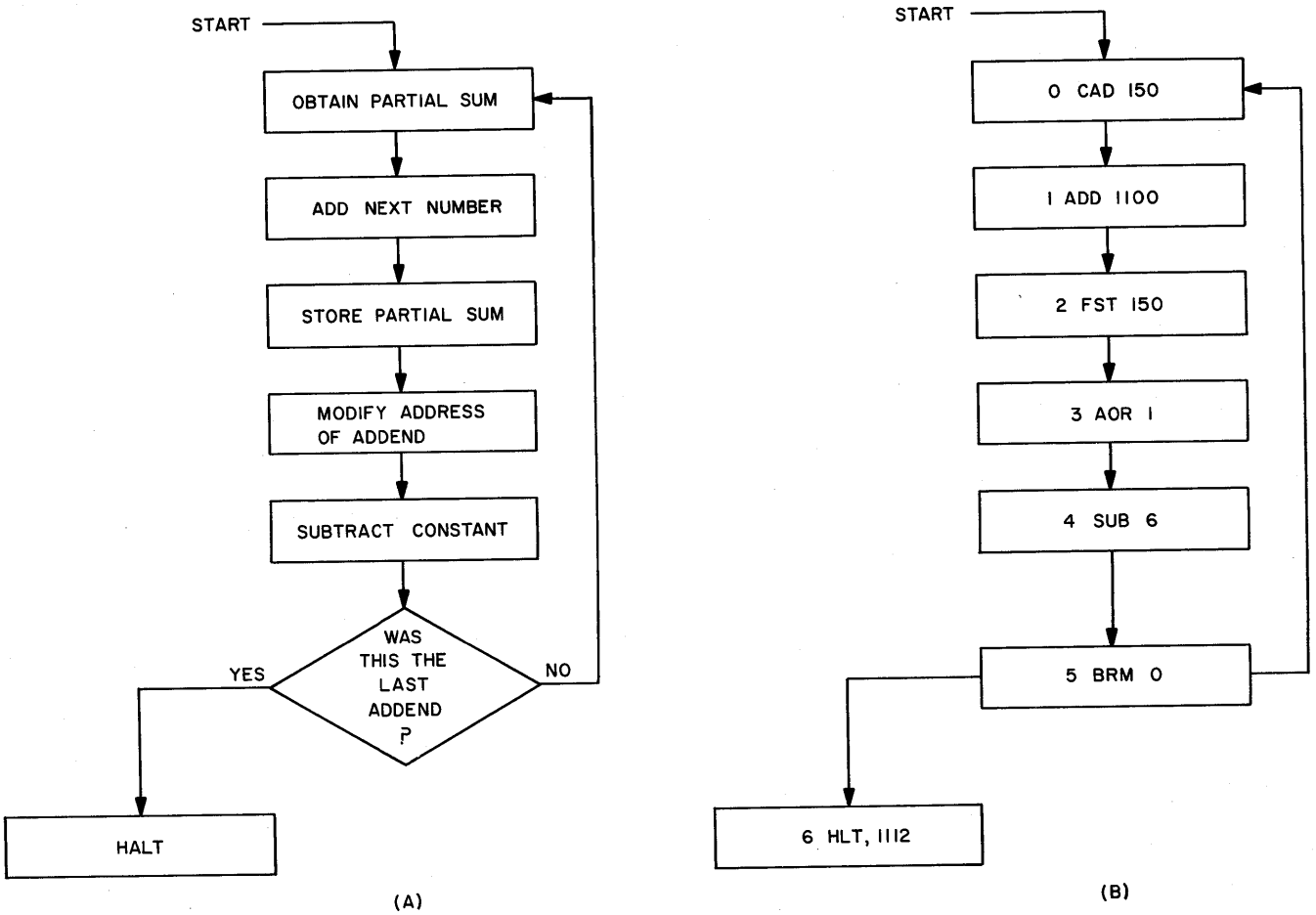


Figure 2–17. Address Modification Using AOR Instructions

TABLE 2–14. STRAIGHT LINE ADDITION PROGRAM

LOCATION	OPERATION	ADDRESS
0.00000	*CAD*	0.01100
0.00001	*ADD*	0.01101
0.00002	*ADD*	0.01102
0.00003	*ADD*	0.01103
0.00004	*ADD*	0.01104
0.00005	*ADD*	0.01105
0.00006	*ADD*	0.01106
0.00007	*ADD*	0.01107
0.00010	*ADD*	0.01110
0.00011	*ADD*	0.01111
0.00012	*ADD*	0.01112
0.00013	*FST*	0.00150
0.00014	*HLT*	–

Notice that while the program using the *AOR* instruction requires only seven memory locations as opposed to the 13 needed for the straight line program, the straight line program is considerably faster. The straight line program requires a total of 27 machine cycles, or 162 μsec. On the other hand, the indexed program requires 13 machine cycles for each number added, or a total of 145 cycles (11 x 13 plus two cycles for the *HLT* instruction). This means 870 μsec are needed to complete this program. Thus, we have two programs which arrive at the same result, one being shorter and the other being faster. As mentioned previously, one of these criteria will have to be sacrificed, depending on the situation.

2.16.3 Counting by Use of the AOR Instruction

Now let us take an example of the other common use of the *AOR* instruction, that of stepping a pass counter. In a certain table, some words have both halves negative; several have only the right half negative; some have only the left half negative; and others have both halves positive. We wish to know the total number of each type of word. The table is located at memory location 0.01000 through 0.01500, and locations 0.02000-0.02003 will be used to store the number of full negative, right negative, left negative, and full positive words, respectively. The preliminary flow chart for this program is shown in figure 2–18.

In this type of program, we can make good use of the various branching instructions to test for the conditions we are interested in. The final flow chart is shown in figure 2–19. Notice that after the four possible combinations of half-words have been checked, we step one of the four pass counters and then branch (or fall through) to the data address modification which uses an *AOR* instruction in exactly the same manner as the program discussed in 2.16.2. The program using the pass counters is listed in table 2–15.

2.17 INDEXING

2.17.1 General

As previously explained, indexing is the process whereby the address portion of the instruction word is modified so that the same instruction may be used repeatedly but with a different operand each time it is executed. An example of this was given in 2.16.2 by

TABLE 2–15. PROGRAM USING THE AOR INSTRUCTION AS A STEP COUNTER

LOCATION	OPERATION	ADDRESS
0.00000	*CAD*	0.01000
0.00001	*BFM*	0.00011
0.00002	*BRM*	0.00014
0.00003	*BLM*	0.00017
0.00004	*AOR*	0.02003
0.00005	*AOR*	0.00000
0.00006	*SUB*	0.00010
0.00007	*BRM*	0.00000
0.00010	*HLT*	–
0.00011	*AOR*	0.02000
0.00012	*SUB*	0.00010
0.00013	*BRM*	0.00005
0.00014	*AOR*	0.02001
0.00015	*SUB*	0.00010
0.00016	*BRM*	0.00005
0.00017	*AOR*	0.02002
0.00020	*SUB*	0.00010
0.00021	*BRM*	0.00005
0.0100 - 0.01500	Data	
0.02000	Pass counter	
0.02001	Pass counter	
0.02002	Pass counter	
0.02003	Pass counter	

using the AOR instruction to modify the address portion of the instruction. While this method is certainly valid, it leaves several things to be desired. For instance, each time we wish to modify the address portion of the instruction, it is necessary to read the word out of memory, perform the addition, and then write the word back in memory. This is time-consuming and would make a program requiring several hundred modifications excessively long. In addition, notice that we can add only one to the address portion of the instruction word. If we wished to select every other register from a table of operands, two *AOR* instructions would be required to increase the address portion of the instruction by two, or else a constant of two would have to be used to

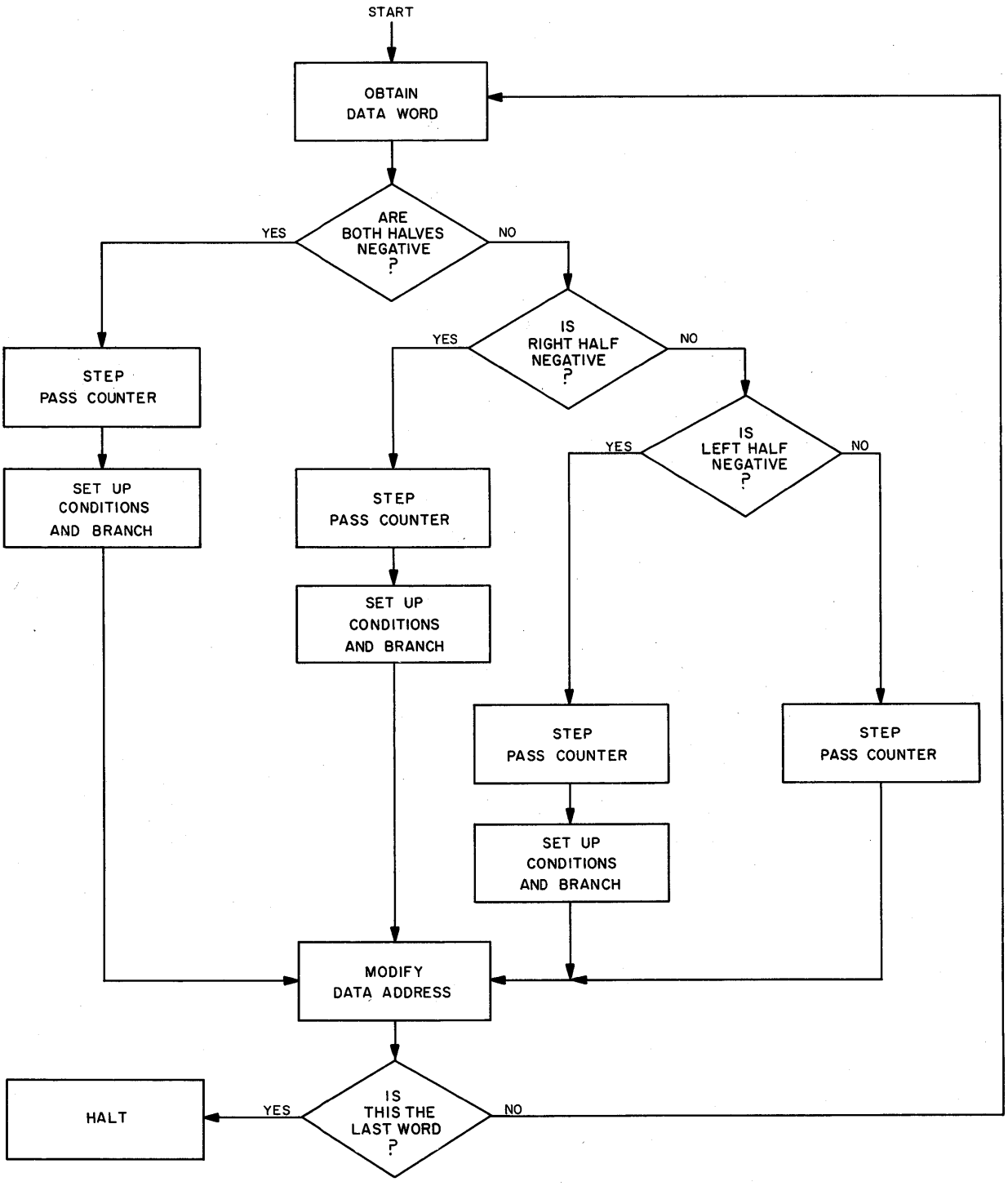


Figure 2–18. Counting by Use of the AOR Instruction, Preliminary Flow Chart

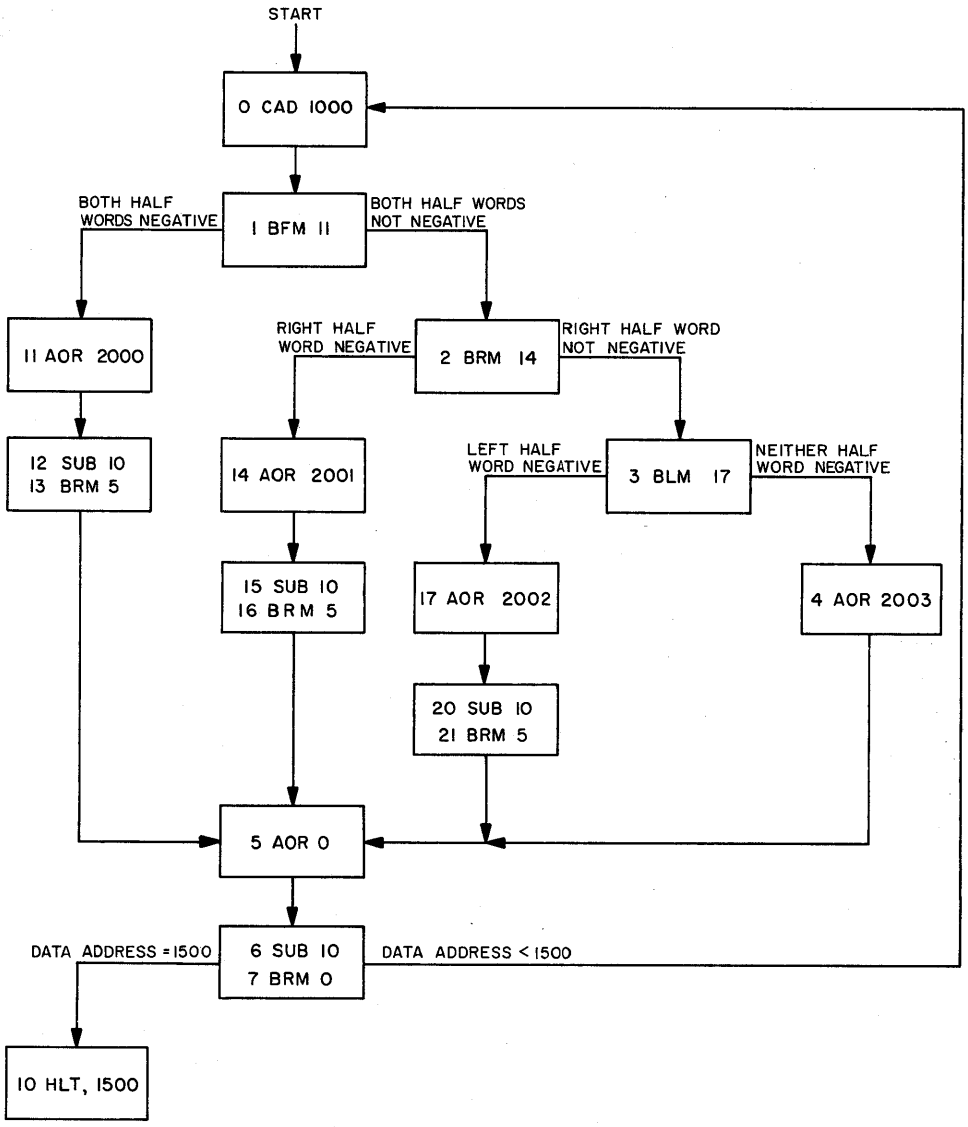


Figure 2–19. Counting by Use of the AOR Instruction, Final Flow Chart

perform the address modification. Finally, since the modification of the address takes place in the arithmetic element, overflow may be encountered when a true overflow condition does not exist.

From the above discussion, it can be seen that indexing by this method is unsatisfactory for our purposes. Therefore, the AN/FSQ-7 and AN/FSQ-8 make use of four index registers to perform address modifications. These registers may be loaded with a value of up to 2^{16} and then added to the address portion of an instruction word. The index register actually contains 17 bits; however, the sign bit is used strictly as a control bit and does not affect the amount by which an address is changed. It is also possible to reduce the contents of the index registers at any time by a separate instruction, causing the next address selected to be lower than the one previously selected by the instruction concerned. Index registers are especially valuable when we are writing computer programs because the address modification of an instruction does not require any additional time over and above the normal execution time of the instruction. Thus, when we wish to modify the address portion of an *ADD* instruction, we can do so and still execute the instruction in 12 μsec, using the modified data address.

By referring to figure 2–10, it can be seen that index registers designated 1, 2, 4, and 5 are transferred to the address register. The right accumulator is sometimes used as an index register and is designated as index register 3. It also may be transferred to the address register. Actually, only one of these five registers is selected at one time and its contents transferred to a portion of the address register, known as the index adder, during PT 7-11. Of course, the instruction from memory is also transferred out during this time, with the address

portion of the word going to the address register. The original data address and the contents of the selected index register are added in the index adder, and the modified address is transferred to the memory address register. All of this occurs during PT 7-11, and therefore does not lengthen execution time of the instruction.

2.17.2 *Reset Index Register* Instruction

The *Reset Index Register (XIN)* instruction provides us with the capability of loading a value into index registers 1, 2, 4, or 5. No provision is made for loading the right accumulator with this instruction because it may be loaded in a variety of ways through the use of other instructions (such as *CAD,* etc). The address portion of this instruction does not refer to a memory address but contains the value which we wish to place in the specified index register. The index register which is to be loaded is designated by the first three bits in the left half-word of the *XIN* instruction. Only one index register may be loaded with one *XIN* instruction. Thus, if we wish to load index register 4 with a value of 100_{10}, we would write the instruction as *4 XIN 144.* The octal operation code for the *XIN* instruction is 754, and it requires 6 μsec to execute. Because it is used to load an index register, this instruction cannot be indexed by another index register. Figure 2–20 shows how the instruction *4 XIN 144* is actually represented in binary.

2.17.3 *Branch on Positive Index* Instruction

2.17.3.1 General

Now that we have an instruction which enables us to load an index register, we need an instruction which will reduce its contents, so that different operands may be obtained by using the same instruction and index register. The instruction that does this is called the *Branch on Positive Index (BPX)* instruction. In addition to reducing the specified index register, the *BPX* instruction will transfer program control to the location specified in the address portion of the *BPX* instruction if the branching condition is met. This condition is that the index register contain a positive value; hence, the name of the instruction. If the branching condition is not satisfied, the program falls through to the next sequential instruction. The index register to be reduced is specified by bits L1-L3 of the instruction, and the amount this register is to be reduced is specified by auxiliary bits L10-L15. Previously, it was stated that bit L10 of the instruction word had a dual function; i.e., it served as a class variation bit and as an auxiliary bit. When it is necessary to utilize the auxiliary bits, the final class variation bit is not needed. This is the case with the *BPX* instruction; its octal operation code is simply 51, leaving bits L10-L15 to specify the amount the index register is to be reduced. A typical *BPX* instruction is written as *1 BPX(01)500,* and it means to branch to location 0.00500 if index register 1 is positive and to reduce the contents of this register by one.

The *BPX* instruction is executed in the following manner. The sign bit of the selected index register is sensed, and, if it is positive, the program counter, which contains an address one greater than the *BPX* instruction itself, is transferred to the right A register. Then the address register, which contains the memory location we wish to branch to, is transferred to the program counter and then to the memory address register. The address register is then cleared and loaded with the complement of bits L10-L15. The specified index register and the complemented number are then added in the index adder, and the resultant difference is placed back in the proper index register. Because no operand is required from memory, the *BPX* instruction is executed in 6 μsec. In addition, the *BPX* instruction cannot be indexed by another index register.

It is important to remember that the sign of the index register is sensed before its contents are reduced, since this fact determines the amount with which the index register is loaded originally. For example, if we wish to select five sequential data addresses by use of an index register, we will load the index register with a value of 3. Some justification for this may appear necessary, but if we examine a sample program, the reason will become apparent. Assume that we wish to take the sum of the data located in addresses 0.01000 - 0.1004. The program is listed in table 2–16.

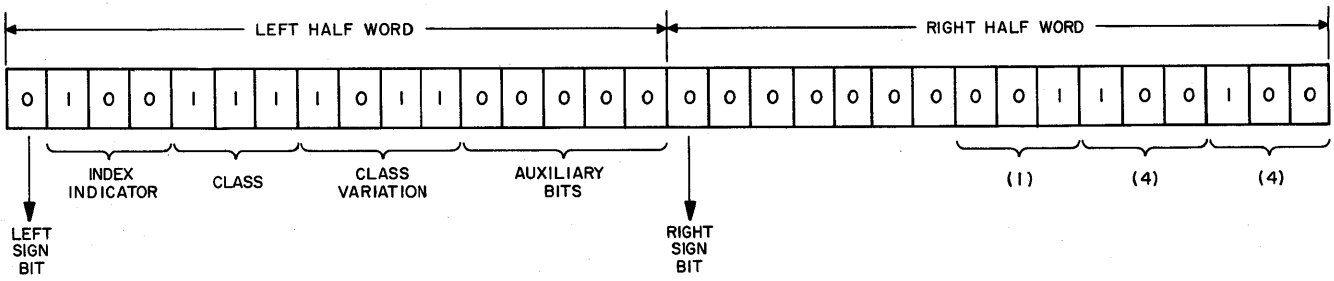


Figure 2–20. Word Layout for XIN Instruction

TABLE 2–16. INDEXED ADDITION PROGRAM

LOCATION	OPERATION	ADDRESS
0.00000	*1 XIN*	0.00003
0.00001	*CAD*	0.01000
0.00002	*1 ADD*	0.01001
0.00003	*1 BPX(01)*	0.00002
0.00004	*FST*	0.10000
0.00005	*HLT*	—

Notice that step 0.00001 placed the contents of location 0.01000 in the accumulator without the use of an indexed instruction, leaving us with only four numbers to be added. Step 0.00002 will select address 0.01004 during the first pass (address 0.01000 plus contents of index register one). At step 0.00003, we check the index register, find it positive, branch back to step 0.00002, and reduce the contents of the index register. On successive passes through the program, data addresses 0.01003 and 0.01002 will be selected. When data address 0.01002 is selected, the contents of index register 1 contains one. The branch condition is satisfied, and the program returns to step 0.00002. However, the index register has been reduced by one which clears the 16 significant bit positions and sets the sign bit to a 1. This is a modified form of —0 but is treated in the same manner as a true —0. The important thing to keep in mind is that the sign bit acts as a control, and when it is set to 1, indicating that the index register has become negative, the branch condition will no longer be satisfied. The sign bit of the index register is not added to the address register; therefore, adding the index register contents (2.00000) to the address portion of the *ADD* instruction will not modify this address, and the original address (0.01001) will be selected. The check of the index register will show its sign bit to be negative, and the program will fall through to the *FST* instruction.

In general, the rule to remember when loading an index register is to set it to one less than the number of passes to be made. In the above program, five numbers were to be added, but the first one was placed in the accumulator without using an index register. Four numbers remained to be added, which required four passes through the program. Therefore, the index register was loaded with three. Once again, it should be noted that this is not a rigid rule, but depends on how the program is written, particularly where the *BPX* instruction is placed in relation to the instruction being modified.

2.17.3.2 Applications of the *BPX* Instruction

The following examples show how the *BPX* instruction may be used in various programs. Assume that we wish to sort through a table which is stored in data locations 0.02500 through 0.03000. We want to place all numbers with both half-words negative into a table starting at location 0.00500 and the rest of the numbers into a table starting at location 0.04000. The sample flow chart for this program is shown in figure 2–21 (part A). We have no way of knowing how many, if any, of these numbers are negative in both half-words; therefore, it is wise to provide a halt after each table address modification in the event that all the numbers were placed in one table or the other. Part B of figure 2–21 gives the final flow chart for this problem. The program is also listed in table 2–17.

As another example of the uses of the *BPX* instruction, assume that we wish to program a delay of 120 μsec into the Central Computer System. There are various ways in which this could be done, but using the *BPX* instruction to branch back to itself is probably the one method which uses the least memory space. A sample program showing this delay is listed in table 2–18.

Step 0.00002 will load index register 5 with a value of 22_8 or 18_{10}. This step consumes 6 μsec. The next step causes the index register to be stepped down by a decrement of one and to branch to the same instruction. What we are doing, in effect, is continually sensing the sign of index register 5, waiting for it to go negative. The sign bit will be positive for 18_{10} times; however, the branch condition will have been met before the index register is stepped from 0.00001 to 2.00000, so the *BPX* instruc-

TABLE 2–17. TABLE-SORTING PROGRAM USING THE *BPX* INSTRUCTION

LOCATION	OPERATION	ADDRESS
0.00000	*1 XIN*	0.00300
0.00001	*2 XIN*	0.00300
0.00002	*4 XIN*	0.00300
0.00003	*1 CAD*	0.02500
0.00004	*BFM*	0.00010
0.00005	*2 FST*	0.04000
0.00006	*2 BPX(01)*	0.00013
0.00007	*HLT*	—
0.00010	*4 FST*	0.00500
0.00011	*4 BPX(01)*	0.00013
0.00012	*HLT*	—
0.00013	*1 BPX(01)*	0.00003
0.00014	*HLT*	—

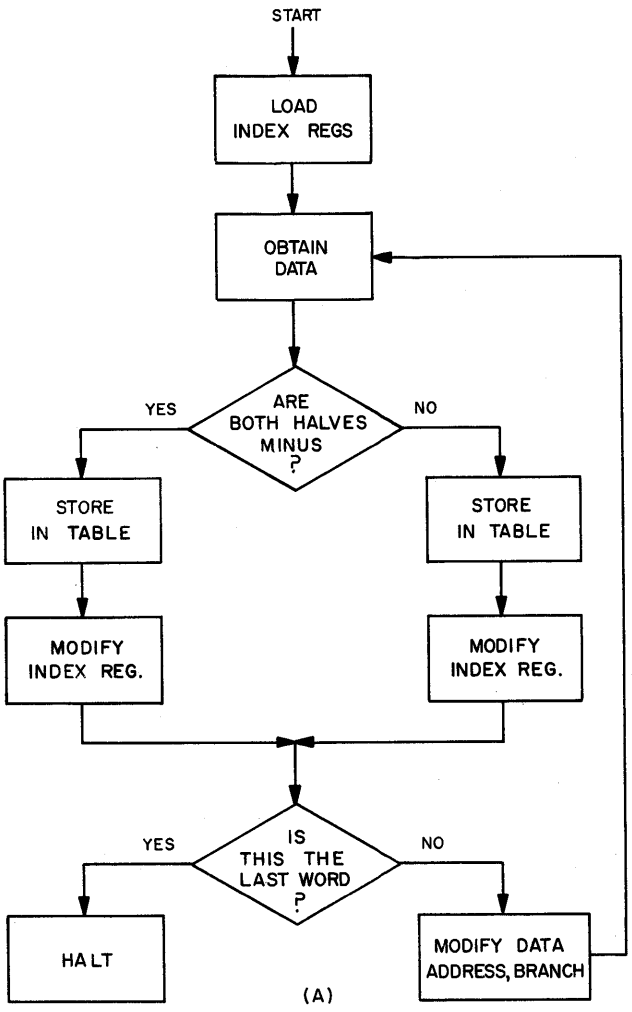
tion will be executed 19_{10} times or a total of 114 μsec. Adding the 6 μsec for the execution of the *XIN* instruction will give the required 120-μsec delay.

2.17.4 Using the *BPX* Instruction as an Unconditional Branch

There is one important feature of the *BPX* instruction which we have not discussed. Although the name of the instruction implies the use of an index register, it is also possible to use this instruction as an unconditional branch. This is accomplished by omitting a number from index indicator bits L1–L3. When this is done, the *BPX* instruction will automatically transfer program control to the address specified in the right half portion of the *BPX* instruction. The execution time remains at 6 μsec, and the octal operation code is simply 51. When the *BPX* instruction is used in this manner, it can often save several steps in a routine. For instance, let us consider the number-sorting routine which was presented in 2.15.5. This program sorted three numbers, found the largest, and stored it in a specified location. The *BLM* instruction was used in this program and 18_{10} steps were required. Now let us solve the same program using the *BPX* instruction as an unconditional branch. The three numbers were A, B, and C, and were stored in the left half portions of locations 0.00100, 0.00101, and 0.00102,

TABLE 2–18. PROGRAMMED DELAY USING THE *BPX* INSTRUCTION

LOCATION	OPERATION	ADDRESS
0.00000	*CAD*	0.01000
0.00001	*ADD*	0.01050
0.00002	*5 XIN*	0.00022
0.00003	*5 BPX(01)*	0.00003
0.00004	*FST*	0.02250
0.00005	*HLT*	—



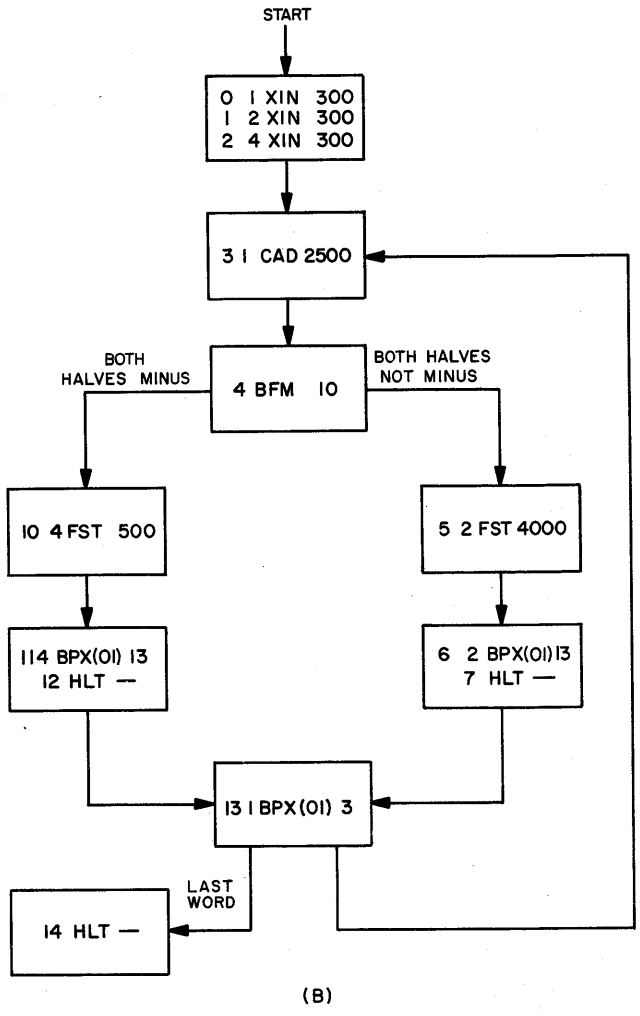


Figure 2–21. Table Sorting by Using the BPX Instruction

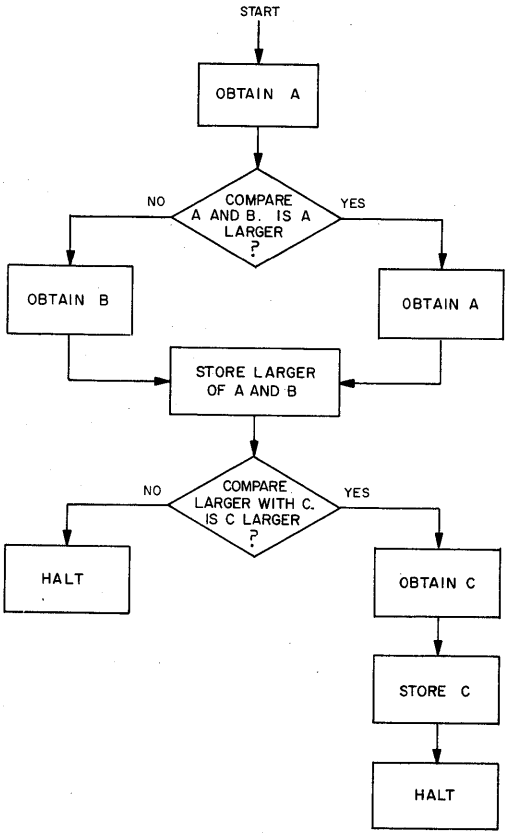


Figure 2–22. Number-Sorting Program Using Unconditional Branch, Preliminary Flow Chart

respectively. The largest number was to be stored in location 0.00150. The preliminary flow chart for this program is shown in figure 2–22. Notice that this flow chart arrives at the same place from two different outcomes; since the computer program proceeds sequentially, some provision must be made to allow us to do this. The *BPX* instruction is what is used, and the final flow chart is listed in figure 2–23. The program is written in 13_{10} steps, a saving of five memory spaces over the former program. This program is also listed in table 2–19. This is just one example of the use of an unconditional branch, but it serves to show the importance of this instruction. It is also used widely when entering and leaving program subroutines and with various other programming techniques. The *BPX* instruction as an unconditional branch is more fully explained in the following chapters.

2.18 SUMMARY OF BASIC INSTRUCTIONS

A listing of the basic instructions which have been discussed in this chapter is given in table 2–20. This table includes the actual name, mnemonic name, octal operation code, execution time, and other pertinent information about each of the 17 instructions covered up to this point.

TABLE 2–19. SORTING PROGRAM WITH UNCONDITIONAL BRANCH

LOCATION	OPERATION	ADDRESS
0.00000	*CAD*	0.00100
0.00001	*SUB*	0.00101
0.00002	*BLM*	0.00005
0.00003	*CAD*	0.00100
0.00004	*BPX*	0.00006
0.00005	*CAD*	0.00101
0.00006	*FST*	0.00150
0.00007	*SUB*	0.00102
0.00010	*BLM*	0.00012
0.00011	*HLT*	—
0.00012	*CAD*	0.00102
0.00013	*FST*	0.00150
0.00014	*HLT*	
0.00100	A	
0.00101	B } Data	
0.00102	C	
0.00150	Result	

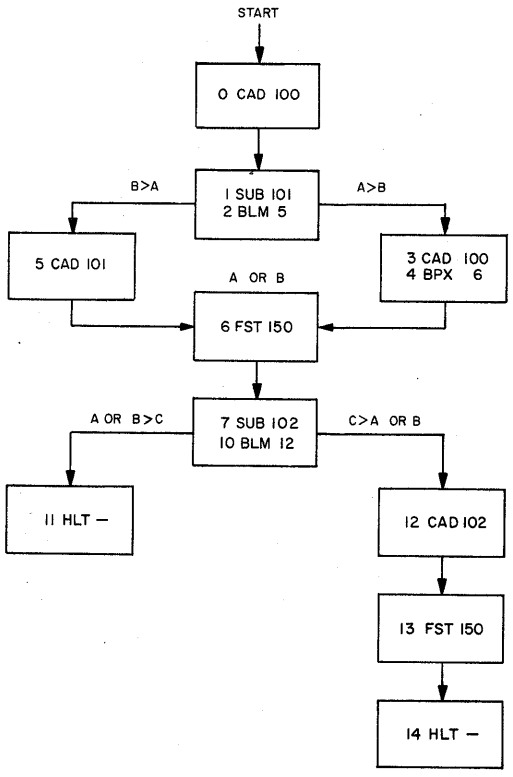


Figure 2–23. Number-Sorting Program Using Unconditional Branch, Final Flow Chart

TABLE 2–20. SUMMARY OF BASIC INSTRUCTIONS

INSTRUCTION NAME	MNEMONIC NAME	OCTAL CODE	EXECUTION TIME	INDEXABLE	CAUSE OVERFLOW
Halt	*HLT*	000	12 μsec	No	No
Clear and Add	*CAD*	100	12 μsec	Yes	No
Add	*ADD*	104	12 μsec	Yes	Yes
Twin and Add	*TAD*	110	12 μsec	Yes	Yes
Clear and Subtract	*CSU*	130	12 μsec	Yes	No
Subtract	*SUB*	134	12 μsec	Yes	Yes
Twin and Subtract	*TSU*	140	12 μsec	Yes	Yes
Full Store	*FST*	324	12 μsec	Yes	No
Left Store	*LST*	330	18 μsec	Yes	No
Right Store	*RST*	334	18 μsec	Yes	No
Add One Right	*AOR*	344	18 μsec	Yes	Yes
Branch on Positive Index	*BPX*	51-	6 μsec	No	No
Branch on Full Zero	*BFZ*	540	12 μsec	No	No
Branch on Full Minus	*BFM*	544	6 μsec	No	No
Branch on Left Minus	*BLM*	550	6 μsec	No	No
Branch on Right Minus	*BRM*	554	6 μsec	No	No
Reset Index Register	*XIN*	754	6 μsec	No	No

PART 3
PROGRAMMING THE CENTRAL COMPUTER SYSTEM

CHAPTER 1
GENERAL DESCRIPTION

1.1 PURPOSE OF CENTRAL COMPUTER SYSTEM

The prime purpose of the Central Computer System is to process, algebraically and logically, the information supplied to it by the Input System via the Drum System to arrive at results which are pertinent to the air defense situation. This processing takes the form of numerical computation with binary-coded data concerning the tactical conditions or reliability of the AN/FSQ-7 and AN/FSQ-8. Both tactical and maintenance information supplied to the Central Computer System is in binary numerical form, and the Central Computer System processes the data as if it were numerical in both content and nature. The Central Computer System is a mathematical machine, although the information with which it deals need not be purely mathematical in nature.

As tactical information is introduced into the Central Computer System, it is processed and correlated in a fixed pattern. The actual manner of processing is predetermined by additional information (a program) introduced into the Central Computer System. This programmed information consists of a series of orders which, when executed in the prescribed sequence, cause the Central Computer System to process the tactical data so as to arrive at the proper results. After the results of the computations have been obtained, they are transferred from the Central Computer System into the Drum System for storage. From this system, they are distributed to the Output and Display Systems. From the Output System, these results are transmitted via telephone and teletype lines to locations where defense action is carried out. The Central Computer System results which are sent to the Display System are presented in visual form to operating personnel for monitoring, interpretation, and control purposes.

In addition to its function of processing raw information, the Central Computer System acts as the control center for AN/FSQ-7 and AN/FSQ-8 as information is processed by the Central Computer System, as ordered, and sent to the Drum System for use by the Input, Output, and Display Systems. These signals control the transfer of data between systems, initiate operational cycles, set up control circuits for impending operations, and generally synchronize the action of each of these systems with that of the Central Computer System.

In summary, the purpose of the Central Computer System may be separated into two parts:

a. Information processing

b. Synchronization and control

Information processing takes place within the Central Computer System and involves the calculation of results from supplied information. The synchronization and control of the Central Computer System co-ordinates the acquisition, transfer, transmittal, and storage of information throughout the AN/FSQ-7 and AN/FSQ-8 so that all operations take place at the proper time and in the proper sequence.

1.2 SYSTEM REQUIREMENTS

Up to this point, we have discussed only those instructions which will permit us to solve relatively simple problems. However, to perform the job of air defense efficiently the AN/FSQ-7 and AN/FSQ-8 must be capable of executing many instructions which can rapidly process tactical data and maintenance information and, at the same time, keep a constant check on the slate of the various systems and elements. The Central Computer System has the capability of large information storage and fast access but requires several additional instructions, beyond what we have already discussed, to make full use of these facilities. The majority of instructions that are executed in the AN/FSQ-7 and AN/FSQ-8 deal with the Central Computer System; therefore, this part will discuss all those instructions which are executed directly in the Central Computer System and involve registers, sense units, etc., within the Central Computer System. Only those instructions which deal with IO devices will not be discussed at this time.

In paragraph 2.17 of the last chapter, a general description of indexing was presented, and some of the applications of this technique were shown. However, there are many uses to which indexing may be applied other than what we have covered so far; in addition, two

more indexing instructions are available for use within the AN/FSQ-7 and AN/FSQ-8. Because indexing of routines is such an important consideration when writing a computer program, the following chapter is devoted to more extensive applications of indexing and an analysis of two more indexing instructions and their variations.

CHAPTER 2
INDEXING TECHNIQUES

2.1 GENERAL

As explained in Part 2, indexing is essentially a technique whereby address modification of an instruction may be made without increasing the time required to execute the instruction. We have discussed problems, which could be solved by the use of index registers, to reduce the storage space required for a program and to cause delays in internal operations. However, aside from the obvious uses of indexing instructions, there are many applications which have not yet been discussed. This chapter will show some of the additional uses to which indexing may be applied and two instructions and their variations which can be used in obtaining even greater program flexibility.

2.2 ADDITIONAL USES OF THE *BPX* INSTRUCTION

The *BPX* instruction is used primarily for two purposes: either to reduce the contents of a selected index register and transfer program control back to the start of a routine or to act as an unconditional branch. However, this instruction may also be used to cause a 6-μsec delay in internal operations. This takes place if either index register 6 or 7 (which are nonexistent) is specified. This variation of the *BPX* instruction is useful when a mistake has been made in a program and it is desired to delete an instruction from the sequential listing without reshuffling the instruction locations. A *6 BPX* or *7 BPX* is put in the same address as the deleted instruction, and when the program reaches this point, no action will take place for 6 μsec, the time normally required to execute the *BPX* instruction. The program will then fall through to the next sequential instruction.

In some cases, we wish to examine the contents of an index register and take some action, depending on what we find, without disturbing the contents of the specified index register. This may be done by using an index interval of 0. For instance, if we wished to check the contents of index register 4 and branch to some location if it was positive, we would write the instruction as *4 BPX (00)*. This instruction will be executed in the same manner as a normal *BPX*. However, the index register will not be reduced because the complement of bits L10–L15 equals —0, and adding this to the index register results in no change in the register contents.

Table 3–1 shows all the possible combinations that may be used with the *BPX* instruction and the corresponding action that will take place. The instruction layout shows an index interval of one in each case where the index interval is applicable; however, it should be remembered that the index interval may be loaded with any number through 77_8 (capacity limit of bits L10–L15).

2.3 *RESET INDEX REGISTER FROM RIGHT ACCUMULATOR* INSTRUCTION

The *Reset Index Register from Right Accumulator (XAC)* instruction is used to load the specified index register with the contents of the right accumulator. The instruction is executed by first clearing the address register and then transferring the contents of the right accumulator into the address register. The address register then transfers its contents to the selected index register, which has also been cleared. The octal operation code for the *XAC* instruction is 764, and it requires 6 μsec to execute. This instruction does not utilize the address portion, since the index interval bits determine where the contents of the right accumulator are to be placed. For this reason, the *XAC* instruction is not indexable.

TABLE 3–1. *BPX* INSTRUCTION CONFIGURATIONS

INSTRUCTION	ACTION
0 BPX (—)	Unconditional branch
3 BPX (—)	Unconditional branch
1 BPX (01)	Branch if index register 1 is positive and reduce its contents by one
2 BPX (01)	Branch if index register 2 is positive and reduce its contents by one
4 BPX (01)	Branch if index register 4 is positive and reduce its contents by one
5 BPX (01)	Branch if index register 5 is positive and reduce its contents by one
1 BPX (00)	Branch if index register 1 is positive
2 BPX (00)	Branch if index register 2 is positive
4 BPX (00)	Branch if index register 4 is positive
5 BPX (00)	Branch if index register 5 is positive
6 BPX (—)	No operations for 6 μsec
7 BPX (—)	No operations for 6 μsec

As an example of the use of the *XAC* instruction, let us assume that we are going to process some tactical data on the AN/FSQ-7 or AN/FSQ-8. We are interested only in data that has been received after midnight, but we do not know how much data has been received. We will assume that the data we are concerned with is the status of missiles at various launching sites and that all status reports transmitted to the AN/FSQ-7 or AN/FSQ-8 after midnight will have a 1 in the LS bit position. Memory locations 0.00300 through 0.00550 are reserved as a block for unsorted missile status reports, and locations 0.06000 through 0.06250 are reserved as a block for those reports received after midnight. Once the reports have been sorted, we wish to arrange them in a form in which they may be further processed. Our problem here is not quite as simple as some we dealt with in preceding chapters, because more than one operation is involved. First, we must determine what data we are going to process, then temporarily store it, and, at the same time, record the amount of data that we will be processing. It is obvious that index registers will be used in this operation to obtain the raw data. Recording the amount of usable data can be done by stepping a cleared location. The flow chart for this problem is shown in figure 3–1, and the program is listed in table 3–2. Wherever possible, preliminary and final flow charts will be consolidated in one figure (as has been done in fig. 3–1) throughout the remainder of the manual.

It can be seen that this program performs several operations with relatively few instructions. Besides performing the preliminary sorting necessary, the program will set an index register so that further processing may be done on the reports stored in locations 0.06000 – 0.6250. It is necessary to reduce the contents of the index register after it is loaded so that the proper number of loops will be executed. Another instruction then branches to a routine which will then do the final processing on the status reports.

The *XAC* instruction may also be used when a certain number of results are known to be possible from a particular arithmetic or logical operation. When one of these results occurs, it is possible to test for it and then branch to a corresponding place in a table of constants and load the index register with a constant by use of the *XAC* instruction. Thus, the contents of the right accumulator may not directly contain the number desired to be placed in the index register but can direct the program to select the proper number. As an example of this, assume that we know that the outcome of a certain logical operation will be either 0.00004, 0.00010, or 0.00020 in the right accumulator. We will assume that the left accumulator contains +0. If the outcome is 0.00004, we wish to go to a routine which will loop three times; if it is 0.00010, we want to loop four times; and if it is 0.00020, we want to loop five times. The program to accomplish this is listed in table 3–3.

TABLE 3–2. DATA SORTING AND COUNTING PROGRAM

LOCATION	OPERATION	ADDRESS
0.00000	*1 XIN*	0.00250
0.00001	*CAD*	0.05771
0.00002	*FST*	0.05770
0.00003	*1 CAD*	0.00300
0.00004	*BLM*	0.00012
0.00005	*1 BPX (01)*	0.00003
0.00006	*CAD*	0.05770
0.00007	*4 XAC*	–
0.00010	*4 BPX (10)*	0.00011
0.00011	*BPX*	0.00030
0.00012	*FST*	0.06000
0.00013	*AOR*	0.05770
0.00014	*AOR*	0.00012
0.00015	*BPX*	0.00005
0.00300 - 0.00550	Initial data storage	
0.06000 - 0.06250	Sorted data storage	
0.05770	Report counter	
0.05771	Constant of +0	

Here, it can be seen that while we are dealing with the values 4_8, 10_8, or 20_8, we wish to load an index register with corresponding values of 3_8, 4_8, or 5_8. In addition, this program contains one more feature that we have not discussed yet, but which is very useful in programming. This is the *HLT* instruction in location 0.00005. We are assuming that the previous program has placed one of the three allowable results in the right accumulator. If this is the case, the program will always branch around the *HLT* instruction, even on the last pass through it. However, if, for some reason, an incorrect value was stored in location 0.01420, the program would make three unsuccessful attempts to compare and then

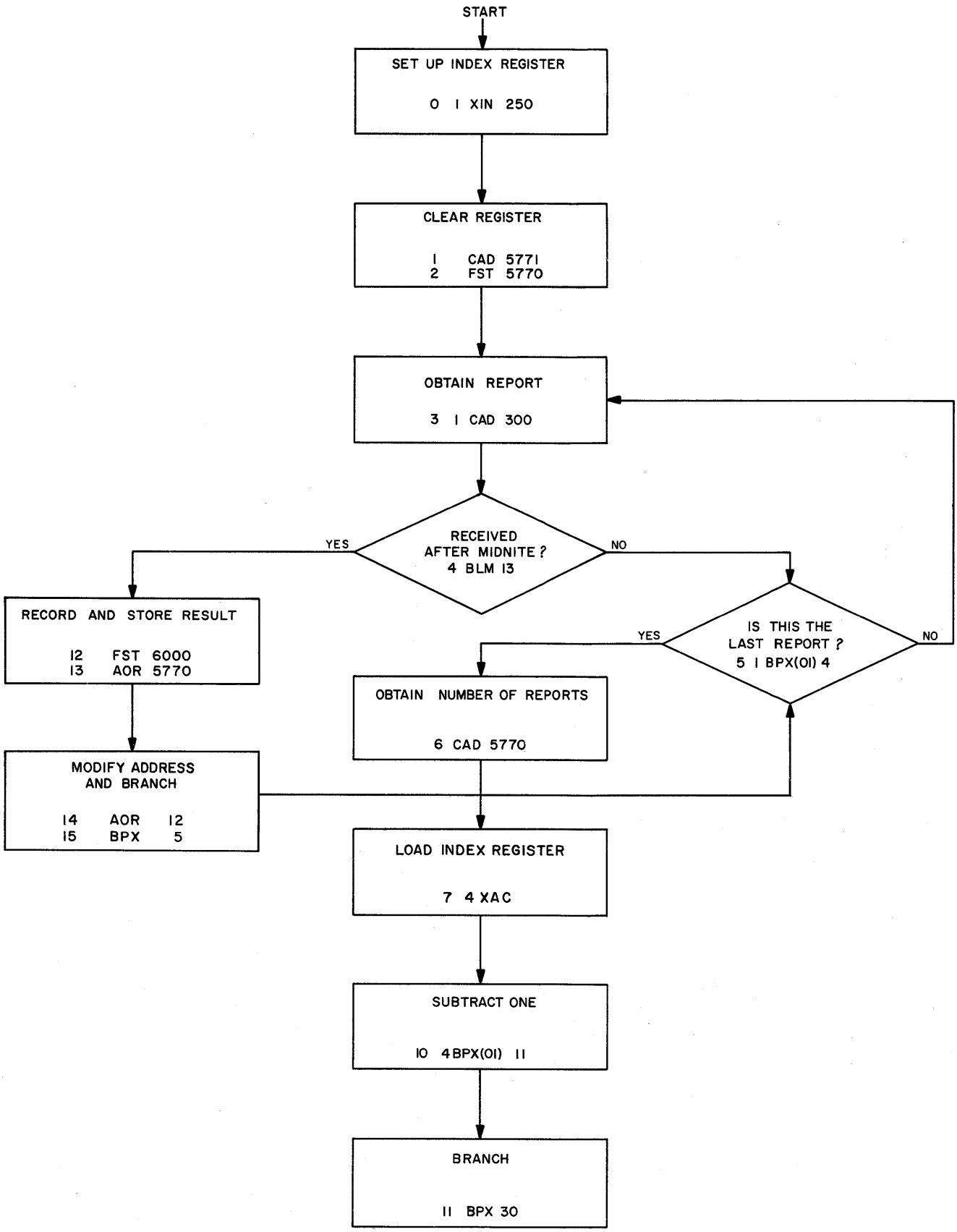


Figure 3–1. Data Sorting and Counting Program Flow Chart

fall through to the *HLT* instruction. This is known as an "error halt" because, under normal circumstances, we would not halt but branch to the desired routine after setting index register 2. Therefore, if the program does halt at this location, we know the result of the previous program was incorrect.

The *XAC* instruction also has a configuration that enables it to be used for more than its one stated purpose. The various configurations are shown in table 3–4.

2.4 USING THE RIGHT ACCUMULATOR AS AN INDEX REGISTER

It has been stated that the right accumulator can be used as an index register; however, so far, we have used only index registers 1, 2, 4, or 5 to perform program loops or address modification. There is no provision in the AN/FSQ-7 or AN/FSQ-8 to reduce the contents of the right accumulator by a branching instruction; therefore, it is of no use to us when we wish to loop through a certain portion of a program or separate routine. However, the right accumulator is extremely useful when it is used to modify addresses of various operands. This may not be apparent at first, since it would appear that we cannot obtain sequential memory locations as is the case with the normal index registers. This seeming disadvantage is actually a useful programming technique, as we shall see in the following examples.

TABLE 3–3. INDEX REGISTER LOADING ROUTINE

LOCATION	OPERATION	ADDRESS
0.00000	*1 XIN*	0.00002
0.00001	*CAD*	0.01420
0.00002	*1 SUB*	0.00060
0.00003	*BFZ*	0.00006
0.00004	*1 BPX (01)*	0.00001
0.00005	*HLT*	–
0.00006	*1 CAD*	0.13350
0.00007	*2 XAC*	–
0.00010	*BPX* (To desired routine)	
0.01420	Result of previous program	
0.00060	+0, 0.00004	
0.00061	+0, 0.00010	
0.00062	+0, 0.00020	
	Constants	
0.13350	+0, 0.00002	
0.13351	+0, 0.00003	
0.13352	+0, 0.00004	

TABLE 3–4. *XAC* INSTRUCTION CONFIGURATIONS

INSTRUCTION	ACTION
0 XAC	No operation for 6 μsec
3 XAC	No operation for 6 μsec
6 XAC	No operation for 6 μsec
7 XAC	No operation for 6 μsec
1 XAC	Reset index register 1 from right accumulator
2 XAC	Reset index register 2 from right accumulator
4 XAC	Reset index register 4 from right accumulator
5 XAC	Reset index register 5 from right accumulator

2.4.1 Table Lookup Procedure

We already know that the right accumulator cannot cause a program to select sequential locations, so we must utilize it when it is desired to select locations out of sequence. Such is often the case when we are looking up various tables such as sine, cosine, etc. The tables are stored in the AN/FSQ-7 and AN/FSQ-8 in the order of magnitude, and if we wish to select some operand at random out of the table, it is necessary merely to load a proportionate value into the right accumulator, modify the *CAD* instruction, and select the desired operand. For instance, let us assume that we wish to perform co-ordinate conversion from polar to rectangular form on several sets of targets. To do this, we need to know the various angles of these targets in relation to north. We shall call these angles ϕ. The actual co-ordinate conversion requires us to use the sine of ϕ and the cosine of ϕ. We have a table containing sine ϕ, cosine ϕ in the left and right half-words, respectively, in incre-

TABLE 3–5. TABLE LOOKUP PROGRAM

LOCATION	OPERATION	ADDRESS
0.00000	*1 XIN*	0.00061
0.00001	*1 CAD*	0.00020
0.00002	*3 CAD*	0.12200
0.00003	*1 FST*	0.01100
0.00004	*1 BPX (01)*	0.00001
0.00005	*HLT*	–
0.12200 - 0.12577	Sine, cosine table	
0.00020 - 0.00101	Target angles	
0.01100 - 0.01161	Storage for sine, cosine values	

ments of 1/256 of a circle or about 1.4°. (Such a table is actually used in the AN/FSQ-7.) This table is stored in locations 0.12200 through 0.12577. Since there are 256 possible units of azimuth possible, each angle ϕ may be represented as a value between 0_8 and 377_8. Thus, 1.4° would be represented as 0.00000; 2.8° would be 0.00001, etc. There are 50_{10} targets stored in locations 0.00100 through 0.00161. Since the targets are not stored in a particular order, we must have some method of random access to the table of sine ϕ, cosine ϕ. Otherwise, it will be necessary to sort the target angles and place them in ascending order. This can be overcome by placing the angle ϕ itself in the right accumulator and using this value to modify the address of the first entry in the table. Then, as different angles are placed in the right accumulator, they will cause a corresponding modification of the table address and enable us to select the proper sine ϕ, cosine ϕ. The program to accomplish this, often called a table lookup program, is listed in table 3–5.

In table 3–5, it can be seen that the larger the angles, the higher will be the address selected in the table. This type of operation saves a great amount of time that would otherwise be spent arranging the angles so that sequential access to the table could be achieved.

2.4.2 Table Makeup Procedure

Many tables, such as the sine and cosine table involved above, are loaded into the AN FSQ-7 and AN/FSQ-8 already in the proper sequence. These tables are "permanent" tables; the information contained in them is always valid and can be used over and over again. However, the AN/FSQ-7 and AN/FSQ-8 require many temporary tables concerning the air defense situation which cannot be prepared externally and then placed in internal memory. These tables must be generated as the information required is received into the Central Computer System. This information often comes in at random times, and several operands are sometimes required before a table can be "built up." As an example of such a table, let us consider the program dealing with counting and sorting missile status reports which was discussed in conjunction with the *XAC* instruction. The "output" or final result of this program was to provide a list of missile reports that had been received after midnight and to keep track of the number of reports involved. It was mentioned at the time that this program would lead to another program which would further process these reports. (See table 3–2.) Such a program is the table makeup program. Let us assume that we have already set an index register to the proper value as a result of the last program and that our reports received after midnight are stored, starting at location 0.06000. However, all we really know about these reports is that they fell within the time limit; we do not have a useful table of reports, since they were received randomly, and we cannot assume that they are in any kind of order. If we assign each missile base a code number and include this code number as part of the right half-word, we can make up a table, assigning the locations by use of the code number. For ease of explanation, we will assume that the code numbers of the missile installations will form a numerical sequence when arranged properly. Our problem is to make this numerical arrangement so that a master program can refer to a known location at any time and find out the status of the missile base by examining the contents of that location. Using ordinary sorting techniques would require an excessive amount of time, just as with the table lookup program. However, we can again make use of the right accumulator to modify the starting address of the table and construct the table so that the code numbers fall in sequence. The program for this function is listed in table 3–6.

The first instruction will place a Missile Status report in the accumulator. We then use the Code Number contained in the right half-word to determine the location of the report in the final report, by using the right accumulator as an index register.

By now, the flexible programming that can be achieved with the right accumulator as an address modifier should be quite clear. The main uses to which this programming technique can be put are those of table lookup and table makeup, as illustrated in the above programs. Any other process whereby the result of an arithmetic or logical process can be used to select or build a table is equally valid. Later, we shall see this application of the right accumulator used again in conjunction with various other instructions.

TABLE 3–6. TABLE MAKEUP PROGRAM

LOCATION	OPERATION	ADDRESS
0.00030	*4 CAD*	0.06000
0.00031	*3 FST*	0.07300
0.00032	*4 BPX (01)*	0.00030
0.00033	*HLT*	–
0.06000 - 0.06250	Status reports received after midnight	
0.07300 - 0.07550	Report table in numerical order	

2.5 *STORE ADDRESS* INSTRUCTION

The *Store Address (STA)* instruction is used to replace the right half portion of the specified memory location with the contents of the right A register. The right A register is used as the modifying register in this instruction because many other registers utilize it during their execution, and its contents are often valuable. One example is the entire class of branching instructions which automatically transfer the contents of the program counter to the right A register during their execution. An application of this is shown below. The *STA* instruction is executed in much the same manner as the *RST* instruction except that the right A register is used to supply the new address portion of the memory location rather than the right accumulator. The *STA* instruction requires 18 μsec to execute and may be indexed. It has an octal operation code of 340.

As an example of the *STA* instruction, let us consider a routine of a program which is to be frequently used in conjunction with a main or master program. Such a routine might be the table makeup routine we discussed previously. As the routine is written now, we halt as soon as the table is made up. However, in the actual operation of the AN/FSQ-7 and AN/FSQ-8, it is not feasible to perform just a few instructions and then halt; the computer program must run continuously. In order to do this, we must be able to get in and out of routines, or subroutines, without halting. This may be accomplished with the use of the *STA* instruction in the following manner. Assume that the last address of a certain program was 0.00100 and that another routine was needed to make up a table from the results of the last program. After this is done, however, we wish to return to another address which will start the execution of a new program. This may be accomplished by placing a *BPX* (unconditional branch) instruction at location 0.00100. The execution of the *BPX* instruction will cause the program counter, which contains 0.00101, to be transferred to the right A register. If the first instruction of the routine to which we branch is a *STA* instruction, it will place the contents of the right A register (i.e., the program counter) in the right halfword of the desired location. By specifying the last instruction (also a *BPX*) of the subroutine as the right half portion of the *STA* instruction, we can assure ourselves of a way to get out of the subroutine and back to another program which starts at location 0.00101. The flow of program control through these routines is shown in figure 3–2.

A routine of this type which can be entered and left with unconditional branches and a *STA* instruction is known as a "closed" subroutine. This means that the routine is closed with respect to the rest of the program and requires a branch to and from it. However, it may be used many times by several different programs. In contrast to the closed routine, the "open" subroutine is one which is sequenced through normal stepping of the program counter. It may perform a discrete function within a program but it cannot be entered from another program and, therefore, has a limited application.

With the programming capability that the AN/FSQ-7 and AN/FSQ-8 have for branching to and from subroutines, it can be seen how important it is for the programmer to know exactly what the contents of various blocks in internal memory contain. With the explanation of more instructions, we will see that the AN/FSQ-7 and AN/FSQ-8 have many instructions that are designed especially to check the status of various blocks and even to check specific bits within an instruction or data word. Also, many instructions, such as the *STA* instruction, are used primarily to preserve or utilize the contents of certain operating registers. A discussion of such an instruction follows.

2.6 *ADD INDEX REGISTER* INSTRUCTION

2.6.1 Instruction Analysis

The *Add Index Register (ADX)* instruction is used to add the contents of the specified index register to the address of the instruction. The execution of this instruction is similar to any address modification specified by an index register. The address portion of the *ADX* instruction and the selected index register are added in the index adders of the address register. However, the modified address is then transferred to the right A register, where it may be dealt with by a *STA* instruction. The execution time of the *ADX* instruction is 6 μsec and is designated by an octal operation code of 770.

The *ADX* instruction is used primarily when it is desired to preserve the contents of an index register for a particular reason. Up to this time, we have had no way of obtaining the contents of an index register once

it has been loaded. However, with the *ADX* instruction, we can specify an address portion of 0.00000 and thus obtain the value of the specified index register which will be transferred to the right A register. Then a *STA* instruction may put the number in a desired location.

2.6.2 Programmed Use of the *ADX* Instruction

The main reason for preserving the contents of an index register is to set up control for some other program or because the index register is required midway through the execution of a program for loop control of another program. This occurs frequently in various programs run on the AN/FSQ-7 and AN/FSQ-8 because there may be several routines or subprograms that require indexing control and there are only four index registers. If we are executing an indexed program which is being controlled by index register 1, and a more important program requests the use of the same index register, it is necessary to preserve the original contents of the index register so that the program in progress at the time of interruption may continue from the point of interruption once the intervening program is completed. This can best be shown by an example of just such an occurrence. Assume that we have a program which is contained in locations 0.00150 through 0.00156. The function of the program is to add several numbers together with the use of an index register. If, at some point in the addition, a large negative number is encountered which causes the right accumulator sum to go negative, we wish to branch to a number-sorting program using the same index register, which is contained in locations 0.01625 through 0.01634. After this program is completed, we wish to return to our original program and complete it from its point of interruption. The reader should not be concerned with the actual function of these two programs; the illustration is to show the transfer of control back and forth between these routines, one of which has priority over the other. The fact that priority exists has been established by the fact that a control word containing a large negative value may have been placed in the table of operands used by the first program. Thus, even a third program enters into the picture. However, this fact is beyond the scope of the example at this time. Assume that this condition may occur and that the program must be written to include this occurrence. The flow chart for this program (or programs) is shown in figure 3–3.

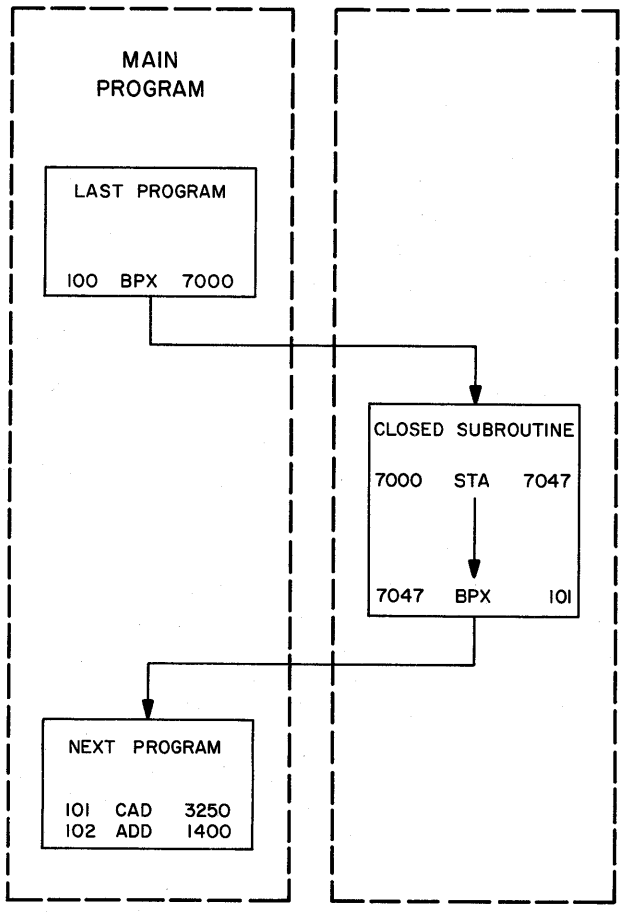


Figure 3–2. Use of STA Instruction

The analysis of this flow chart will show that program 1 may run to completion and halt at location 0.00156. This is true if the sum, which is checked after each addition by the *BRM* instruction at location 0.00153, remains positive. As soon as the sum goes negative, a branch directly into a subroutine takes place, which stores the contents of the program counter (location branched from plus one). If we had only one program such as program 1, this first step in the subroutine would not be necessary since we would always know where the branch took place. On the other hand, if we had 10 programs which performed the same function as program 1 and had the same provisions for branching, this step would be necessary to assure us that we returned to the correct memory location in the correct program. Then the sum of program 1 is returned to the value it contained before the branch, and the contents of index register 1 are preserved. Here again, we must assume that some action may take place in a third program which will clear the large negative value out of the table of operands used by program 1. If this was not done, we would continually branch to the subroutine. After program 2, the priority program, has been executed to completion, index register 1 is restored to the value it contained at the time the branch from 1 took place. The sum is also restored and we branch back to our program at the point of interruption. The index register is reduced just as if the operand which has contained the negative control word had been positive; however, the sum will show only the sum of the numbers up to the point the control word was en-

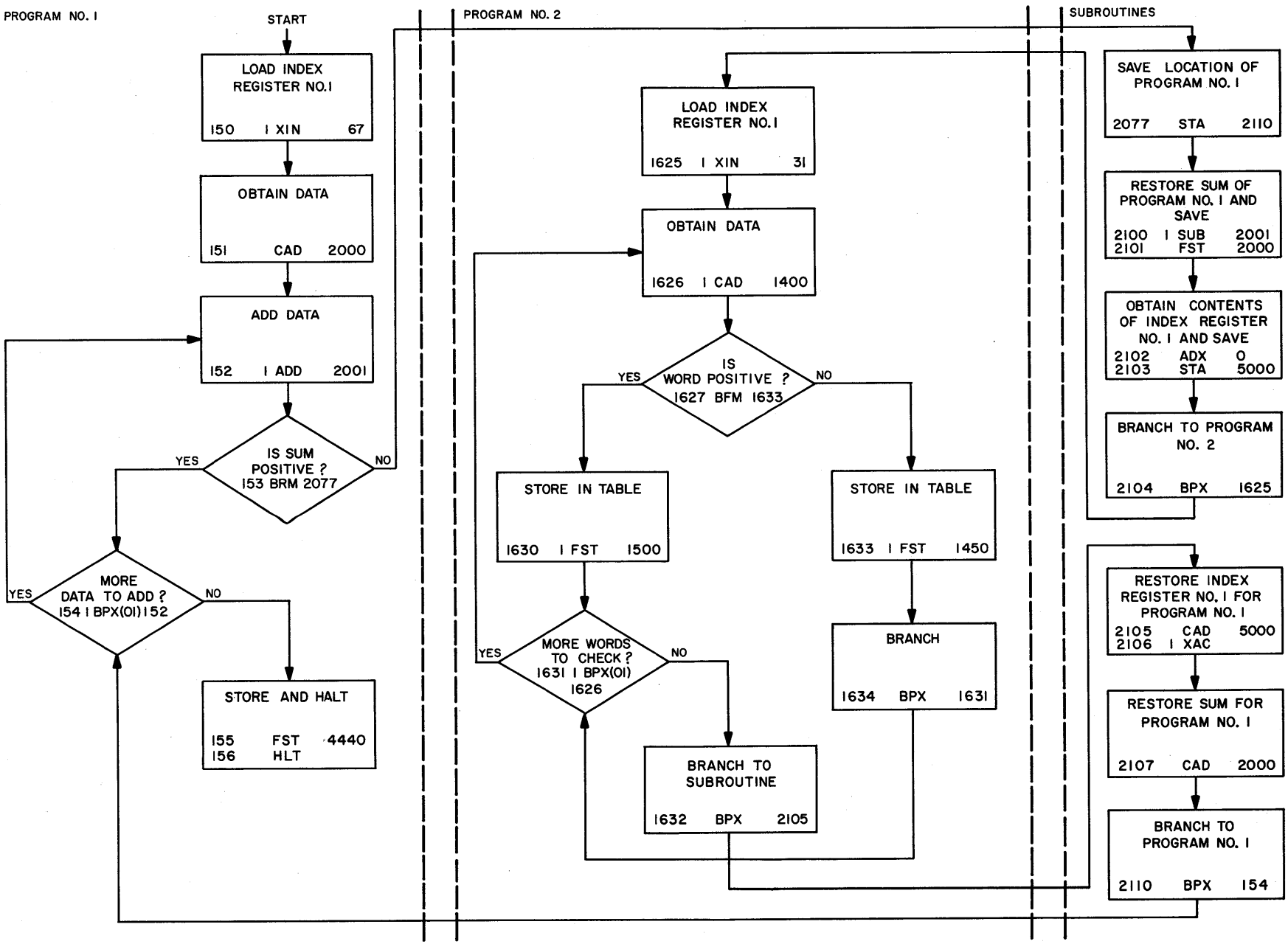


Figure 3–3. Flow Chart Showing One Index Register Used in Two Programs

countered. Therefore, we may conclude that while this program is interested in obtaining an arithmetic sum of some operands, it also is "searching" for a signal that enough data is available for a program with a higher priority to be executed.

2.7 SUMMARY OF INDEXING TECHNIQUES

At this time, the reader may feel that an example such as the one given above has little or no application to the AN/FSQ-7 and AN/FSQ-8. However, many of the instructions for these two machines were designed especially for the air defense problem, and, therefore, examples of their uses will necessarily resemble tactical operations. It is beyond the scope of this manual to present even a general idea of how the tactical air defense problem is programmed because of its magnitude. The various indexing techniques and program controls discussed in this chapter were presented from a practical viewpoint. It should be remembered that maintenance programs are often a simulation of tactical operations and may utilize programs and subroutines similar to the actual tactical program. As more instructions are explained in the remaining chapters, the program examples will have more and more practical significance.

CHAPTER 3
INSTRUCTIONS

3.1 *CLEAR AND ADD MAGNITUDE* INSTRUCTION

The *Clear and Add Magnitude (CAM)* instruction is used to place in the accumulators the positive absolute magnitude of the contents of the memory location specified by the right half portion of the instruction. For an operand that is already positive, the *CAM* instruction is equivalent to a *CAD* instruction; for a negative operand, the *CAM* instruction is equivalent to a *CSU* instruction. Thus, the value contained in the accumulators after the execution of this instruction is always positive, regardless of its original sign. The reason for this is to ensure a known starting condition for an operation which deals with magnitudes only. The *CAM* instruction requires 12 μsec to execute and may be indexed. It is specified by an octal operation code of 160.

3.2 *CLEAN AND ADD MAGNITUDE* INSTRUCTION

The *Difference Magnitude (DIM)* instruction is used to generate the difference in absolute magnitudes between a number in the accumulator and the operand contained in the memory location specified by the right half portion of the *DIM* instruction. Since the number in the accumulator may not be positive, it is necessary to make the contents of the accumulator positive for proper execution of the *DIM* instruction. However, the original contents of the accumulator may need to be preserved for some other operation; therefore, the accumulators are first transferred to the B registers. It was stated during the discussion on binary arithmetic that the B registers serve as extensions of the accumulators; the *DIM* instruction makes use of this facility. After the accumulator contents are duplicated in the B registers, the accumulators are made positive, if necessary; the operand to be used is placed in the A registers, made positive, if necessary, and then complemented. Thus, the effect of adding these two positive numbers together will be to subtract the contents of the A register from the accumulator. If the accumulator contents are still positive after the execution of this instruction, we know that the absolute magnitude of the operand from memory was smaller than the absolute magnitude of the accumulator contents. On the other hand, if the accumulator is negative, we know that its contents were larger in absolute magnitude than the operand from memory. The *DIM* instruction can be executed in 12 μsec, and it may be indexed. It is designated by an octal operation code of 164.

3.3 PROGRAM EXAMPLE USING ABSOLUTE MAGNITUDES

Assume that we are processing air traffic data, and a request comes to the Central Computer System to clear a "corridor" of air space 20 miles wide for use by a flight of cargo planes. It is the responsibility of the Central Computer System to define this corridor so that it may be displayed on the various monitor consoles within the AN/FSQ-7 or AN/FSQ-8. However, we must first notify all planes presently near the corridor that they must change course in order to clear the area. We wish to make up a list of all tracks close to the corridor and display this list on a digital display tube so that a ground controller may direct the necessary planes out of the way. We will assume that the corridor is to run north of the AN/FSQ-7 or AN/FSQ-8 installation, east to west, and that its southern boundary will be 50 miles to our north. This layout is shown in figure 3–4. Therefore, it is necessary for us to contact all planes with a "y" co-ordinate of either 50 or 70 miles since they are on the boundary of the corridor. We can do this with the use of the magnitude instructions. There is one precaution we must take, however, and that is to discard all tracks that are located south of us, since we do

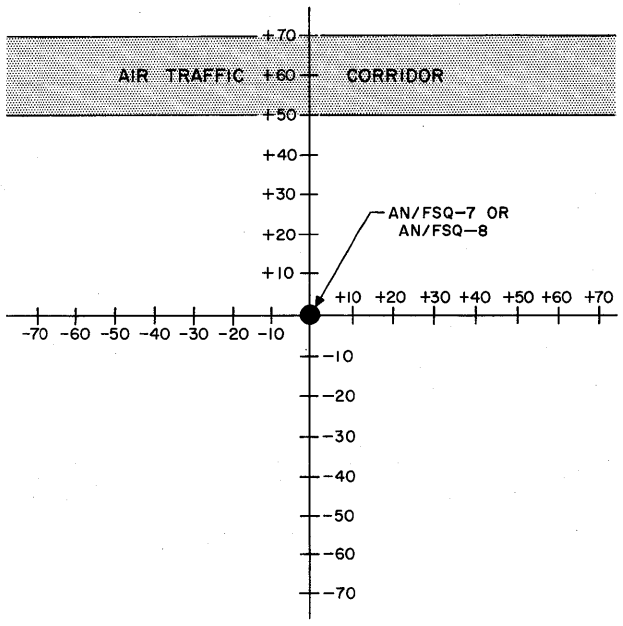


Figure 3–4. Layout of Air Traffic Problem

not wish to regard tracks with a y co-ordinate of —50 or —70 miles. Assume that the co-ordinates for all tracks we have are located in addresses 0.03000 through 0.03600. For the purposes of this problem, we will also assume that these locations contain only the y co-ordinates in the left half-words and that the right half-words contain +0. This table of y co-ordinates has been made up by a subroutine which will not be explained at this time. The program flow chart to accomplish the task we have outlined above is shown in figure 3–5.

First, the index registers are loaded to the value which will enable us to process the maximum number of tracks that could be contained within the Central Computer System. The y co-ordinates of these tracks are examined, and those with positive values are stored, starting at 0.04600. Each time we find a positive co-ordinate, we step an indicator register at 0.00100. Then the index register which is to control the iterations of the second part of the program is set by subtracting +1 from the contents of the indicator register. The starting address of the table to be used with index register 4 is determined by subtracting the indicator register from the highest address in the table. We determine whether our co-ordinates are either +70 or +50 by two *DIM* instructions. For example, if we were testing a co-ordinate of +50, the first *DIM* instruction at location 0.00021 would leave —10 in the left accumulator. The second *DIM* instruction would leave —0 in both the left and right accumulators. When the *BFZ* condition is satisfied, we *AOR* in the location containing the co-ordinate just tested, changing the right half-word from +0 to 0.00001. Thus, this program produces a table containing marker bits in R15 of each location which has a y co-ordinate of +70 or +50. Then another program can test for the presence of marker bits in the table and display the corresponding track numbers.

3.4 *ADD B REGISTERS* INSTRUCTION

The *Add B Registers (ADB)* instruction is used to add the contents of the B registers to the accumulators. The address part of this instruction is not needed because no operand is required from core memory. However, the contents of the B registers are treated just as if they were contained in a core memory location. The contents are transferred to the A registers and then added to the accumulators in the normal manner. Overflow may occur with the execution of this instruction. The *ADB* instruction, which is not indexable, requires 12 μsec to execute. It has an octal operation code of 114.

3.5 *MULTIPLY* INSTRUCTION

The *Multiply (MUL)* instruction is used to obtain the product of the contents of the accumulator and the contents of the specified memory location. Execution of this instruction leaves a 30-bit product in the combined accumulator and B register. The least significant bit of the B register (B15) is identical to the sign of the multiplier (original contents of the accumulator) and is not considered part of the product. The actual process involved in multiplication of two binary numbers is by addition and shifting, as explained in Part 1, Chapter 3. The *MUL* instruction is executed in the AN/FSQ-7 and AN/FSQ-8 in the following manner. The accumulator contents are first made positive and then transferred to the B registers. Then the addition and shifting is performed in accordance with the contents of bit 15 of the B register. These shifts are controlled by 2-mc pulses from the time pulse generator. However, some of these pulses are not associated with a particular machine cycle; therefore, most of the shifting part of the *MUL* instruction takes place during an arithmetic pause. The product generated by a *MUL* instruction is always positive at the completion of the shifting; then the true sign, which has been determined algebraically, is restored to the product. Execution time of the *MUL* instruction is 16.5 μsec, plus or minus ½ μsec. This variance allows for delay in synchronization of the 2-mc pulses. The *MUL* instruction is designated by an octal code of 250 and may be indexed.

3.6 *TWIN AND MULTIPLY* INSTRUCTION

The *Twin and Multiply (TMU)* instruction multiplies the left half-word contained in the specified memory location by the contents of the left and right accumulator. Thus, the *TMU* instruction causes the left half-word to be used as the multiplier and in both the left and right arithmetic elements. Aside from this difference, the *TMU* instruction is executed in the same manner as the *MUL* instruction. The execution time of the *TMU* instruction is 16.5 μsec, plus or minus ½ μsec, and is an indexable instruction. It is designated by an octal operation code of 254.

3.7 *DIVIDE* INSTRUCTION

Execution of the *Divide (DVD)* instruction produces the quotient of the contents of the specified memory address (divisor) and the contents of the combined accumulators and B registers (dividend). The execution of this instruction leaves an unsigned quotient of 16 significant bits in the B register and the remainder and sign of the quotient in the accumulators. Binary division in the AN/FSQ-7 or AN/FSQ-8 takes place as described in Part I, Chapter 3. The Central Computer System carries out the process of trial subtractions and shifting with the use of 2-mc pulses from the time pulse generator, as with the *MUL* instruction. However, a trial subtraction cannot be performed with one pulse, so the pulses are separated into groups of five pulses each, called divide time pulses (DVTP). Each DVTP cycle, whose pulses are numbered DVTP 0 – DVTP 4, causes

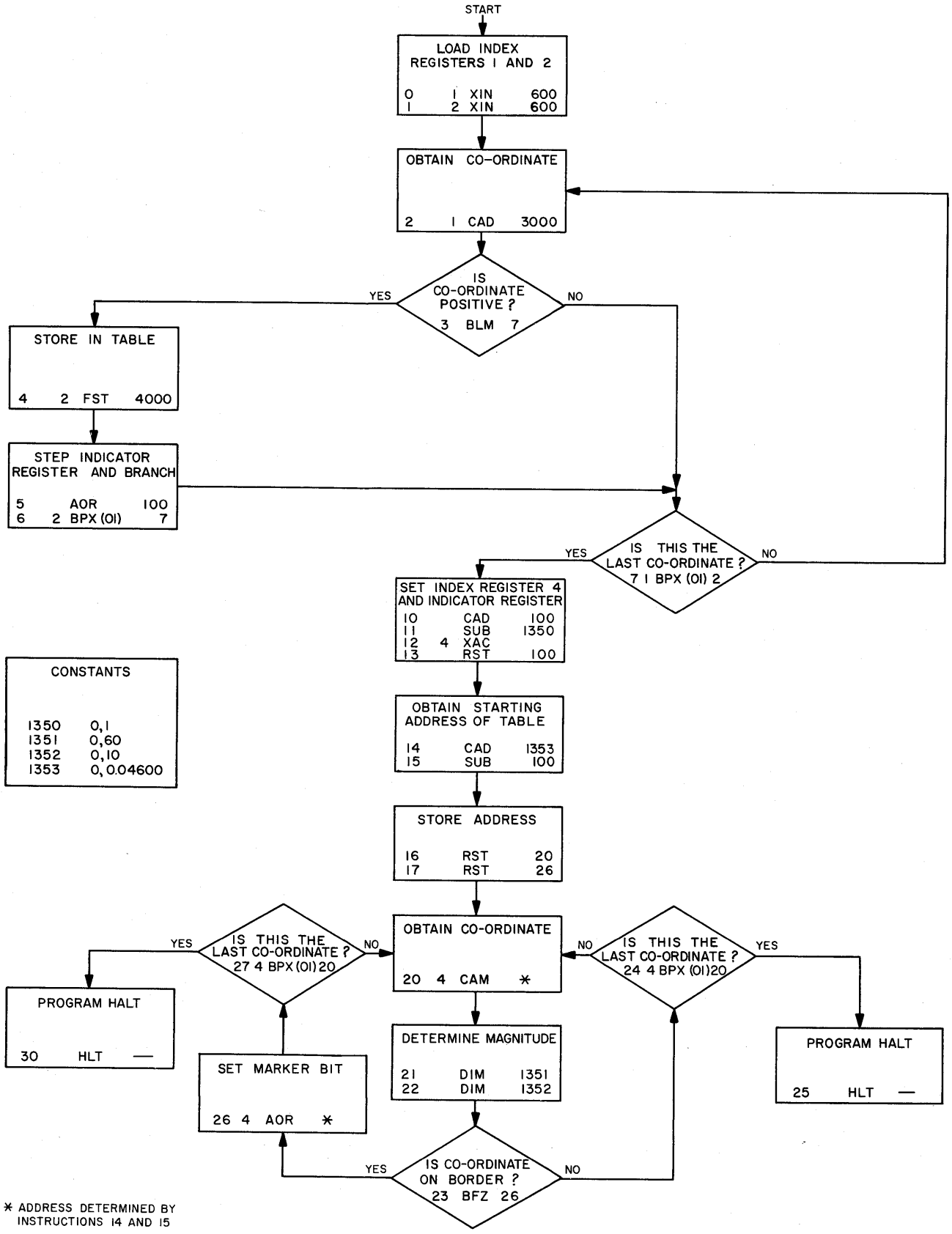


Figure 3–5. Air Traffic Program

the execution of one trial subtraction. Sixteen trial subtractions are required, yielding an arithmetic pause time of 39.0 μsec plus or minus ½ μsec. The addition of the normal PT-OT cycles brings total execution time of the *DVD* instruction to 51.0 μsec plus or minus ½ μsec. The *DVD* instruction is indexable and is specified by an octal operation code of 260.

3.8 *TWIN AND DIVIDE* INSTRUCTION

The *Twin and Divide (TDV)* instruction, which operates like all twin instructions, uses the left half-word contained in the specified memory location as the divisor for both the left and right arithmetic element. The *TDV* instruction is executed in the same manner as the *DVD* instruction. Execution time of the *TDV* instruction is 51.0 μsec plus or minus ½ μsec; this instruction is indexable. The octal operation code for this instruction is 264.

3.9 *SHIFT LEFT AND ROUND* INSTRUCTION

3.9.1 General

The *Shift Left and Round (SLR)* instruction provides the means for manipulating a number within the arithmetic element and rounding off that number of 15 significant bits. You will recall that the result of a multiplication leaves a 31-bit product (including sign) in the combined accumulators and B registers, and the result of a division leaves a 16-bit unsigned quotient plus the remainder in the combined accumulators and B registers. Neither of these results is compatible with the word length of the AN/FSQ-7 or AN/FSQ-8, since we deal with half-words of 16 bits, and these numbers are almost twice as long as the maximum length we can handle. Therefore, some method must be found to reduce these numbers to 15 significant bits and yet preserve the most accuracy it is possible to give to the magnitude of the number. This is accomplished by the *SLR* instruction, whose execution is explained below.

3.9.2 Execution

The *SLR* instruction shifts the number in the combined accumulators and B registers left the number of places specified by the right half portion of the instruction and rounds off the accumulator contents to 15 significant bits. "Shifting" in the AN/FSQ-7 or AN/FSQ-8 refers to displacing the contents of a bit position to the left or right within the same register; in the *SLR* instruction, the bits are shifted left only. The basic shift operation transfers the contents of one flip-flop to the adjoining flip-flop; this operation may be repeated as many times as specified. Thus, a shift to the left of two places will transfer the contents of bits 1 and 2 out of the register, with bit 3 moving into the bit 1 position, bit 4 moving into the bit 2 position, etc., through the entire register, except for the sign bit position which remains unchanged. Roundoff in the AN/FSQ-7 and AN/FSQ-8 is performed in much the same manner as we perform roundoff with a number on paper. In the machine, the sign bit of the B register is sensed, and if it is a 1, the contents of the accumulator are increased by 1. If the sign bit of the B register is a 0, the accumulator contents are left unchanged. Remember that the sign of the B register does not indicate polarity of its contents when used with multiplication and division instructions, but rather is a significant bit. Actual execution of the *SLR* instruction is done by first shifting the combined accumulators and B registers to the left the number of places indicated by the right half-word of the instruction. These shifts are controlled by the 2-mc pulses from the time pulse generator, with each pulse causing a shift of one place to the left. After the shifting operation has been completed, the combined registers are complemented, if negative, and the carry 1 line to the accumulator is pulsed if the sign bit of the B register is a 1. Then the sign of the accumulator contents is restored, if necessary. Assuming that the sign bit of the B register is a 1, rounding off a positive number will increase its magnitude by 1 and rounding off a negative number will decrease its magnitude by 1. The *SLR* instruction may be executed in 6 μsec if no shifts are called for; if shifting is specified, the execution time depends on the number of shifts involved. If an *SLR 1* instruction is specified, the execution time is 6.5 μsec; *SLR 2* requires 7.5 μsec execution time. This execution time for shifts of three or more can be determined by the formula: $t = 6.0 + \frac{N-1}{2}$, where N equals the number of decimal shifts that are called for. For example, *SLR 5* would require 6.0 + 2.0 or 8.0 μsec to execute. Since no memory location is referred to, this instruction may not be indexed. Overflow may occur as a result of the *SLR* instruction, which is designated by an octal operation code of 024.

3.10 SAMPLE PROGRAMS INVOLVING MULTIPLICATION AND DIVISION

3.10.1 Multiplication Programs

3.10.1.1 Basic Multiplication

Assume that we wish to write a program which will compute the value $AC^2–2AC, BD$. We will place our data in locations 0.01000 and 0.01537, and they shall contain A, B and C, D, respectively. Our answer will be placed in location 0.44444. The program to compute this value is shown in table 3–7.

Notice that we have used an *SLR* instruction after each multiplication, so that our product can be used with another instruction immediately following. No shifts have been specified in this example, since our most significant bits are already in the accumulator and merely need to be rounded off.

TABLE 3–7. BASIC MULTIPLICATION PROGRAM

LOCATION	OPERATION	ADDRESS	
0.00070	*CAD*	0.01000	
0.00071	*MUL*	0.01537	
0.00072	*SLR*	0.00000	
0.00073	*FST*	0.44444	
0.00074	*MUL*	0.01537	
0.00075	*SLR*	0.00000	
0.00076	*SUB*	0.44444	
0.00077	*SUB*	0.44444	
0.00100	*LST*	0.44444	
0.01000	A	B	Constants
0.01537	C	D	

3.10.1.2 Function Evaluation Programs

One of the common functions that we evaluated in mathematics is the general form of the quadratic equation, which is: $y = ax^2 + bx + c$. This function can be easily programmed on the AN/FSQ-7 and AN/FSQ-8. The method of solution is fairly obvious; however, the sequence of computation does make a difference in the time it takes to solve this problem. If we program this function by generating the most obvious products and sums, we follow the sequence given below:

1. Obtain x
2. Multiply by x
3. Multiply x^2 by a
4. Store partial result
5. Obtain b
6. Multiply by x
7. Add ax^2 to bx
8. Add c
9. Store result

While this solution is certainly valid, it is by no means the best way to arrive at the correct answer. Upon examination, it can be seen that the equation $y = ax^2 + bx + c$ can be rewritten to read $y = (ax + b)\,x + c$. The problem can now be solved by computing in the following manner:

1. Obtain a
2. Multiply by x
3. Add b
4. Multiply by x
5. Add c
6. Store result

In the latter method of solution, no intermediate storing is required, and only two multiplication steps are required instead of the three needed when solving by the first method. Saving multiplication steps is an important consideration when writing a program, because each multiplication step that is saved also saves a rounding-off step. The function evaluation program as it should be written is given in table 3–8.

TABLE 3–8. FUNCTION EVALUATION PROGRAM

LOCATION	OPERATION	ADDRESS	
0.00000	*CAD*	0.15320	
0.00001	*MUL*	0.04000	
0.00002	*SLR*	0.00000	
0.00003	*ADD*	0.15321	
0.00004	*MUL*	0.04000	
0.00005	*SLR*	0.00000	
0.00006	*ADD*	0.15322	
0.00007	*FST*	0.62000	
0.00010	*HLT*	0.62000	
0.15320	A	A	Constants
0.15321	B	B	
0.15322	C	C	
0.04000	X	X	

Of course, this program will generate two solutions simultaneously in both the left and right arithmetic element. If we wished to construct a table of the values of y, we would probably store the accumulators separately in sequential locations and then rearrange the table so that all values of y were in order and in the same half-word.

3.10.1.3 Co-ordinate Conversion Program

When targets on tracks are reported to the AN/FSQ-7, they are given in terms of range and azimuth, or polar co-ordinates. It is necessary for us to have these co-ordinates converted to rectangular form so they may be more easily dealt with in future computations. This may be accomplished by multiplying the sine and cosine of the angle of azimuth by the range so that $x = r \sin \phi$ and $y = r \cos \phi$. If we use a table of sine and cosine values such as was used previously for the program listed in table 3–5, our sine and cosine values will be located in the left and right half-words of memory locations 0.12200 through 0.12577. We will also assume that we wish to perform conversions on 50_{10} tar-

TABLE 3–9. CO-ORDINATE CONVERSION PROGRAM

LOCATION	OPERATION	ADDRESS
0.00000	*1 XIN*	0.00061
0.00001	*1 CAD*	0.00020
0.00002	*3 CAD*	0.12200
0.00003	*1 TMU*	0.00020
0.00004	*SLR*	0.00000
0.00005	*1 FST*	0.01100
0.00006	*1 BPX (01)*	0.00001
0.00007	*HLT*	–
0.00020 -		
0.00101	Range	Ø (azimuth angle)
0.01100 -		
0.01161	X co-ordinate	Y co-ordinate
0.12200 -		
0.12577	Sin Ø	Cos Ø

gets which have their range and azimuth angle in the left and right half-words, respectively, of locations 0.00020 – 0.00101. The program to generate these x and y co-ordinates is given in table 3–9.

The proper sine and cosine are selected for each target by first obtaining the angle of the target and then using the table lookup procedure, utilizing the right accumulator as an index register. This step will place the sine of the angle in the left half-word and the cosine in the right half-word. A *TMU* instruction will multiply both of the accumulators by the range of the target, leaving R sin ϕ and R cos ϕ in the left and right accumulators. These results are then rounded off and stored. The example given above shows how easily an actual problem may be handled by the Central Computer System when the various facilities such as dual computation, twinning, and table lookup procedures are employed.

3.10.2 Division Programs

3.10.2.1 Requirements for Division

Up to this point, we have not discussed the problem of division in the AN/FSQ-7 and AN/FSQ-8 from any standpoint except the actual execution of the instructions. However, we know that the Central Computer System has been designed to handle only those numbers which fall between the limits of +1 and –1. In multiplication, no difficulty was encountered since the product of two fractional numbers is always another fraction. But in division the quotient of two fractions may be a number larger in magnitude than the AN/FSQ-7 and AN/FSQ-8 can handle. Therefore, we must make sure that the operand to be used as a divisor is larger in magnitude than the dividend to always obtain a fractional answer. If this precaution is not taken prior to execution of the *DVD* or *TDV* instruction, the result in the combined accumulator and B register may be meaningless.

3.10.2.2 Division Example

We will assume that we wish to evaluate the following formula:

$x = \frac{C}{A} + \frac{A}{D}$. Memory locations 0.00427 and 0.00077 contain A, B, and C, D, respectively. The result

TABLE 3–10. DIVISION PROGRAM

LOCATION	OPERATION	ADDRESS
0.00030	*CAD*	0.00077
0.00031	*SUB*	0.00427
0.00032	*BLM*	0.00034
0.00033	*HLT*	–
0.00034	*CAD*	0.00033
0.00035	*TAD*	0.00427
0.00036	*SUB*	0.00077
0.00037	*BRM*	0.00041
0.00040	*HLT*	–
0.00041	*CAD*	0.00077
0.00042	*DVD*	0.00427
0.00043	*SLR*	0.00017
0.00044	*RST*	0.43300
0.00045	*CAD*	0.00033
0.00046	*TAD*	0.00427
0.00047	*DVD*	0.00077
0.00050	*SLR*	0.00017
0.00051	*ADD*	0.43300
0.00052	*RST*	0.43300
0.00053	*HLT*	–
0.00427	A	B
0.00077	C	D
0.43300		Result

is to be placed in the right half-word of location 0.43300. Table 3–10 lists the program to evaluate this formula.

The first part of this program checks the numbers to be divided and halts if any divisor is not larger than its dividend. If both divisors are larger, the program will continue. After each division is executed, the *SLR 17* instruction moves the unsigned 16-bit quotient from the B registers into the accumulators and rounds off the result to 15 significant places, the maximum half-word length we can store in one memory.

3.11 SHIFT INSTRUCTIONS

3.11.1 General

The shift instructions are used to position words within the combined accumulator and B register or within the accumulator alone. The principle of the shift instructions is the same as that of the *SLR* instruction: bits are simultaneously transferred to their adjacent positions, with the number of shifts determined by the contents of the right half-word. The *SLR* instruction does not fall into the general shift instruction classification since it may specify only a shift left of the combined accumulators and B registers and also performs a rounding-off operation. All of the other shift class instructions merely position the word to the left or right the number of specified places and do not round off. When the contents of the accumulator are shifted to the right, the sign of the accumulator remains unchanged but is shifted into each position vacated by the shift. In addition, bits shifted out of bit 15 of the accumulator (or bit 15 of the B register if the two are combined) are lost. When a shift to the left occurs, the sign bit remains unchanged but is shifted into each position of the accumulator (or B register) that is vacated. Bits shifted out of accumulator bit 1 are lost.

The placing of the sign bit into the high or low order bits is, in effect, placing 0's in those positions. If the number is positive, 0 bits will be shifted into each bit position vacated. If the number is negative, 1's are placed in each vacated position. However, a 1 bit in a negative number represents a magnitude of 0 since the number is in complement form.

Shift instructions are used for two general purposes. In logical or nonarithmetic operations, the bits may be positioned in order to conduct a test of certain bits or bit combinations. In arithmetic operations, numbers are shifted to increase or decrease their magnitudes; this process is known as scaling. Scaling within the Central Computer System is discussed in Part 5, Chapter 2.

Because shift instructions do not refer to a core memory location, they may not be indexed. In addition, overflow will not occur upon the execution of a shift instruction since the sign bits are not changed during the execution.

The time required for the execution of the shift instructions depends on the number of shifts involved. If six or less shifts are specified, the instruction may be executed in 6 μsec. If more than six shifts are required, the Central Computer System uses an arithmetic pause to complete the operation. Execution time for shifts of six or more can be determined by the formula: $t = 6.0 + 0.5\ (N-5)$, where N equals the number of decimal shifts that are called for. For example, *DSR 10* would require 6.0 + 0.5 (8–5) or 7.5 μsec to execute. It should be noted that only bits R10 through R15 are decoded to determine the number of shifts; therefore, the maximum number of shifts that may take place during any one instruction is 63_{10} or 77_8. If more than 77_8 shifts are called for, only the value represented in bits R10 through R15 will actually be performed. For instance, if we try to shift 105_8 positions, only 5_8 shifts will be executed. A description of the various shifting instructions used in the AN/FSQ-7 and AN/FSQ-8 is presented below.

3.11.2 *Dual Shift Left* Instruction

The *Dual Shift Left (DSL)* instruction combines the accumulator and B register of both arithmetic elements into a 32-bit register which may be shifted only to the left. The sign bits of both accumulators are duplicated in bit 15 of the B registers as the shifting operation begins, and bits shifted out of accumulator bit 1 positions are lost. The *DSL* instruction is designated by an octal operation code of 400. Execution of this instruction is illustrated in figure 3–6, part A.

3.11.3 *Dual Shift Right* Instruction

The *Dual Shift Right (DSR)* instruction combines the accumulator and B register of both arithmetic elements into 32-bit registers for the purpose of shifting numbers to the right. The sign bits of both accumulators are duplicated in accumulator bit 1 positions, and bits shifted out of B register bit 15 positions are lost. An octal operation code of 404 is used to specify a *DSR* instruction. This instruction execution is shown in figure 3–6, part B.

3.11.4 *Left Element Shift Right* Instruction

The *Left Element Shift Right (LSR)* instruction combines the left accumulator and left B register into a 32-bit shifting register. The right accumulator and right B register are not affected by the execution of this instruction. The sign of the left accumulator is duplicated in the L1 bit position, and all bits shifted out of bit 15 of the left B register are lost. The *LSR* instruction is designated by an octal operation code of 440. Execution of this instruction is shown in figure 3–6, part C.

3.11.5 *Right Element Shift Right* Instruction

This instruction is similar to the *LSR* instruction except that it deals with the right arithmetic element only. Thus, the right accumulator and right B register

are combined into a 32-bit shifting register whereas the left arithmetic element is not affected. The *Right Element Shift Right (RSR)* instruction duplicates the sign bit of the right accumulator in the R1 bit position, and all bits shifted out of bit position 15 of the right B register are lost. The *RSR* instruction, which has an actual operation code of 444, is shown in figure 3–6, part D.

3.11.6 *Accumulators Shift Left* Instruction

The *Accumulators Shift Left (ASL)* instruction shifts both accumulators to the left the number of places specified by the right half-word. The execution of this instruction does not affect the contents of the B registers. The sign bits are duplicated in bit position 15 of the accumulators, and all bits shifted out of bit positions L1 and R1 are lost. Because this instruction does not combine the accumulators and B registers, only the 15 magnitude bits of the accumulators may be shifted. Thus, an *ASL* 17_8 (or greater) will duplicate the contents of the sign bits in all positions of the accumula-

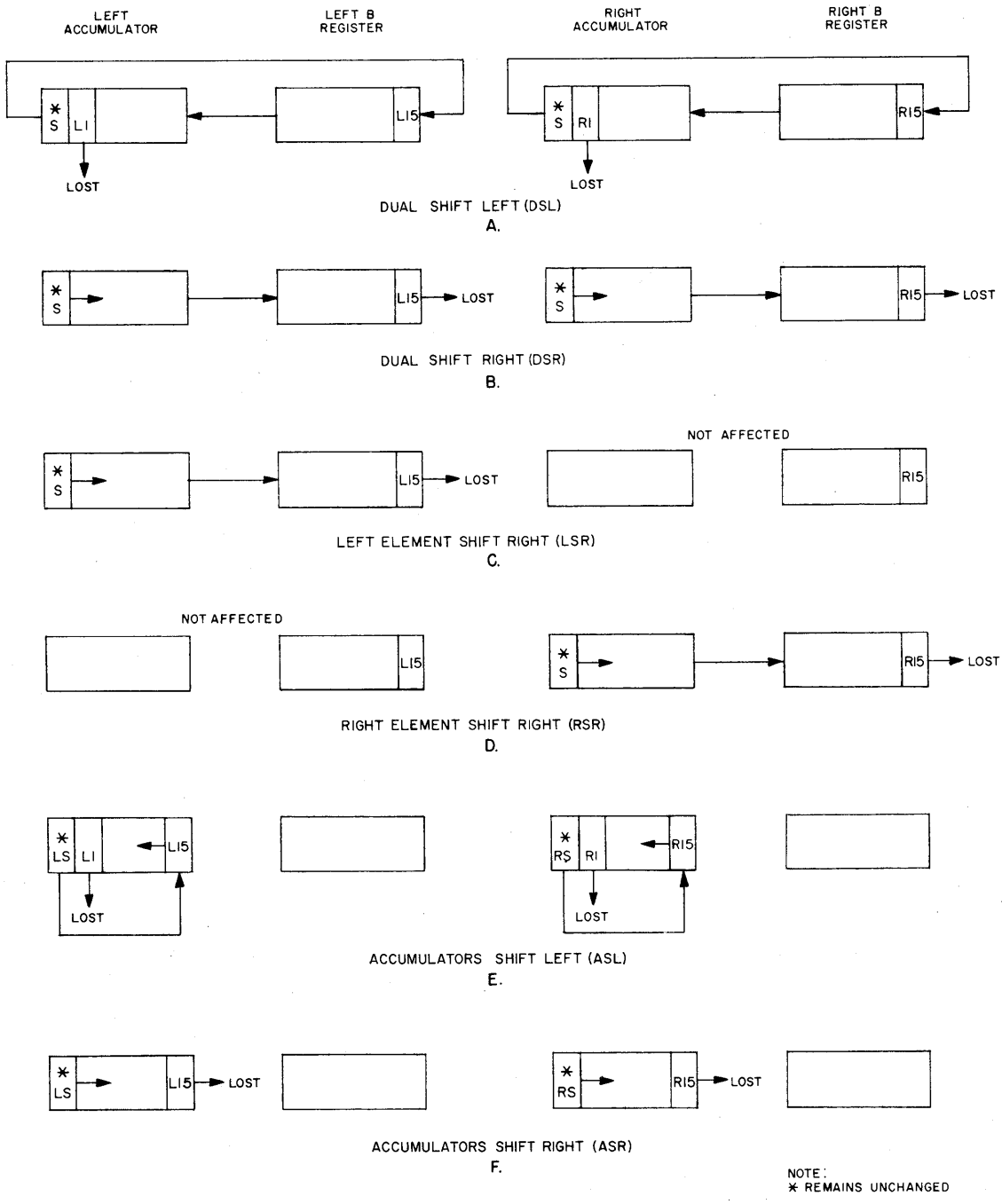


Figure 3–6. Execution of Shift Instructions

tors. The *ASL* instruction is specified by an octal operation code of 420. Its execution is illustrated in figure 3–6, part E.

3.11.7 *Accumulators Shift Right* Instruction

This instruction shifts the contents of both accumulators to the right without disturbing the contents of the B registers. The sign bit is duplicated in bit positions L1 and R1 of the accumulators, and bits shifted out of bit positions L15 and R15 are lost. As with the *ASL* instruction, a shift of 17_8 or greater will make all bit positions in both accumulators identical with their respective sign bits. The *Accumulators Shift Right (ASR)* instruction is designated with an octal operation code of 424 and is shown in figure 3–6, part F.

3.11.8 Program Examples of Shift Instructions

Let us assume that we wish to find out how many numbers stored in memory locations 0.01000–0.01031 are larger in absolute magnitude than a particular constant. We can do this by placing the constant in the accumulators and then using a *DIM* instruction to compare magnitudes. However, the action of the *DIM* instruction is such that the original contents of the accumulators are first duplicated in the B registers. Therefore, after testing to see if the number meets our conditions, we can *DCL* and restore the number to the accumulators without going into memory again. The flow chart to accomplish this is shown in figure 3–7.

As another example of the use of shift instructions, let us examine a table that contains information in the

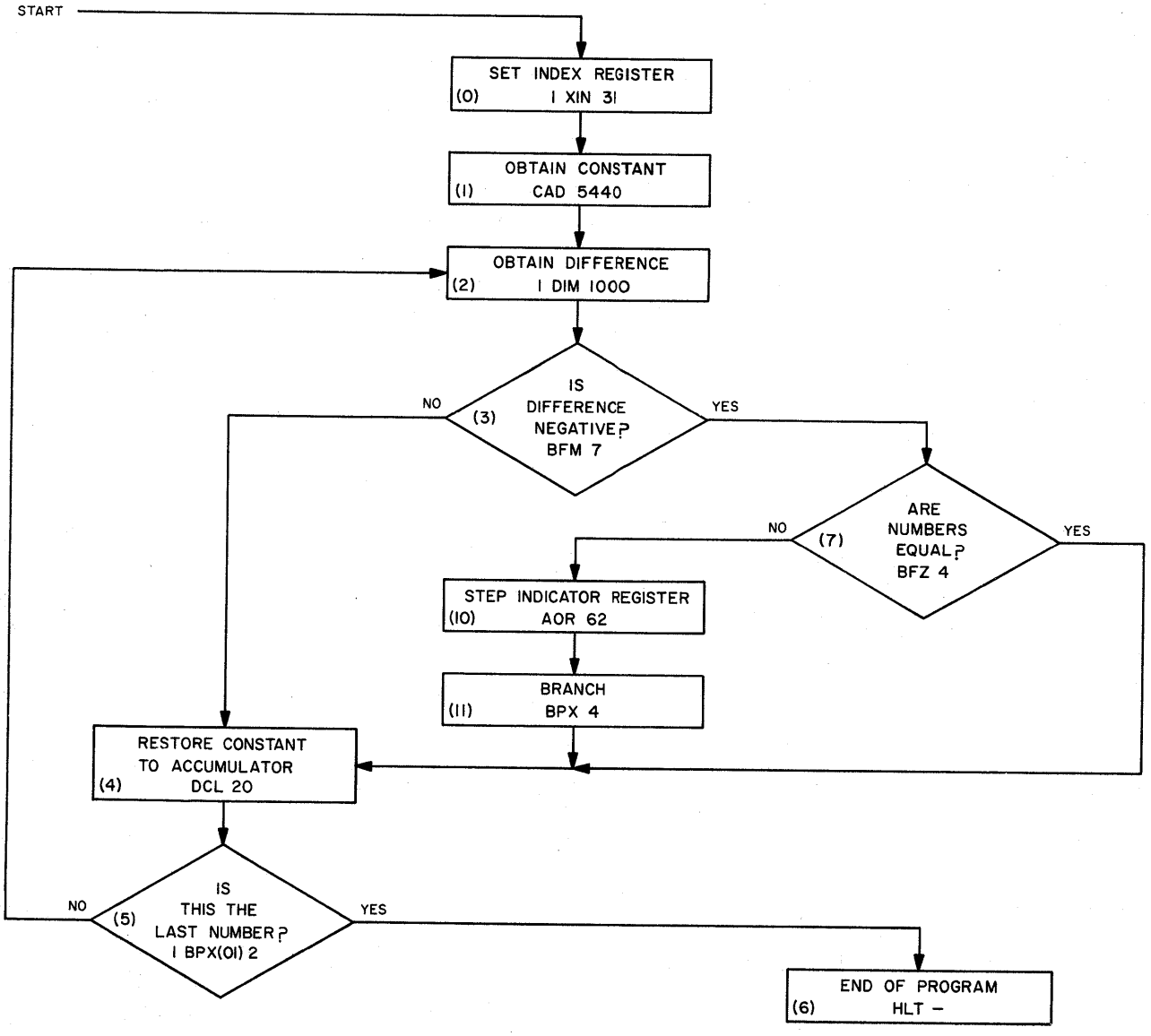


Figure 3–7. Magnitude Sorting Program

right half-word only. The left half-words contain +0. The right half-words have marker bits (1's) placed at one particular position within a half-word, but the bit position used does not have to be the same in each word. What we want to do is make up another table telling us in what bit position the marker bit is located in each half-word. The program will search locations 0.34040 through 0.34047 and store the results in locations 0.00050 through 0.00057. A constant of 20_8 is stored in the right half-word of memory location 0.00300. The program flow chart for this example is shown in figure 3–8.

After obtaining the location, we check to make sure that it contains an operand by use of the *BFZ* instruction at step 0.00002. If the branch condition is not satisfied, we know that the right half-word is not 0 and contains a marker bit. Then we shift the right accumulator one bit position to the right and step an indicator

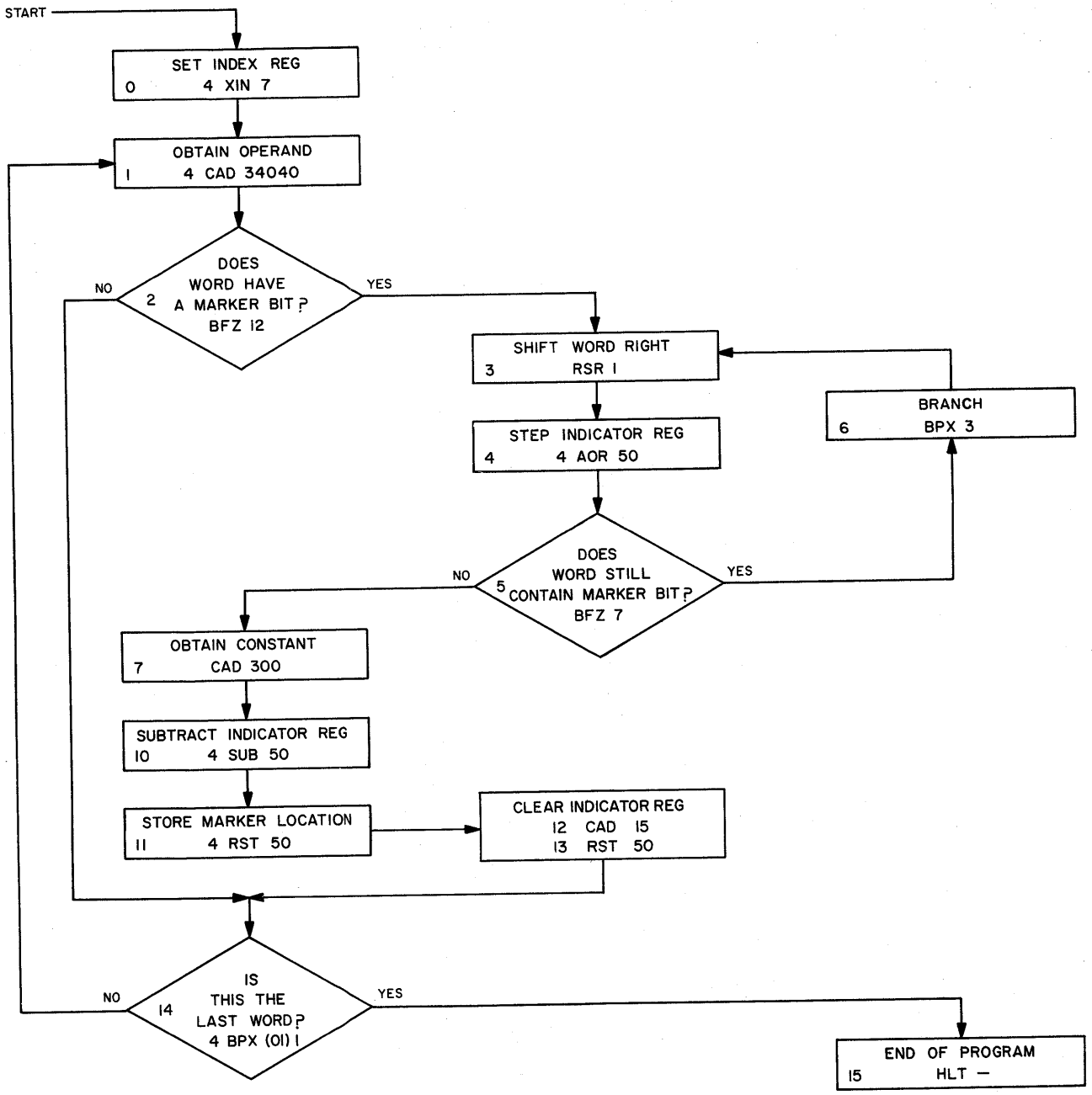


Figure 3–8. Marker Bit Identification Program

register. If the next check for 0 reveals that the right half-word has been shifted so that the marker bit is no longer in the word, the position of this marker can be determined by obtaining a constant of 20_8 and subtracting the contents of the indicator register from it. Thus, if the marker bit for a particular word was located in bit position R9, seven shifts would be required to remove this bit from the right half-word. Subtracting seven from our constant of 20_8 would leave 11_8 (or 9_{10}) in the accumulator. The marker location is then stored, the indicator register is cleared, and the next operand selected if the program is not completed.

Although only two examples of shift instructions have been given, it should be clear in what manner these instructions deal with the accumulator and accumulator B register contents. Further and more complete coverage of shift instructions will be given in the chapter dealing with scaling of numbers to be used in the Central Computer System.

3.12 CYCLE INSTRUCTIONS

3.12.1 General

There are two instructions which are similar in operation to the shift instructions, except that the sign bit is shifted in the same manner as the magnitude bits. In addition, no bits are lost by the shifting process but are re-entered into the low order position of the registers involved. The shifting of bits within the registers is accomplished in the same manner as with the shift instructions.

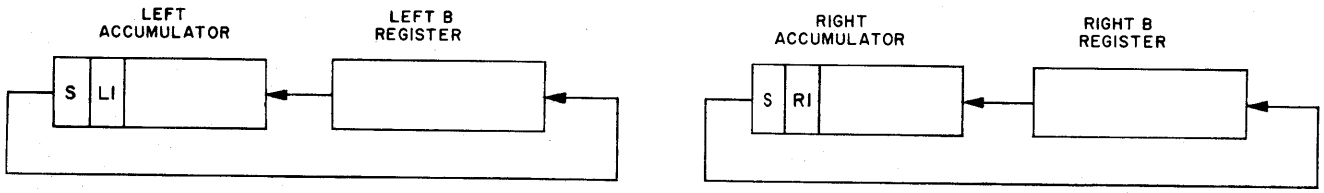
Cycle instructions do not refer to a memory location and therefore cannot be indexed. Their principal use is to exchange the contents of the left and right accumulators and to exchange the contents of an accumulator and its associated B register. As with the shift instructions, the cycle instructions have a variable execution time, depending on the number of shifts involved.

3.12.2 *Dual Cycle Left* Instruction

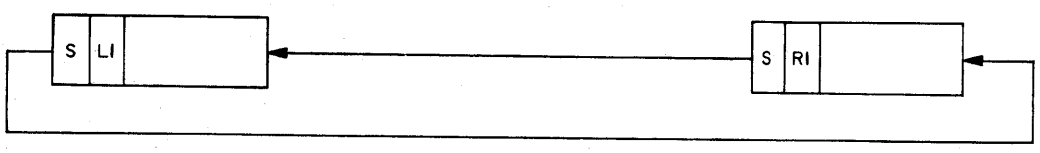
The *Dual Cycle Left (DCL)* instruction combines the accumulator and B register of each arithmetic element into a 32-bit cycling register. (See fig. 3–9, part A.) Bits shifted out of the accumulator sign bit positions are re-entered into the bit-15 position of the B registers. Bits shifted out of the sign bit position of the B registers are shifted into bit 15 of the accumulators. When 40_8 shifts are specified, the execution of the *DCL* instruction will be equivalent to no shifts at all, since all bits will be restored to their original position. When greater than 40_8 shifts are specified, the final result will be the number specified in the right half-word minus 40_8. For instance, when *DCL* 50_8 is specified, 50_8 shifts will take place, but will have the same effect as a *DCL* 10_8 instruction. The *DCL* instruction will not cause an overflow and may be executed in 6 μsec if six shifts or less are called for. If more than six shifts are specified, execution time is variable. The *DCL* instruction is designated by an octal operation code of 460.

3.12.3 *Full Cycle Left* Instruction

The *Full Cycle Left (FCL)* instruction is used to interchange bit positions of the left and right half-words in the arithmetic element. Thus, the two accumulators are combined as shown in figure 3–9, part B. The contents of the B registers are not affected by the execution of the *FCL* instruction. Bits leaving the LS bit position are re-entered into the R15 bit position, and bits leaving the RS bit position are entered into the L15 bit position. If 20_8 shifts are called for, the *FCL* instruction, specifying 40_8 shifts, will bring the bit posi-



DUAL CYCLE LEFT (DCL)
A.



FULL CYCLE LEFT (FCL)
B.

Figure 3–9. Execution of Cycle Instructions

tions into their original places and will be equivalent to no shifting action. When more than 40_8 shifts are called for, the effect is the same as it is for the *DCL* instruction. Execution time of the *FCL* instruction can be 6 μsec if six shifts or less are needed; otherwise, it is variable. The *FCL* instruction, which will not cause on overflow, is designated by an octal operation code of 470.

3.12.4 Examples of Cycle Instructions

Assume that we have operands A, B located in memory address 0.10300 and operands E, F located in memory address 0.40000. We wish to obtain and store the quantity $(2BE)^2$, $(AB)\ E^2$ in memory address 0.00700, which is cleared. Notice that this quantity contains the operand B in the left half-word, yet the original operand B is contained in the right half-word of location 0.10300. We have no technique available for directly placing the right half-word operand into the left half of the arithmetic element, so some other method must be used to position the quantity as we desire. This is done by cycling between accumulators, as shown in table 3–11.

As another example of the use of cycle instructions, we will consider a problem that is similar to one actually solved in the AN/FSQ-7 and AN/FSQ-8. We have several maintenance programs which are identified by a 4-digit code in the left half-word (referred to as the program number) and miscellaneous information in the right half-word relating to the format of the program. This identification word is placed at the beginning of each maintenance program and is used to set up a display so that the operator will know what program is about to be run. For instance, suppose the program number is 6753. In order to display this number on the face of the display tube, we need to make up a word with a specific combination of bits. The combination actually required is not important at this point; we will assume that the proper combination needed for each digit is stored in a table. Our problem is to gain access to this table and make up the proper word. The program to accomplish this is listed in table 3–12.

TABLE 3–11. APPLICATION OF CYCLE INSTRUCTION

LOCATION	OPERATION	ADDRESS
0.02670	*CAD*	0.10300
0.02671	*TMU*	0.40000
0.02672	*SLR*	0.00000
0.02673	*RST*	0.00700
0.02674	*ADD*	0.00700
0.02675	*FCL*	0.00020
0.02676	*LST*	0.00700
0.02677	*MUL*	0.00700
0.02700	*SLR*	0.00000
0.02701	*FST*	0.00700
0.02702	*HLT*	—
0.10300	A	B
0.40000	E	F
0.00700	Result	

The program is executed in the following manner. Index register 5 is loaded to a value of three, since we wish to make four passes through the routine: the number of digits in the program number. Then the result register is cleared, and the identification word is brought into the accumulator. Remember that the program number is contained in the left half-word. An *FCL* 20_8 instruction places the program number in the right half-word and the *DCL* 20_8 instruction will further move it into the right B register. The accumulator is then cleared, and a *DCL* 4_8 moves the sign bit and the first three bits of the B register into the accumulator. This is really the first digit of our program number, or 6. Four bits are required to designate a digit in this case, because 7_{10} is the largest number that can be represented by three bits, and the program number might contain an 8_{10} or 9_{10}, which would require four bits. Thus, the sign bit of this half-word is a magnitude bit rather than a true sign bit. Now that the first digit of our program number is in the least significant position of the right accumulator, we can go through a table lookup procedure using the contents of the right accumulator as an index register. This will place the proper bit combination for six in the arithmetic element. These bit combinations consist of three bits in the left half-word and three bits in the right half-word. (These bits are actually the x and y co-ordinates necessary to obtain the digit 6 and display it.) At step 0.00271, we shift the accumulators left three places to make room for the next pair of x and y co-ordinates. A check of the index register reveals that it is still positive, so a branch is made to obtain the next digit, etc.

At the completion of this program, the proper bits will be contained in memory location 0.03341. The last three bit positions of each half-word will contain 0's, since we are only dealing with a 4-digit program number. The word contained in location 0.03341 can then be used by another routine which will select the proper pairs of x and y co-ordinates and, hence, display the proper digit.

TABLE 3–12. DISPLAY MAKEUP PROGRAM

LOCATION	OPERATION	ADDRESS
0.00260	*5 XIN*	0.00003
0.00261	*CAD*	0.00274
0.00262	*FST*	0.03341
0.00262	*CAD*	0.00065
0.00263	*FCL*	0.00020
0.00264	*DCL*	0.00020
0.00265	*CAD*	0.00274
0.00266	*DCL*	0.00004
0.00267	*3 CAD*	0.01000
0.00270	*ADD*	0.03341
0.00271	*ASL*	0.00003
0.00272	*FST*	0.03341
0.00273	*5 BPX (01)*	0.00265
0.00274	*HLT*	—
0.03341	Result	—
0.00065	Program number	Format
0.01000 - 0.01011	Table with bit combinations	

3.13 LOGICAL INSTRUCTIONS

3.13.1 General

Some instructions can be executed within the Central Computer System of the AN/FSQ-7 or AN/FSQ-8 that have a purely logical function. In other words, although they may use arithmetic to arrive at a certain result, that result is usually not interpreted as an arithmetic answer. The branch instructions and shift instructions could be thought of as logical instructions; however, they do not employ arithmetic processes during their execution. The instructions we are going to discuss involve binary arithmetic to modify or leave unchanged certain bits within a specified word. The arithmetic employed is not pure binary but a type referred to as logical arithmetic. In logical arithmetic, the rules of binary arithmetic apply except that no consideration is given to carries from one bit position to the next. Thus, the logical sum of a "one" and a "one" is still a "one," with no carry. Because each bit position is treated independently, certain portions of a word within the Central Computer System may be dealt with without affecting the rest of the word. We have already mentioned the logical addition process, but the one that is used most widely is logical multiplication. Logical multiplication involves multiplying one bit position by another and finding the product in accordance with the rules of binary arithmetic but without shifting the product. For instance, the logical product of 0.00101 and 1.10011 is 0.00001. The contents of the two least significant bits were 1; therefore, the product in that position is a 1. All other positions contain a 0 because at least one of the factors in each of those products contained a 0. The significance of logical multiplication will be seen more clearly during explanation of the various instructions.

3.13.2 *Extract* Instruction

3.13.2.1 Execution

The *Extract (ETR)* instruction is used to obtain the product of a logical multiplication between the contents of the accumulator and a control word contained in the memory location specified by the right half of the instruction. Each bit position in the control word which contains a 0 will clear the corresponding bit position in the accumulator, since multiplying anything by 0 will give a result of 0. Each bit position in the control word which contains a 1 will leave the corresponding accumulator bit unchanged. If the accumulator bit was a 0, multiplying by 1 will still yield a 0, and if the accumulator bit was 1, the result will still be a 1. Execution of the *ETR* instruction takes place in the following manner. The control word is read out of memory into the A registers. The A registers are then logically multiplied by the accumulators, with the results left in the accumulators. The *ETR* instruction, which may be indexed, requires 12 μsec to execute. It is designated by an octal operation code of 004.

3.13.2.2 Program Example

Suppose that we wish to sort through a block of numbers stored in memory and place all those that are even in one table and all those that are odd in another table. Assume that the numbers are contained in the right half-words of the memory locations and that the left half-words are cleared. We can determine whether a number is odd or even by examining the contents of the least significant bit. If the bit is a 1 we know it is odd; if it is a 0 we know the number is even. We can use the *ETR* instruction to examine the contents of the least significant bit of the number by the method shown in table 3–13.

Notice that the control word contains all 0's except for the least significant bit position of the right half-word. This position, which contains a 1, will leave bit 15 of the right half-word unchanged during execution of an *ETR* instruction. After the *ETR* instruction has been executed, the accumulators will be cleared by the control word with the exception of bit 15. If bit 15 originally contained a 1, it will still contain a 1, and the

BFZ condition immediately following the *ETR* instruction will not be satisfied. If this is the case, the program will fall through to the routine for storing the number in the odd table. If the bit was 0 originally, it will still be 0, and the program will branch to location 0.00772, where the number will be stored in the even table.

The control word used in executing the *ETR* instruction is commonly referred to as a "mask." In effect, it masks out those bits which are not to be utilized during the execution of the instruction and leaves unchanged the active bits, or those that are used.

3.13.3 *Load B Registers* Instruction

The *Load B Registers (LDB)* instruction is used to load the B registers with the contents of the memory location specified by the right half portion of the instruction. The *LDB* instruction could be used when it is desired to place a word in the B registers prior to execution of the sort shown in table 3–12, when a program number was cycled into the B register from the accumulators with a *DCL* 20_8 instruction. The number could have been stored and then placed in the B registers by use of the *LDB* instruction rather than by cycling. However, the main use of the *LDB* instruction is to place a control word or mask into the B registers in preparation for the execution of another logical instruction, called *Deposit,* which is explained below. The *LDB* instruction is indexable and requires 12 μsec to execute. It is designated by an octal operation code of 030.

TABLE 3–13. NUMBER DETERMINATION PROGRAM

LOCATION	OPERATION	ADDRESS
0.00760	*1 XIN*	0.00400
0.00761	*2 XIN*	0.00400
0.00762	*5 XIN*	0.00400
0.00763	*1 CAD*	0.25000
0.00764	*ETR*	0.00500
0.00765	*BFZ*	0.00772
0.00766	*1 CAD*	0.25000
0.00767	*2 RST*	0.40000
0.00770	*2 BPX (01)*	0.00775
0.00771	*HLT*	—
0.00772	*1 CAD*	0.25000
0.00773	*5 RST*	0.70000
0.00774	*5 BPX (01)*	0.00775
0.00775	*1 BPX (01)*	0.00763
0.00776	*HLT*	0.00763
0.00500	0.00000 (Control word)	0.00001
0.25000 - 0.31000	Data Storage	—
0.40000 - 0.40400	Storage for odd numbers	
0.70000 - 0.70400	Storage for even numbers	

3.13.4 *Deposit* Instruction

3.13.4.1 Execution

The *Deposit (DEP)* instruction allows the replacement of part of a word in core memory with a corresponding part of the accumulator contents on a bit-by-bit basis rather than by a full word or a half-word. The *DEP* instruction refers to a mask in the B register (usually placed there by an *LDB* instruction) to determine which bits from the accumulator will be stored. If the bit in the B register contains a 1, the corresponding bit in the accumulator will be stored in the memory word; if the mask contains a 0, the corresponding bit in the memory location will be unchanged. The memory location to be modified is specified by the right half portion of the *DEP* instruction. Execution of the *DEP* instruction takes place in the following manner. The accumulator contents are first complemented; then the mask from the B register is transferred to the A register. A logical multiplication takes place between the accumulators and A registers, with the result that the accumulators will be cleared except for those bits in the mask which contained a 1. In effect, this first logical multiplication erases those bits which are not active. The accumulator contents are then complemented again, and the specified memory location is transferred to the A registers. It should be noted that this transfer is not a normal one, since the A registers are not cleared before the transfer from the memory buffers, as is usually the case. This step is equivalent to a logical addition, since the mask was already in the A registers, and will cause each position of the memory location contents to be changed to a 1 wherever a 1 exists in the mask. Then another logical multiplication takes place between the A registers and accumulators, leaving the results in the accumulators. The *DEP* instruction execution is completed with the storing of the accumulator contents back into the specified location.

The *DEP* instruction may be indexed and requires a total of 18 μsec to execute due to the fact that the word is obtained from memory, modified, and then returned to memory. An octal operation code of 360 is used to designate a *DEP* instruction. Because the execution of this instruction is rather difficult to follow, a typical example using a half-word is shown in table

TABLE 3–14. *DEPOSIT* INSTRUCTION EXECUTION

ACTION	ACCUMULATOR	A REGISTER	MEMORY LOCATION
Start	0.010 101 111 000 011	0.000 000 000 000 000	0.000 100 011 111 101
Complement	1.101 010 000 111 100	0.000 000 000 000 000	0.000 100 011 111 101
Transfer	1.101 010 000 111 100	0.111 000 000 000 111	0.000 100 011 111 101
Logical Multiply	0.101 000 000 000 100	0.111 000 000 000 111	0.000 100 011 111 101
Complement	1.010 111 111 111 011	0.111 000 000 000 111	0.000 100 011 111 101
Logical Add	1.010 111 111 111 011	0.111 100 011 111 111	0.000 100 011 111 101
Logical Multiply	0.010 100 011 111 011	0.111 100 011 111 111	0.000 100 011 111 101
Store	0.010 100 011 111 011	0.111 100 011 111 111	0.010 100 011 111 011

3–14, when the B register contains a mask of 0.111 000 000 111.

3.13.4.2 Program Example

Let us consider a group of programs which are being run in the AN/FSQ-7 or AN/FSQ-8. One of these programs requires tables of tactical data which are compiled by three other programs. Therefore, these three programs are really inputs to the fourth program. The data which is compiled by the three input programs comes into the Central Computer System in random fashion; we do not know when the three tables will be completed. However, as soon as they are, we want to branch immediately to the program which utilizes them. What we need is something to tell us when one of the programs has been completed and also something to tell us when all three programs are completed. Thus, we need a subroutine which performs a decision-making function to tell use whether the tables are ready to be used.

The conditions for the problem are as follows. There are three registers in memory which contain +0 when the associated table has not been completed; when the table is completed, the right half-word will be set to −0. In addition, there is an indicator register which contains the status of various tables within the Central Computer System. This register contains +0 in the left half-word and various values in the right half-word, depending on the status of the tables. There are 16 bit positions in a half-word; each one contains the status of one table. If the table is complete, the bit position contains a 1; if the table is incomplete, the bit position contains a 0. We will assume that the bit positions assigned to the three tables we are concerned with are R11, R14, and R15. This register is what we will be checking to see if the three tables are complete. The program to accomplish this is shown in figure 3–10.

This program may be entered from any of the three table makeup programs whose last few instructions are shown. Notice that as soon as each program is completed, it stores −0 in a corresponding location. The decision-making program sets index register 5 to two since we will be checking the status of three bits within the permanent indicator register. Each temporary storage register associated with one of the table makeup programs is checked for −0 and sets the corresponding bit in the permanent status register if the temporary register contains −0. Notice that we are taking the contents of a half-word and reducing it to just one bit. That is, the status of a table was indicated by a negative or positive zero in the right half-word of a certain register and the *DEP* instruction reduces this indication to just one bit. If the table was completed, −0 will be in the right accumulator, the proper mask will be loaded into the B register, and the bit specified by this mask will be stored in the permanent indicator register. Then a constant is placed in the accumulators and is used to direct the placement of bits R11, R14, and R15 into the accumulator. The constant is then subtracted, and if all three bits contained a 1, a −0 will result in the accumulator. The program can then take appropriate action.

3.13.4.3 Summary of *Deposit* Instruction

The program example discussed above may seem cumbersome and unnecessarily long when applied to that problem alone. However, it should be remembered that adding more masks and constants would enable us to deal with any bit or combination of bits stored in the permanent indicator register. Moreover, we can change the status of one bit within the indicator register by use of the *DEP* instruction without affecting the remaining bits. Thus, the chief use of the *DEP* instruction is to provide for a storage of less than one half-word in any bit position or combination of positions without destroying the remaining bits. This facility is especially useful when it is desired to update information (change co-ordinates, etc.) within a word when only part of the

word requires changing and all the other bit positions are still significant.

3.14 *EXCHANGE* INSTRUCTION

3.14.1 Execution

The *Exchange (ECH)* instruction is used to interchange the contents of the accumulators with the contents of the memory location specified by the right half portion of the instruction. Thus, the *ECH* instruction is equivalent to an *FST* instruction followed by a *CAD* instruction, with the exception that the same memory location is specified in each case. Execution of this instruction takes place in the following manner. The word from core memory is first transferred to the A

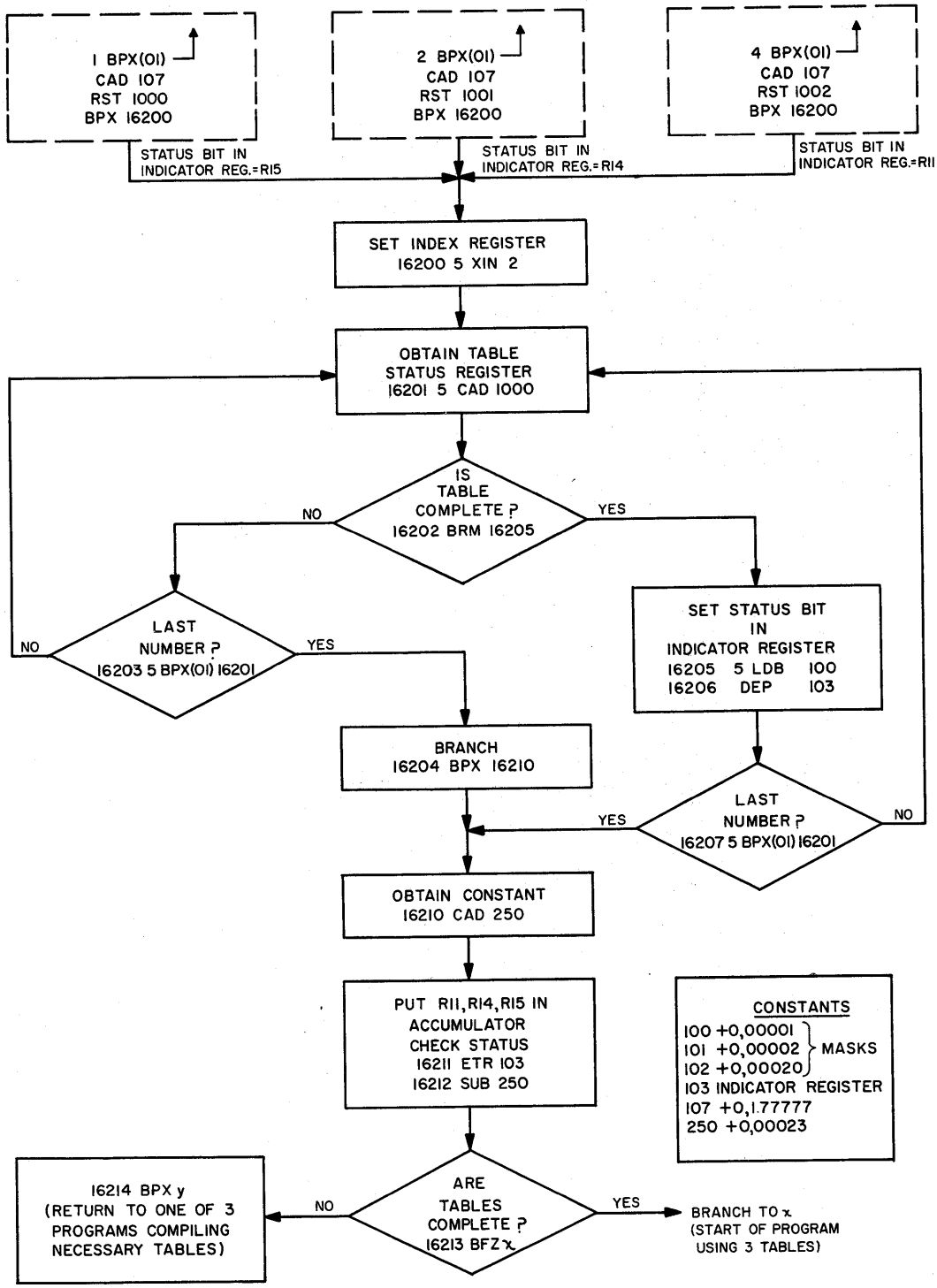


Figure 3–10. Indicator Register Testing Program

registers, then the accumulator contents are transferred to the memory buffer registers, and on into the location just read. Finally, the A registers are added to the cleared accumulators, ending the execution. Because a memory cycle is required to read the word out and another one is needed to store the accumulator contents, the *ECH* instruction is a 3-cycle instruction and thus requires 18 μsec execution time. This instruction, which may be indexed, is designated by an octal operation code of 350.

3.14.2 Use of *Exchange* Instruction

As an example of the *ECH* instruction, let us consider two tables that are stored in core memory. We wish to exchange the locations of these tables without disturbing their arrangement. The tables are originally located in 0.13470 through 0.13600 and 0.30000 through 0.30110. The program to exchange the two tables is given in table 3–15.

TABLE 3–15. RELOCATION PROGRAM

LOCATION	OPERATION	ADDRESS
0.00000	*1 XIN*	0.00110
0.00001	*1 CAD*	0.13470
0.00002	*1 ECH*	0.30000
0.00003	*1 FST*	0.13470
0.00004	*1 BPX (01)*	0.00001
0.00005	*HLT*	—

3.15 COMPARE INSTRUCTIONS

3.15.1 General

The compare instructions are a group of eight instructions which perform basically the same operations with various parts of a word. Comparison takes place between the contents of the accumulators and the contents of the memory location specified by the right half-portion of the instruction. If a satisfactory comparison is made, the program counter is stepped once, as in normal practice. However, if a satisfactory comparison is not made, the program counter is stepped an additional time, thereby causing the program to skip the instruction immediately following the compare instruction. Four of the compare instructions involve one additional step after the comparison, in which the difference (if any) between the accumulator contents and the specified operand is obtained. The difference may then be tested to determine which operand was the larger. A description of each of the compare instructions is given below.

3.15.2 Basic Compare Instructions

3.15.2.1 *Compare Left Half-Words* Instruction

The *Compare Left Half-Words (CML)* instruction is used to compare the contents of the left accumulator with the contents of the left half-word contained in the specified memory location. The execution of this instruction is started by complementing the left accumulator and clearing the left A register. The left half-word from memory is then transferred to the left A register. A comparison is made between the contents of the A register and accumulator, and if satisfactory, the program counter is stepped once, and the next sequential instruction is selected for execution. If the contents of the two registers do not compare, the program counter will be stepped twice, causing the instruction immediately following the *CML* instruction to be skipped. Regardless of the result of the comparison, the left accumulator is recomplemented so that its final contents will be identical with the original contents. Execution time of the *CML* instruction is 12 μsec, and it may be indexed. An octal operation code of 044 is used to designate this instruction.

3.15.2.2 *Compare Right Half-Words* Instruction

The *Compare Right Half-Words (CMR)* instruction is used to compare the contents of the right accumulator and the contents of the right half-word specified by the address part of the instruction. The right accumulator is complemented, and the right A register is cleared. Then the right half of the specified memory location is transferred to the right A register where a comparison between that register and the right accumulator is made. If the comparison is successful, the next instruction is selected for execution. If the comparison shows that the two half-words were not equal, the instruction immediately following the *CMR* instruction is skipped. This instruction is terminated when the right accumulator is again complemented to restore its contents to their original value. The *CMR* instruction may be indexed and requires 12 μsec to execute. It is specified by an octal operation code of 042.

3.15.2.3 *Compare Full Words* Instruction

This instruction is used to compare a full word from memory with the contents of the accumulators. The *Compare Full Words (CMF)* instruction is executed in much the same manner as the *CML* and *CMR* instructions. The accumulator contents are complemented and the A registers are cleared. Then the word from memory is transferred to the A registers and a comparison is made. A successful comparison will cause the next instruction in sequence to be executed; a no-compare indication will cause the program to skip the

instruction immediately following the *CMF* instruction. It should be remembered that the entire word must compare in order to cause a successful indication; if one half-word compares and the other does not, the result is still a no-compare indication. The *CMF* instruction is executed in 12 μsec and may be indexed. It is designated by an octal operation code of 046.

3.15.2.4 *Compare Masked Bits* Instruction

The *Compare Masked Bits (CMM)* instruction is used to compare any combination of bits within a full word. These bits are specified by a mask which is loaded into the B registers. If a bit position in the B register mask contains a 1, the corresponding bit position in the associated accumulator will be compared with the same bit position of the memory word; if the mask contains a 0, the bit is not active as far as the compare circuitry is concerned. However, it should be noted that all bits within the B registers, accumulators, and specified memory location are dealt with during the execution of the *CMM* instruction. Those bits which are not active are automatically made to compare. The execution of the *CMM* instruction takes place in the following manner. The accumulators are complemented and the A registers are cleared. The mask from the B registers is then transferred to the A registers. A logical multiplication takes place between the accumulators and A registers, with the result being contained in the accumulators. The A registers are then complemented and the contents of the specified memory location are logically added to the A registers. Comparison between the contents of the A registers now takes place, and if the comparison is successful, the next sequential instruction is executed. If the comparison is not successful, the instruction immediately following the *CMM* instruction is skipped. The accumulator is then complemented, terminating the instruction. In this case, complementing the accumulator will not necessarily restore it to its original contents. Only those bits which were specified in the B register masks as being active are returned to their original values; the remaining bits will all contain 1's regardless of their original contents. The automatic comparison of the inactive bits is forced by the combination of logical addition and multiplication in the A registers. An octal code of 040 is used to designate the *CMM* instruction, which is indexable. Twelve μsec are required for the execution of this instruction, which is shown in table 3–16. This table illustrates the execution on a half-word only; however, the full word is processed in the same manner. We will assume that the B register is loaded with a mask of 0.000 111 111 000 111.

In this particular example, the bits selected compared with each other, so the next instruction executed by the program would be the one immediately following the *CMM* instruction. Notice that although the bits are checked to see if they contain the same value in each active position, the actual comparison is made by taking the complement of the bits to be checked. Thus, if a bit position in the accumulator contains its complement in the A register when the comparison is made, the original value of the two bits is the same. Notice also that the inactive bits in the accumulator are cleared to 0's regardless of their original value, and the corresponding bit positions in the A register are set to 1's, thus forcing an automatic compare of the nonactive bits.

3.15.2.5 Program Examples

Suppose we are stepping two indicator registers by two different programs. We want to go through a definite sequence of programs when the contents of the two registers reach the same value, otherwise we will continue to execute the two programs which are step-

TABLE 3–16. *COMPARE MASKED BITS* INSTRUCTION EXECUTION

ACTION	ACCUMULATOR	A REGISTER	MEMORY LOCATION
Start	0.101 101 001 110 010	0.000 000 000 000 000	0.100 101 001 000 010
Complement	1.010 010 110 001 101	0.000 000 000 000 000	0.100 101 001 000 010
Clear	1.010 010 110 001 101	0.000 000 000 000 000	0.100 101 001 000 010
Transfer	1.010 010 110 001 101	0.000 111 111 000 111	0.100 101 001 000 010
Logical Multiply	0.000 010 110 000 101	0.000 111 111 000 111	0.100 101 001 000 010
Complement	0.000 010 110 000 101	1.111 000 000 111 000	0.100 101 001 000 010
Logical Add	0.000 010 110 000 101	1.111 101 001 111 010	0.100 101 001 000 010
Compare	0.000 010 110 000 101	1.111 101 001 111 010	0.100 101 001 000 010
Complement	1.111 101 001 111 010	1.111 101 001 111 010	0.100 101 001 000 010

TABLE 3–17. REGISTER COMPARISON PROGRAM

LOCATION	OPERATION	ADDRESS
0.00001	*CAD*	0.00075
0.00002	*CMR*	0.00076
0.00003	*BPX*	To new program
0.00004	*BPX*	Return to register-stepping programs
0.00075	Indicator	Register one
0.00076	Indicator	Register two

ping the indicator registers. We can cause our program control to branch to a short subroutine which will test the equality of the two indicator registers at specific intervals. Assuming that the indicator registers are contained in the right half-words of the memory locations, the subroutine which will test the contents of the registers is given in table 3–17.

As another example of the compare instructions, assume that we are reading in various programs from a reel of magnetic tape. This tape contains various maintenance programs, and each program has a program number. We have a control word in memory which tells us that we are searching for the first program number that starts with the digits 56_{10}. The programs are read in one at a time, with the program number always being stored in the same location. Our problem is to compare each program number read in with a control word of 56_8 in the proper bit positions (L1-L6) until we reach the first program starting with these digits. When we have determined that we have the proper program, we wish to execute it. The program to compare these numbers is given in table 3–18.

3.15.3 Compare Difference Instructions

3.15.3.1 *Compare Difference Left Half-Words* Instruction

The *Compare Difference Left Half-Words (CDL)* instruction tests the left half portion of the memory location specified in the instruction to see if it is equal to the contents of the left accumulator. If the contents compare favorably, the next sequential instruction is executed. On the other hand, if the contents of the left half-words are not identical, the instruction immediately following the *CDL* instruction is skipped. After the comparison check is made, the entire word in the accumulator is subtracted from the contents of the specified memory location. Thus, the difference between the full words will be generated, although only the left half portions were compared. If the final contents of the accumulators are in negative form, the original value in the accumulator was equal to or greater than the operand. A positive remainder in the accumulator indicates that the operand was larger than the original contents of the accumulators.

Execution of the *CDL* instruction takes place in the following manner. The accumulator contents are complemented, and the A registers are cleared. Then the word from the specified memory location is transferred to the cleared A registers. A comparison is made between the left A register and left accumulator to see if they contain the complement of each other, and the program counter is stepped accordingly. This instruction is then terminated by pulsing the carry-0 line in the address, thus adding the contents of the full word from the A registers to the accumulators. However, since the accumulators are in complement form, this process is, in effect, subtracting the accumulator contents from the A registers (operand from core memory). The *CDL* instruction requires 12 μsec to execute and may be indexed. It is specified by an octal operation code of 045.

3.15.3.2 *Compare Difference Right Half-Words* Instruction

The purpose of the *Compare Difference Right Half Words (CDR)* instruction is to test the equality of the right accumulator contents and the operand contained in the right half of the memory location specified by the instructions. As with all compare instructions, the instruction immediately following the *CDR* instruction will either be executed or skipped, depending on the outcome of the comparison. Execution of the *CDR* instruction is started by complementing the accumulators

TABLE 3–18. TAPE PROGRAM SEARCH ROUTINE

LOCATION	OPERATION	ADDRESS
0.00410	*CAD*	0.06250
0.00411	*LDB*	0.03341
0.00412	*CMM*	0.03325
0.00413	*BPX*	Execute this program
0.00414	*BPX*	Read in next program
0.06250	0.56000	Control word
0.03341	0.77000	(Mask of 0.111 111 000 000 000)
0.03325	Current program number	

and clearing the A registers. The specified memory location contents are then transferred to the A registers, and the right A register and right accumulator are compared. Following this, both A registers are added to the complemented accumulators, thereby generating the difference between the numbers. If the final contents of the accumulator are still in complement form, we know that their original contents were equal to or larger than the operand; if the final accumulator contents are positive, it means that the operand was larger than the original value in the accumulators. Twelve μsec are required to execute the *CDR* instruction, which is indexable and is designated by an octal operation code of 043.

3.15.3.3 *Compare Difference Full Words* Instruction

This instruction is used to compare both the left and right accumulators with the contents of the memory location specified by the instruction. The execution of the *Compare Difference Full Words (CDF)* instruction is similar to the execution of all the compare instructions. The accumulator contents are complemented, the A registers are cleared, and the contents of the specified operand are transferred to the A registers. Then both half-words of the accumulators are compared with both A registers. If a no-compare is generated from either or both halves, the instruction immediately following the *CDF* instruction will be skipped. Otherwise (both halves compare), the next sequential instruction will be executed. Finally, the A registers are added to the complemented accumulators, thereby generating the difference between each half-word of the accumulator and the corresponding half-word from memory. The final accumulator contents will be in negative form if they were originally larger; they will be in positive form if the operand was larger. The *CDF* instruction is specified by an octal operation code of 047. It may be indexed and requires 12 μsec to execute.

3.15.3.4 *Compare Difference Masked Bits* Instruction

The *Compare Difference Masked Bits (CDM)* instruction tests the equality of specified bits in the accumulators with the corresponding bits of the operand contained in the designated memory location. The bits that are to be tested are determined by the mask which is contained in the B registers. For each bit position that is to be compared, the corresponding bit position in the B register mask must contain a 1. Bits not involved in the comparison are designated by 0's in the mask. After the designated bits are compared, and the program counter has been stepped in accordance with the results, the difference between the masked bits of the accumulators and the operand from memory is obtained.

Execution of the *CDM* instruction takes place in the following manner. The accumulator contents are complemented, and the A register is cleared. Then the mask from the B register is transferred to the cleared B registers. A logical multiplication takes place between the A register and the accumulators, leaving the results in the accumulators. The A registers are then complemented and the word from memory is logically added to them. This combination of logical multiplication and addition will cause an automatic compare of the unmasked bits, just as with the *CMM* instruction. After the comparison takes place, the A registers are added to the complemented accumulators, thus generating the difference between the original contents of the specified bits of the accumulators and the corresponding bits of the operand. The final contents of the accumulators will be in negative form if the original bit contents of the accumulators were equal to or larger than the bit contents of the operand. If the operand bits were larger, the result in the accumulators will be positive. The final results of the accumulators for this instruction indicate the relative magnitude of the bits compared only if the sign bit is not included in the mask (inactive). When the sign bit is active, the final contents of the accumulators are variable, depending on such things as end-carry, overflow, etc., that may occur when the A register is added to the complemented accumulator. The *CDM* instruction requires 12 μsec to execute and may be indexed. An octal operation code of 041 is used to indicate a *CDM* instruction. An example of the execution of this instruction is given in table 3–19. Only a half-word is shown, but the entire word is compared in the same manner. In this example, the B register has been loaded with a mask of 0.110 110 110 000 001.

In this case, the bits did not compare, but the final accumulator contents are positive because the value contained in the memory location was larger than the original contents of the accumulator. We know this statement is true because the sign bit was not involved in the comparison.

3.15.3.5 Program Examples

We are going to test the contents of two specified memory locations. We want to branch to one of five subroutines, depending on the results of our test. The five possible outcomes from a comparison of two full words are comparison, no-compare with both accumulators negative, no-compare with the left accumulator negative, no-compare with the right accumulator negative, and no-compare with both accumulators positive. A program which will provide for branching to the correct subroutine, depending on the result, is given in table 3–20.

TABLE 3–19. *COMPARE DIFFERENCE MASKED BITS* INSTRUCTION EXECUTION

ACTION	ACCUMULATOR	A REGISTER	MEMORY LOCATION
Start	1.010 111 100 011 111	0.000 000 000 000 000	0.100 001 101 010 000
Complement	0.101 000 011 100 000	0.000 000 000 000 000	0.100 001 101 010 000
Clear	0.101 000 011 100 000	0.000 000 000 000 000	0.100 001 101 010 000
Transfer	0.101 000 011 100 000	0.110 110 110 000 001	0.100 001 101 010 000
Logical Multiply	0.100 000 010 000 000	0.110 110 110 000 001	0.100 001 101 010 000
Complement	0.100 000 010 000 000	1.001 001 001 111 110	0.100 001 101 010 000
Logical Add	0.100 000 010 000 000	1.101 001 101 111 110	0.100 001 101 010 000
Compare	0.100 000 010 000 000	1.101 001 101 111 110	0.100 001 101 010 000
Obtain Difference	0.001 001 111 111 111	1.101 001 101 111 110	0.100 001 101 010 000

In this example, the registers compare; therefore, the program counter will be stepped normally, and we will execute the branch immediately following the *CDF* instruction to subroutine A. If the numbers did not compare, however, the *BPX* instruction at location 0.62502 would be skipped. We would then branch to subroutine B if both accumulators were negative, to subroutine C if the left accumulator was negative, to subroutine D if the right accumulator was negative, and, finally, to subroutine E if both accumulators were positive. We would probably store the difference between the two registers compared as the first step in subroutines B, C, D, and E. This would not be necessary in the case of subroutine A because we know the register contents are equal.

As an example of the use of the *CDM* instruction, let us check the equality of bit positions R10-R15 of two registers. We wish to execute a certain subroutine if the contents are equal and another one if the contents are unequal and the number placed in the accumulator is higher in magnitude than the number being compared with it. This program is listed in table 3–21.

TABLE 3–20. REGISTER COMPARE AND BRANCH ROUTINE

LOCATION	OPERATION	ADDRESS
0.62500	*CAD*	0.07300
0.62501	*CDF*	0.07301
0.62502	*BPX*	To subroutine A
0.62503	*BFM*	To subroutine B
0.62504	*BLM*	To subroutine C
0.62505	*BRM*	To subroutine D
0.62506	*BPX*	To subroutine E
0.07300	0.74044	0.65032
0.07301	0.74044	0.65032

TABLE 3–21. PARTIAL WORD COMPARE PROGRAM

LOCATION	OPERATION	ADDRESS
0.00001	*CAD*	0.03314
0.00002	*LDB*	0.06000
0.00003	*CDM*	0.03315
0.00004	*BPX*	To subroutine A
0.00005	*BRM*	To subroutine B
0.03314	0.00650	0.02540
0.03315	0.43120	0.01630
0.06000	0.00000	0.00077

The numbers selected will yield a no-compare, and the program control will skip the *BPX* instruction at 0.00004. Since the number placed in the right accumulator was larger than the number compared with it, and the sign bit was not part of the mask, the final contents of the right accumulator will be negative, and the *BRM* instruction will be executed. Notice that no provision has been made to check for a no-compare positive accumulator condition, since this was not one of the stipulations of the problem.

3.16 TEST BITS INSTRUCTION

3.16.1 *Test One Bit* Instruction

The *Test One Bit (TOB)* instruction is used to test a particular bit within the operand specified by the

right half portion of the instruction. The test consists of adding the bit content to the least significant bit position of the program counter. Thus, if the bit being tested contains a 1, it will be added to the program counter, causing it to skip the instruction immediately following the *TOB* instruction. If the bit being tested contains a 0, the program counter will not be stepped an additional time, and the next sequential instruction following the *TOB* instruction will be executed. The bit to be tested is determined by decoding the contents of auxiliary bits L11 through L15. A maximum of 37_8 or 31_{10} can be specified by these bits, which is enough to select any one of the 32 bits within an operand (00 also indicates a bit selection). The *TOB* instruction is designated by an octal operation code of 050. The last octal 0 simply indicates that bit L10 of the instruction word contains a 0 but has no significance as far as selection of the bit to be tested is concerned. Execution of the *TOB* instruction requires 12 μsec, and it may be indexed.

3.16.2 *Test Two Bits* Instruction

The *Test Two Bits (TTB)* instruction is used to test two particular bits of the operand specified by the right half portion of the instruction. The test consists of adding the bit contents to the least significant bits of the program counter. With two bits, four combinations are possible and can cause the program counter to be stepped in any of the following ways with the following results:

a. Bits contain 00. Program counter will be stepped in the normal manner, causing the instruction immediately following the *TTB* instruction to be executed.

b. Bits contain 01. Program counter will be stepped one additional time, causing the instruction immediately following the TTB instruction to be skipped.

c. Bits contain 10. Program counter will be stepped two additional times, causing the two instructions immediately following the *TTB* instruction to be skipped.

d. Bits contain 11. Program counter will be stepped three additional times, causing the three instructions immediately following the *TTB* instruction to be skipped.

The bits to be tested are determined by decoding the contents of auxiliary bits L11 through L15 of the instruction word. As with the *TOB* instruction, these bits may specify only one of the 32-bit positions within the operand; however, in the case of the *TTB* instruction, the second bit to be tested is automatically determined to be the bit adjacent to and to the left of the selected bit. For example, if bit R12 was designated by the auxiliary bits, the bit combination to be tested is R11 and R12. This does not apply to the selection of either the left or right sign bits; therefore, it is not possible to specify a *TTB* instruction using the sign bit position as the bit to be tested. However, the sign bits may be tested by a *TOB* instruction. An octal code of 054 is used to designate a *TTB* instruction. The octal four indicates that bit L10 of the instruction word is a 1 but has no bearing on the bit combination selected for testing. Thus, the *TTB* instruction is simply a variation of the *TOB* instruction; bit L10 of the instruction word determines whether the test will be performed on one bit (L10 = 0) or the selected bit and the one adjacent to and to the left of it (bit L10 = 1). The *TTB* instruction requires 12 μsec to execute and may be indexed.

3.16.3 Bit Selection for *TOB* and *TTB* Instructions

As stated previously, the contents of bits L11 through L15 determine what bit of the 32-bit position is selected, and bit L10 determines whether that bit only is tested or whether it and its adjacent bit are tested. Table 3–22 lists the various octal codes that can be represented in positions L11 through L15 and the bit(s) they will select.

It should be noted that the contents of bit L10 must be added to the octal notations listed above when giving the octal word layout of either the *TOB* or *TTB* instruction. For example, if we wish to use a *TTB* instruction (code 054) on bit R5 (code 25), the resulting code is 0565. Adding an octal 0 in front of these four digits for the index indicator bits gives us a word layout of 00565. Similarly, if we wish to test L12 alone, we specify *TTB* (code 050) and bit L12 (code 14) to get 00514.

3.16.4 Program Examples

Previously, we illustrated several programs which required that a test be made on one specific bit to determine a course of action. There are several ways we can get a bit into position for testing. One way is to cycle it into a sign bit position and then use a branch instruction. Another way is to use the *ETR* instruction and set all the inactive bits in the word to be tested to 1's except the particular bit we are interested in, and then execute a *BFZ* instruction. Now, with the addition of the *TOB* instruction, we can test any bit we desire without any shifting or masking. For instance, assume that we wish to check a certain program which had been read into memory to make sure that 17-bit operation had been specified for all the instructions within the program. We can do this very simply because we know that the presence of a 1 bit in L12 of an instruction word which does not ordinarily make use of the auxiliary bits indicates that 17-bit operation is desired. (We will assume that the program we are to test contains

TABLE 3–22. *TOB* AND *TTB* BIT SELECTION

L11-L15	BIT(S) SELECTED	
	TOB	*TTB*
00	LS	LS*
01	L1	LS, L1
02	L2	L1, L2
03	L3	L2, L3
04	L4	L3, L4
05	L5	L4, L5
06	L6	L5, L6
07	L7	L6, L7
10	L8	L7, L8
11	L9	L8, L9
12	L10	L9, L10
13	L11	L10, L11
14	L12	L11, L12
15	L13	L12, L13
16	L14	L13, L14
17	L15	L14 ,L15
20	RS	RS*
21	R1	RS, R1
22	R2	R1, R2
23	R3	R2, R3
24	R4	R3, R4
25	R5	R4, R5
26	R6	R5, R6
27	R7	R6, R7
30	R8	R7, R8
31	R9	R8, R9
32	R10	R9, R10
33	R11	R10, R11
34	R12	R11, R12
35	R13	R12, R13
36	R14	R13, R14
37	R15	R14, R15

**The sign bits may not be selected in conjunction with the TTB instruction. If specified, they will be executed as a TOB instruction.*

only instructions with 0's in all the auxiliary bit positions except where 17-bit operation is desired.) The program which will perform this check is given in table 3–23.

As long as the instruction being tested contains a 1 bit in the L12 position, the program will continue to cycle. However, when a 0 bit is encountered, the program counter will not be stepped an additional time but will select the *BPX* instruction at 0.00503. This instruction will branch to a routine which adds the index register contents at the time to the starting address of the program, thus obtaining the address of the instruction which did not contain a 1 bit in the L12 position. This address is then stored in location 0.03000, and the program halts. We can now manually correct the contents of the instruction stored in location 0.03000 and return to the program.

As an example of the *TTB* instruction application, let us assume that after we have performed the program discussed above, we again want to run through it and store the address of every instruction which specifies the right accumulator (3) as its index register. Since the index indicator contains three bits (L1 through L3), we cannot use a *TOB* to determine what we need to know. As a matter of fact, we could not use a *TTB* instruction if we were searching for index registers 4 or 5, since they need all three bits to identify them. But index register 3 requires only two bits (L2 and L3); therefore, we can utilize this instruction. The program to perform this check is given in table 3–24.

This program will examine the contents of bits L2 and L3 with the *TTB* instruction at location 0.00512. If any other bit combination (00, 01, or 10) is found, the program immediately branches to a step which reduces the index register by one and selects a new instruction

TABLE 3–23. INSTRUCTION WORD CHECKING PROGRAM

LOCATION	OPERATION	ADDRESS
0.00501	*1 XIN*	0.00123
0.00502	*1 TOB 14*	0.01340
0.00503	*BPX*	0.00506
0.00504	*1 BPX (01)*	0.00502
0.00505	*HLT*	–
0.00506	*1 ADX*	0.01340
0.00506	*STA*	0.03000
0.00507	*HLT*	–
0.01340 - 01463	Location of program	

in the program for testing. If a bit combination of 11 is found, the address of the instruction it is found in is stored, starting at location 0.00623, and the next instruction is selected for testing. Notice that we used two different index registers in this program, one to control the memory locations selected for testing and the other to control the storage of applicable addresses. We could have used the same index register to store the addresses; however, the chances are that the index register controlling the address selection will be stepped several times for each time that we store an address (which contains an instruction using index register 3). Therefore, our table of addresses would not be in sequential order. To achieve sequential order, a separate index register was employed.

You will notice that the last two programs discussed have dealt with operations on instructions themselves. This is perfectly valid, since an instruction is treated just like an operand when it is brought out of memory during an OT cycle. The fact that instructions in a specific program were being tested for a particular condition is no different than testing a block of true operands for some condition. Both instructions and operands are in numerical form, and on an OT cycle where no decoding of the word brought from memory is performed, the instructions are, in effect, operands.

3.17 *CLEAR AND ADD CLOCK* INSTRUCTION

3.17.1 Execution

The *Clear and Add Clock (CAC)* instruction is used to transfer the contents of the clock register to the right accumulator. This instruction is similar in execution to a *CAD* instruction except that no operand is required from memory. Instead, the accumulators are cleared, and the clock register contents are transferred to the right memory buffer register. From this point, the action that occurs is identical with that of a *CAD* instruction. Execution of the *CAC* instruction requires 12 μsec, and it is designated by an octal operation code of 170. Indexing is not applicable to this instruction, since only one clock register may be selected.

Although the *CAC* instruction does not refer to a memory location to obtain its operand, the address portion of the instruction is still used. Before this instruction was available for the AN/FSQ-7 or AN/FSQ-8, the clock register could be obtained by using any instruction (usually a *CAD)* and an address of 0.60000. Now that a definite instruction is available, the address of 0.60000 is no longer necessary. However, an address of 3.77777 is used as the right half portion of the *CAC* instruction. It should be noted that this address has no significance as concerns the selection of the clock, but is used to inhibit parity checks and logical addition in the memory buffer registers. Address 3.77777 is the highest address in test memory (for the AN/FSQ-7) and will prevent either the large or small memory unit from being accidentally selected. In the AN/FSQ-8, there is no large memory unit installed; therefore, the address part of the *CAC* instruction is meaningless. In addition, control is still available to select the clock register in the AN/FSQ-8 by using a *CAD 0.60000* instruction. However, to select the clock register in the AN/FSQ-7, a *CAC 3.77777* instruction must be given.

TABLE 3–24. SAMPLE *TTB* PROGRAM

LOCATION	OPERATION	ADDRESS
0.00510	*1 XIN*	0.00123
0.00511	*2 XIN*	0.00123
0.00512	*1 TTB (03)*	0.01340
0.00513	*BPX*	0.00521
0.00514	*BPX*	0.00521
0.00515	*BPX*	0.00521
0.00516	*1 ADX*	0.01340
0.00517	*2 STA*	0.00500
0.00520	*2 BPX (01)*	0.00521
0.00521	*1 BPX (01)*	0.00512
0.00522	*HLT*	—

3.17.2 Interpretation of Clock Register Contents

Before a program example of the *CAC* instruction is given, a general discussion regarding the reading of the clock register contents will be helpful. As previously mentioned, the clock register consists of 16 flip-flops connected into an endless counter; i.e., when the clock register contains the maximum value, the next stepping pulse will clear the clock register and restart the count. Pulses are supplied to the clock register at a rate of 32 pulses per second (pps), and the maximum time that can be represented by 16 bits is $65{,}536_{10}$ pulses (approximately 34.13 minutes). Since the stepping rate is 32 pps, 32 pulses supplied to the clock register equal 1 second of real time. The number 32 has an octal equivalent of 40_8; therefore, if the clock register contains 0.00040, it actually represents 1 second. It is necessary to convert the number represented by the clock register to its decimal equivalent and then divide by 32 to determine the actual time in seconds. (This is for use only when a time indication in actual minutes and seconds is desired. In the Central Computer System, the clock register is used directly in figuring time computations.)

TABLE 3–25. TIME DETERMINATION ROUTINE

LOCATION	OPERATION	ADDRESS
0.00001	*CAC*	3,77777
0.00002	*RST*	0.00150
0.00003	*BPX*	To program
Program	(*BPX* to 0.00004 at end)	
0.00004	*CAC*	3,77777
0.00005	*RST*	0.00151
0.00006	*SUB*	0.00150
0.00007	*BRM*	0.00012
0.00010	*RST*	0.00155
0.00011	*HLT*	—
0.00012	*CAD*	0.00153
0.00013	*SUB*	0.00150
0.00014	*ADD*	0.00151
0.00015	*RST*	0.00155
0.00016	*HLT*	—
0.00150	0.00000	Start time
0.00151	0.00000	End time
0.00153	0.00000	1.77776
0.00155	0.00000	Running time

For example, if the clock register contained a value of 0.03600 and we desired to find out what real time this represented, we would convert this number to decimal, which is 1920. Dividing 1920 by 32 gives us 60, the number of seconds represented. Thus, a clock register reading of 0.03600 represents 1 minute of time. Also, it should be remembered that the sign bit position of the clock register does not indicate polarity but rather is the most significant bit position of the register.

3.17.3 Program Example

In many instances, it is desirable to know the time required for a certain program or routine to be executed. We can do this by obtaining the contents of the clock register prior to execution of the program and comparing it with the clock register contents after the program is completed. The program to determine the running time is shown in table 3–25.

The routine is executed in the following manner. The starting time of the program is obtained and stored in location 0.00150. Then the program is executed and, at its conclusion, branches back to the routine which will determine the running time. The current clock reading, representing the end of the program, is obtained and stored in location 0.00151. The start time of the program is then subtracted from the end time, and the result is stored in location 0.00155 if the results are positive. However, it is possible that our program may have started as the clock register was nearly stepped to its maximum count and may have ended after the clock register was reset. If this is the case, we will get a negative indication in the right accumulator. The *BRM* instruction will take us to location 0.00012 where we *CAD* the maximum value of the clock register (1.77777) and subtract the start time from it. The difference will be the amount of time the program ran before the clock was reset. Next, we add the end time of the program to this number and have, as a result, the actual time involved in running the program. Let us take two sample values for the clock register (assuming that the clock was reset during execution of the program being timed). The clock reading at the start of the program was 0.67754, and at the completion of the program, it was 0.00071. Subtracting the first from the second will give us a negative result, so we must subtract the start time from the maximum content of the clock. This subtraction (1.77777 minus 0.67754) will leave us with 1.10023. Adding the end time to this will result in a total elapsed time of 1.10114. Converting this to decimal gives us a running time of approximately 1154 seconds, or 19.23 minutes.

3.18 *OPERATE* INSTRUCTION

The *Operate (PER)* instruction is used to initiate a wide variety of actions within the AN/FSQ-7 and AN/FSQ-8, such as starting a test pattern generator or rewinding a magnetic tape reel. The *PER* instruction variation which is to be performed is designated by bits L10-L15 of the instruction word (auxiliary bits). The execution of the *PER* instruction depends chiefly on a decoding matrix, which allows for the transmission of a pulse to a unit specified by the auxiliary bits. As previously mentioned, this pulse can cause several actions to take place, depending on what unit is receiving the pulse. Execution time of the *PER* instruction requires 12 μsec. Because the address portion of this instruction is meaningless, it may not be indexed. An octal operation code of 01 is used to specify a *PER* instruction. These bits are placed in L4-L9 of the instruction word, leaving all the auxiliary bits free to specify what action is to take place.

Many of the variations of the *PER* instruction affect units outside the Central Computer System. Only those variations which have an effect on the Central Computer System itself are discussed below. The *PER* variations that affect other portions of the AN/FSQ-7 or AN/FSQ-8 are discussed in Part 4. All the *PER*

variations are decoded in the manner described above; only the action of the variations themselves is discussed.

3.18.1 Condition Lights (1-4)

Four condition lights are provided on the duplex maintenance console which are used to indicate the course of a specific program. These four condition lights are turned on by the execution of a *PER (01 – (04)*, respectively. For example, execution of a *PER (03)* instruction will cause condition light 3 to be turned on, giving a visual indication that a certain action has resulted in the Central Computer System in accordance with the program being run.

As an example of the use of condition lights, assume that we are comparing the magnitude of two numbers. We wish to determine which is the larger and store it in a specified location. However, if the two numbers are equal, we wish to turn on condition light 2 and cause the program to halt. The routine to check these numbers is given in table 3–26.

If a negative number is in the accumulator as a result of the *DIM* instruction, we check to see if it is —0. If it is, we know the magnitudes of the two numbers are equal, so we branch to 0.00011 and turn on condition light 2. If the numbers are not equal, we obtain the larger, store it in location 0.7300, and branch to the next routine.

TABLE 3–26. EQUALITY CHECK ROUTINE

LOCATION	OPERATION	ADDRESS
0.00000	*CAM*	0.01350
0.00001	*DIM*	0.01351
0.00002	*BRM*	0.00006
0.00003	*CAD*	0.01350
0.00004	*FST*	0.07300
0.00005	*BPX*	To next routine
0.00006	*BFZ*	0.00011
0.00007	*CAD*	0.01351
0.00010	*BPX*	0.00004
0.00011	*PER (02)*	—
0.00012	*HLT*	—
0.01350	0.00000	A
0.01351	0.00000	B
0.07300	Storage	—

3.18.2 Set Inactivity

There are two types of machine inactivity which we can check for in the active AN/FSQ-7 or AN/FSQ-8. The first type involves checking to see if the time pulse distributor action has been interrupted for a period of at least 1/32 of a second. This check is initiated by the execution of a *PER (05)* instruction. Once this instruction has been given, it sets up circuitry which is checked every clock pulse to see if the time pulse distributor is active. If the time pulse distributor stops for some reason, an alarm is generated.

The second type of inactivity we can check for is to see if the Central Computer System is hung up in some type of idle loop. This check is also initiated by the execution of a *PER (05)* instruction. If the Central Computer System is in an idle loop, an alarm will be generated after the second consecutive 8-second clock pulse unless the inactivity alarm circuitry is reset by a *PER (05)* instruction before the second 8-second clock pulse is generated, or the inactivity test is terminated. Thus, the *PER (05)* instruction actually initiates two kinds of inactivity check. If we are testing for time pulse distributor inactivity, an alarm will occur as soon as the time pulse distributor has been stopped for 1/32 of a second. If we are checking for an idle loop, the alarm will occur approximately 8 to 16 seconds after the execution of the *PER (05)* instruction, if an idle loop condition exists. Normal programmed use of the *PER (05)* instruction is such that it intervenes between consecutive 8-second clock pulses, thus preventing the generation of an idle loop inactivity alarm, if normal operating conditions exist. This is accomplished by programming an iterative loop which executes a *PER (05)* instruction every 8 seconds. If an idle loop occurs, the program will not be able to branch back to the *PER (05)* instruction, thus allowing two consecutive 8-second clock pulses to step the inactivity controls and generate an alarm. Occurrence of both these alarms depends on whether or not the inactivity check is terminated prior to the time the alarms would take place.

3.18.3 Reset Inactivity

The control needed to terminate a check for inactivity is provided by the *PER (06)* instruction which resets the inactivity alarm circuitry. The *PER (06)* instruction must be used in conjunction with the *PER (05)* instruction.

3.18.4 Intercommunication Flip-Flops (1-4)

There are two Central Computer Systems within each AN/FSQ-7 and AN/FSQ-8, as previously explained, and while one is performing the active air defense program, the other is in the standby status. In the normal course of operations, it is sometimes necessary for one system to communicate with the other one. This may be accomplished by the execution of one of the *PER (10)*

– (13) instructions. The execution of one of these instructions in one Central Computer System will set a corresponding flip-flop (FF) in the other Central Computer System to indicate that a specific condition exists. For example, if we give a *PER (11)* instruction in the active computer, it will set intercommunication FF 2 in the standby computer. The standby computer may then sense its flip-flops to see if any of them are set. In this case, FF 2 would be set, and the standby computer would be aware that a certain condition exists in the active computer.

While the intercommunication flip-flops are most generally employed in a program to indicate that a drum transfer from the active Central Computer System to the standby Central Computer System is ready to take place, the flip-flops can be used at the discretion of the programmer to indicate whatever he desires. To illustrate the use of the intercommunication flip-flops, assume that the active Central Computer wishes to notify the standby Central Computer that it is going to make a switchover. It may do this by setting a flip-flop in the standby computer; FF 3, for example. This would require the active computer to execute a *PER (12)* instruction. When the standby computer senses its flip-flops and finds FF 3 set, it will branch to a routine, preparing itself to take over the air defense function. (It should be remembered that a notification of switchover between computers may not actually be done in this manner; switchover was simply picked as a possible condition that could be indicated from one computer to the other.)

3.18.5 Test Clock Register

As previously explained, the clock register is normally stepped at a 32-pps rate. When we wish to test the stepping action of the clock register, it is desirable to do it at a much faster rate than 32-pps. The execution of a *PER (14)* instruction enables us to test the clock register at a pulse rate of approximately 167 kilocycles, or one pulse every 6 μsec. The action of the *PER (14)* instruction is to disconnect the clock register from its 32-pps input, clear it to +0, and set up a branch control line to its input, so that every time a *BPX* instruction is executed the clock register will be stepped once. When a second *PER (14)* instruction is executed, the clock will be reconnected to its 32-pps input but will not be cleared. A sample program to check the clock is given in table 3–27.

We first obtain the current clock register content and store it in the right half portion of location 0.02153. Then we set up the clock register for testing by execution of the first *PER (14)* instruction. An index register is loaded with 1000_8, and we branch in a loop until the program falls through the *1 BPX (01)* instruction, at which time the clock should have been stepped 1000_8 times. We then obtain the contents of the clock and subtract a value of 1000_8 from it. If the clock register has been stepped correctly, the right accumulator will contain –0 and the *BFZ* condition will be satisfied. If the right accumulator does not contain –0, the clock was stepped incorrectly, and we execute an error halt at location 0.02150. Assuming that the clock was stepped correctly, it is then necessary to restore it to its original contents and proceed. If we branched from location 0.02147 (indicating correct stepping), we then execute a second *PER (14)* instruction to reconnect the clock. A third *PER (14)* instruction immediately follows to disconnect the clock and clear it. This is necessary because the second *PER (14)* instruction would not clear the clock. An index register is set to the value originally contained in the clock, and we step it up to this value again by execution of a *1 BPX (01)* instruction at 0.02154. A fourth *PER (14)* instruction will reconnect the clock again to its usual input. It should be remembered that this program may be executed in much less time than 1/32 of a second, so it is possible to disconnect the clock, test it, and reconnect it between the pulses it would ordinarily receive, without destroying the continuous stepping action of the clock at a 32-pps rate.

TABLE 3–27. CLOCK REGISTER STEPPING ROUTINE

LOCATION	OPERATION	ADDRESS
0.02140	*CAC*	3.77777
0.02141	*RST*	0.02152
0.02142	*PER (14)*	–
0.02143	*1 XIN*	0.01000
0.02144	*1 BPX (01)*	0.02144
0.02145	*CAC*	3.77777
0.02146	*SUB*	0.01000
0.02147	*BFZ*	0.02151
0.02150	*HLT*	–
0.02151	*PER (14)*	–
0.02152	*PER (14)*	–
0.02153	*1 XIN*	Determined by original clock setting
0.02154	*1 BPX (01)*	0.02153
0.02155	*PER (14)*	–
0.01000	0.00000	0.01000

3.18.6 Inhibit Alarms

There are several alarms which may be generated in the AN/FSQ-7 and AN/FSQ-8 that can cause automatic branching if it is desired. When this is the case, the occurrence of a specific alarm will cause a branch to a designated test memory location. On the other hand, it is possible to generate these alarms and take no action at all, or to generate an alarm and halt. These options are provided for by switches on the duplex maintenance console. For each condition that may generate an alarm (such as inactivity), there is a switch which determines whether the alarm is active or not. If the switch is in the inactive position, the occurrence of an alarm condition will be ignored and the computer program will proceed. If the switch is in the active condition, the alarm condition is acknowledged, and one of two actions take place, depending on the setting of another switch on the duplex maintenance console. This is the STOP/BRANCH switch, and when set to STOP, it will cause the program to halt upon the occurrence of an alarm. If set to the BRANCH position, the program will cause an automatic branch to a designated test memory location upon receipt of the alarm.

When we wish to inhibit the automatic branching or halting of a computer program upon receipt of an alarm, we can execute a *PER (15)* instruction. This will disable most of the alarm circuitry, and the program will proceed sequentially. Thus, execution of a *PER (15)* instruction will override any switch settings on the duplex maintenance console, allowing the program to determine when an alarm may be generated.

3.18.7 Reset Alarms

As explained above, execution of a *PER (15)* instruction will disable a major portion of the alarm circuitry. When it is desired to reset this circuitry so that alarm actions may take place, we execute a *PER (16)* instruction. This causes the computer program to once again follow the settings of the switches on the duplex maintenance console.

3.18.8 Generate Alarm 1 and 2

There are five types of alarms which will cause automatic branching if so desired. These five types of alarms are grouped into two classes of alarms, known as alarm 1 and alarm 2. Alarm 1 consists of memory parity, addressable drum parity, and inactivity. Alarm 2 is made up of status drum parity and tape parity. When one of these alarms is generated in the active computer, it sets a flip-flop in the standby computer which may be sensed by a *BSN* instruction. Further, if an alarm condition occurs in the active computer, it will cause an automatic alarm branch to take place in the standby computer, if its switches are set to do so, and the standby computer will prepare itself to become the active computer. However, there are many instances in which the active computer would like to notify the standby computer to take over the air defense functions that are not included in the list of actions that will cause the generation of an alarm 1 condition. For example, if a selected IO device fails while we are involved in an operation with it, it may be desirable to make a switch-over and repair this device immediately. Since the failure of an IO device does not cause an alarm 1 generation to the standby computer, it is necessary for the program to generate this alarm. This may be accomplished by the execution of a *PER (37)* instruction which will generate an alarm 1 condition that is transferred to the standby computer. It will also set the alarm 1 and alarm 2 flip-flops in the active computer.

It is anticipated that in the near future the automatic generation of the alarm 1 condition will be discontinued. Then, when an alarm occurs, a branch to a subroutine will take place which will determine what caused the alarm condition in the active computer and try to correct the malfunction. If the error cannot be corrected, a *PER (37)* instruction will be executed, telling the standby computer that an error has occurred in the active computer which cannot be fixed and instructing the standby computer to prepare to become active.

3.19 *BRANCH ON SENSE* INSTRUCTION

The *Branch on Sense (BSN)* instruction provides for a branch of program control to the address specified in the right half portion of the instruction if a designated condition exists. This condition is determined by decoding the auxiliary bits (L10-L15) of the instruction word. We can sense for such things as overflow in either or both accumulators, checking a tape drive unit to see if it is ready to be used, and various other conditions. The execution of the *BSN* instruction takes place in the following manner. The auxiliary bits are decoded in the same matrix used by the *PER* instruction. These bits will partially condition one unit selected by the bits. If the unit being sensed is active, it completes the conditioning necessary for a positive indication. The *BSN* instruction then strobes all the sense units in parallel and causes a branch of program control if a unit is on. Only one unit may be on at the time the check for branch takes place. If a unit is found to be active, the contents of the program counter are transferred to the right A register and the contents of the address register are transferred to the program counter. (This is the same action which results when a condition is satisfied for any of the other branch instructions such as *BRM* and *BPX*.) If the unit being sensed was not active, the branch will not be executed, and the next sequential instruction will be selected. Execution of the *BSN* instruction requires 12 μsec, and it is not indexable. The *BSN* instruction is designated by an octal operation

code of 52 in bit positions L4-L9. Bits L10-L15 are thus made available to specify what unit is to be sensed.

As with the *PER* instruction, many of the codes specified by the auxiliary bits of the *BSN* instruction do not apply to the Central Computer System. Only the applicable codes are discussed below. The remainder will be discussed in the part of the manual relating to programming of the equipment involved. The execution of the *BSN* instruction is the same for all codes, however, so only the action of the *BSN* variations is covered.

3.19.1 Condition Lights ON (1-4)

The four condition lights on the duplex maintenance console are turned on by the execution of the applicable *PER (01) – (04)* instruction. When it is desired to check the condition lights by computer control to see if they are on, we execute one of the *BSN (01) – (04)* instructions. For example, if condition light 4 is on, execution of the *BSN (04)* instruction will cause the program to branch to the address specified in the right half portion of the *BSN (04)* instruction, since the branching condition will be met. Execution of the *BSN (04)* instruction will also turn condition light 4 off.

The condition lights are provided to furnish a visual indication to the programmer if a certain status exists; however, changes in this status often take place too rapidly for the programmer to observe. Therefore, it is desirable to be able to check the condition lights by program control and also to turn off a condition light, signifying that a certain status no longer exists.

3.19.2 Inactivity ON

When an inactivity alarm occurs after the execution of a *PER (05)* instruction, it sets an inactivity alarm flip-flop. We have the option of inhibiting an inactivity alarm, just as with the other alarms that have been discussed. However, if an inactivity alarm does occur and we wish to determine by means of program control what action to take, we can execute a *BSN (05)* instruction. This instruction will examine the inactivity alarm flip-flop and branch to the memory location specified in the right half portion of the instruction. Execution of a *BSN (05)* instruction will also clear the inactivity alarm flip-flop.

3.19.3 Left Overflow ON

When an overflow occurs in the left accumulator, it sets the left overflow flip-flop. The setting of this flip-flop is controlled, however, by the contents of bit L13 in instruction words which may cause a left overflow. If bit L13 is a 1 and a left overflow occurs, the left overflow flip-flop and the overflow alarm indicator flip-flop will be set. If bit L13 contains a 0, the left overflow circuitry is disabled, and the computer will have no indication that a left overflow has occurred.

We can check the status of the left overflow flip-flop by the execution of a *BSN (12)* instruction. If the flip-flop is set and bit L13 of the instruction being executed contains a 1, the branching conditions for the *BSN (12)* instruction will be met, and a branch to the memory location specified in the right half-word of the *BSN (12)* instruction will take place. The program can exercise control over an overflow alarm by branching to the desired location when an overflow occurs. This is true because L14 of the instruction word determines whether the program will recognize a left overflow indication or not. If bit L14 contains a 1 (as well as bit 13), the alarm circuitry will check to see if the overflow switch on the duplex maintenance console is active. If it is, an overflow alarm will be generated. However, if bit 14 contains a 0, or if the overflow switch was inactive, only the left overflow flip-flop would be set. In this case, a *BSN (12)* instruction is the only way we have of detecting an overflow in the left accumulator. A *BSN* (12) instruction also clears the overflow flip-flop if it is set.

3.19.4 Right Overflow ON

It is possible to determine whether an overflow of the right accumulator has occurred by the execution of a *BSN (13)* instruction. The action of this instruction is identical with that of the *BSN (12)* instruction. Bit L13 of the current instruction being executed must contain a 1 or the right overflow flip-flop will not be set, thus disabling all indications of a right overflow. Further, bit L15 of the instruction word must also be a 1 to cause an automatic branch upon receipt of a right overflow indication. If, for some reason, an automatic branch cannot take place (overflow switch inactive or bit L15 contains a 0), only the right overflow flip-flop will be set, allowing the computer to exercise program control of branching upon receipt of a right overflow. Execution of a *BSN (13)* instruction will cause the program to branch control if the right overflow flip-flop is on and will also clear the flip-flop.

3.19.5 Memory Parity Error

A check for memory parity error may be made by the execution of a *BSN (15)* instruction. If a memory parity has occurred, the program will branch control to the address specified by the right half portion of the *BSN (15)* instruction and will clear the memory parity error flip-flop.

3.19.6 Addressable Drum Parity Error

We can check to see if a parity error has occurred while reading an addressable drum field by the execution of a *BSN (16)* instruction. If an addressable drum parity error has occurred, the addressable drum parity error flip-flop will be set. The *BSN (16)* instruction will sense this flip-flop to see if it is set and branch control to

the address specified in the right half-word of the *BSN (16)* instruction if the flip-flop is set. Execution of this instruction will also clear the flip-flop.

3.19.7 Tape Parity Error

A parity error encountered while we are involved in an IO operation with one of the tape drives will set the tape parity error flip-flop. This flip-flop can be sensed by the execution of a *BSN (17)* instruction. If the flip-flop is set, the program will transfer control to the address specified in the right half portion of the instruction and clear the flip-flop.

3.19.8 Sense Switch ACTIVE (1-4)

The four sense switches located on the duplex maintenance console may be sensed by the execution of one of the *BSN (21) – (24)* instructions. These instructions will check sense switches 1 through 4, respectively, and execute a branch to the location specified by the right half portion of the instruction if the sense switch is active. Execution of the *BSN (21) – (24)* instructions has no effect on the status of the sense switch but will merely determine its status.

3.19.9 Status Drum Parity Error

If a parity error occurs while reading a status drum field, the status drum parity error flip-flop will become set. The program can then check this flip-flop by the execution of a *BSN (25)* instruction. If the flip-flop is set, the program will branch to the location specified in the right half-word of the instruction and will clear the flip-flop.

3.19.10 Duplex Switching Completed ACTIVE

When switchover is effected between computer A and computer B, several seconds are required before all the relays have been transferred. When the machine that is assuming the active status wishes to know at what time the switchover is completed, a *BSN (30)* instruction may be executed. This instruction will check the status of a sense unit and branch to the location specified in the right half portion of the *BSN (30)* instruction if the switchover is completed. This provides the computer with a means of determining by program control when to initiate the active air defense program.

3.19.11 Alarm 1 ON

When the switches on the duplex maintenance console are set to prohibit the generation of an alarm 1 condition (inactivity, memory parity, or addressable drum parity), it is necessary to execute a *BSN (41)* instruction to detect the occurrence of an alarm 1 condition. When an alarm 1 condition is present, it sets the alarm 1 flip-flop. Execution of the *BSN (41)* instruction will check the status of the flip-flop and branch to the memory location specified in the right half portion of the instruction if the flip-flop is set. The *BSN (41)* instruction also clears the alarm 1 flip-flop.

3.19.12 Alarm 2 ON

When we wish to sense for an alarm 2 condition (tape parity or status drum parity), we can execute a *BSN (42)* instruction. This instruction will check the status of the alarm 2 flip-flop and branch to the location specified by the right half portion of the *BSN (42)* instruction if the flip-flop is set. The *BSN (42)* instruction also clears the alarm 2 flip-flop.

3.19.13 Intercommunication Flip-Flops ON (1-4)

When one computer desires to notify the other that a specific condition exists, it may execute one of four instructions, *PER (10) – (13)*, which will turn on intercommunication FF's 1 through 4 in the other computer. These flip-flops can be sensed by the appropriate *BSN (43) – (46)* instruction. If the flip-flop being sensed is set, the computer executes a branch of program control to the address specified in the right half portion of the instruction. Execution of a *BSN (43) – (46)* will also clear the intercommunication flip-flop being sensed.

CHAPTER 4
17-BIT ADDRESS SELECTION

(This chapter does not apply to AN/FSQ-8)

4.1 GENERAL

Prior to the inclusion of an expanded memory unit in the AN/FSQ-7, a total of 4096_{10} registers were available for use in each of the two memory units. These 4096_{10} registers could be designated by a total of 12 bit positions in the address portion of the instruction ($4096_{10} = 2^{12}$). These bits occupied positions R4 through R15. If the address specified by these 12 bits was in core memory 1, this fact was designated by a binary code of 000 in bits R1 through R3. On the other hand, if core memory 2 was selected, a binary code of 001 was specified. Test memory was selected by a binary code of 010 in positions R1 through R3, and the clock register was selected by a code of 110. This arrangement utilized all 15 significant bits of the address, which was just sufficient to select all the registers that were available. The expanded memory unit has a total of $65{,}536_{10}$ registers, which requires 16 significant bit positions ($65{,}536_{10} = 2^{16}$) which is the entire right half portion of the instruction word, including the sign bit. To provide for the selection of the small memory and test memory, it was necessary to utilize the LS bit position as the most significant bit of the address. Bit LS is still physically contained in the left half-word, but can be used as a bit in the address portion of the instruction if it is desired. However, there are instances when using the LS bit as an address bit has no advantages; therefore, these instructions are still handled with 16-bit addresses. For example, there is no reason why we should specify 17-bit operation with an instruction such as *DCL,* since the 17th bit would never be involved in decoding. It should be obvious by now that 17-bit operation is desirable only where absolute addresses can be changed or modified by various instructions. Operands within the AN/FSQ-7 are still treated as 16-bit numbers, and any discussion of 17-bit operation does not include operands but merely internal memory addresses. The purpose of this chapter is to explain the use of the 17-bit option which is available with some instructions. (Technically, all instructions which do not utilize the auxiliary bits can be given the 17-bit option; however, the option is meaningful in only a few instances.) In addition, some of the pitfalls of programming 17-bit and 16-bit instructions together are pointed out; an indication of how these pitfalls can be avoided is also provided.

4.2 INDEXING CHANGES

We know that the basic process of indexing within the AN/FSQ-7 involves adding the contents of a specified index register to the contents of the address register; in this way, we can modify the address portion of an instruction while executing the same instruction several times. Each time a different index register value is added to the address specified in the right half of the instruction, we can obtain a different operand. The index registers were originally 16-bit registers and could be loaded with a maximum value of 0.77777. If the index registers had not been changed, we would not be able to index in the higher addresses in memory. In order to permit the index register to be set to any number from 0.00000 through 1.77777, an extra bit was added to the index register. This bit is located between the sign bit and bit 1 position of the index registers and is referred to as bit 0. This extra bit is added to index registers 1, 2, 4, and 5 only. The right accumulator (index register 3) is not affected by this change.

Another change brought about when the index registers were increased to 17 bits is that the sign bit is cleared to a 1 when the remaining bits are cleared to 0. Thus, the cleared state of the index registers is expressed as 2.00000, with the number 2 being represented by a 1 and 0 in the index register sign and zero positions, respectively. In addition, the index register sign bit is no longer transferred to the address register when a modification is to take place. A requirement for transfer of a 17-bit number to the index register from the address register is that the bit position LS of the address register must contain a 0. When the normal reduction of the index register takes place, the restriction that bit LS must contain a 0 means that the index register cannot be stepped lower than its cleared state of 2.00000, which now becomes the no-branch condition.

As an example of programming with a 17-bit index register, consider the *ADX* instruction. This instruction specifies that the indicated index register be added to the address register and that the sum be placed in the right A register. However, since we can no longer transfer the index register sign bit, the transfer consists only of 16 bits. The transfer from the address register to the

right A register will still be made up to 17 bits; thus, it is important to realize that although the address in the right A register contains 17 bits, it was generated by adding a 16-bit and a 17-bit number together rather than two 17-bit numbers.

4.3 OTHER REGISTERS CHANGED

4.3.1 Address Register

Since we can now specify a 17-bit address in the AN/FSQ-7, it was mandatory that the address register also be increased one bit position to accommodate the higher numbered addresses which will be encountered. The bit used is the LS bit and is the most significant position of the address register.

4.3.2 Program Counter

It was also necessary to increase the program counter to 17 bits so that the higher memory addresses can be obtained when the instructions are being selected by the normal transfer of the program counter to the memory address register.

4.3.3 Right A Register

The right A register is used to contain addresses of instructions after execution of the *ADX* and all the branch instructions. Because we will encounter 17-bit addresses, it was necessary for the right A register to have one additional bit position connected to it also. Once again, it should be emphasized that this change was made to handle higher memory addresses and not to accommodate operands. When the A registers are involved in the normal arithmetic processes, the 17th bit in the right A register has no significance.

4.4 INSTRUCTION OPTIONS

4.4.1 *STA* Option

The *STA* instruction may store either a 16- or 17-bit number from the right A register into the memory location specified by the right half portion of the instruction. If bit L12 contains a 1, the entire 17-bit right A register will be stored; if bit L12 contains a 0, only bits RS through R15 will be stored. Since the *STA* instruction is utilized after the execution of a *BPX* or an *ADX* instruction, the right A register usually contains an address. Therefore, for all practical purposes, we can say that 17-bit operation should always be used in conjunction with the *STA* instruction.

4.4.2 *AOR* Option

When the *AOR* 17-bit option is used, the contents of the LS, RS through R15 bit contents will have a 1 added into bit R15. When the 17-bit option is not used, the addition takes place with the contents of RS through R15 only. If the *AOR* register is being used merely to step a pass counter, the 17-bit option is not necessary, and the counter may be stepped up to a limit of 0.77777 without causing overflow. However, if the *AOR* instruction is used in conjunction with an address modification, the 17-bit option should definitely be employed. As an example, suppose we were going to *AOR* a register which contains an address of 1.77777 (highest address in the expanded memory unit.) If we specify 17-bit operation with this instruction, the addition process will take the carry-out from the RS bit position and put it in the LS bit position, giving us an answer of 2.00000, which is a valid address. On the other hand, if 17-bit operation had not been specified, the addition of 1 to address 1.77777 would have resulted in an answer of 0.00001, which would be incorrect. Thus, it is important to consider what the *AOR* instruction will be modifying before assigning the type of operation to be used.

4.4.3 *RST* Option

The *RST* option is contingent on almost the same thing as the *AOR* option. If an operand is to be stored from the right accumulator, 16-bit operation is satisfactory. However, if an address is involved, it is important to exercise the 17-bit option, since failure to do so could result in a loss of the most significant bit of the address being stored.

4.4.4 *ADD, SUB,* and *ADB* Option

The options available for use with the *ADD* and *SUB* instructions again depend on whether or not we are dealing with true operands or addresses in the arithmetic element. As previously mentioned, we do not use 17-bit operands in the AN/FSQ-7; therefore, we do not wish to perform arithmetic operations on operands and specify 17-bit operation, since this will lead to erroneous results. On the other hand, if we have an address in the right accumulator which we are going to modify with the use of an *ADD, SUB,* or *ADB* instruction, it is mandatory that we exercise the 17-bit option. The reason for this is the same as for the *AOR* instruction: We do not want to lose a bit of the address when making modifications to that address by use of the *ADD* or *SUB* instruction. It should be noted that the carry-out of the LS bit is connected to the right end-carry input of the R15 position during 17-bit execution of the *ADD* and *SUB* instructions. The carry-out of L1 bit position to the LS bit position is inhibited, so, in effect, the L1 through L15 addresses perform no useful function during execution of an *ADD, SUB,* or *ADB* instruction which is using the 17-bit option.

4.5 OVERFLOW CONTROL

Arithmetic modification of 17-bit addresses brings up a problem which did not concern us with 16-bit operation. The problem concerns the generation of an overflow alarm when such an indication is not desired. This can best be illustrated by an example similar to the one discussed in the *AOR* operation. If we wish to

place address 0.77777 in the right accumulator and then add a value of 0.00002, we can expect to arrive at an answer of 1.00001. While this answer is correct, and it results in a valid address, it would cause an overflow alarm by the carry of 1 into the RS bit position just as would be the case if two operands of 0.77777 and 0.00002 were deleted. In the case of the operands, we want to know that we have exceeded the capacity of the machine; therefore, we generate an overflow alarm. However, when adding two memory addresses, this condition is not a true computer overflow and the alarm should be eliminated. An option has been provided so that we can suppress the overflow alarms when performing arithmetic modification of 17-bit addresses. It involves the use of bit L13 of the auxiliary bits in the instruction word. When this bit is set to 1, the overflow alarms are suppressed, and when it is set to 0, the alarm circuitry functions normally.

To show how this alarm suppression option is utilized, let us consider the original problem of modifying an address of 0.77777 by 0.00002. The routine to accomplish this and suppress the overflow is shown in table 3–28.

TABLE 3–28. OVERFLOW ALARM SUPPRESSION ROUTINE

LOCATION	OPERATION	ADDRESS
0.00370	*CAD*	0.01425
0.00371	*ADD 04*	0.05055
0.00372	*RST 10*	0.01425
0.00373	*HLT*	—
0.01425	*CAM*	0.77777
0.05055	0.00000	0.00002

The *ADD 04* instruction specifies that the overflow suppression bit (L13) is active and will not cause an overflow alarm when the address and operand are added together. In addition, the *RST 10* instruction specifies that the 17-bit option is being exercised (by the presence of a 1 in bit L12) and thus will store the contents of the LS, RS through R15 in the LS, RS through R15 portions of the designated address.

4.6 INSTRUCTION SUMMARY

A summary of the instructions related to the Central Computer System is given in table 3–29.

TABLE 3–29. SUMMARY OF CENTRAL COMPUTER SYSTEM INSTRUCTIONS

INSTRUCTION NAME	MNEMONIC NAME	OCTAL CODE	EXECUTION TIME	INDEXABLE	CAUSE OVERFLOW
Reset Index Register from Right Accumulator	*XAC*	764	6 μsec	No	No
Store Address	*STA*	340	18 μsec	Yes	No
Add Index Register	*ADX*	770	6 μsec	No	No
Clear and Add Magnitudes	*CAM*	160	12 μsec	Yes	No
Difference Magnitudes	*DIM*	164	12 μsec	Yes	No
Add B Registers	*ADB*	114	12 μsec	No	Yes
Multiply	*MUL*	250	17 ± 0.5 μsec	Yes	No
Twin and Multiply	*TMU*	254	17 ± 0.5 μsec	Yes	No
Divide	*DVD*	260	51.5 ± 0.5 μsec	Yes	No
Twin and Divide	*TDV*	264	51.5 ± 0.5 μsec	Yes	No
Shift Left and Round	*SLR*	024	Variable	No	Yes
Dual Shift Left	*DSL*	400	Variable	No	No
Dual Shift Right	*DSR*	404	Variable	No	No
Left Element Shift Right	*LSR*	440	Variable	No	No
Right Element Shift Right	*RSR*	444	Variable	No	No
Accumulators Shift Left	*ASL*	420	Variable	No	No
Accumulators Shift Right	*ASR*	424	Variable	No	No

TABLE 3–29. SUMMARY OF CENTRAL COMPUTER SYSTEM INSTRUCTIONS (cont'd)

INSTRUCTION NAME	MNEMONIC NAME	OCTAL CODE	EXECUTION TIME	INDEXABLE	CAUSE OVERFLOW
Dual Cycle Left	*DCL*	460	Variable	No	No
Full Cycle Left	*FCL*	470	Variable	No	No
Extract	*ETR*	004	12 μsec	Yes	No
Load B Registers	*LDB*	030	12 μsec	Yes	No
Deposit	*DEP*	360	18 μsec	Yes	No
Exchange	*ECH*	350	18 μsec	Yes	No
Compare Left Half-Words	*CML*	044	12 μsec	Yes	No
Compare Right Half-Words	*CMR*	042	12 μsec	Yes	No
Compare Full Words	*CMF*	046	12 μsec	Yes	No
Compare Masked Bits	*CMM*	040	12 μsec	Yes	No
Compare Difference Left Half-Words	*CDL*	045	12 μsec	Yes	Yes
Compare Difference Right Half-Words	*CDR*	043	12 μsec	Yes	Yes
Compare Difference Full Words	*CDF*	047	12 μsec	Yes	Yes
Compare Difference Masked Bits	*CDM*	041	12 μsec	Yes	Yes
Test One Bit	*TOB*	050	12 μsec	Yes	No
Test Two Bits	*TTB*	054	12 μsec	Yes	No
Clear and Add Clock	*CAC*	170	12 μsec	No	No
Operate	*PER*	01-	12 μsec	No	No
Branch on Sense	*BSN*	52-	12 μsec	No	No

PART 4
PROGRAMMING THE INPUT-OUTPUT SYSTEMS

CHAPTER 1
INTRODUCTION

1.1 GENERAL INFORMATION

This part will present an introduction to, and a brief summary of, the operating principles of the Input-Output Systems associated with the AN/FSQ-7 and AN/FSQ-8. The specific information necessary for proper programming of each system will be presented in a separate chapter. The term, Input-Output System, does not limit the coverage of this part only to the physical equipment referred to as the Input System and the Output System in the AN/FSQ-7 and AN/FSQ-8. Rather, it is intended to refer to all systems that are external to the Central Computer System. For example, Display System programming is covered in this part, since the displays do represent an output as far as the Central Computer System is concerned. The Drum System is not discussed as an individual system since it is used in part by all the other systems within the AN/FSQ-7 and AN/FSQ-8. Instead, the information pertaining to the Input System drum fields will be discussed in the chapter dealing with programming of the Input System, information pertaining to the Output System drum fields will be discussed in the chapter dealing with programming of the Output System, etc. In addition, the card machines and tape units which are physically included as part of the Central Computer System are discussed in a separate chapter since they also serve as inputs and outputs to the Central Computer System.

1.2 INPUT-OUTPUT ELEMENT

1.2.1 Description

The input-output (IO) element of the Central Computer System is responsible for the control and transfer of all information both to and from devices outside the Central Computer System. This element was not discussed in conjunction with the Central Computer System since its function is so closely related to the external devices. The block diagram of the IO element is shown in figure 4–1. It can be seen that this element is composed primarily of various registers and counters which supply the Central Computer System with both tactical data and program instructions. Transfers to and from internal memory from the IO element usually involve blocks of words, whether they are data words or instructions. For this reason, two types of machine cycles are required to handle these transfers in place of the PT, OTA, and OTB cycles discussed up to this point.

1.2.2 Break Cycles

The execution of a memory cycle is required for each word transferred by the IO element during an IO operation. During the execution of this memory cycle, internal operations are suspended while the Central Computer System executes a machine cycle called a break cycle. A word supplied to core memory from the IO element is written into memory during a break-in (BI) cycle. Conversely, a word that is read out of core memory is delivered to the IO element during what is known as a breakout (BO) cycle. As previously stated, both the BI and BO cycles are coincident with the occurrence of a memory cycle and are divided into pulse sequences just as the other three types of machine cycles. For example, an action which is executed at BO 8 means that this action occurs at the ninth time pulse issued during a BO cycle. Unlike PT cycles, which occur once for each instruction executed, and OT cycles, which occur if the instruction decoding matrix determines whether the instruction requires OT cycles, the BI and BO cycles are executed by request of a device external to the Central Computer System. This is, of course, the IO device, which is either reading information out of memory or writing information into it. Requests for break cycles may occur at any time during normal internal operations and are executed at the completion of the internal machine cycle in progress at the time the break request was received. (One exception to the execution of break cycles is the fact that they may be executed during an arithmetic pause.) Upon completion of the break cycle the Central Computer returns to its normal operation.

It should be emphasized that the occurrence of BI and BO cycles is governed by the speed at which the IO device can accept information from the IO element

or transfer information to it. Since all the IO devices are slower than core memory, break cycles will usually not occur one after the other but will intervene between regular machine cycles at various intervals.

1.2.3 Registers and Counters

1.2.3.1 IO Register

The IO register is the intermediate register through which all information entering or leaving the Central Computer System must pass. In transfers from the Central Computer to an IO device, the information is obtained directly from the memory buffer registers. Transfers into core memory may be made by directly loading the IO register with the information or first loading a buffer register, which then transfers its contents to the IO register at the proper time. The IO register contains 33 positions, to accommodate a full computer word plus parity.

1.2.3.2 IO Buffer Register

The IO buffer register performs two main functions: it adapts the high speed of the computer to the relatively low speed of the various external devices by acting as a temporary storage device for information being transferred between the computer and a particular external device, and, in conjunction with the drum comparison circuits, it determines whether a computer-to-drum information transfer is to take place. During break-in operations, the IO buffer register is used to transfer words from the Drum System, card machines, manual inputs, and the burst-time counter into core memory via the IO register.

1.2.3.3 IO Address Counter

When words are to be transferred into or out of core memory, it is necessary to specify which location in the core memory these words are to come from or go to. Before an IO operation can be initiated, the IO address counter must be loaded with the address in core memory from which or to which the first word is to be transferred. During the break cycle, the content of this counter is transferred to the core memory so that the word will be read out of, or stored in, the proper core memory location. As each word is transferred between core memory and the selected IO device, a 1 is added to the contents of the IO address counter. Thus, when an IO operation is in progress, the IO address register supplies sequential memory addresses to the memory address register, just as the program counter supplies the address of the next instruction to be executed during a program. (Refer to fig. 2–10.) The IO address counter is loaded with an instruction which will be explained later in this chapter.

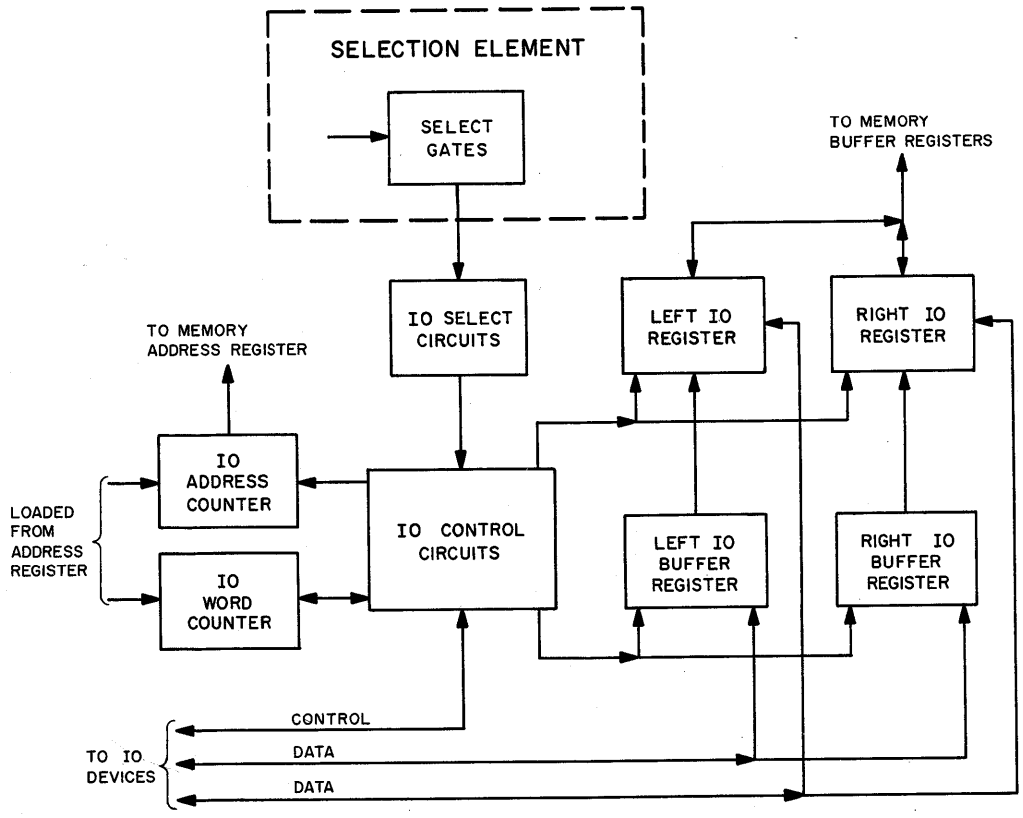


Figure 4–1. Input-Output Element, Information Flow

1.2.3.4 IO Word Counter

The IO word counter performs the function of counting the number of words which have been transferred in an IO operation. The complement of the number of words to be transferred between the Central Computer System and the IO device of the AN/FSQ-7 or AN/FSQ-8 is placed in the IO word counter. Each time a word is transferred between core memory and an IO device, a 1 is added to the IO word counter. When the number of words specified has been transferred, the IO word counter contains +0. As a result of the last stepping, the word counter generates an end-carry pulse, which is transferred to the selection element, indicating that the IO operation has terminated.

1.2.4 Control Circuitry

1.2.4.1 Break Command Generators

The break command generators determine when a BI or BO cycle will occur. As mentioned previously, the IO device to be used requests a break cycle, and this request is honored at the end of the internal cycle then in progress. Break requests from the various IO devices are applied to the break command generators, which then gate time pulses out of the instruction control element. These pulses actually form the break pulses which occur during a break cycle. Thus, the IO device requests a break cycle through the break command generators, which initiate, synchronize, and terminate the IO word transfer.

1.2.4.2 Data Transfer Control Circuits

These circuits provide the controls to direct the transfer of data within the IO element. Since each IO device operates in a different manner, it is necessary to provide separate controls for each type of transfer. In addition, the main drums may be read or written on by three different modes, and control is provided for each of these modes.

1.3 IO INSTRUCTIONS

1.3.1 *Select* Instruction

The *Select (SEL)* instruction chooses the IO device (except for drums) which is to be operated during the subsequent IO operation. The various devices which may be selected are specified by bits L10-L15 of the instruction word. Each of the codes dealing with the IO device will be covered in the chapter on the programming of that device. The effect of the *SEL* instruction is to prepare the designated IO device for a data transfer and, in so doing, deselect all the remaining IO devices. The actual selection of a particular IO device is accomplished by decoding the auxiliary bits in the same matrix used for the *PER* and *BSN* instructions. Once a *SEL* instruction has been given for a particular IO device, that device will remain selected until another *SEL* instruction specifying a different device is executed. The *SEL* instruction does not utilize the right half-portion of the instruction word and, therefore, may not be indexed. Execution of this instruction requires 12 μsec. It is designated by an octal operation code of 62.

1.3.2 *Select Drums* Instruction

The *Select Drums (SDR)* instruction is used to perform the same function as the *SEL* instruction, except that the selections are limited to fields on the magnetic drums. Bit R1 of the instruction word determines whether the main drum group or the auxiliary drum group has been selected. If the bit contains a 1, the auxiliary drums are selected; if the bit contains a 0, the main drums are selected. The auxiliary bits (L10-L15) further designate which particular drum field is to be used in the IO operation. If the drum is an addressable one (i.e., the words stored on the drum are referred to by a specific register number), the address of the first drum register to be used is specified in the right half-portion of the *SDR* instruction. If the drum is read by identity, the identification code is placed in the right half-word of the instruction. When an *SDR* instruction is given, the particular drum field selected will remain selected until another *SEL* or *SDR* instruction is executed. This instruction requires 12 μsec for execution and may be indexed. An octal code of 61 is used to designate the *SDR* instruction.

1.3.3 *Load IO Address Counter* Instruction

The *Load IO Address Counter (LDC)* instruction is used to replace the contents of the IO address counter with the value specified in the right half-portion of the instruction word. This value is the address of the first core memory location involved in an IO operation. The *LDC* instruction may be indexed and requires 6 μsec to execute. It is designated by an octal operation code of 600.

1.3.4 *Read* Instruction

The *Read (RDS)* instruction initiates the transfer of information from an IO device which has been previously selected by an *SEL* or *SDR* instruction. The address part of the *RDS* instruction is used to designate the number of words to be transferred from the IO device. This number is placed in the IO word counter in complement form, and each word transferred to core memory causes a 1 to be added to this counter, thus effectively decreasing the value in the counter. When an addressable drum field is the IO device used to supply words to core memory, bits L13-L15 of the *RDS* instruction specify in what manner the drum registers are to be read; that is, we can read consecutive registers, or every 8, 16, or 64 registers, if we desire. This method of reading is known as interleave, and the code to be placed in bits L13-L15 for the desired pattern of interleaving is shown in table 4–1.

When an IO transfer to core memory is initiated by the *RDS* instruction, a control flip-flop known as the IO interlock is turned on. While the IO interlock is on, no additional IO operations may be started. Thus, if the IO interlock is on, the execution of the *SEL, SDR, LDC,* or another *RDS* instruction (plus an instruction requesting words to be transferred to an IO device from core memory) is delayed until the IO interlock is cleared. Either a disconnect pulse from the IO device involved in the operation or the end-carry from the IO word counter (signifying all words have been read) will clear the IO interlock. Internal machine operations which do not affect the IO devices may proceed while the IO interlock is set. (One exception to this is the execution of the *HLT* instruction. Though the *HLT* instruction does not deal directly with an IO device, its execution while an IO transfer is in progress would cause a loss of information. Therefore, the *HLT* instruction is also delayed until the IO interlock is cleared.)

The *RDS* instruction requires 6 μsec to execute and is not indexable. It is designated by an octal operation code of 670.

1.3.5 *Write* Instruction

Execution of the *Write (WRT)* instruction will initiate a transfer of information from core memory to the IO device selected by the *SEL* or *SDR* instruction. The address portion of this instruction is used to designate the number of words to be written, and, as with the *RDS* instruction, this number is placed in the IO word counter in complement form. The word counter is stepped by 1 for each word transferred out of core memory. When an addressable drum is used as the IO device receiving information from core memory, bits L13-L15 of the *WRT* instruction word specify the interleave pattern to be used. The interleave code for the *WRT* instruction is the same as for the *RDS* instruction and is shown in table 4–1.

In addition to initiating the word transfer from core memory to the IO device, execution of the *WRT* instruction will also turn on the IO interlock. The interlock will delay execution of any IO instruction which is used to prepare for another IO operation or the *HLT* instruction. It is cleared when all words have been written or the IO device sends a disconnect signal.

TABLE 4–1. INTERLEAVE CODE

BINARY CODE (Bits L13-L15)	INTERLEAVE PATTERN
000	Consecutive registers
001	Every 8th register
010	Every 16th register
100	Every 64th register

The *WRT* instruction is executed in 6 μsec and is not indexable. It is specified by an octal operation code of 674.

1.3.6 *Clear and Subtract Word Counter* Instruction

The *Clear and Subtract Word Counter (CSW)* instruction is used to transfer the contents of the IO word counter to the right accumulator. This is done when it is desired to determine the number of words actually transferred during an IO operation.

Remember that the *RDS* or *WRT* instruction placed the complement of the number of words to be transferred in the IO word counter. Each time a word is transferred during an IO operation, a 1 is added to the contents of the IO word counter. However, when an *RDS* or *WRT* instruction is given, a 1 is added to the IO word counter before any words are transferred; therefore, the IO word counter actually contains the 2's complement of the number of words to be transferred. (You will recall that, during the discussion of binary arithmetic, it was stated that the 2's complement on a number was simply the 1's complement plus 1.) The 2's complement is used in the IO word counter so that no end-carry will be necessary when the last word is read and the last 1 is added to the counter. Thus, reading the last word will clear the IO word counter to +0. In the event that the specified number of words is not transferred, the IO word counter will contain the 2's complement of the number of words still remaining to be read.

The *CSW* instruction takes place in the following manner. The right accumulator is cleared (plus the left accumulator if 17-bit operation is specified), and the contents of the IO word counter are transferred to the right accumulator. Since the IO word counter already contains a complemented number, transferring its contents is the same as subtracting a positive number. It is important to realize that the action of the *CSW* instruction does not subtract the contents of the IO word counter from the cleared right accumulator, but merely transfers them. The *CSW* instruction does not utilize the right half-portion of the instruction word; therefore, it may not be indexed. The execution time of this instruction is 6 μsec and is specified by an octal operation code of 020.

1.4 IO PROGRAMMING

1.4.1 General

When we desire to program a transfer either to or from core memory, it is necessary to set up a definite

program pattern using the instructions described above. First of all, we must know what device we are going to operate; therefore, we must have an *SEL* or *SDR* instruction to connect the proper device to core memory via the IO element. We must also know where in memory the transfer of words is to start and whether we are going to read or write; thus we must also have an *LDC* instruction in our program. Finally, an *RDS* or a *WRT* instruction is given to initiate the transfer, either to or from memory. The *SEL* or *SDR* instruction and the *LDC* instruction may be given in any sequence, but the *RDS* or *WRT* instruction must be the last instruction in the program since it actually starts the transfer and sets the IO interlock, thus preventing any other IO instructions from being executed. The basic IO pattern of instructions in a program then is as follows:

a. *LDC*
b. *SEL* or *SDR*
c. *RDS* or *WRT*

Naturally, each of these instructions has many possible values that can be specified, or many devices that can be selected; but the basic IO program must consist of all three instructions, and the *RDS* or *WRT* instruction must be given last, as previously stated. After the *RDS* or *WRT* instruction has been executed, the program will continue to perform internal machine operations as long as they do not affect the IO devices in any way. Sometimes, however, it is desirable to take definite courses of action while an IO program is in progress and then return to the program immediately following the IO program steps. Various methods of operating the Central Computer System while an IO transfer is in progress are available, and some of these are discussed in 1.4.2.

1.4.2 Normal Operation

1.4.2.1 Real-Time Considerations

When we execute the basic IO program steps listed above, the IO interlock becomes set and we cannot perform any other IO operations until it is cleared. However, during any one IO operation, it is usually impossible to tell exactly when the IO interlock will be cleared. To avoid a loss of operating time with the Central Computer System when IO operations are being performed, two methods of programming may be used. The first method involves executing some instructions after the *RDS* or *WRT* instruction has been given, which will require more time than the word transfer between the IO device and core memory. For example, the card reader is capable of supplying a rate of 150 cards per minute. If we wanted to read in 100 cards of information, we could perform some arithmetic operations on data already within memory (which would require approximately 1 minute to execute) and be assured that our IO operation is terminated when the intermediate program is finished. The sequence of an operation such as this is:

a. *SEL*
b. *LDC*
c. *RDS*
d. Routine requiring approximately 1 minute
e. Use data read in if desired

1.4.2.2 *Branch If IO Interlock On* Instruction

The above method of operation, though satisfactory in many instances, may not be desirable because the data being read in may be in core memory for some time before it is utilized. For this reason, the *Branch If I/O Interlock On (BSN 14)* is provided. This variation of the *BSN* instruction checks the IO interlock and branches to the memory location specified by the right half-portion of the instruction if the IO interlock is on. If the IO interlock is not on, the program falls through this instruction and selects the next sequential instruction location. Thus, if we wished to perform an IO operation and utilize the Central Computer System for some other operation only until the data we wanted was in core memory, we could utilize the *BSN (14)* instruction. After giving the basic IO instructions necessary for the initiation of any IO operation, a *BSN (14)* instruction immediately following will check to see whether the IO interlock is on. As long as it is on, we will branch to a subroutine, execute a pass (or passes), and then check the IO interlock again. As soon as the interlock is turned off, we can proceed with the program using the data just obtained. As an example, suppose we are reading in some information from the card reader and wish to process this information as soon as possible. However, some data already exists in the Central Computer System which also needs to be processed, but not immediately. While we are waiting for the data from the card reader to be placed in memory, we can process, at least partially, the data on hand. The method for doing this is given in table 4–2.

The operation of the program is such that we will check the IO interlock after each data word processed by the program contained in locations 0.37001 through 0.37006. If the interlock is still on, we process another word. (Check to see whether both halves are minus; if not, store the word.) In the event that all the words were processed by the alternate program before the card reader had transferred all the words to core memory, we attempt to execute an *HLT* instruction. This instruction will be delayed until the IO transfer is complete; then it will be executed. Thus, if our program halts at step 0.37006, we know that all the words were read in from the card reader and that we also had processed all the data in locations 0.02000 through 0.06000.

If the program does not halt, we know that the IO interlock was turned off before all the words were processed, and we fell through the *BSN (14)* instruction at 0.00105, executing the branch to the program utilizing the data just read in. In this case, it would be necessary to preserve the contents of the index registers at the time we branched from location 0.00106, so that the data-processing going on could be finished at some other time.

The alternate data-processing program will require 24 μsec for one pass through it if the word is negative, or 42 μsec if the word is positive. Thus, we will be checking the status of the IO interlock at least every 42 μsec, sensing for the termination of the IO transfer.

1.4.3 IO Pause

An IO pause occurs when there is an IO transfer in progress and no information is available in memory with which we may occupy the Central Computer System. As explained above, normal operation during an IO transfer will also utilize the Central Computer System to some extent. However, when the information needed to perform some program is the same information that is being obtained during an IO transfer, this is a pause condition. The Central Computer System must be prevented from executing the instructions immediately following the instructions which initiate the IO transfer. This is most easily done by giving another IO instruction or an *HLT* instruction immediately after the *RDS* or *WRT* instruction. A sample program which will start an IO pause is shown in table 4–3.

TABLE 4–2. DATA READ-IN PROGRAM

LOCATION	OPERATION	ADDRESS
0.00100	*SEL* (Card Reader)	
0.00101	*LDC*	0.01000
0.00102	*RDS*	0.00300
0.00103	*1 XIN*	0.04000
0.00104	*2 XIN*	0.04000
0.00105	*BSN (14)*	0.37001
0.00106	*BPX*	To program using data supplied by card reader
0.37001	*1 CAD*	0.02000
0.37002	*BFM*	3.7005
0.37003	*2 FST*	0.60400
0.37004	*2 BPX (01)*	0.37005
0.37005	*1 BPX (01)*	0.00105
0.37006	*HLT*	–
0.01000-0.01300	Storage for data being read in	
0.02000-0.06000	Storage for data already in memory	
0.60400-0.64400	Storage for processed data	

TABLE 4–3. IO PAUSE PROGRAM

LOCATION	OPERATION	ADDRESS
0.0000	*SEL* (IO Device)	–
0.0001	*LDC*	0.01000
0.0002	*RDS*	0.03000
0.0003	*LDC*	0.04250

The selected IO device will start transferring data to core memory, starting at location 0.01000. We have specified that 3000_8 words are to be read; after the reading has started, we attempt to load the IO address counter with another location. Since the *LDC* instruction cannot be executed while the IO interlock is on, internal machine operations will cease until the IO interlock is cleared by the carryout from the IO word counter. Then, the *LDC* instruction will be executed, specifying that the same IO device is to prepare for another transfer, starting at location 0.04250. Termination of the IO transfer does not deselect the IO device, and it will remain selected until another *SEL* instruction is given which designates a different IO device. In some instances, IO pauses cannot be avoided; however, we usually attempt to make as much use of the Central Computer System as possible while an IO transfer is in progress.

1.4.4 IO Hangup

If the IO interlock is set by an *RDS* or *WRT* instruction and the IO device is not able to deliver or accept information, an IO hangup occurs when we attempt to execute another IO instruction. In this case, the IO interlock cannot be cleared because no IO operation will take place, which will result in stepping the IO word counter to 0. This is actually an infinite IO pause, and, since the IO operation will not terminate, we will not be able to execute further instructions. Once this condition occurs, there is no programmed method of clearing the IO interlock. Instead, a pushbutton on the duplex maintenance console must be used to manually reset the IO interlock. An IO hangup will occur if any of the following conditions exist:

a. A nonexistent device is selected.

b. The selected device is not ready.

TABLE 4–4. WORD COUNT TRANSFER ROUTINE

LOCATION	OPERATION	ADDRESS
0.00547	*SDR*	
0.00550	*LDC*	0.01000
0.00551	*RDS*	0.05000
0.00552	*BSN (14)*	0.00552
0.00553	*CSW*	–
0.00554	*ADD*	0.12450
0.00555	*BRM*	0.00270
0.00556	*5 XAC*	–
0.00557	*5 BPX (01)*	–
0.00270	No words read, do another routine	
0.12450	0.00000	0.04777

c. The selected device is not prepared (applicable only to magnetic tapes).
d. The *RDS* or *WRT* instruction is not legal for the selected device.

1.4.5 Use of CSW Instruction

Although the actual use of the *CSW* instruction is in conjunction with programming the Central Computer System, its application is discussed at this point, since a better understanding of its function will result. When an IO transfer is terminated and less than the prescribed number of words have been transferred, we do not have any direct way of examining the IO word counter to determine the contents and so must use the *CSW* instruction. As an example, assume that we wish to perform some operation on data obtained from a magnetic drum field. In the right half portion of the *RDS* instruction, we specified the number of words we desired to read from the drum. The capacity of one drum field is 2048_{10} or 4000_8 registers; therefore, if we try to read more than 4000_8 words from one drum field, we know that the IO word counter will always contain some value at the termination of the IO process. The *CSW* instruction can then be used to determine exactly how many words were transferred. A program to accomplish this is shown in table 4–4.

After the drum transfer has been initiated, the Central Computer will be in an IO pause to be cleared. When it is cleared, we obtain the IO word counter contents and add a constant of 0.04777 to it. This constant will give the actual number of words read, because its value is such that it will offset the first stepping of the IO word counter before any IO transfer actually takes place. (This constant is merely one less than the number of words specified to be transferred, as can be seen by examining the right half-position of the *RDS* instruction.) In the event that no words were read, the IO word counter would contain 1.73000, as it would be stepped once anyway. Adding the constant of 0.04777 would give us –0, and the *BRM* condition at 0.00555 would be satisfied. If one or more words are read, the result in the right accumulator will be positive and indicate the number of words actually transferred. If we wish to use these words in some sort of indexed loop, it is necessary to set the index register to a value of one less than the desired number of repetitions; therefore index register 5 is loaded from the right accumulator and its contents are reduced by one.

It should be noted that 17-bit operation is mandatory for use with the *CSW* instruction in the AN/FSQ-7. If this is the case, transferring the contents of the IO word counter (which is a 17-bit counter) to the LS and R1-R15 bits of the accumulator makes it necessary for us to check the left sign bit of the accumulator for a minus condition. The left sign bit will be negative only if no words were transferred, just as the right sign bit would be with 16-bit operation. Therefore, at location 0.00555, we would substitute a *BLM* instruction for the *BRM* instruction.

1.4.6 IO Test Instructions

1.4.6.1 General

Certain instructions which are available for use with the AN/FSQ-7 and AN/FSQ-8 are mainly for maintenance personnel and are generally not used in a program performing IO operations in a tactical situation. These instructions do not apply to any specific IO device; therefore, they will be discussed in the following paragraphs.

1.4.6.2 *Select IO Register* Instruction

Although the IO register is generally used as the intermediate storage register between core memory and a selected IO device, it may itself be selected as an IO device by the *Select IO Address Register (SEL 04)* instruction. We usually use an *SEL (04)* instruction when we desire to clear core memory or place a fixed pattern into core memory. When any *SEL* instruction is given, the IO register is cleared, and the following *RDS* or *WRT* instruction will again clear it prior to every word transferred. However, when an *SEL (04)* is given, the IO register is *not* cleared by the subsequent *RDS* or *WRT* instruction. Therefore, if we wish to clear or place a definite pattern into memory, we may do so by selecting the IO register and then reading the contents

of the IO register into as many locations as desired. For example, to completely clear the 256^2 memory unit of the AN/FSQ-7, we could execute the program listed in table 4–5.

If we wished to load some fixed pattern containing 1's and 0's into memory, it is first necessary to place the desired pattern in the IO register. This may be done by giving an *SEL (04)* instruction, and then writing the desired pattern into the IO register by using a memory location which contained the pattern. The pattern could then be placed in the desired locations by using the IO register as the device supplying data, which in this case is the same word every time. However, it is extremely important to remember that the IO register is not selected again before initiating the *RDS* instruction, since doing this will clear the IO register and destroy the word it contains. A sample program to load a fixed pattern into core memory is given in table 4–6.

Although it is possible to place any desired pattern in the IO register by writing one word from core memory into it as we have done in the above program, it is interesting to note that, if we write more than one word in the IO register, the register will contain the logical sum of all the words read at the termination of the IO transfer. This is true because the IO register is *not* cleared before each word transfer takes place when it is selected as the IO device. As an example of this feature, assume that we had cleared the 256^2 memory unit by the program listed in table 4–5. If we then proceed to write the entire contents of this memory unit back into the IO register, the logical sum of all the words transferred will be contained in the IO register. Since we were writing all 0's in the IO register to begin with, it should still remain cleared after all the words have been transferred.

TABLE 4–5. PROGRAM TO CLEAR 256^2 MEMORY

LOCATION	OPERATION	ADDRESS
2.00100	*SEL (04)*	—
2.00101	*LDC*	0.00000
2.00102	*RDS*	2.00000
2.00103	*HLT*	—

TABLE 4–6. PROGRAM TO LOAD MEMORY WITH A FIXED PATTERN

LOCATION	OPERATION	ADDRESS
2.01000	*SEL (04)*	—
2.01001	*LDC*	2.01050
2.01002	*WRT*	0.00001
2.01003	*LDC*	0.00000
2.01004	*RDS*	2.00000
2.01005	*HLT*	—
2.01050	1.25252	1.25252

1.4.6.3 *Clear IO Interlock* Instruction

In some instances, it is desirable to program the clearing of the IO interlock once it has been set by an *RDS* or *WRT* instruction. This will normally occur when an IO operation is taking place that is of secondary importance, and we do not wish to wait until the IO operation is terminated. For example, if we wished to check out the various IO control devices and see whether they were enabling us to make IO transfers properly, we could start an IO transfer and then clear the interlock with the *Clear IO Interlock (PER 27)* instruction as soon as it had been determined that all the controls were working properly. In tactical operations, this instruction would probably not be used to any extent.

1.4.6.4 *Lock IO Address Counter* Instruction

The *Lock IO Address Counter (PER 75)* instruction is used to lock the IO address counter at the address it contains at the time the instruction is given, until the IO interlock is cleared. The *PER (75)* instruction usually precedes the *RDS* or *WRT* instruction. If the *PER (75)* instruction is given before an *RDS* instruction, all the words read from the selected IO device will be placed in one core memory location; if the *PER (75)* instruction is given before a *WRT* instruction, the contents of the specified memory location will be written in successive locations of an IO device. The latter technique is very helpful in writing a fixed pattern on drum fields, writing a test pattern with the printer, etc.

CHAPTER 2

PROGRAMMING THE AM DRUMS AND IC FIELDS

2.1 AUXILIARY MEMORY DRUM FIELDS

2.1.1 General

There are 50 drum fields associated with the Central Computer System of the AN/FSQ-7 and AN/FSQ-8, which are used as auxiliary memory devices. Each of these fields contains 2,048 registers; therefore, a total of 102,400 registers are available for storage of information. The information contained on these 50 fields usually consists of subroutines, mathematical tables, constants, blocks of processed data, or other information which is used quite often by the Central Computer System, but not often enough to warrant its inclusion in core memory. In order to obtain or store information on these drum fields, the usual IO process must take place; for that reason, these drum fields are considered to be IO devices rather than a part of memory, although their logical function is to serve as storage media.

The physical Drum System in the AN/FSQ-7 and AN/FSQ-8 is divided into two groups, the main drums and auxiliary drums. There are six physical drums in each group, although some of the fields located in the main drum groups are designated as auxiliary fields. As explained in the execution of the SDR instruction, bit R1 of the instruction word specified whether the main or auxiliary drum group is being selected. Therefore, the codes which are used to designate the auxiliary fields may contain either a 0 or a 1. Since auxiliary drum fields may transfer data only to or from the Central Computer System, they cannot transfer any information to another system in the AN/FSQ-7 or AN/FSQ-8. In addition, all the auxiliary fields are addressable, making it possible to start an IO operation with any drum register we desire. Table 4–7 lists the auxiliary memory fields and the codes used to select them.

2.1.2 Program Examples

As an illustration of the use of the auxiliary memory fields, assume that we wish to load a group of mathematical tables onto the auxiliary memory fields. The tables are originally contained in a deck of punched cards; therefore, it will be necessary for us to first place the data in core memory and then write it on the designated auxiliary field. A program to accomplish such a transfer is given in table 4–8.

TABLE 4–7. AUXILIARY MEMORY DRUM FIELDS

DRUM FIELD	OCTAL CODE	
	R1	L10-L15
Auxiliary memory 1	0	02
Auxiliary memory 2	0	03
Auxiliary memory 3	0	04
Auxiliary memory 4	0	05
Auxiliary memory 5	0	06
Auxiliary memory 6	0	07
Auxiliary memory 7	0	10
Auxiliary memory 8	0	11
Auxiliary memory 9	0	12
Auxiliary memory 10	0	13
Auxiliary memory 11	0	14
Auxiliary memory 12	0	15
Auxiliary memory 13	1	41
Auxiliary memory 14	1	42
Auxiliary memory 15	1	43
Auxiliary memory 16	1	44
Auxiliary memory 17	1	45
Auxiliary memory 18	1	46
Auxiliary memory 19	1	51
Auxiliary memory 20	1	52
Auxiliary memory 21	1	53
Auxiliary memory 22	1	54
Auxiliary memory 23	1	55
Auxiliary memory 24	1	56
Auxiliary memory 25	1	61
Auxiliary memory 26	1	62
Auxiliary memory 27	1	63
Auxiliary memory 28	1	64

TABLE 4–7. AUXILIARY MEMORY DRUM FIELDS (cont'd)

DRUM FIELD	OCTAL CODE	
	R1	L10-L15
Auxiliary memory 29	1	65
Auxiliary memory 30	1	66
Auxiliary memory 31	1	71
Auxiliary memory 32	1	72
Auxiliary memory 33	1	73
Auxiliary memory 34	1	74
Auxiliary memory 35	1	75
Auxiliary memory 36	1	76
Auxiliary memory 37	1	02
Auxiliary memory 38	1	03
Auxiliary memory 39	1	04
Auxiliary memory 40	1	05
Auxiliary memory 41	1	06
Auxiliary memory 42	1	07
Auxiliary memory 43	1	10
Auxiliary memory 44	1	11
Auxiliary memory 45	1	12
Auxiliary memory 46	1	13
Auxiliary memory 47	1	14
Auxiliary memory 48	1	15
Auxiliary memory (spare)	0	20
Auxiliary memory (spare)	0	21

TABLE 4–8. DRUM LOADING ROUTINE

LOCATION	OPERATION	ADDRESS
0.00200	*SEL* (Card Reader)	–
0.00201	*LDC*	0.01000
0.00203	*RDS*	0.04000
0.00204	*SDR (41)*	0.40000
0.00205	*LDC*	0.01000
0.00206	*WRT*	0.04000
0.00207	*HLT*	–

The *SEL* instruction will select the card reader, and 4000_8 words will be read in core memory, beginning at location 0.01000. Immediately after the execution of the *RDS* instruction, we attempt to select an auxiliary drum field. However, execution of this instruction will be delayed until completion of the *RDS* instruction. Notice that the right half-portion of the *SDR* instruction is 0.40000. This indicates that bit R1 contains a 1, and that register 0000_8 of the selected drum field has been chosen as the first register into which information will be written. The combination of bit R1 and the auxiliary bits tells us that auxiliary memory field 13 has been selected as the IO device. Our transfer is to start from location 0.01000 in core memory, and we wish to transfer all 4000_8 words to drum field 13. Since this number (4000_8) is the total number of registers in one drum field, this program will place information in every register of auxiliary memory field 13.

As an example of reading information from an auxiliary memory field, let us assume that we are about to do some sort of calculation which will require the use of a sine and cosine table. We have used sine and cosine tables in previous program examples, but always assumed that the tables were located in core memory. Now let us assume that the sine and cosine table is stored on one of the auxiliary drum fields. All that is really necessary for us to obtain this data is to program a conventional IO sequence. No preliminary instructions are necessary to check the status of the drums, since we consider the drums to be always ready to receive or supply information. The program to obtain the sine, cosine table is given in table 4–9.

TABLE 4–9. DRUM READING ROUTINE

LOCATION	OPERATION	ADDRESS
0.04250	*SDR (02)*	0.03400
0.04251	*LDC*	0.00500
0.04252	*RDS*	0.00377
0.04253	*HLT*	–

In this case, bit R1 is a 0, and the auxiliary bits contain (02); therefore, auxiliary drum field 1 will be selected by the *SDR* instruction. We are reading 377_0 words from the drum, and they will be stored starting at memory location 0.00500. The execution of the *HLT* instruction will be delayed until the specified numbers of words have been read from the drum.

2.2 INTERCOMMUNICATION DRUM FIELDS

2.2.1 General

As previously mentioned, there are actually two Central Computer Systems within each AN/FSQ-7 and

AN/FSQ-8. These two systems are normally both in operation with one performing the air defense function and the other operating in standby status. For ease of reference, one system is referred to simply as "computer A" and the other as "computer B." Periodically, the active system will transfer information to the standby system so that both systems have the latest available information regarding the air defense situation. In addition, when a scheduled switchover (change of function between computer A and computer B) takes place, all the information presently contained in the active system must be transferred to the system about to become active. In order to allow this interchange of information between computer A and computer B, two special drum fields are needed. These fields are called the intercommunication (IC) fields, and they differ from the rest of the drum fields in that the information placed on them is read under control of the other Central Computer System. For example, if information is to be transferred from computer A to computer B, the sequence of transfer is first from core memory A to the IC drum field associated with computer A. This field is referred to as the IC (own) field. Then, the information is read from this field under the control of computer B into core memory B. When computer B reads the IC field, it is said to be reading the IC (other) field. In normal operation, we read and write on the IC (own) fields, and read the IC (other) field. In addition, it is possible for one Central Computer System to read its own IC (other) field for test purposes. It should be kept in mind that there are only two physical IC drum fields, but they represent four logical drum fields to the two Central Computer Systems. Figure 4–2 shows the relationship of the IC fields within the AN/FSQ-7 and AN/FSQ-8.

2.2.2 Selecting the IC Drum Fields

2.2.2.1 Select IC (Other) Field

The IC (other) field is selected by an *SDR* (16) instruction. This prepares the IC (other) field for a reading operation only. If the *SRD (16)* is executed in computer A, the IC drum field in computer B is selected for reading. The *SDR (16)* instruction requires 12 μsec to execute and may be indexed.

2.2.2.2 Select IC (Own) Field

The IC (own) field is selected by an *SDR (26)* instruction. This instruction will prepare the IC (own) field for a reading or writing operation. Information is usually written on the IC (own) field by one Central Computer System for reading by the other Central Computer System. However, the ability to read the IC (own) field enables it to serve as a type of auxiliary drum storage for intercommunication information, if so desired. The *SDR (26)* instruction requires 12-μsec execution time and may be indexed.

2.2.2.3 Select IC (Own Test) Field

The IC (own test) field is selected by an *SDR (76)* instruction. It is used to prepare the IC (other) field for a reading operation; however, the *RDS* instruction is given by the Central Computer System which contains that particular IC drum, and the information is read back into the same Central Computer System to form a test loop. Thus, the *SDR (16)* and the *SDR (76)* instructions actually select the same drum field, but in the case of *SDR (16)* the field is selected by the other computer, and in the case of *SDR (76)* it is selected by the same computer. By referring to figure 4–2, it can be seen that computer A can place information on its IC (own) field, and then obtain this same information from its IC (other) field for test purposes. The *SDR (76)* instruction is indexable, and requires 12 μsec to execute.

2.2.3 Programming IC Transfers

Assume that we wish to make a normal information transfer from the active computer to the standby computer. In this and the following examples, we will assume that computer A is always the active computer and that computer B is always in standby. It is important

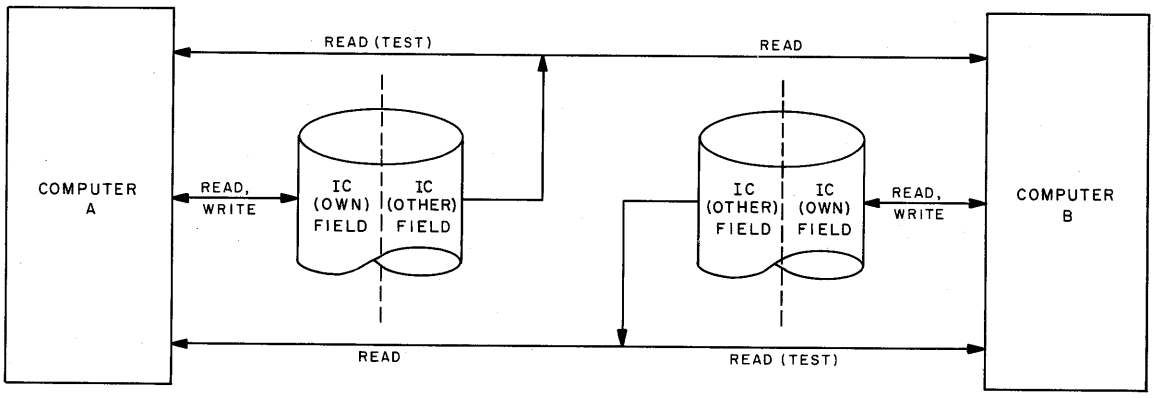


Figure 4–2. Relationship of IC Drum Fields

to keep in mind that a program will be running in computer B as well as computer A. A program to transfer one drum field of information via the IC drums is shown in figure 4–3.

The IC (own) field of computer A is selected by the *SDR (26)* instruction, and we transfer 4000_8 words to the drum. The first word is obtained from memory location 0.01000 and is transferred to drum register 0000. As soon as the IO transfer is started, an *SDR (02)* instruction is given to cause an IO pause while the transfer is taking place. When the IO transfer is completed, the *SDR (02)* instruction will be executed. (Actually, internal operations could take place during this transfer, if desired.) After that, we set intercommunication flip-flop 1 in computer B to notify it that the drum field has been loaded, and that computer B may obtain the information. While this program is running in computer A, computer B is in an idle loop, sensing the status of its intercommunication flip-flop 1. When the flip-flop becomes set by computer A, the branch condition at step 0.00120 of the program running in computer B is satisfied. The IC (other) field is then selected, and the information is read in core memory of computer B, starting at location 0.03500. Immediately after the

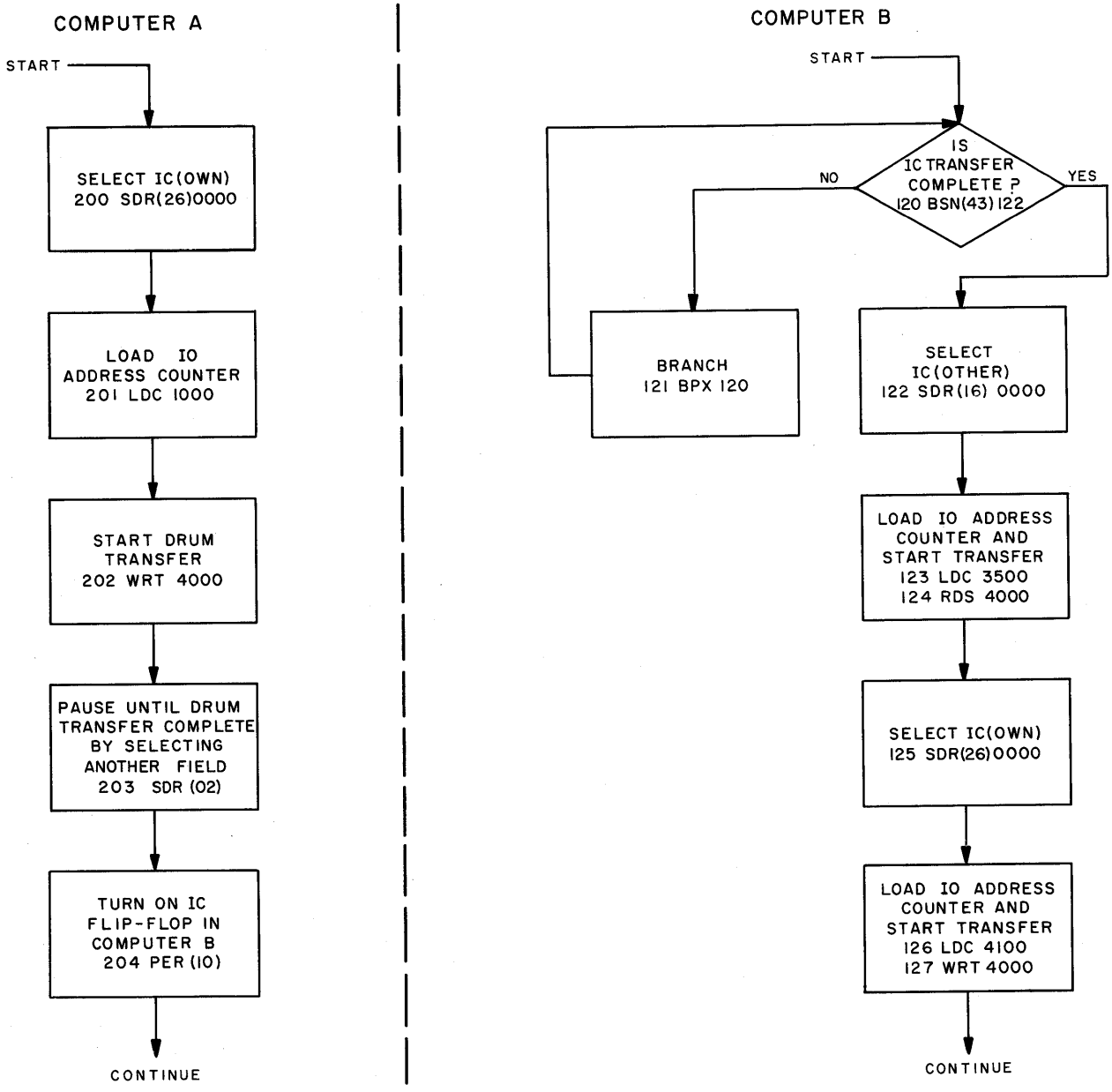


Figure 4–3. Intercommunication Routine

4000_8 words are in core memory, computer B places them on its IC (own) drum field to be stored. This can be done because the normal transfer of information from one Central Computer System to another is from the active machine to the standby machine. Therefore, the IC (own) field of computer B may be used as an auxiliary memory drum field.

If we wish to perform a check on an IC drum, we can form a loop and compare the information in one Central Computer System. Since this is a test procedure, it would normally be performed on the standby computer. For example, if we wish to check the information transfer and drum storage capabilities of the IC drum, we can use a program similar to the one listed in table 4–10.

This program will load the IC (own) field with 4000_8 words taken from core memory locations 0.16000 through 0.21777. Then the *SDR (76)* instruction enables the same machine (computer B in this case) to read the information back into memory, using the circuitry which would normally be used by computer A. This information is transferred to locations 0.02000 through 0.05777. Thus, if this information transfer has been made correctly, we should have two identical blocks of data in core memory. These two blocks of data are compared word by word by use of the *CMF* instruction. As long as the words compare, we will select another word. If, for some reason, the words do not compare, the program will come to an error halt at location 0.00563.

TABLE 4–10. INTERCOMMUNICATION TEST LOOP

LOCATION	OPERATION	ADDRESS
0.00550	*SDR (26)*	0.00000
0.00551	*LDC*	0.16000
0.00552	*WRT*	0.04000
0.00553	*SDR (76)*	0.00000
0.00554	*LDC*	0.02000
0.00555	*RDS*	0.04000
0.00556	*BSN (14)*	0.00556
0.00557	*1 XIN*	0.03777
0.00560	*1 CAD*	0.16000
0.00561	*1 CMF*	0.02000
0.00562	*BPX*	0.00564
0.00563	*HLT*	–
0.00564	*1 BPX (01)*	0.00560
0.00565	*HLT*	–

CHAPTER 3
PROGRAMMING CARD MACHINES AND TAPES

3.1 INTRODUCTION

This chapter discusses the programming principles that are used in connection with the IO devices that are logically a part of the Central Computer System. The units that will be discussed are:

Card Reader (IBM Type 713)
Card Punch (IBM Type 723)
Line Printer (IBM Type 718)
Magnetic Tapes (IBM Type 728)

The card reader is used to load programs and data into the Central Computer System. The card punch and the line printer are used to check out programs (by either punching information on cards or by printing information on paper) that are stored in memory. The punched cards or the printout can then be visually checked by a program. The magnetic tape unit is used to store large blocks of information in a very small space.

The tape unit is the only device discussed in this chapter that can be written on or read from by the Central Computer System. The other devices (reader, punch, and printer) function in only one direction; the reader presents information to the computer and the punch or printer receives information from the computer.

3.2 INFORMATION STORAGE

Information that is contained in the various IO devices appears in many different forms. Before a programmer attempts to write a comprehensive IO program, a knowledge of the way information is stored in each device is helpful.

3.2.1 Punched Cards

The basic IBM card contains 80 vertical columns of 12 rows each. (See fig. 4–4.) Any information punched on such a card is binary in form since the presence or absence of a punch in a particular position is equivalent to a 1 or a 0 in that position. Although an IBM card can hold 960 binary bits (80 x 12), its use in storing purely binary information has assumed importance only with the advent of binary digital computers. The card was originally designed for storage of decimal and alphabetical information. The designations of the rows as numeric or zone rows are derived from this original use of the card. For example, a decimal digit can be stored in each column of the card by placing a punch in the numeric row of that column corresponding in value to the value of the decimal digit to be recorded (or in the 0 row if a 0 is to be recorded). In effect, the decimal digit is represented within the column by using a vertical positional notation of its magnitude; i.e., a 6

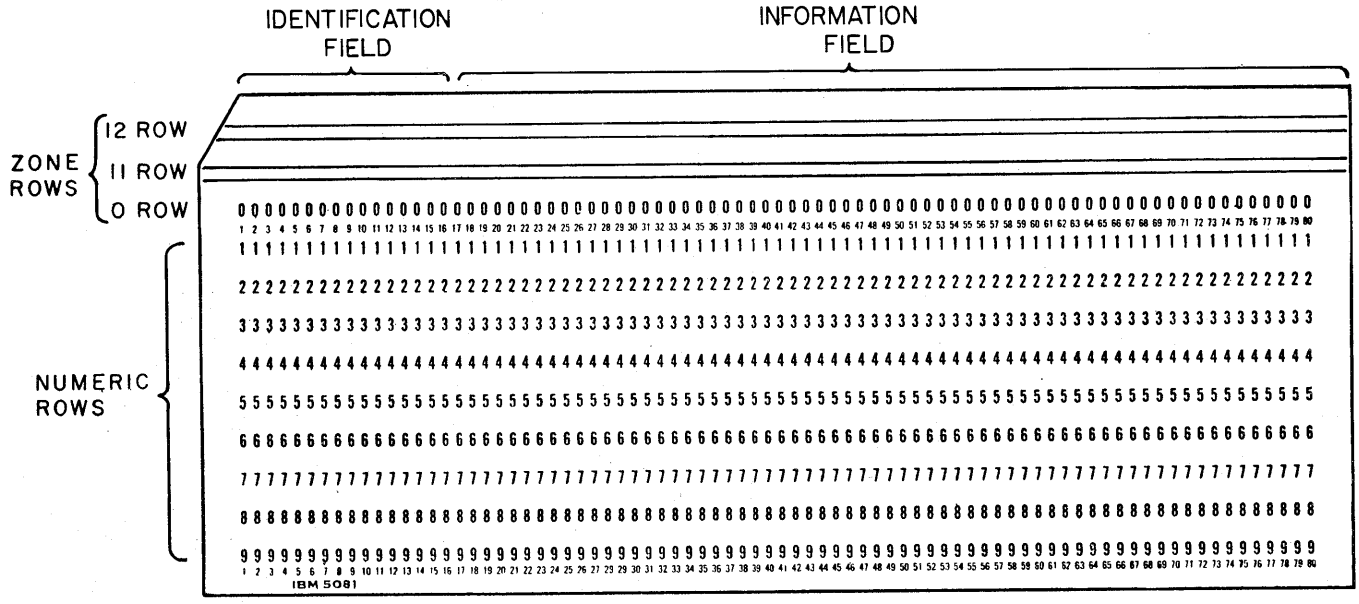


Figure 4–4. IBM Card Showing Hollerith Code Zones and Field Division

can be stored in column 10 of the card by punching the 6-row of column 10.

Alphabetical characters are stored on an IBM card with two punches per column; one punch in a zone row to indicate selection of one-third of the alphabet and one punch in a numeric row to indicate which letter within that third is represented. Thus, the letter A is indicated by a punch in the 12-row and the 1-row, the letter N by a punch in the 11-row and 5-row, and the letter T by a punch in the 0-row and 3-row. In addition to letters, 12 special characters are indicated by either a single punch in a zone row punch, or a zone row punch plus two numeric row punches. The coding for all of these symbols, known as the Hollerith code, is shown in table 4–11.

TABLE 4–11. HOLLERITH CODE FOR PUNCHED CARDS

		ZONE ROW		
NUMERIC ROW	NONE	12	11	0
None		+	–	0
1	1	A	J	/
2	2	B	K	S
3	3	C	L	T
4	4	D	M	U
5	5	E	N	V
6	6	F	O	W
7	7	G	P	X
8	8	H	Q	Y
9	9	I	R	Z
8 & 3	+	.	$	,
8 & 4	-	□	*	%

Using Hollerith code, up to 80 alpha-numeric characters may be represented on a single card with one character per column. The card machines associated with the Central Computer System can utilize only 64 columns on each card for storage of information introduced into or received from that system. Specifically, columns 17 through 80 of each card are used for this purpose. (See fig. 4–4.) Columns 1 through 16, while not normally available to the Central Computer System, are used for card identification which allows processing of punched cards by machines independently of the Central Computer System. In general, the information field of a card is further subdivided in accordance with the type of information presented on it. The subdivisions of the information field are discussed in connection with each of the three basic card forms. These are:

a. Instruction card

b. Binary card

c. Octonary card

3.2.1.1 Instruction Card

The card form on which most programs are prepared for initial insertion into the Central Computer System is the instruction card. Each instruction card may contain one instruction in alpha-numeric form plus comments to clarify the function of that instruction within a program. The location field on the card specifies the storage location of the instruction. The instruction field contains the operation code in mnemonic code form and the index indicator and auxiliary bits in octonary form. The address field specifies the contents of the address half of the instruction. The identification field contains program identification and the preparation date. (See fig. 4–5.)

It should be noted that a program presented on instruction cards cannot be executed directly by the Central Computer System. The program must be translated from its alpha-numeric form into binary form before it can be executed.

3.2.1.2 Binary Card

The binary card utilizes the binary nature of punched cards in storing information. The identification field, whose contents are identical with the identification field of the instruction card, is the only field on a binary card containing information in Hollerith code. All information on the information field of the binary card is in binary form. (See fig. 4–6.)

Since there are 64 columns within the information field of the binary card, each row contains two Central Computer System words. To distinguish between the two words in a single row, the word in columns 17 through 48 is called the left word while the word in columns 49 through 80 is called the right word. Another means of differentiating involves describing the former as the word in 9-row left and the latter as the word in 9-row right, where 9 specifies the row in which the two words are found.

Although the total capacity of the information field on a binary card is 24 binary words, usually no more than 22 instruction words are placed on one card. The remaining space is taken up by control information which usually requires one full row within the information field. In certain applications of binary cards, another full row is used for control information, thus restricting the capacity of the card to only 20 words.

In either case, program storage on binary cards offers two advantages over program storage on instruction cards. These are (1) binary cards contain more words per card than instruction cards, and (2) the words appear in a form which allows their direct execution. It is for these reasons that an assembly program (one which translates instructions from the form in which they are presented on instruction cards) produces a binary deck as its output.

3.2.1.3 Octonary Card

The third type of card used with the card machines of the Central Computer System is known as an octonary card. The octonary card has an intermediate value in capacity and usefulness between the instruction card and the binary card. Like the instruction card, program words inserted on octonary cards, must be translated by the Central Computer System before they can be executed; like the binary card, the octonary card carries more than one word. In general, octonary cards are used for insertion of short programs which can be prepared conveniently in octonary form. They are also used for insertion of patch routines, short sequences of instructions to be added to a program being assembled from instruction cards.

The octonary card contains an identification field whose contents are identical with those of binary or instruction cards. The information field of the octonary card is divided into several subfields. Four of these subfields can each contain one full computer word in

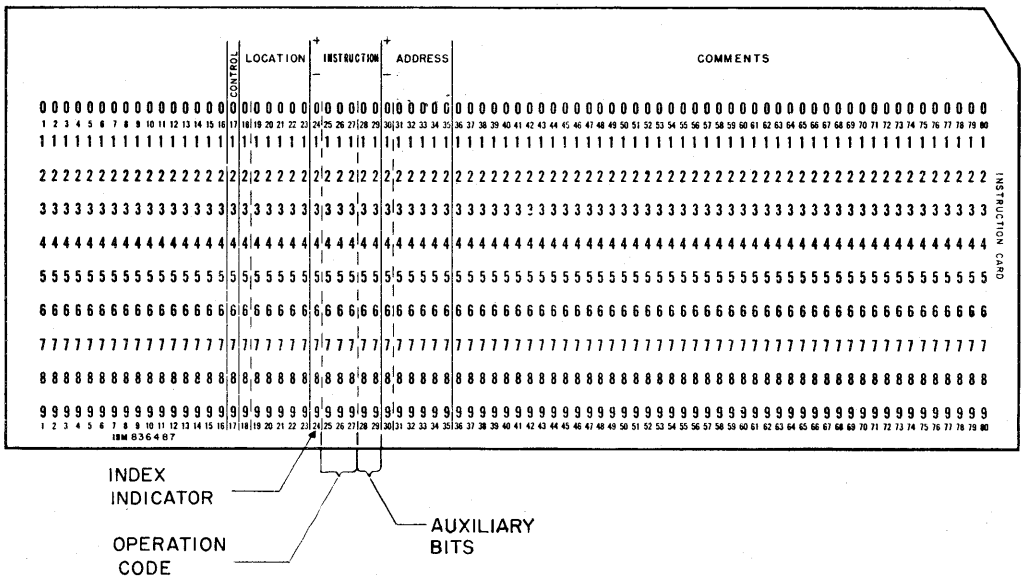


Figure 4–5. Instruction Card

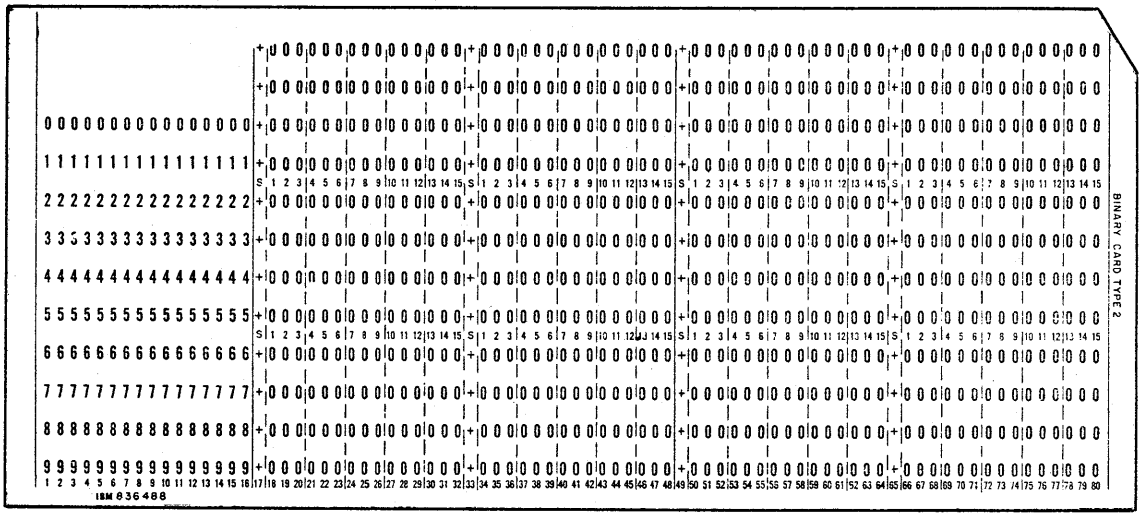
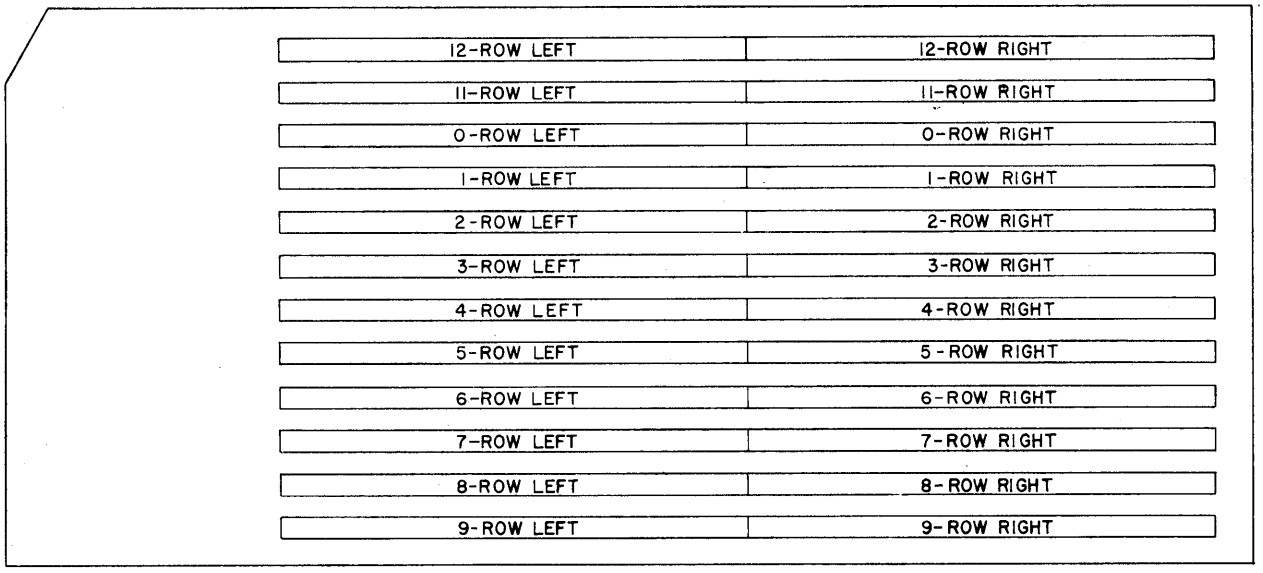


Figure 4–6. Binary Card



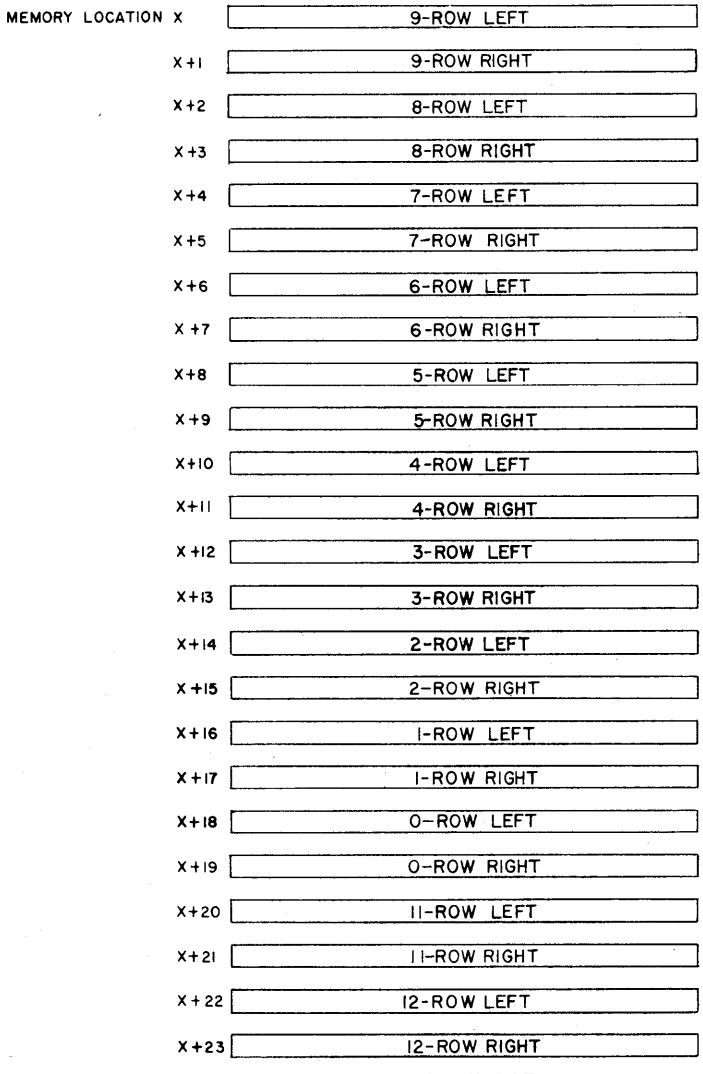


Figure 4–7. Relation of Card Image to IBM Card

octonary form. The first eight columns within the information field of the octonary card contain control information of the type contained in the reserved row of a binary card and contain the assigned storage location for the first word on the card. This location in absolute or symbolic form is identical in function to the location field on an instruction card. Only the location for the first word on the card is specified; the remaining three words are assigned locations in sequential order from the specified location.

3.2.1.4 Card Image

When the contents of an IBM card are transferred into the Central Computer System, they appear in a form known as a card image. Figure 4–7 shows the relationship between the words on a card and the words in a card image. In effect, the card image in memory contains the words from the card in the order 9-row through 12-row. The word from the left half of each row precedes the word from the right half of the modes of operation of the card machines and the Central Computer System; i.e., the card machines can handle 64 bits at a time while the Central Computer System can handle only 32 bits at a time. Since the card image can contain all the information from a card in a known pattern, it is possible to program the Central Computer System to interpret this information just as if the information were still in the pattern present on the card itself.

3.2.2 Line Printing

The line printer prints alphabetic and numeric characters when supplied with information from the Central Computer System. The information is stored in core memory in card image patterns. The card images are coded by using the Hollerith code. The Central Computer must supply a card image (64 columns) to print 64 characters on the printer. The printer prints a line of characters at a time by simultaneously positioning all of the 120 type wheels. The type wheels which are not supplied with information are internally wired to print out blank spaces. A print wheel is shown in figure 4–8. Notice that the numeric code used corresponds with the first nine rows on a card. The last three rows (0, 11, 12) are referred to as zone rows. By placing a 1 bit in the numeric row and a 1 bit in the zone row any character on the type wheel can be printed out.

3.2.3 Magnetic Tape

Information is stored on magnetic tape in the form of small magnetized spots. A computer word is stored on the tape in groups of five 7-bit characters and one 4-bit character. The bits of a computer word are always positioned in the same place in the group (fig. 4–9). Only six of the seven bits in a 7-bit character and only

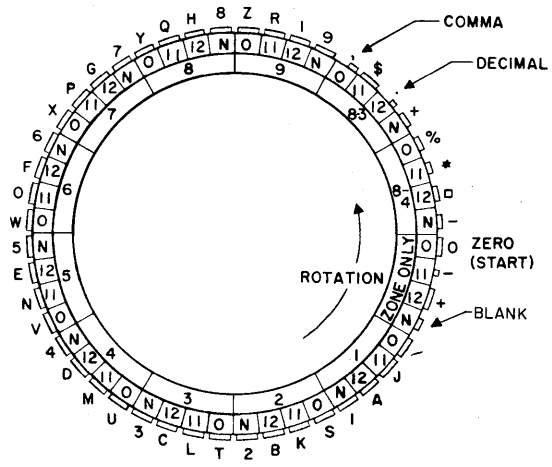


Figure 4–8. Type Wheel, Pictorial Diagram

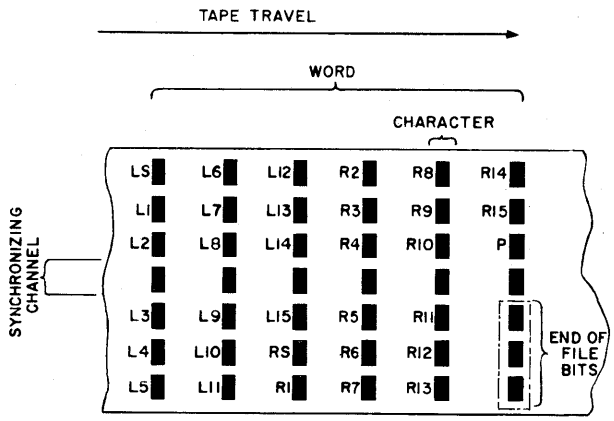


Figure 4–9. Tape Word Bit Positions

three of the four bits in a 4-bit character contain information. The remaining bit down the center of the tape is used for synchronizing the tape drive to ensure correct tape reading or backspacing.

A group of characters is called a word; a word or many consecutively written words is called a record; and information is transferred to or from the tape one record at a time. The record can contain any number of words up to the physical limits of the tape. A group of consecutively written information records on a tape is called a file. A file thus is defined as the physical portion of the tape that is usable for storage. The tape can contain more than one file (this again is determined by the programmer). The three bits in the first character of figure 4–9 (enclosed by dashed lines) can be written at any point, and these bits determine the physical size of a file. The normal operation of the tape is to program only one file; hence, the end-of-file (EOF) record would occur only once.

There are two reflective markings on the tape which are sensed by a photomultiplier tube. One mark, the load point (LP), indicates when the tape is in position to be written on or read from; the other mark is placed approximately 14 feet from the end of the tape and indicates the end of tape (EOT). If an operation is in progress (reading or writing), it will continue over the end-of-tape mark, but no new operations will occur after the mark is sensed.

During a rewind operation the load point is the control that stops the tape drive in a position to start reading or writing.

3.3 PROGRAMMING TECHNIQUES

3.3.1 General

The card machines and tapes associated with the Central Computer System of the AN/FSQ-7 and AN/FSQ-8 are programmed in the conventional manner. All units to be involved in an IO operation with the Central Computer System must receive the basic sequence of *SEL, LDC,* and *RDS* or *WRT* instructions. In addition, it is possible to check the status of each unit selected to see if it is ready to participate in an IO operation. The checking of each unit is performed in basically the same manner; however, there is some variation among the machines as to what constitutes a ready condition. For this reason, the check for the readiness of each machine will be discussed separately.

3.3.2 Card Reader

3.3.2.1 Description

The 713 Card Reader, shown in figure 4–10, allows the insertion of information from punched cards directly into the Central Computer System. The card reader reads a card a row at a time within a cycle of 400 ms, with a maximum reading rate of 150 cards per minute. Only columns 17 through 80 of each row are read, allowing the storage of the information from one row within two Central Computer System words. The reading of a card 9-row first produces a card image in core memory. Approximately 15 ms elapse between the reading of successive rows on a card. The remaining time within the card reader cycle is used to move the card into and out of the reading section of the card reader.

3.3.2.2 Selection

The card reader is selected for reading only by a *SEL (01)* instruction. As a rule, it is desirable to execute this instruction before an *LDC* instruction is given, so that the card reader will have sufficient time to send the ready signal to the Central Computer System before the check for readiness takes place. When the *SEL (01)* instruction is given, all other IO devices are deselected and remain so, until another *SEL* or *SDR* instruction is given.

3.3.2.3 Sense for Card Reader Not-Ready

The card reader can be checked to see if it is ready for an IO operation by execution of a *BSN (11)* instruction. The term "card reader ready" means that the card reader is capable of transferring information to core memory immediately upon receipt of an *RDS* instruction. To be in a ready state, the following conditions must be satisfied:

a. Card reader must be under computer control.
b. Card must be ahead of the read brushes.
c. Card stacker must not be full.
d. Power must be on.
e. Fuses must be intact.

If all of the above conditions are not present, the branching condition for the *BSN (11)* will be satisfied. However, a *BSN (11)* instruction should not immediately follow a *SEL (01)* instruction, since this will result in the status of the card reader being checked before it has time to indicate whether it is ready or not. For this reason, the desired sequence of instructions when programming the card reader is:

a. *SEL (01)*
b. *LDC* (with desired address)
c. *BSN (11)* (to desired location)

Execution of the *LDC* instruction in between the *SEL (01)* and *BSN (11)* instructions will provide sufficient time for the card reader to indicate its status.

If the *BSN (11)* instruction is not used when programming the card reader, and the card reader is not ready, execution of the *RDS* instruction will cause an IO hang-up since the IO interlock will be set and cannot be cleared.

3.3.2.4 Reading

When an *RDS* instruction is given to the card reader, it must be remembered that usually only 24_{10} binary words are contained in one card; therefore, execution of an *RDS* 30_8 instruction will cause the contents of one card to be completely read by the card reader and transferred to core memory. If the *RDS* instruction does not specify that some multiple of 30_8 words be read, only as many words as are specified will be transferred to core memory, but an entire card will always be read. Thus, if we executed an *RDS* 45_8 instruction, only 45_8 binary words would be transferred, but two entire cards would pass through the card reader. Also, if the *RDS* 0_8 instruction is given, one card will pass through the reader without having any words read from it.

3.3.2.5 Program Example

A typical program utilizing the card reader to obtain information is given in table 4–12.

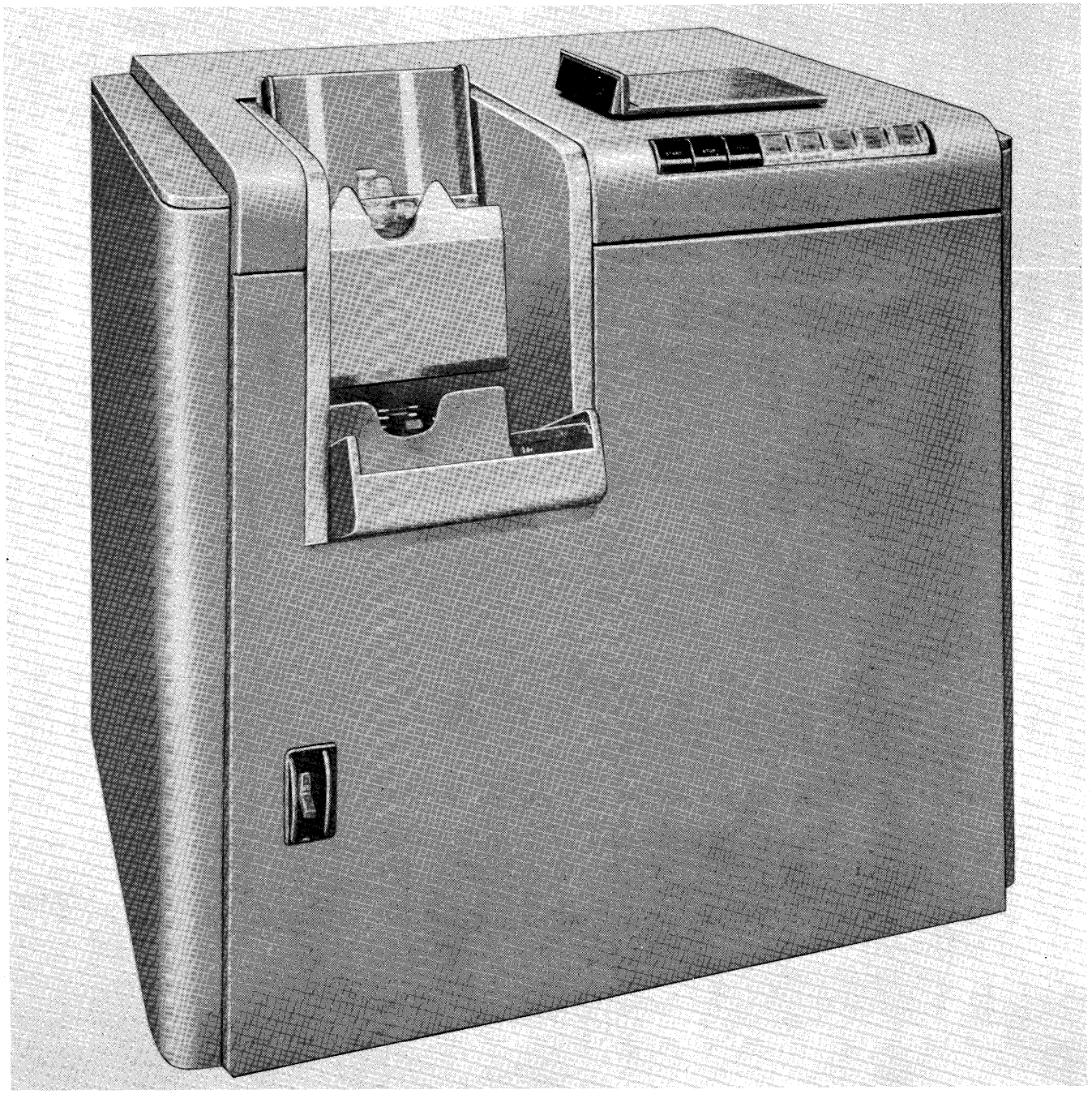


Figure 4–10. Card Reader, Type 713

After the card reader is selected, and the starting address in core memory has been loaded into the IO address counter, we sense the status of the card reader. If the reader is not ready for use, the *BSN (11)* will branch to itself, and the program will continue to loop at this step until the card reader becomes ready. Then, the *RDS* instruction will be executed, and the program will halt after all 300_8 words (10 binary cards) have been read into core memory.

3.3.3 Card Punch

3.3.3.1 Description

The 723 Card Punch, shown in figure 4–11, allows the Central Computer System to present processed information in punched card form. The card punch has an operating cycle of 600-μsec duration, allowing a maximum punching rate of 100 cards per minute. Since the card punch normally punches only columns 17 through 80 of each row, the Central Computer System must

TABLE 4–12. CARD READER TRANSFER ROUTINE

LOCATION	OPERATION	ADDRESS
0.00325	*SEL (01)*	–
0.00326	*LDC*	0.27500
0.00327	*BSN (11)*	0.00327
0.00330	*RDS*	0.00300
0.00331	*HLT*	–

supply two words for each row. The order in which the card punch handles a card, a row at a time and the 9-row first, requires the preparation of a card image in core memory in order to have punched information appear correctly on a card. Approximately 42.8 ms elapse between the punching of successive rows on a card. The remaining time within the card punch cycle is utilized in moving the card into and out of the punching section of the card punch.

Unpunched cards of the desired type are placed in the card punch hopper, face down with the 9-row to the right. When the punching of a card is called for by the Central Computer System program, the card is moved, bottom edge first, between the punches and the dies. The card is punched in a particular column of the row under the punches if the words from the Central Computer System contain a 1 in the bit position corresponding to that column.

After the card passes through the punching station, it goes to a reading station. The card punch can, under program control, connect the punch brushes, reading columns 1 through 16, to the punch magnets over the corresponding columns. With this provision, the information in columns 1 through 16 of a newly punched card can be duplicated by gang-punching on the next card passing through the punch station. A programming provision also exists for switching the inputs for the punch magnets over columns 17 through 32 to the punch magnets over columns 1 through 16. Thus, the Central Computer System can place information in the identification field of one card and then have this information gang-punched on each succeeding card. After a card has been punched and read by the punch brushes, it is placed in the stacker from which it may be removed for storage or for reinsertion into the Central Computer System through the card reader.

3.3.3.2 Selection

The card punch is selected for a writing operation by the execution of a *SEL (02)* instruction. As with the card reader, the *SEL (02)* instruction should be the first instruction given in the sequence which will initiate an IO operation using the card punch. When the *SEL (02)* instruction is given, all other IO devices are deselected and remain deselected until another SEL or SDR instruction is given.

3.3.3.3 Sense for Card Punch Not-Ready

Once the card punch is selected, it is usually desirable to see if it is ready for an IO operation by executing a *BSN (11)* instruction. This instruction will indicate whether the card punch is electrically and mechanically capable of participating in an IO operation. The function of the *BSN (11)* instruction is to check the existence of the following conditions in the card punch:

a. Card must be ahead of the punching station.
b. Punch must be under computer control.
c. Stacker must not be full.
d. Power must be on.
e. Fuses must be intact.

If all of the above conditions are not present at the time a *BSN (11)* instruction is executed, the branching condition will be satisfied, and program control will be transferred to the location specified by the right half-portion of the *BSN (11)* instruction. As with the card reader, a *BSN (11)* should not directly follow the *SEL (02)* instruction, so that the status check will not take place before the card punch has time to indicate its readiness. Therefore, the desired sequence for programming the card punch is as follows:

a. *SEL (02)*
b. *LDC* (with desired address)
c. *BSN (11)* - (to desired location)

If the *BSN (11)* instruction is not used prior to initiating an IO operation, executing a *WRT* instruction (which sets the IO interlock) may result in an IO hang-up. If the card punch is not ready, the IO interlock cannot become cleared.

3.3.3.4 Writing

Writing on the card punch is initiated by a *WRT* instruction just as with all IO devices. Normally, only columns 17 through 80 of the card are punched, enabling the card punch to handle two computer words at a time. Once the card punch has been started into its cycle, it will completely punch a card, or as many words as are specified by the *WRT* instruction. If less than 30_8 words are punched, the card will be advanced after the punching operation is completed. Thus, it is not possible to program more than one IO operation on the card punch and have the results punched into the same card.

In addition to the normal punching operations that can be performed on the card punch, it is also possible to gang-punch, or to duplicate the same information in more than one card. We use gang-punching to place the identity of a programmed deck and various other

information in the identification field of a card (columns 1–16). Since normal operation of the card punch places information in columns 17-80 only, it is necessary to program the card punch to place information in columns 1-16. This is accomplished by the use of a *PER (73)* instruction. When a *PER (73)* is executed immediately after a *WRT* instruction, the information which would normally be punched in columns 17-32 of the card is, instead, punched into columns 1-16.

Once the identification field of a card has been punched by the execution of a *PER (73)* instruction, some provision must be made to duplicate this same information in every desired card. This is accomplished by the execution of a *PER (74)* instruction, which causes the punch to read the information in the identification field of the card just punched and to transfer this information to the punches controlling punching in columns 1-16. Thus, to gang-punch information in the iden-

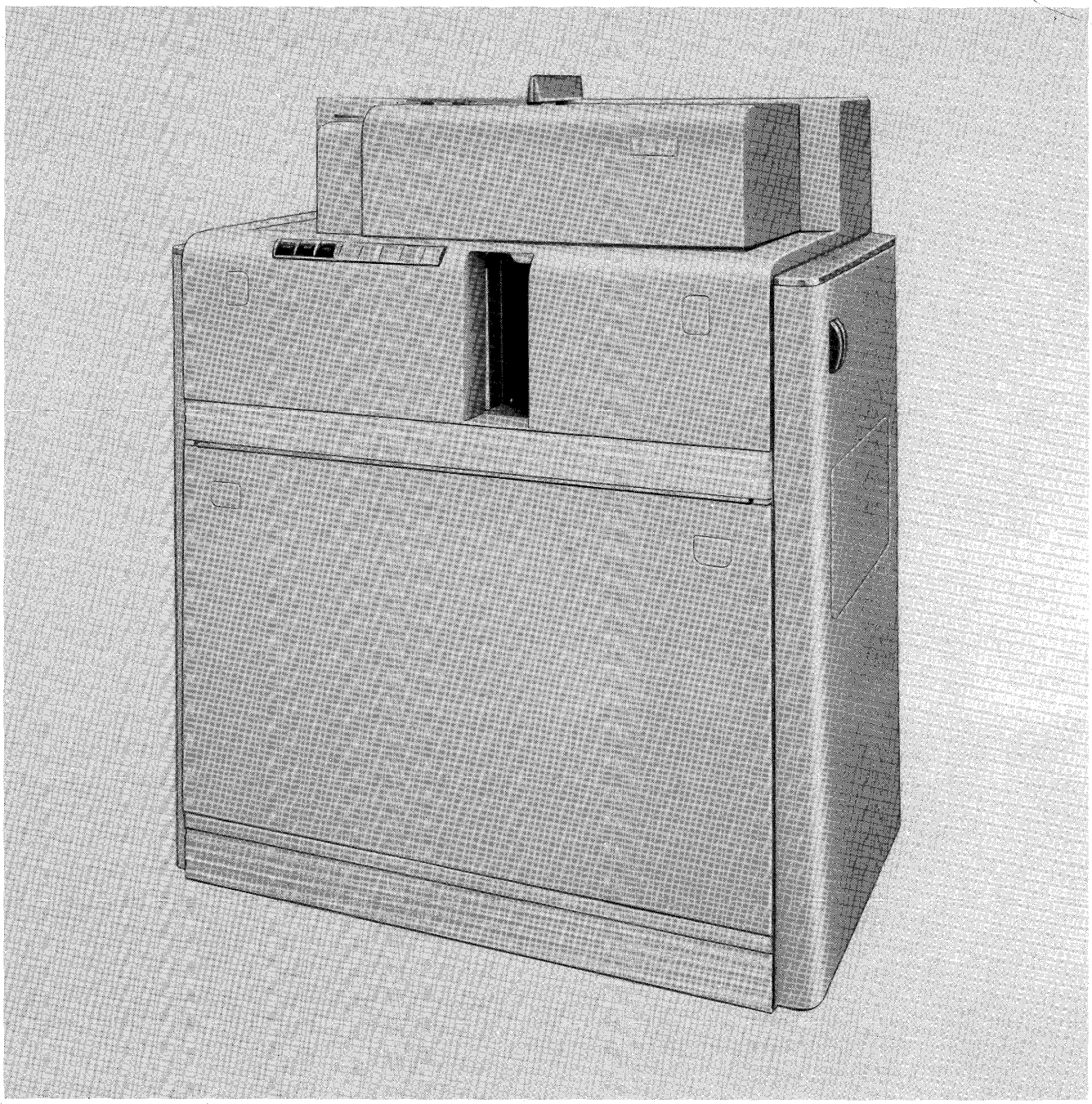


Figure 4–11. Card Punch, Type 723

tification fields of a card deck, it is necessary first to have a card image in memory, a *PER (73)* instruction to place the information in the first card, and a *PER (74)* instruction to initiate gang-punching.

3.3.3.5 Program Examples

An example of the use of the card punch in a normal IO operation is given in table 4–13.

TABLE 4–13. CARD PUNCH TRANSFER ROUTINE

LOCATION	OPERATION	ADDRESS
0.01504	*SEL (02)*	–
0.01505	*LDC*	0.01200
0.01506	*BSN (11)*	0.01506
0.01507	*WRT*	0.04540
0.01510	*HLT*	–

This program will punch the contents of memory locations 0.01200 through 0.05737 into a deck consisting of 100_{10} cards. The *BSN (11)* instruction will cause the program to loop continuously if the card punch is not ready, until the punch is made ready. Then, after all words have been punched, the program will halt.

As an example of gang-punching the identification fields of a deck of cards, assume that the desired card image is contained in memory location 0.00300 through 0.00327. We wish to gang-punch a deck of 50_{10} cards with the information this card image represents. The program to accomplish this is given in table 4–14. The first *SEL (02)* and *LDC* instructions will prepare the card punch to start its operations. If the *BSN (11)* at step 0.07352 shows the card punch to be ready, we initiate a writing operation of 30_8 words (or one complete card). Immediately after the *WRT* instruction is given, a *PER (73)* is executed which will cause punching to take place in columns 1-16. It should be remembered that punching will also take place in the other columns if information is present. However, the usual process is to have only the identification field of the first card punched. To do this, zeros must be present in the card image locations which would normally contain information. We pause while the first card is being punched to wait until the IO interlock is turned off. Another *LDC* instruction will give us the starting location of the information that is to be punched in columns 17-80. The *WRT* instruction will initiate punching of the second

TABLE 4–14. GANG-PUNCHING ROUTINE

LOCATION	OPERATION	ADDRESS
0.07350	*SEL (02)*	–
0.07351	*LDC*	0.00300
0.07352	*BSN (11)*	0.07352
0.07353	*WRT*	0.00030
0.07354	*PER (73)*	–
0.07355	*BSN (14)*	0.07355
0.07356	*LDC*	0.01000
0.07357	*WRT*	0.02260
0.07360	*PER (74)*	–
0.07361	*HLT*	–

card, while the *PER (74)* will start duplicating the identity field of the first card into the second card. Thus, we are gang-punching columns 1-16 of the 50 cards with the same information and punching columns 17-80 with the information contained in specified core memory locations.

To further illustrate the relation between the identity field of a card and the applicable core memory card image, assume that the core memory locations 0.00300 - 0.00330 in the above problem contained the values listed in table 4–15.

The punching from this card image which would result in the identity field is shown in figure 4–12. Memory location 0.00300 corresponds to the punching which would normally be placed in columns 17-32 but is, instead, placed in columns 1-16 by the *PER (73)* instruction. As previously explained, the columns of the card which are to be left blank must be provided for by zeros in the proper core memory locations. The half-words actually used in punching the identity field (starting with the 9-row) are in 0.00300, 0.00302, 0.00304, etc.

3.3.4 Line Printer

3.3.4.1 Description

The 718 Line Printer, shown in figure 4–13, provides the means for the Central Computer System to prepare information in printed alpha-numeric form. The

TABLE 4–15. CARD IMAGE FOR IDENTITY PUNCHING

LOCATION	LEFT HALF-WORD	RIGHT HALF-WORD
0.00300	0.20004	0.00000
0.00301	0.00000	0.00000
0.00302	0.10004	0.00000
0.00303	0.00000	0.00000
0.00304	0.04777	0.00000
0.00305	0.00000	0.00000
0.00306	0.02104	0.00000
0.00307	0.00000	0.00000
0.00310	0.01024	0.00000
0.00311	0.00000	0.00000
0.00312	1.77404	0.00000
0.00313	0.00000	0.00000
0.00314	0.20177	0.00000
0.00315	0.00000	0.00000
0.00316	0.10101	0.00000

TABLE 4–15. CARD IMAGE FOR IDENTITY PUNCHING (cont'd)

LOCATION	LEFT HALF-WORD	RIGHT HALF-WORD
0.00317	0.00000	0.00000
0.00320	0.04001	0.00000
0.00321	0.00000	0.00000
0.00322	0.02017	0.00000
0.00323	0.00000	0.00000
0.00324	0.01101	0.00000
0.00325	0.00000	0.00000
0.00326	1.77577	0.00000
0.00327	0.00000	0.00000

line printer prints a line at a time by the simultaneous positioning of as many of its 120 type wheels as is possible within one card cycle of 400 ms. The normal printing rate of the line printer is 150 lines per minute of 64 characters each. The Central Computer System must normally supply a full card image for each line of

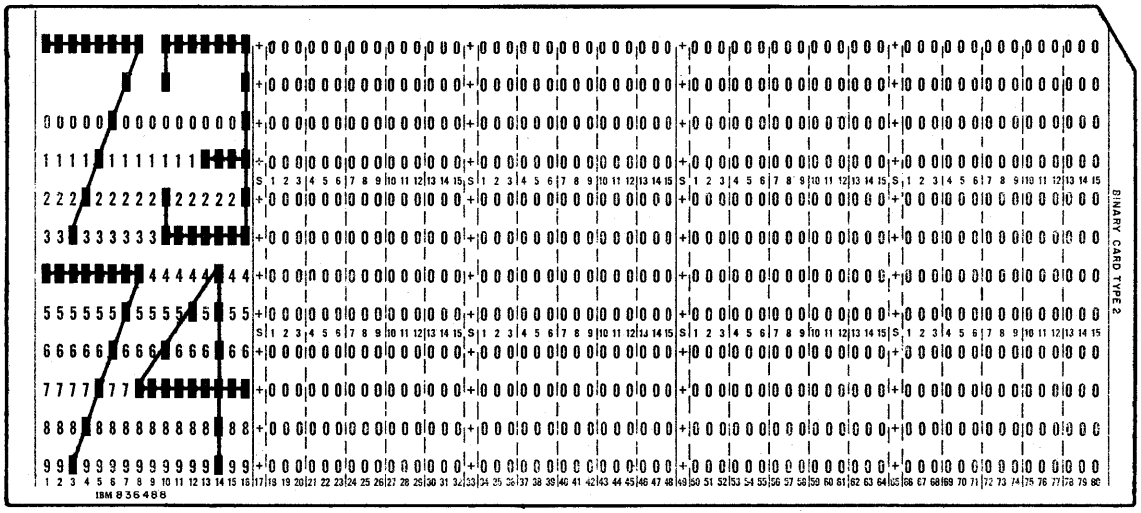


Figure 4–12. Card Punched in Identity Field

64 characters. As can be seen by referring to the type wheel shown in figure 4–8, all characters selectable by Hollerith code are arranged on the type wheel numeric sectors with the four characters within each sector selectable by the zone indication.

At the start of a print cycle, all 120 wheels are lined up with the character 0 in printing position. All the wheels begin rotating as soon as information is supplied to their positioning controls. The information applied to the positioning controls of each type wheel arrives in the order used in reading one column on a punched card; i.e., 9-row, 8-row, etc., and then 0-row, 11-row, and 12-row. When a bit is encountered in a numeric row position, the print wheel starts turning at a constant speed. The presence of a zone bit will slow the print wheel down so that printing of the character selected by the combination of bits can be obtained.

3.3.4.2 Selection

The line printer is selected for a writing operation only by the execution of a *SEL (03)* instruction. The

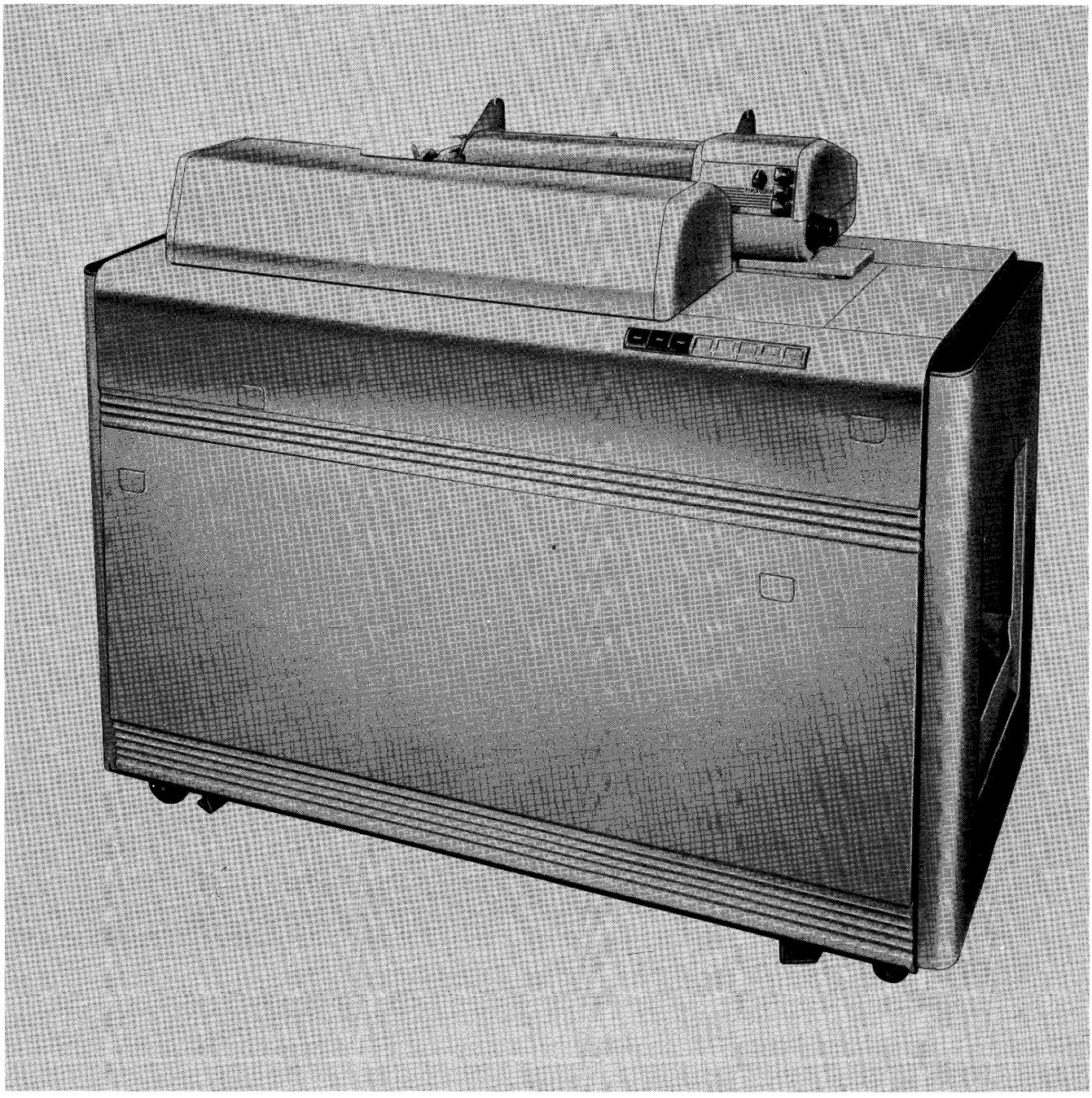


Figure 4–13. Line Printer, Type 718

sequence of instructions used to initiate a printing operation is basically the same as for the reader and punch; it is desirable to execute the *SEL (03)* instruction first so that the printer can indicate its status to the Central Computer System if a *BSN (11)* instruction is given. The *SEL (03)* instruction will deselect all other IO devices until another *SEL* or *SDR* instruction is given.

3.3.4.3 Sense for Line Printer Not-Ready

It is possible to check the status of the line printer once it has been selected by a *BSN (11)* instruction. To be in a ready condition, the following conditions must exist:

a. Forms to be printed on must be in the printer.
b. Printer must be under computer control.
c. Printer must not be in TEST status.
d. Printer carriage must not be stopped by a CARRIAGE STOP pushbutton on the printer.
e. Power must be on.
f. Fuses must be intact.
g. Line printer control panel must be in place and PR ON hub wired.

If any of the above conditions are not present, a branch of program control will occur. The *BSN (11)* instruction should not immediately follow a *SEL (03)* instruction, so that an erroneous branch will not take place. Thus, the desired sequence for operating the printer is:

a. *SEL (03)*
b. *LDC* (with desired address)
c. *BSN (11)* - (to desired location)

If we attempt to write with the printer when it is not ready, the IO interlock will become set and cannot be cleared.

3.3.4.4 Writing

Writing on the printer is initiated by the *WRT* instruction, and the printing takes place in accordance with the number of words supplied to it. If a *WRT* 0_8 instruction is executed, no information will be printed out, but the line printer will advance the paper form one line.

The appearance of the information printed on the line printer is controlled largely by the wiring of the printed control panel. For this reason, no definite rules can be formulated concerning the initiation of an IO operation other than to state that the execution of a *WRT* instruction will cause the printer to start a print cycle. In addition to the actions which normally occur during a print cycle, a provision is made to program pulses at certain exit hubs on the line printer control panel. These hubs are pulsed by a *PER (51) – (62)* instruction. For example, execution of a *PER (51)* instruction will cause a pulse to appear at exit hub one approximately seven milliseconds after execution of the instruction. The control panel may be wired from this hub to perform a variety of printer operations. The remaining print exit hubs that can be pulsed under program control operate in the same manner.

3.3.4.5 Program Example

A basic line printer operation to print the contents of memory locations 0.70000 through 0.70027 is given in table 4–16.

TABLE 4–16. LINE PRINTER ROUTINE

LOCATION	OPERATION	ADDRESS
0.00450	*SEL (03)*	–
0.00451	*LDC*	0.70000
0.00452	*BSN (11)*	0.00452
0.00453	*WRT*	0.00030
0.00454	*HLT*	–

There are two line printer entry hubs which provide impulses to a set of sense entry hubs on the line printer control panel in much the same manner as the IO exit hubs previously described. It is possible to sense these entry hubs by computer control and take specific courses of action, if the impulses are present at the hubs. These hubs are sensed by execution of a *BSN (31)* or *BSN (32)*, respectively. For example, if an impulse is present on entry hub 2 and we execute a *BSN (32)* instruction, the branching condition will be satisfied and a branch of program control will take place. It is not possible to list actions that will cause an impulse to appear at the sense entry hubs since this is controlled largely by the wiring of the line printer control panel.

3.3.5 Magnetic Tapes

3.3.5.1 Description

The 728 Magnetic Tape Drive units, shown in figure 4–14, which are used with the AN/FSQ-7 and AN/FSQ-8 provide long-term, slow-access information storage. Unlike the card reader, card punch, and line printer, the tape units can participate in both reading and writing operations with the Central Computer System. The method of recording on tape is nondestructive, so that once binary information is recorded it may be used indefinitely without re-recording.

The tape is fed through the reading and writing heads at a rate of 75 inches a second. With a normal tape length of approximately 2,400 feet, this means that one entire reel of tape can be written on or read from in about 6 minutes. The amount of information that can be recorded on the tape reel is approximately

the same as that which could be contained in 20,000 standard IBM 80-column punched cards. Thus, it can be seen that storage on magnetic tape provides not only faster access than do punched cards but also requires much less space to store an equivalent amount of information.

Three mechanical operations of the tape drive unit, selectable by the program of the Central Computer System, are:

a. Move tape forward (read or write)
b. Backspace
c. Rewind

The tape must be accelerated to the correct speed before either writing or reading can begin. A normal delay of 10 ms is allowed (with an additional 40-ms delay allowed if writing or reading begins from the load point). Reading each word requires 324 μsec with a 150-μsec delay after the last word of a record. Writing requires an extra 150 μsec, totaling 300 μsec after the last word. This time, which includes the time taken by the tape in coasting to a stop, produces the blank space denoting the end of the record.

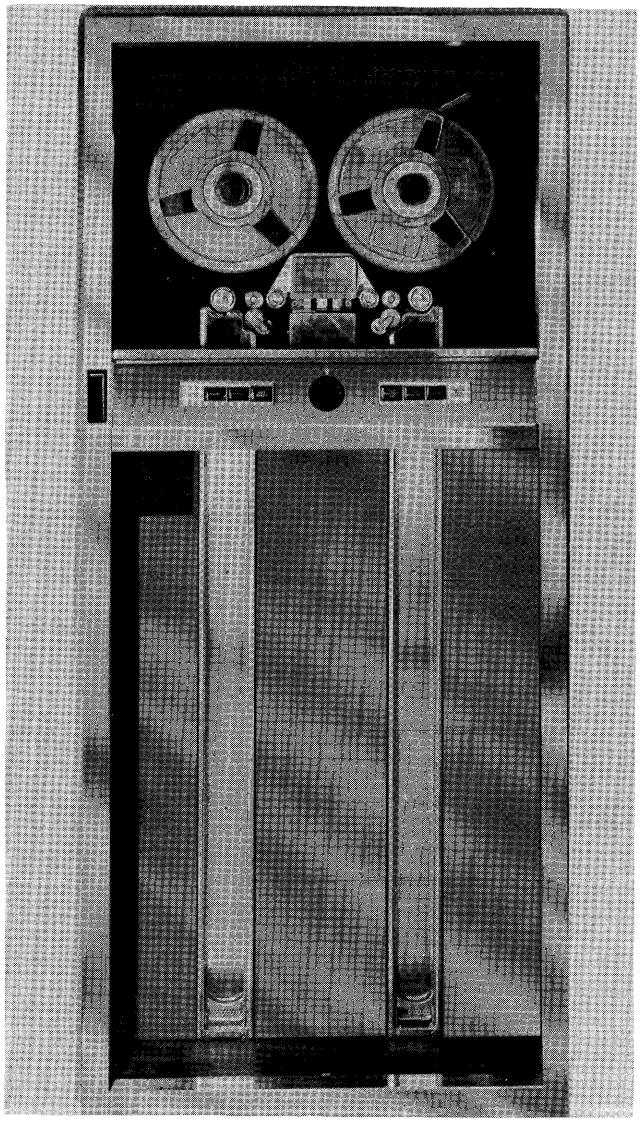


Figure 4–14. Tape Drive Unit, Type 728

A backspace operation moves the tape back over one record, stopping the tape at the beginning of that record. If the tape is in write status when the backspace is given, the tape will move forward for 6 ms and will take another 25 ms to shift into reverse before moving backward. The record is read backward but is not sent to the Central Computer System. Only the synchronizing track is read to sense the beginning of the record. When the beginning of the record is reached, 1 ms elapses before the stop-tape signal is generated, and 2 ms elapse before the mechanism shifts into forward. Twenty-five ms are required to complete the shift.

The rewind operation causes the tape to rewind onto the file reel until the photoelectric cell senses the load point. The tape stops and then shifts to forward status.

3.3.5.2 Selection

There are provisions for as many as six tape drive units with each Central Computer System. These tape drives are numbered 1 – 6 and are selected by a *SEL (11) – (16)*, respectively. When a tape drive unit is selected for an IO operation, all other IO devices are deselected and remain so until another *SEL* or *SDR* instruction is given. As in the case of the other machines, it is advisable to execute the *SEL (11) – (16)* instruction prior to the LDC instruction so that the circuit indicating the status of the selected tape drive may have time to stabilize.

3.3.5.3 Sense for Tape Unit Not-Ready

We can sense to see if a given tape unit is ready to take place in an IO operation by the execution of a *BSN (11)* instruction. To be ready for an IO operation the following conditions must be present on the selected drive unit:

a. Unit must be under computer control.
b. Transport mechanism must be loaded.
c. Tape must not be broken.
d. Light for photoelectric sensing of the reflective spots must be on.
e. Power must be on.
f. All fuses must be intact.
g. Door of the unit must be closed.

If any of the above conditions is not present, the branching conditions for execution of the *BSN (11)* instruction will be satisfied, and a branch of program control will take place.

3.3.5.4 Sense for Tape Unit Not-Prepared

There is another class of conditions which are necessary foı the selected tape drive unit to take part in an IO operation besides those listed above. These conditions are as follows:

a. The tape drive unit is not in rewind status.

b. The tape drive unit has not sensed end-of-tape or end-of-file marks or written end of file.

If either of these two conditions is not present (that is, the tape drive is in rewind, or the end-of-tape – end-of-file marks are sensed), the tape unit is considered not prepared. The existence of these two conditions may be checked with a *BSN (10)* instruction. It should be emphasized that a selected tape drive unit may be ready and still not be prepared. Therefore, *BSN (11)* is not always sufficient to determine if the tape may take part in an IO operation without causing hang-up. Application of these two instructions will be discussed further in the paragraph dealing with tape programming.

3.3.5.5 Reading

The reading process on a tape drive unit is initiated by an *RDS* instruction, just as with any IO unit. However, the action of the *RDS* instruction is slightly different from that of other IO devices. We can specify the number of words we wish to read in the right half-portion of the instruction, but, regardless of this, one full record is always read from the tape unit by each *RDS* instruction. For example, if we desire to read 200_8 words, and the record we are about to read contains 300_8 words, the reading operation will continue for the entire record, even though only 200_8 words are actually transferred to core memory. Even though no information is being transferred to core memory, the IO interlock remains set until the end-of-record (EOR) gap is reached. Then, the tape unit will generate a disconnect and will clear the IO interlock to stop the tape drive. In this case, stepping the IO word counter to 0 does not terminate the IO operation. On the other hand, if the IO word counter does not contain 0 after one complete record is read, the tape drive is still stopped, and the disconnect signal is transmitted to the Central Computer System indicating that the IO operation is over. For example, if we execute an *RDS* 4000_8 instruction, and the record to be read only contained 3500_8 words, the tape drive would stop after reading 3500_8 words. In order to read the remaining words, another *RDS* instruction must be given. An *RDS* 0_8 instruction will cause the tape drive to read the next record without transferring any words to core memory; the effect of an *RDS* 0_8 instruction is to skip the next record as far as the Central Computer System is concerned.

If a reading operation is in process and EOT is sensed, or the record being read is an EOF, the tape is determined to be not-in-file-area (NIFA), and the tape becomes not prepared. However, reading continues until the EOR gap is reached. No new operations may take place on this tape unit until the tape has become prepared. Methods for preparing the tape will be discussed in subsequent paragraphs.

In summary, the most important thing to remember when programming an RDS operation for a tape unit is that a separate *RDS* instruction must be given for every record that is on the tape.

3.3.5.6 Writing

Execution of the *WRT* instruction sets the IO interlock, places the tape drive in the write status, and starts the tape drive. The words are placed on the tape in consecutive order until all the words that are specified by the right half-portion of the *WRT* instruction have been written. When the IO word counter goes to 0 the tape unit is pulsed to create an EOR gap.

If the EOT mark is sensed while writing a record, the tape unit will become not-prepared, but writing will continue until the record is completely written. However, no more operations may take place with that unit until the tape becomes prepared.

Unlike a reading operation, we cannot specify a *WRT* 0_8 instruction when using a tape drive. This instruction will cause the IO interlock to become set and remain set, because the tape drive unit will not be able to generate a disconnect pulse. This will cause the computer to be in an IO hang-up. Therefore, a *WRT* 0_8 instruction should never be given with a tape unit.

All writing operations automatically erase previous information, starting from the point where writing began. For this reason, an electromechanical device (called a file protection ring) is provided for use with a tape drive unit to prevent erroneous writing operations on a tape, and, thus, destroy previously recorded information. When the file protection ring is in place on the tape reel, writing may take place, but when it is removed, writing and erasing will not take place. If a *WRT* instruction is programmed on a tape unit that is being protected, the *WRT* instruction acts as a disconnect signal to the Central Computer System. File protection does not prevent normal reading operations however.

3.3.5.7 Tape Instructions

Because of the flexibility of the tapes, there are several instructions which will cause actions to take place on the specified tape drive unit. Some of these actions are dependent on the IO interlock, and some may be executed at any time. The cases which will call for use of these instructions may vary from time to time; therefore, a paragraph will be devoted to each instruction, outlining the most general applications of that instruction.

The desired instruction sequence for performing a reading or writing operation with a tape unit is basically the same as for all IO devices. As stated previously, two classes of conditions exist which will prevent the tapes from taking part in a reading or writing operation. The first condition, "tapes not ready," should always be sensed with a *BSN (11)* instruction before an IO operation is initiated. The desirability of sensing for "tapes not prepared" depends on the previous action that has taken place on the tape unit. If we are selecting the tape unit for the first time, it is usually desirable to check for both conditions. A program to check these conditions is given in table 4–17.

TABLE 4–17. TAPE UNIT READINESS CHECK

LOCATION	OPERATION	ADDRESS
0.01000	*SEL (11)*	–
0.01001	*LDC*	0.42500
0.01002	*BSN (11)*	0.01002
0.01003	*BSN (10)*	0.01005
0.01004	*WRT*	0.02000
0.01005	*SEL (12)*	–

In this case, after checking to see if the tape unit was not ready, we also sense to see if it is not prepared. If the not-prepared condition exists, this means that the tape is past the usable writing area; therefore, a new tape unit should be selected. Thus, a branch to the *SEL (12)* instruction is provided. Since we had just selected this tape for an IO operation, it is not possible that rewinding could be causing the not-prepared condition. It should be stressed that this may not always be the case; sometimes a different procedure may be necessary.

As a general rule, the programmer will know what has taken place on the specified tape drive unit before a subsequent IO operation is attempted. If we knew positively that we were writing on a blank tape, the *BSN (10)* instruction would not be necessary. Since this is not the case, a *BSN (10)* instruction is desirable.

Once we have determined that a tape unit is not prepared, two instructions may be used to make the unit prepared, if we do not wish to select another tape unit. The first of these is the *Set Prepared (PER 67)* instruction. Execution of this instruction does not set the IO interlock or require that the IO interlock be cleared before the *PER (67)* is given. The *PER (67)* instruction clears the not-prepared condition and permits the tape to proceed in a forward direction. However, it should be noted that if a *PER (67)* instruction is executed when the EOT caused the not-prepared condition, a subsequent *RDS* instruction will cause the tape to wind onto one reel.

The second instruction which can be used to make the tapes prepared is a *PER (70)* instruction which causes the selected tape drive unit to backspace one record at reading speed. The IO interlock becomes set during this operation, but no information is transferred to core memory. Execution of a *PER (70)* instruction will thus move the tape back into the usable file area from where an RDS or a WRT operation may be initiated.

When the tape drive is determined to have read or written all of the usable area, or as much of the area as desired, it is possible to rewind the tape reel to the LP by execution of a *PER (71)* instruction. The *PER (71)* instruction will make the tapes not-prepared during the time rewinding is actually taking place; the instruction also turns on the IO interlock. (It is not necessary for the IO interlock to be cleared prior to execution of this instruction.) It should be noted that the IO interlock is cleared approximately 40 to 80 ms after the rewind operation is started. This does not indicate that the rewind operation has been completed, but only that the selected tape drive unit began to rewind. The rewind operation is relatively fast compared to the reading or writing speed (approximately 500 inches per second); however, if the entire tape had to be rewound, it would still require about a minute to complete, a much longer time than that in which the IO interlock is set by execution of *PER (71)*.

If the selected tape drive unit was prepared (tape still in usable area) before execution of a *PER (71)* instruction, the tapes will be not-prepared only during the rewind condition. Then, they will become prepared again. On the other hand, if the tape had passed the EOT mark, or had read or written an EOF record, it would already be not-prepared. In this case, execution of a *PER (71)* does not make the tape prepared after the rewind operation has returned it to the LP.

One precaution should be taken when the *PER (71)* is to be used in a program. Since execution of this instruction does not require that the IO interlock be cleared prior to its execution, it is possible to initiate an IO operation with a tape drive unit, and then rewind the same tape. This may result in permanent tape damage; therefore, extreme care should be used when executing a *PER (71)* instruction.

When all the usable information that is to be recorded on tape has been written, we usually write an EOF record. This is not done by use of a *WRT* instruction, since normal writing on the tape will never place information into the three bit positions that indicate EOF. (Refer to fig. 4–9.) Instead, a *PER (72)* in-

struction is executed, which writes this special one-word record and also makes the tapes not-prepared. The *PER (72)* does not examine the IO interlock before execution; therefore, care should also be exercised when giving this instruction since its use at an improper time may cause a loss of information.

3.3.5.8 Tape Programming

Besides the reading and writing operations which have been discussed, many other operations using the instructions described above may take place with a specific tape unit or units. Certain sequences of instructions which are desirable when performing any of these operations are outlined below.

When rewinding one tape unit, it was stressed that we should be sure no other IO operation is taking place with that unit. This may be done by making a time check of the IO interlock and then starting the rewind. A program to accomplish this is given in table 4–18.

TABLE 4–18. TAPE REWIND PROGRAM

LOCATION	OPERATION	ADDRESS
0 00420	*SEL (13)*	–
0.00421	*LDC*	0.01000
0.00422	*BSN (11)*	0.00422
0.00423	*PER (71)*	–
0.00424	*BSN (14)*	0.00424
0.00425	*PER (67)*	–
0.00426	*BSN (10)*	0.00426
0.00427	*RDS*	0.03000

After the *SEL (13)* instruction has been executed, we know the IO interlock is off, so no further check is required to ensure we will not be interrupting any IO operation. The *BSN (11)* instruction will branch to itself until the tape becomes ready, at which time we will execute the *PER (71)* instruction. After that, we loop at step 0.00424 until the IO interlock goes off. This tells us that the rewind operation has started and that the *PER (67)* may be executed; the execution of this instruction will clear the condition indicating we were at EOF or EOT (if this condition exists). The following *BSN (10)* instruction will hold the program until the rewind operation is completed, at which time an IO operation may be initiated.

It is possible to rewind more than one tape unit at a time by executing the instructions in the sequence given in table 4–19.

TABLE 4–19. PROGRAM TO REWIND MORE THAN ONE TAPE

LOCATION	OPERATION	ADDRESS
0.01000	*SEL (11)*	–
0.01001	*BSN (11)*	0.01001
0.01002	*PER (71)*	–
0.01003	*BSN (14)*	0.01003
0.01004	*SEL (12)*	–
0.01005	*BSN (11)*	0.01005
0.01006	*PER (71)*	–
0.01007	*BSN (14)*	0.01007

This program will select tape unit one and sense to see if it is ready. Notice that the *BSN (11)* instruction immediately follows the *SEL* instruction, which is usually not advised. However, since no intervening instruction can be used to any advantage, and the *BSN (11)* branches to itself, no false indication of a not-ready condition will result. If the unit is ready, the rewind is started, and the program loops until the IO interlock is cleared, indicating that another tape unit may be selected. Then, the same process is followed as often as desired.

To properly backspace a tape unit, the suggested sequence is given in table 4–20.

TABLE 4–20. TAPE BACKSPACE ROUTINE

LOCATION	OPERATION	ADDRESS
0.14000	*SEL (16)*	–
0.14001	*LDC*	0.21205
0.14002	*BSN (11)*	0.14002
0.14003	*PER (70)*	–
0.14004	*RDS*	0.00370

An RDS instruction after a backspace operation is the only one which should be given to start an IO transfer. This is true because it is not advisable to try and write a record over an old record, even if the same numbers of words are involved. Changes in synchronization and mechanical differences may result in destroying some information on the next record. Records may be reliably written after a backspace operation only if the record backspaced over was the last record in a file.

When it is necessary to search for a record on a given tape, there are various ways that can be used. In our example, we will assume that the records on the tape are in numerical order, and that they are maintenance programs. We are searching for a particular maintenance program which may be identified by the first word of the record. When the correct maintenance program is located, we wish to start executing it. We will assume that the first instruction will be the second word in the record. A tape search routine which will select a given program according to its identity word is given in table 4–21. (The identity of the desired program is contained in the right half-word; the left half-word contains 0's.)

TABLE 4–21. TAPE RECORD LOCATION PROGRAM

LOCATION	OPERATION	ADDRESS
0.00350	*SEL (15)*	–
0.00351	*LDC*	0.01000
0.00352	*BSN (11)*	0.00352
0.00353	*RDS*	0.20000
0.00354	*BSN (14)*	0.00354
0.00355	*CAD*	0.01000
0.00356	*CDF*	0.00065
0.00357	*BPX*	0.01001
0.00360	*BFM*	0.00372
0.00371	*BPX*	0.00351
0.00372	*BSN (11)*	0.00372
0.00373	*PER (70)*	–
0.00374	*BSN (14)*	0.00374
0.00375	*PER (70)*	–
0.00376	*BSN (14)*	0.00376
0.00377	*BPX*	0.00351
0.00065	0.00000	Constant

This program makes the assumption that the tape was in a usable area before the program started and was not being rewound. After the tape has been selected and found ready, we read in one record. The value 20000_8 assures us that we are trying to read more words than will be found in one record. After the record is read in, we compare the first word with a constant (which is the identity of the program we want). If the comparison is successful, we branch to the second word from the record and start executing the program. If the accumulator contents are negative, this indicates the number was originally larger than the operand. This means we have obtained a record beyond the one we desire, so it is necessary to backspace until the desired record is reached. However, if the contents are positive we know that the accumulator contents were originally smaller than the constant; therefore, we must read in the next record.

Assuming that it is necessary to backspace to locate the record we want, we first sense to see if the unit is still ready. If it is, we execute a *PER (70)*, which will place us back at the beginning of the record we just read. Another *PER (70)* is necessary to move us to the next sequentially lower record. Then, we branch back to the *LDC* instruction and read the next record and compare.

CHAPTER 4
PROGRAMMING THE INPUT SYSTEM

4.1 DESCRIPTION

The Central Computer System of the AN/FSQ-7 and AN/FSQ-8 is capable of processing reports on aircraft flying within a given area in order to defend that area against air attack. Data on potential targets must be acquired, generally by radar, and introduced into the Central Computer System. This function is performed by the Input System. (The Input System of the AN/FSQ-8 handles only processed data reports and does not receive raw radar data as does the AN/FSQ-7. Therefore, the following discussion of the radar input elements will apply to the AN/FSQ-7 only.) The Input System is divided into three elements, each handling a different type of information. The long-range radar input (LRI) element receives data from long-range radar sets as well as from height-finder radars located near the long-range radars. The long-range radars are arranged to provide coverage of every point within the surveillance area. Those portions of the surveillance area which are not covered by long-range radars are covered by short-range radars. These gap-filler radars are so called because they fill gaps in the long-range radar coverage. Since data from gap-filler radars is different in form from that supplied by long-range radars, this data is received and processed by the gap-filler input (GFI) element.

The crosstell (XTL) input element of the AN/FSQ-7 receives data already processed by adjacent centrals. Crosstell messages generally describe aircraft moving out of the surveillance area of the transmitting central into the area covered by the receiving central. The XTL element of the AN/FSQ-8 processes messages relayed from AN/FSQ-7's in the division controlled by the AN/FSQ-8. In addition, the XTL element of the AN/FSQ-8 receives messages from other AN/FSQ-8's describing the overall air defense situation.

4.2 DATA FORMS

4.2.1 LRI Data Drum Word Layout

As previously mentioned, the AN/FSQ-8 is not concerned with raw radar reports; therefore, this discussion of LRI data words is applicable only to the AN/FSQ-7.

The processing of radar data information as it is received from the various radar sites is processed automatically by the LRI input element. No programming is required to arrange this information into the format necessary for presentation to the LRI drum fields. However, once the radar data is placed on the drum, all further processing is under control of the Central Computer System. For that reason, a description of the word layout of a radar message, as it is presented to the Central Computer System, is included below. Information processed by the LRI element is obtained from the LRI fields by the Central Computer System in the forms shown in figure 4–15, part A. A search radar report contains the following information:

a. Drum word 1
   1. Run length, L11 through L13.
   2. Time delay, R2 through R6.
   3. Message label, R7 through R11, identifying message as search radar IFF, IFF with SIF, or height reply.
   4. Site identity, R12 through R15, identifying location of data source.
   5. Odd-parity bit.

b. Drum word 2
   1. Range, L1 through L10.
   2. Azimuth, L11 through L15 (least significant) and R9 through R15 (most significant).
   3. Clock time, R1 through R6.
   4. Odd-parity bit.

Those bits indicated as blanks, or unused, are written as 0's on the drum field.

Information may be placed on the LRI drum fields which consists of an IFF (Identification Friend or Foe) report. This report is used to positively identify a radar return as being either a friendly or hostile aircraft. IFF reports are generated at the radar sites and, therefore, are transmitted along with actual radar returns. The only difference between a search radar report and an IFF report is in the first drum word. The message label indicates that the report is an IFF report with or without Selective Identification Feature (SIF). If a SIF response is included, it occupies bits L1 to L13. An IFF report without SIF contains all 0's in these bit positions. An IFF word layout, as it is placed on the LRI drum field, is shown in figure 4–15, part B. A height reply shown in figure 4–15, part C, contains the following information:

a. Drum word 1

1. Message label, R7 through R11, identifying report as a height reply.
2. Site identity, R12 through R15.
3. Odd-parity bit.

b. Drum word 2

1. Special reply, L1 through L3, containing a pre-arranged coded message.
2. Formation, L4 through L5, a coded indication of the predominant arrangement of the aircraft, reported as one target.
3. Separation, L6 through L7, a coded indication of the distance between the aircraft making up the target.
4. Number of aircraft, L8 through L10, an approximation of the number of aircraft.
5. New height, R9 through R15 and L11 (least significant bit).
6. Address, L12, indicating which of the height finders at the radar site is replying.
7. Request number, L13 through L15, allowing identification of the reply with the target on which a height report was requested.
8. Odd-parity bit.

Those bits labeled as blank, or not used, contain 0 when written onto the LRI fields.

4.2.2 GFI Data Drum Word Layout

This discussion of GFI data is applicable to the AN/FSQ-7 only. The form in which a GFI message appears on a GFI field is shown in figure 4–16. Each drum word contains the following information:

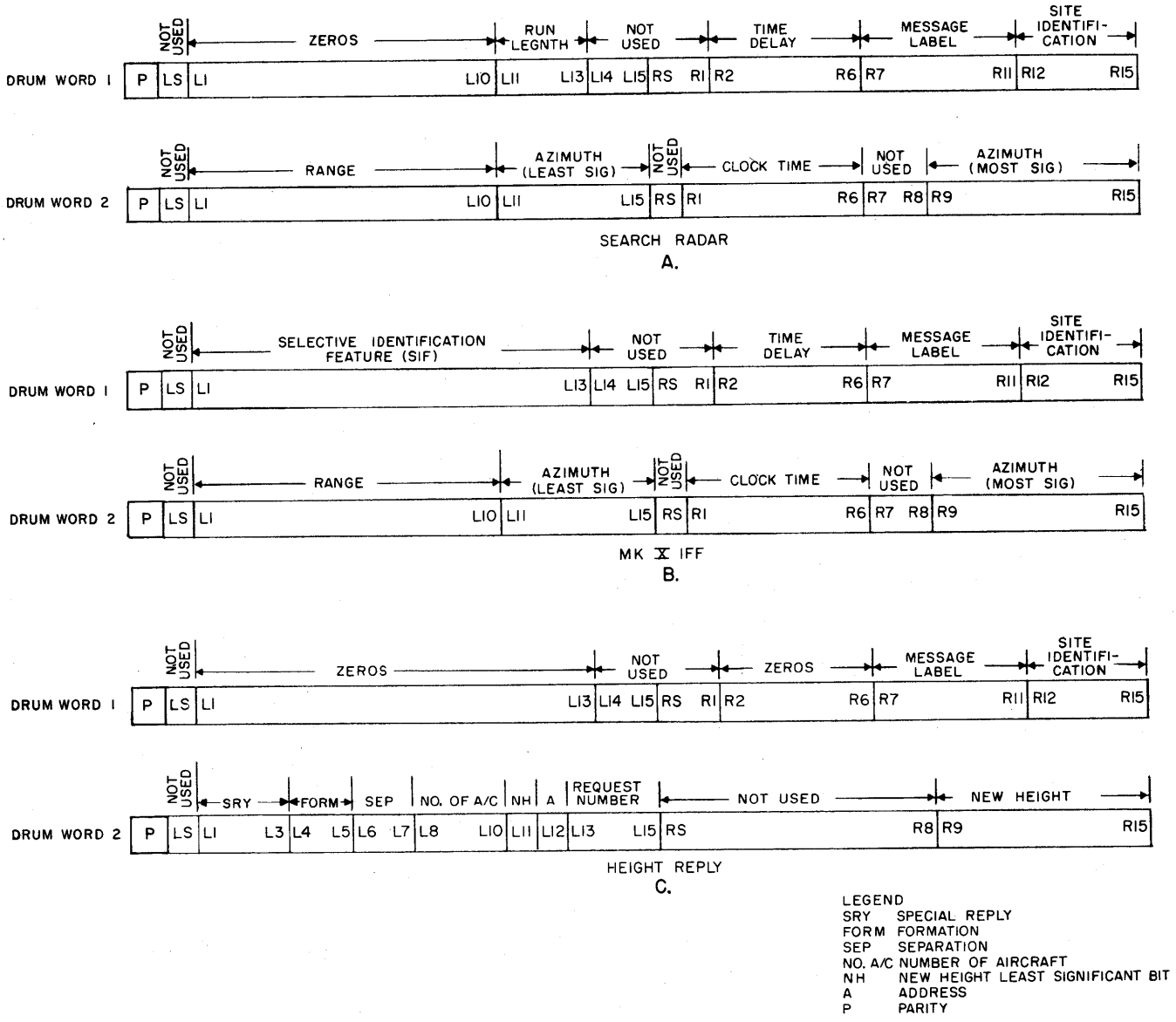


Figure 4–15. LRI Message Drum Field Layouts

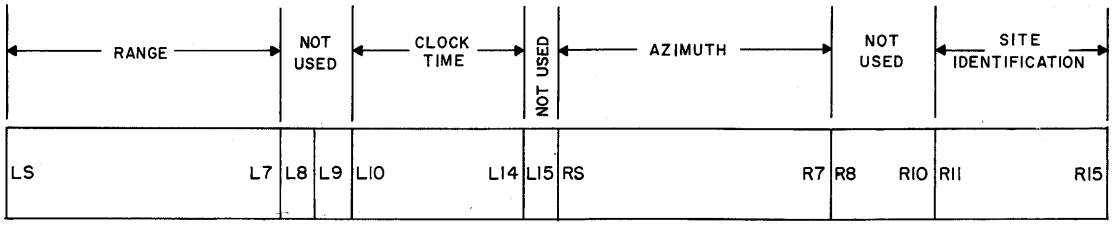


Figure 4–16. GFI Message Drum Field Layout

a. Range, LS through L7.
b. Clock time, L10 through L14, equivalent to clock time in an LRI message but added to a GFI message by the Drum System.
c. Azimuth, RS through R7.
d. Site identity, R11 through R15, identifying the origin of the target co-ordinates within the message.

It should be noted that no parity is added to a GFI message.

4.2.3 XTL Data Drum Word Layout

An XTL message may be received by both the AN/FSQ-7 and AN/FSQ-8. However, since the type of information that may be present in each message varies, it is not possible to give any specific word layout for an XTL message. A normal XTL message is made of three drum words, as is shown in figure 4–17. Each drum word contains the following information:

a. Drum word 1
   1. Message word 1, LS through L15.
   2. Clock time, R5 through R10.
   3. Site identity, R11 through R15.
   4. Odd-parity bit.
b. Drum word 2
   1. Message word 3, LS through L15.
   2. Message word 2, RS through R15.
   3. Odd-parity bit.
c. Drum word 3
   1. Message word 5, LS through L15.
   2. Message word 4, RS through R10.
   3. Address, R11 through R15. (Bit R15 is designated as the All Party bit. If its contents are a 1, all sites tied together by the XTL telephone lines receive the message.)
   4. Odd-parity bit.

4.3 DRUM FIELD SELECTION

4.3.1 LRI Drum Fields

There are two physical drum fields which are used to contain incoming radar messages. There are 36 LRI channels which hold the data from the radar sites until it can be accepted by one of the two drum fields. The LRI element is wired so that channels 1 through 18 transfer 2-word messages onto LRI Field 2, and channels 19 through 36 transfer their 2-word messages onto LRI Field 1. Only one channel in a group may transfer data to a drum field at any one time. The method by which two words are placed in consecutive registers on the LRI drum field is known as writing by status. In addition to the normal channels on a drum field that are used to contain the bits within a word, there are two channels, known as the OD (outside-to-drum) channel and the CD (computer-to-drum) channel. The presence of a 0 bit in the OD status channel indicates that the register associated with it is empty or contains a word already read by the Central Computer System. A 1 bit in the OD status channel indicates that the register contains a word not yet read by the Central Computer System. Writing on the OD status channel is controlled by the Central Computer System, while reading is performed by the circuitry associated with the LRI channels. The CD channel is written on by the LRI CD drum circuitry to indicate to the Central Computer System that a register has information for the Central Computer System. If the register contains no information, the CD status channel contains a 0. When an LRI message is placed on one of the LRI fields, a 1 bit is placed in the CD status channel. When the Central Computer reads this drum message, a 0 bit is placed into the OD status channel, signifying that a new message may be placed in that register (or registers, in the case of LRI drum messages). If the LRI drum is not selected, there is a connection between the CD channel read head and the OD channel write head. Thus, if the Central Computer did not read a register that contained a message, it would indicate by writing a 1 in the OD channel, indicating that this register was still full and should not be written on by the LRI input channel. There is a similar connection between the OD read head and the CD write head on the OD side of the drum. In this way, it is assumed that the Central Computer System will not read the same message twice, nor will the Input System attempt to place a new message into a register that has not been read yet.

Writing on the LRI drum fields is actually per-

formed by modified status, since the LRI messages are composed of two words, and thus require two consecutive drum registers. The action of the CD and OD status channels remains the same with the exception that only every other drum register will contain bits in the channels. If the register containing the bits is determined to be ready for reading or writing, the next consecutive register is assumed to be ready also. The two drum registers are actually treated as one large register and are referred to as a drum "slot."

4.3.1.1 Reading by Status

The Central Computer System has three methods by which it can read the LRI drum fields. The first is by status only which indicates that the Central Computer System will accept every slot which contains a message. Since the incoming messages arrive at random, reading by status may give the Central Computer information from several different radar sites. A code of *SDR (34)* is used to select LRI drum field 1 for reading by status, and a code of *SDR (36)* will select LRI drum field 2 for reading by status.

4.3.1.2 Reading by Identity

The incoming LRI drum messages contain a message label in bits R7 through R11 which tells whether the message is a report from search radar, a height finder, or an IFF. In addition, bits R12 through R15 contain the site identity. It is possible to be selective about what information the Central Computer System will accept from the LRI drums by placing an identity word in the right half of the *SDR* instruction. This identity word is then transferred to the drum control register where each incoming message is compared to see if its bit contents match the identity word. Thus, it is possible to make the drum perform some degree of message sorting before the information enters the Central Computer System. There are two modes of reading by identity that can be specified. The first one utilizes a control word for bits R12 through R15 only. That means we can read in messages for any given radar site. The code used to select LRI drum field 1 for reading by identity in bits R12 through R15 is SDR (35). An SDR (37) will select LRI drum field 2 for reading by identity of bits R12 through R15.

The sorting performed by the drums can be increased still further by specifying an identity code in bits R7 through R15 in the right half-portion of the *SDR* instruction. Since R7 through R11 specifies what type of message is being transferred, an identity code of R7 through R15 will select a particular type of

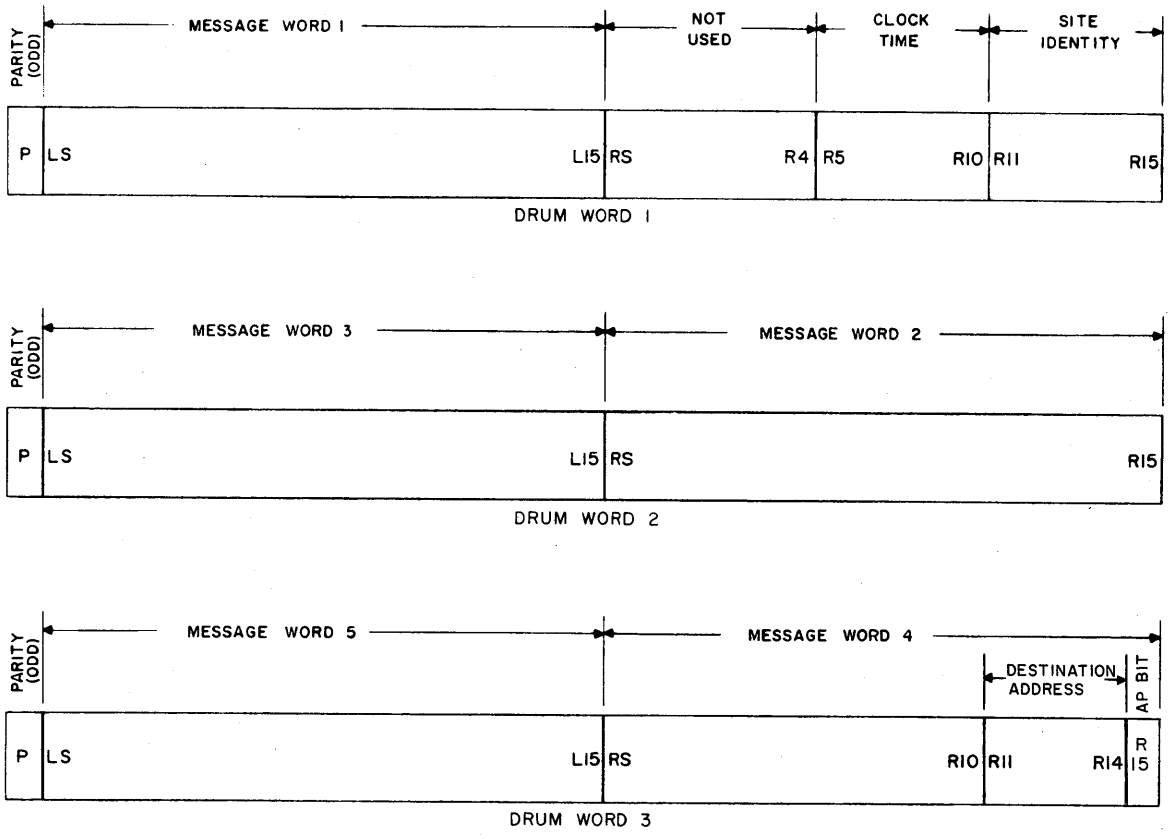


Figure 4–17. XTL Message Drum Field Layout

message from a particular site. It is not possible to select a particular type of message from all sites since the information would be meaningless. Selecting LRI drum field 1 by this identity code is performed by execution of an *SDR (50)* instruction. A code of *SDR (51)* selects LRI drum field 2 for reading by identity of bits R7 through R15.

4.3.2 GFI Drum Field

There is one physical GFI drum field which is selected for reading by status or identity. A code of *SDR (32)* will select the GFI drum field for reading by status, meaning that all current messages on the drum field will be accepted by the Central Computer System.

If it is desired to select messages off the drum from one particular GFI radar site, this may be done by execution of an *SDR (33)* instruction. The right half-portion of the *SDR (33)* must contain the identification of the desired site in bits R11 through R15.

4.3.3 XTL Drum Field

There is one physical XTL drum field which can be selected for reading by status or identity. It should be noted that XTL messages are composed of three drum words, thus making it necessary to provide a 3-word slot on the XTL drum field for each XTL message. The Central Computer System determines the placing of slots on the XTL drum field by placing a bit, called the marker bit, in a channel similar to the status channels. The presence of a 1 bit in the marker channel indicates to the XTL channels supplying XTL messages that the register containing the marker is the first register of a 3-register slot. The marker bits are usually placed on the XTL drum field before normal operation is started. An *SDR (40)* instruction selects the XTL drum, and the markers are placed in the marker channel under program control. The contents of the LS bit determine the bit that is placed in the marker channel. A sample program to place marker bits on the entire XTL drum field is given in table 4–22.

The index register is loaded so that the first drum register to have a marker bit written in it will be 3373, the highest numbered usable slot on the XTL drum. The *1 BPX (03)* instruction will cause successively lower 3-register slots to be written.

To select the XTL field for reading by status, an *SDR (24)* instruction is executing, indicating that all 3-register slots which contain a current message will be accepted. If it is desired to read XTL messages from a particular site, the identity mode of reading is selected by the execution of an *SDR (25)* instruction. A code word corresponding to the contents of bit R11 through R15 of drum word 1 is placed in the right half-portion of the *SDR (25)* instruction, enabling the drum control register to make the comparison.

4.4 INPUT TEST PATTERN GENERATOR

The Input System has a test pattern generator which may be operated in several modes. Some of these modes are completely under program control, while other modes are partially under program control. The test pattern generator provides testing of all three input elements, if desired.

4.4.1 LRI Testing

Putting the test pattern generator completely under computer control is accomplished by selecting the mode II, type 1 operation. (The mode is determined by switches on the test pattern generator.) In this mode, operation is initiated by execution of a *PER (24)* instruction. The contents of the LRI message are determined by the execution of *PER (65)* and *PER (66)* instructions. Each *PER (65)* instruction will generate one bit of sync. Each *PER (66)* instruction will generate one bit of LRI data. To simulate a complete LRI message, several successive *PER (65)* and *PER (66)* instructions must be executed until the desired LRI message is formed. If it is desired to place a preset LRI message into the LRI channel under test, a *PER (63)* instruction is executed. This will initiate a message which has been set up by the bit selection switches on the tester. Each *PER (63)* instruction will initiate only one message. The Central Computer System can return to complete program control of the message being generated by execution of another *PER (24)* instruction.

To provide for synchronization of the timing between generation of LRI messages from the test pattern generator and the formation of messages within the Central Computer System, a *BSN (34)* instruction is provided. This instruction senses to see if the LRI

TABLE 4–22. CROSSTELL MARKER PROGRAM

LOCATION	OPERATION	ADDRESS
0.00700	*1 XIN*	0.03373
0.00701	*1 SDR (40)*	0.00000
0.00702	*LDC*	0.01000
0.00703	*WRT*	~~0.00003~~
0.00704	*1 BPX (03)*	0.00701
0.00705	*HLT*	–
0.01000	1.00000	0.00000
0.01001	0.00000	0.00000
0.01002	0.00000	0.00000

timing flip-flop is set. If it is, a branch to the address specified by the right half-portion of the *BSN (34)* instruction takes place.

4.4.2 GFI Testing

The GFI test pattern generator may also be operated in several modes. In mode II, type 1 (again selected manually), a GFI message is initiated by generation of a *PER (64)* instruction. The desired GFI message must then be formed by successive execution of *PER (65)* and *PER (66)* instructions. Each *PER (65)* instruction will provide one bit of GFI target information. When a message which has been set up manually by bit selection switches is to be entered into the GFI channel under test, a *PER (63)* instruction may be executed. This will terminate program control of the GFI message content and initiate continuous GFI patterns from the switches. When it is desired to return to program control of the GFI message, a *PER (64)* instruction may be executed.

The *BSN (34)* instruction may be executed during GFI testing to determine when the GFI range flip-flop is set. This is to synchronize the actions of the Central Computer System with the GFI test pattern generator. If the range flip-flop is set, the program will branch to the location specified by the right half-portion of the instructions.

In some instances, the Central Computer System must know at exactly what time to shift from one type of GFI operation to another. This shift usually takes place at GFI north azimuth time (when GFI sweep passes 0 degrees). When this occurs, the GFI north azimuth flip-flop is set. The north azimuth flip-flop can be sensed by a *BSN (47)* instruction and will branch to the location specified by the right half-portion of the *BSN (47)* instruction if the north azimuth flip-flop is set.

4.4.3 XTL Testing

Testing of the XTL element is conducted in the same way as testing for the LRI element. Mode II, type 1 operation is selected by switch settings and is initiated by execution of a *PER (25)* instruction. In this mode, the XTL messages are made up by execution of *PER (65)* and *PER (66)* instructions. Each *PER (65)* instruction will form one bit of XTL sync and each *PER (66)* instruction will cause the formation of one bit of XTL data. When a change to an XTL message set up by switches is desired, a *PER (63)* instruction is executed. This instruction will allow only one XTL message to enter a channel and additional *PER (63)* instructions are required for each message desired. Shifting back to complete program control can be accomplished by execution of another *PER (25)* instruction.

The *BSN (34)* instruction can also be executed with XTL testing. The instruction will check the XTL timing flip-flop to see if it is on. If it is on, a branch of program control to the address specified by the right half-portion of the *BSN (34)* instruction will result. This instruction is utilized to synchronize actions of the test pattern generator and the Central Computer System.

CHAPTER 5
PROGRAMMING THE OUTPUT SYSTEM

5.1 DESCRIPTION

The operation of the AN/FSQ-7 and AN/FSQ-8 requires the transmission of processed information to such sites as air bases, airborne interceptors, antiaircraft artillery units, radar sites, and other centrals. The Central Computer System prepares this information and delivers it in parallel to drum fields associated with the Output System. The Output System obtains this information from the drum fields in parallel form and converts the information into serial form for transmission via telephone facilities to its ultimate destination. The Central Computer System places output information on the output buffer (OB) fields status. The Output System reads this information from the OB fields by status identification.

Three types of output information are handled by the Output System: ground-to-air time division (G/A TD) information used for airborne interceptors; ground-to-ground (G/G) information for transmission to other centrals or to height-finder radar sites; and teletype (TTY) information for transmission to non-automatized centers. Ground-to-air information is transmitted via telephone facilities to radio transmission equipment for relay to airborne receivers; G/G information, by telephone facilities to receiving points; TTY information, as conventional teletype signals via telephone facilities. The Output System also supplies control information to the Central Computer System, allowing that system to preassign times of transmission of specific items of information.

5.2 OPERATION

The Output System comprises two elements, the output control element and the output storage element. The output control element reads output drum words from the OB fields, performs certain checks on each word accepted, and directs the output information contained in each word to the appropriate section of the output storage element. When requested by the Central Computer System, the output control element also supplies control information to the Central Computer System.

The output storage element, since it handles several different types of information, is divided into several sections. Each section receives output information presented to it by the output control element, performs further checks on the information, and converts the information into serial form for telephone transmission.

Since failure of any element or section of the Output System would seriously impair its functioning, the Output System is in duplex, i.e., one Output System is within computer A and the other is within computer B. The Output System of the active computer feeds the telephone transmission equipment. The Output System of the standby computer is not used operationally but may be used in certain test operations.

5.3 INFORMATION FORMS

5.3.1 Bursts

Each of the three types of information processed by the Output System is transferred out of the Output System in blocks of fixed duration and content, known as bursts. Within the interval known as a burst period, a single block of information is prepared and transmitted by each section. Bursts from each section are numbered in sequence. These burst numbers are used to control the sequence of transmission of information items and, in certain cases, can also be used to control the exact time of transmission of a specific item. Each word prepared by the Central Computer System for delivery to the Output System contains an assigned burst number. These words are read into the Output System by status identification, using the burst number as the identity code. Each storage section accepts only those words whose assigned burst number corresponds to the number of the burst being prepared by the section at that time, as indicated by a counter in the section (the burst counter). Thus, the assignment of burst numbers to the output drum words can control the sequence and time of transmission of information from each section.

The burst counters for each section are read in sequence by the Central Computer System and are selected by a *SEL (21)* instruction. Execution of the *SEL (21)* instruction will enable the Central Computer System to read the contents of all the burst counters. The burst period for each output storage section is defined as the interval between successive steppings of its burst counter. Two intervals are included within a burst period, search time and readout time. Search time is the interval within which words are accepted by a storage section for transmission during the following readout

time. Readout time is the interval during which words, previously assembled, are transmitted.

5.3.2 Output Drum Word

Information is delivered to the Output System in the form shown in figure 4–18. The right half-word contains the information to be assembled with information from other words for transmission from the Output System in a burst. The left half-word controls the routing of the right half-word within the Output System. This left half-word can be divided into four major parts:

a. Parity bit
b. Output section address
c. Output register address
d. Assigned burst number

The parity bit allows a check on the accuracy of transfer of the word from the OB drum fields to the Output System. The word is not accepted by the Output System unless the parity count for the entire 33-bit word is odd.

The output section address, LS through L2, designates the section of the output storage element through which the right half-word is to be transferred.

The addresses of the various sections are shown in table 4–23.

The output register address, L3 through L7, designates the register within the specified storage section which is to receive the right half-word. Although 32 separate register addresses numbered from 0 through 31 may be specified by five binary bits, not all the registers are used in any section. Unused register addresses within each section are designated illegal register addresses.

The assigned burst number, a number between L8 and L15, determines the order in which output drum words having different burst numbers will be read and transmitted. With eight bits available, 256 burst numbers can be specified before the counting cycle must repeat. The assigned burst number of an output drum word must coincide with the number of the burst currently being assembled in the storage section selected by the section address of that word in order for the word to be accepted by the output control element.

5.4 DRUM TRANSFERS

When an output message is to be placed on one of the OB drum fields, a unique drum field selection is used. Although there are three physical OB drum fields, the Central Computer System treats these fields as two logical fields, the OB even field and the OB odd field. The OB odd field consists of all the odd-numbered registers on all three drum fields, while the OB even field consists of all the even-numbered registers on all three fields. The Central Computer System selects the OB odd field by execution of an *SDR (30)* instruction and selects the OB even field by execution of an *SDR (31)* instruction. When one of the above instructions is executed, the program will thus proceed to place a given output message in alternate registers on the OB fields. The Central Computer System places the output messages onto the OB fields by means of status.

When the Output System reads the OB fields, it always reads the entire three fields. It is possible for the Output System to read every register from the drum fields; however, only alternate registers can contain words of an output message going to any one particular output section. This results from the manner in which the message was placed on the drum (in either odd or even registers) and was done so that the transfer rate of information from the drum would be compatible with

TABLE 4–23. OUTPUT SECTION ADDRESS CODES

OUTPUT SECTION	ADDRESS CODE
Illegal address	000
Illegal address	001
Ground-to-ground	010
Teletype	011
Illegal address	100
*Ground-to-air (time division)	101
Illegal address	110
Illegal address	111

**Not included in AN/FSQ-8.*

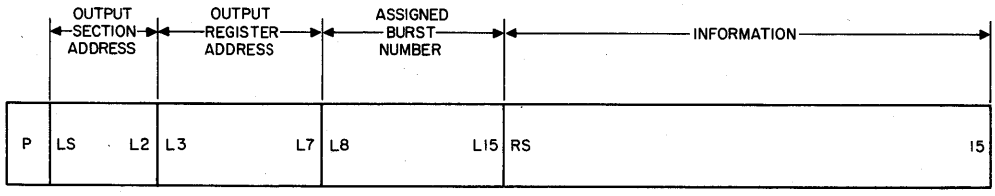


Figure 4–18. Output Drum Word Layout

the slower acceptance rate of the tape cores used to store messages in the output sections.

5.5 OUTPUT ALARMS

5.5.1 Overall Output Alarm

When any alarm condition arises in the Output System, it sets an output alarm flip-flop which may be tested by the Central Computer System. A *BSN (33)* instruction checks to see if the output alarm flip-flop is set and branches to the address specified in the right half-portion of the instruction if there is an alarm condition in the Output System. The execution of a *BSN (33)* instruction checks to see if the output alarm flip-flop is set and branches to the address specified in the right half-portion of the instruction, if there is an alarm condition in the Output System. The execution of a *BSN (33)* instruction will also clear the output alarm flip-flop.

5.5.2 Nonsearch Alarm

When a word is found on the output buffer fields during nonsearch time that has the same burst number as the burst currently being transmitted, a nonsearch compare alarm is generated. A flip-flop indicating whether a nonsearch compare alarm exists may be sensed by execution of a *BSN (50)* instruction. If a nonsearch alarm does exist, the program will branch control to the address specified in the right half-word of the instruction and turn off the alarm flip-flop.

5.5.3 OB Drum Parity Alarm

The OB drum parity alarm indicates detection of a parity error received from the OB drum fields. This alarm will set an OB drum parity flip-flop which may be sensed by a *BSN (51)* instruction. If a parity error did occur, the *BSN (51)* instruction will cause the program to branch control to the address specified in the right-half portion of the instruction and will clear the flip-flop.

5.5.4 Illegal Address or Section Alarm

If a drum word contains a section or register address that is not legal for the Output System, an illegal address or section alarm will be generated and will turn on a corresponding flip-flop. This flip-flop may be sensed by a *BSN (52)* instruction, and the program will branch to the address designated by the instruction. If this alarm condition exists, execution of the *BSN (52)* instruction will also clear the illegal address or section alarm flip-flop.

5.5.5 G/G Parity Alarm

A G/G message being sent to the telephone lines from the output storage element is also parity-checked, and if it is bad, sets a G/G parity alarm flip-flop. A *BSN (54)* instruction senses this flip-flop and branches to the specified location if it is set. Execution of the *BSN (54)* instruction will also clear the flip-flop.

5.5.6 TTY Parity Alarm

A parity error in a TTY message being transmitted over the telephone lines will result in the setting of a TTY parity flip-flop. This flip-flop can be sensed by a *BSN (55)* instruction, which will cause the program to branch to the specified address in the right half-word of the *BSN (55)* instruction and will clear the alarm flip-flop.

5.5.7 G/A TD Parity Alarm

The G/A TD parity alarm flip-flop is set when there is a defective transmission of a G/A TD message. This flip-flop may be sensed by a *BSN (56)* instruction. If the flip-flop is set, the *BSN (56)* instruction will cause the program to branch to the address contained in the right half-word of the instruction. A *BSN (56)* instruction will also clear the flip-flop.

CHAPTER 6

PROGRAMMING THE DISPLAY AND WARNING LIGHT SYSTEMS

6.1 DESCRIPTION

The Display System of the AN/FSQ-7 and AN/FSQ-8 is the medium through which all data collected by the Input System and processed by the Central Computer System must be presented to personnel directing the defense against air attack. The AN/FSQ-7 data which must be collected includes the following: the locations of unidentified, friendly, and hostile aircraft within the area; the status of defensive weapons that can be employed against hostile aircraft; and information on conditions affecting the usefulness of each weapon, such as the extent of visibility, velocity of the wind, etc.

The AN/FSQ-8 Display System presents only data on aircraft and weapons already under control of the AN/FSQ-7. The AN/FSQ-8 Display System presents a summary of all air defense information which has been processed by several AN/FSQ-7's. By presenting this information in digested form to the personnel directing air defense, the Display System makes it possible for the operators to make informed decisions on the actions to be taken in each case. Their decisions are indicated to the Central Computer System for implementation via the manual data input element of the Display System.

The information to be presented by the Display System can be divided into two types: one type, presenting a geographical picture of air movements within the coverage area, must be capable of changing rapidly to reflect accurately the air movements themselves; the other type, presenting such relatively static information as weather conditions or weapons status, need not be changed as often as the first type. The two types of information are known as situation display information and digital display information, respectively. Situation display information provides a complete picture of air movements in a plan-position-indicator (PPI) form. It is identical in function to the plotting-board display of a manual air defense filter center. Digital display information provides statistical data in tabular form. It is identical in function to the tote-board display of a manual air defense filter center.

6.2 OPERATION

6.2.1 General

The nature of the information handled by the Display System dictates its division into two subsystems, situation display and digital display. Both subsystems receive processed information from the Central Computer System via the Drum System. This information is supplied to the generator element in each subsystem. Each generator element, in turn, converts the information presented to it into the form required by the indicator element of that subsystem.

6.2.2 Situation Displays

The situation display subsystem consists of the situation display generator element (SDGE) and the situation display indicator element (SDIE). The situation display generator is a centrally located unit which converts processed situation display information into the form required by the situation display indicator element. Each situation display indicator element is housed in a display console. (See fig. 4–19.)

6.2.3 Digital Displays

The digital display subsystem includes the digital display generator element (DDGE) and the digital display indicator element (DDIE). The digital generator element is a centrally located unit which converts digital display information into the form required by the digital display indicator element. The digital display indicator element is also housed with the situation display indicator section in a display console. However, some digital display indicator sections are separately housed in what are known as auxiliary consoles, and some are in the Command Post desk.

6.2.4 Display Consoles

Most display consoles house an SDIS and a DDIS as well as a panel of pushbuttons for inserting information via the manual data input element into the Central Computer System. Some consoles also have light guns for designating specific targets by means of the manual data input element. Two special consoles are included for specific purposes. One console provides the situation display which is photographed to provide a large board display by projection of the photograph. Another console is equipped with a camera which makes permanent records of air defense situations for later analysis or for use in training of console operators.

6.2.5 Manual Inputs

6.2.5.1 General

All input information to the Central Computer System AN/FSQ-7 or AN/FSQ-8 that is not processed

automatically by the Input System is handled by the manual input element. However, since most incoming manual information is in the form of requests from the Display System, the manual input element is logically a part of the Display System.

An operator at a situation display console may communicate a decision, a request for other information, or some information to be associated with a given situation display message via the manual data input keyboard mounted on the console. A number of prearranged actions are written into the program of the Central Computer System. Each action is performed only if called for by a manual input data selection panel message. Since the function of each console and the programmed actions required by that function can be predetermined, the keyboard message from that console need only indicate which of the possible actions is to be performed at a given time. Thus, the console operator can depress a particular pushbutton with the knowledge that although only one binary bit may be changed from 0 to 1, a relatively complicated message has been conveyed.

Since the Central Computer System may read a data selection panel before the operator has completed mak-

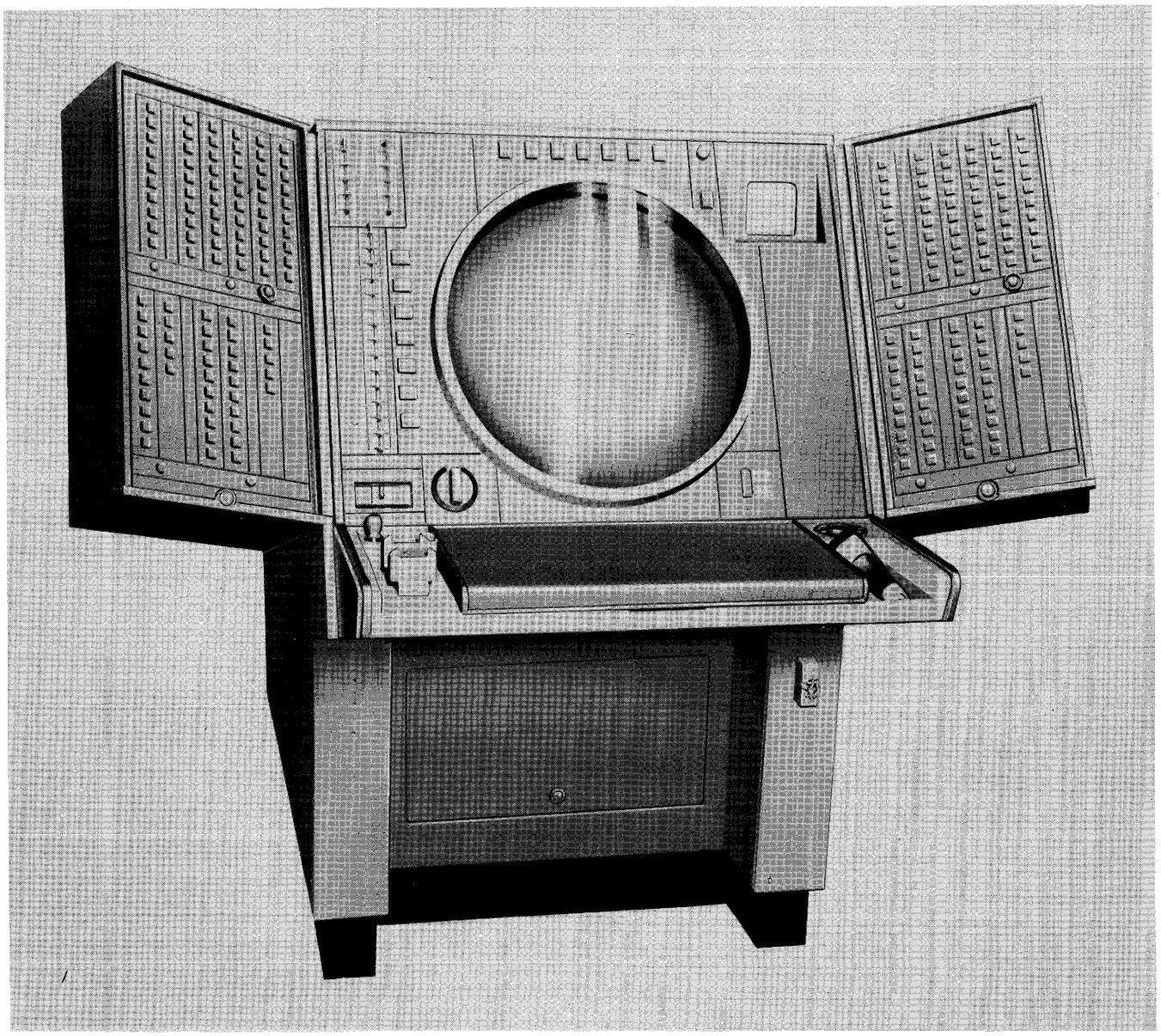


Figure 4–19. Display Console

ing up the message, some provision must be made to prevent action on an incomplete message. This is done by requiring the console operator to indicate the completion of a keyboard message in one of two ways, by depressing the ACTION pushbutton on the keyboard as the last step in making up the message or by pulling the trigger on the light gun at the console. Either method sets a core in the manual input core array which indicates that the message from that keyboard is complete and may be acted on. The choice of method depends upon the type of message being sent via the keyboard. If the message requires no reference to a particular target, the ACTION pushbutton is used. If the message must be related to a particular target, the light gun is used.

6.2.5.2 Light Guns

If a console operator were required to identify a specific target to the Central Computer System by inserting an estimate of its co-ordinates, or its track number, on keyboard switches, errors in estimation or insertion might result. The light gun, a semi-automatic target designation device, eliminates this source of error.

A console operator can specify some action on a selected target by setting up a keyboard message and then designating the target with the light gun at the console. The light gun is positioned so that the red aiming beam it projects just covers the present radar data message to be designated or the vector origin of the tabular track data message to be designated. The console operator then pulls the light gun trigger (this indicates keyboard message completion in the same manner as does depressing the ACTION pushbutton) and intensifies all point features. When the selected target is written during a situation display cycle, the photomultiplier tube within the light gun detects the initial blue flash and generates a pulse coincident in time with the presence of identification information on the selected target in the situation display generator element (SDGE). The light gun pulse is applied to the SDGE as an information transfer signal that gates the target identification out of the SDGE through the manual data input element onto the manual input drum field. The console operator is informed that transfer has been initiated when the aiming beam on the light gun is extinguished and an indicator on its rear is lighted. The operator readies the light gun for another transfer by releasing the trigger on the light gun, restoring the aiming beam, and extinguishing the rear indicator.

6.2.5.3 Area Discriminator

One type of Central Computer System operation obtains information from the Display System via the manual data input element without operator intervention. This operation is automatic track initiation. The required display information is obtained by the Central Computer System from an area discriminator, a large light gun covering an entire situation display tube face.

Automatic initiation is performed in light air traffic areas since the decisions involved are purely routine and efficiency of detection is high. The automatic initiation area discriminator is set to display uncorrelated present radar data. Those areas in which automatic initiation is not to be performed are masked out. When the area discriminator is operated, it transfers reports on all unmasked targets displayed during one situation display cycle. The area discriminator is not used with the AN/FSQ-8.

6.2.5.4 Manual Input Matrix

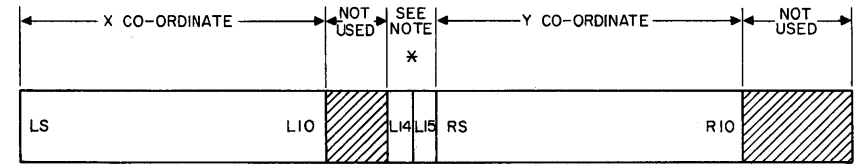
As explained above, the manual input matrix (core array) acts as a time buffer for keyboard messages presented to the Central Computer System from the Display System. The manual input matrix (MIM) is read under program control and transfers its information to core memory for analysis by the Central Computer System.

6.2.6 Warning Light System

Although the Warning Light System is a separate system in the AN/FSQ-7 and AN/FSQ-8, it is utilized primarily to notify personnel at various Display System consoles of special program conditions or machine malfunctions. This is accomplished by lighting neon lights on the various consoles and by generating audible alarms. Because of the close association of the Warning Light System with the Display System, it is discussed in this chapter.

The Warning Light System consists of three elements: the control element, the storage element, and the interconnection and indicator element. The control and storage elements are duplex; the interconnection and indicator element is simplex. The control element synchronizes Warning Light System operation with Central Computer System operation. The storage element consists of 256 flip-flops arranged in eight 32-bit flip-flop registers (called warning light registers), which control the operation of the warning devices. The interconnection and indicator element consists of a patchboard interconnection unit and the various indicators (neons and audible alarms) located at the consoles. Each indicator at each console is connected to a specific flip-flop in the warning light registers through the interconnection unit. The status of each indicator is determined by the status of the flip-flop to which it is assigned. Thus, if a flip-flop is set, the associated neon lights remain illuminated until the flip-flop is cleared. If the flip-flop also controls an audible alarm, the alarm is turned on by the flip-flop but may be turned off manually by depressing a momentary-contact pushbutton at the console. The alarm remains off until the controlling flip-flop is cleared and reset.

In order to change the condition of one or more warning lights, the Central Computer System must institute an IO operation, during which the contents of the eight warning light registers are replaced by the contents of eight core memory registers. The core memory registers contain the image of the contents of the warning light registers. If it is desired to change the condition of one or more warning lights, the bits in the core memory image are changed accordingly. Periodically, the program calls for the transfer of the eight words in core memory to the eight warning light resistors. The words transferred to the warning light registers contain not only the 1 bits corresponding to the lights that are to be changed but also the 1 bits corresponding to the lights that were on and which will remain unchanged.



* NOTE:
L14 - "1" IDENTITY
"0" NO IDENTITY
L15 - "1" CORRELATED
"0" UNCORRELATED

Figure 4–20. Radar Data Message Drum Layout

WORD	BIT POSITION																														
	LS	L1	L2	L3	L4	L5	L6	L7	L8	L9	L10	L11	L12	L13	L14	L15	RS	R1	R2	R3	R4	R5	R6	R7	R8	R9	R10	R11	R12	R13	R14	R15
0	*	A_1			A_2			A_3			A_4			E			*	A_1			A_2			A_3			A_4			E		
		X_2^2	X_2^1	X_2^0	X_2^2	X_2^1	X_2^0	X_2^2	X_2^1	X_2^0	X_2^2	X_2^1	X_2^0	X_2^2	X_2^1	X_2^0		y_2^2	y_2^1	y_2^0	y_2^2	y_2^1	y_2^0	y_2^2	y_2^1	y_2^0	y_2^2	y_2^1	y_2^0	y_2^2	y_2^1	y_2^0
1		X																Y														
		2^0	2^{-1}	2^{-2}	2^{-3}	2^{-4}	2^{-5}	2^{-6}	2^{-7}	2^{-8}	2^{-9}	2^{-10}	2^{-11}	2^{-12}				2^0	2^{-1}	2^{-2}	2^{-3}	2^{-4}	2^{-5}	2^{-6}	2^{-7}	2^{-8}	2^{-9}	2^{-10}	2^{-11}	2^{-12}			
2		CATEGORY					*	DAB																								
		2^4	2^3	2^2	2^1	2^0		1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26
3		DAB																														
		27	28	29	30	31	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47	48	49	50	51	52	53	54	55	56	57	58
4		DAB																														
		59	60	61	62	63	64	65	66	67	68	69	70	71	72	73	74	75	76	77	78	79	80	81	82	83	84	85	86	87	88	89	90
5		X							C_2			C_3			C_4			Y							C_2			C_3			C_4		
		2^0	2^{-1}	2^{-2}	2^{-3}	2^{-4}	2^{-5}	2^{-6}	X_2^2	X_2^1	X_2^0	X_2^2	X_2^1	X_2^0	X_2^2	X_2^1	X_2^0	2^0	2^{-1}	2^{-2}	2^{-3}	2^{-4}	2^{-5}	2^{-6}	y_2^2	y_2^1	y_2^0	y_2^2	y_2^1	y_2^0	y_2^2	y_2^1	y_2^0
6	*	B_1			B_2			B_3			B_4			B_5				B_1			B_2			B_3			B_4			B_5		
		X_2^2	X_2^1	X_2^0	X_2^2	X_2^1	X_2^0	X_2^2	X_2^1	X_2^0	X_2^2	X_2^1	X_2^0	X_2^2	X_2^1	X_2^0		y_2^2	y_2^1	y_2^0	y_2^2	y_2^1	y_2^0	y_2^2	y_2^1	y_2^0	y_2^2	y_2^1	y_2^0	y_2^2	y_2^1	y_2^0
7	*	D_1			D_2			A_5			A_6			C_1				D_1			D_2			A_5			A_6			C_1		
		X_2^2	X_2^1	X_2^0	X_2^2	X_2^1	X_2^0	X_2^2	X_2^1	X_2^0	X_2^2	X_2^1	X_2^0	X_2^2	X_2^1	X_2^0		y_2^2	y_2^1	y_2^0	y_2^2	y_2^1	y_2^0	y_2^2	y_2^1	y_2^0	y_2^2	y_2^1	y_2^0	y_2^2	y_2^1	y_2^0

* = CONTROL BIT

WORD	POSITION	CONTENT	MEANING
0	LS	0	TRACK MESSAGE
		1	INFORMATION MESSAGE
	RS	0	TABULAR FORM
		1	VECTOR FORM
2	L5	0	SUPPRESS DISPLAY
		1	ALLOW DISPLAY
6	LS	0	LIGHT GUN MAY BE USED
		1	LIGHT GUN MAY NOT BE USED
7	LS	0	CHARACTERS TO LEFT OR RIGHT OF VECTOR
		1	CHARACTERS ABOVE OR BELOW VECTOR

Figure 4–21. Tabular Message Drum Layout

6.3 MESSAGE TYPES

6.3.1 Situation Displays

Situation display messages are one of two types: radar and data messages, and track data messages. Radar data messages placed on the radar data (RD) drum fields consist of one word only; track data messages placed on the track data (TD) drum fields consist of eight words. Because the AN/FSQ-8 receives reports on processed air defense data from AN/FSQ-7's, the discussion concerning radar data does not apply to this machine, only to the AN/FSQ-7.

6.3.1.1 Radar Data Messages

Radar data messages are sometimes called point messages since each such message designates the point locating a target. The symbol displayed at the point identifies the radar source (search radar or IFF) and indicates whether the target report is correlated; i.e., whether a track data message can be developed from the radar report. The age of the message is distinguished by the brightness of its display: if bright, the message is a present report; if dim, the message is a past report. The direction in which the target is moving is indicated by the position of the present (bright) report on the target. Radar data messages are useful primarily in manual initiation of tracks and in analyzing trouble-track situations. They are used most in conjunction with the radar inputs program operations and with the smoothing and trouble detection program operations. The drum layout of a radar data message is given in figure 4–20.

6.3.1.2 Track Data Messages

Track display messages are capable of presenting a great deal of information on the targets they describe. Two types of track display messages are distinguished, tabular messages and vector messages. Each subtype occupies eight drum words. A tabular message, however, can cause display of one vector and up to 13 characters (although 18 characters are contained within the message), whereas a vector message can cause display of four vectors and four characters.

A tabular message may be either a tabular track message describing a target in track or a tabular information message describing anything which can be represented by up to 13 characters. The drum layout of a tabular message is given in figure 4–21.

A vector message can cause display of up to four independently positioned vectors and four characters associated with the fourth vector. Vector messages are used for a variety of information displays, including flight histories, geographical boundaries, raid symbols, and attention devices. The drum layout of a vector message is given in figure 4–22.

WORD	BIT POSITION																															
	LS	LI	L2	L3	L4	L5	L6	L7	L8	L9	LI0	LII	LI2	LI3	LI4	LI5	RS	RI	R2	R3	R4	R5	R6	R7	R8	R9	RI0	RII	RI2	RI3	RI4	RI5
0		G_1 X_2^2	X_2^1	X_2^0	G_2 X_2^2	X_2^1	X_2^0	G_3 X_2^2	X_2^1	X_2^0	G_4 X_2^2	X_2^1	X_2^0				*	G_1 y_2^2	y_2^1	y_2^0	G_2 y_2^2	y_2^1	y_2^0	G_3 y_2^2	y_2^1	y_2^0	G_4 y_2^2	y_2^1	y_2^0			
1	X_1 2^0	2^{-1}	2^{-2}	2^{-3}	2^{-4}	2^{-5}	2^{-6}	2^{-7}	2^{-8}	2^{-9}	$\dot{X}_1$ 2^0	2^{-1}	2^{-2}	2^{-3}	2^{-4}	2^{-5}	Y_1 2^0	2^{-1}	2^{-2}	2^{-3}	2^{-4}	2^{-5}	2^{-6}	2^{-7}	2^{-8}	2^{-9}	$\dot{Y}_1$ 2^0	2^{-1}	2^{-2}	2^{-3}	2^{-4}	2^{-5}
2	CATEGORY 2^4	2^3	2^2	2^1	2^0	*	DAB 1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26
3	DAB 27	28	29	30	31	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47	48	49	50	51	52	53	54	55	56	57	58
4	DAB 59	60	61	62	63	64	65	66	67	68	69	70	71	72	73	74	75	76	77	78	79	80	81	82	83	84	85	86	87	88	89	90
5	X_2 2^0	2^{-1}	2^{-2}	2^{-3}	2^{-4}	2^{-5}	2^{-6}	2^{-7}	2^{-8}	2^{-9}	$\dot{X}_2$ 2^0	2^{-1}	2^{-2}	2^{-3}	2^{-4}	2^{-5}	Y_2 2^0	2^{-1}	2^{-2}	2^{-3}	2^{-4}	2^{-5}	2^{-6}	2^{-7}	2^{-8}	2^{-9}	$\dot{Y}_2$ 2^0	2^{-1}	2^{-2}	2^{-3}	2^{-4}	2^{-5}
6	X_3 2^0	2^{-1}	2^{-2}	2^{-3}	2^{-4}	2^{-5}	2^{-6}	2^{-7}	2^{-8}	2^{-9}	$\dot{X}_3$ 2^0	2^{-1}	2^{-2}	2^{-3}	2^{-4}	2^{-5}	Y_3 2^0	2^{-1}	2^{-2}	2^{-3}	2^{-4}	2^{-5}	2^{-6}	2^{-7}	2^{-8}	2^{-9}	$\dot{Y}_3$ 2^0	2^{-1}	2^{-2}	2^{-3}	2^{-4}	2^{-5}
7	X_4 2^0	2^{-1}	2^{-2}	2^{-3}	2^{-4}	2^{-5}	2^{-6}	2^{-7}	2^{-8}	2^{-9}	$\dot{X}_4$ 2^0	2^{-1}	2^{-2}	2^{-3}	2^{-4}	2^{-5}	Y_4 2^0	2^{-1}	2^{-2}	2^{-3}	2^{-4}	2^{-5}	2^{-6}	2^{-7}	2^{-8}	2^{-9}	$\dot{Y}_4$ 2^0	2^{-1}	2^{-2}	2^{-3}	2^{-4}	2^{-5}

* CONTROL BIT

WORD	POSITION	CONTENT	MEANING
0	RS	0	TABULAR FORM
		1	VECTOR FORM
2	L5	0	SUPPRESS DISPLAY
		1	ALLOW DISPLAY

Figure 4–22. Vector Message Drum Layout

The x and y co-ordinates designated in these two types of messages refer to the co-ordinates of characters in a character matrix inside the SD cathode-ray tube. The character matrix for the SD CRT is shown in figure 4–23.

6.3.2 Digital Displays

Digital display information is presented on the screens of 5-inch digital display cathode-ray tubes (DD CRT's) located in situation display consoles, auxiliary consoles, and at the Command Post desk. Digital displays are used primarily to furnish additional information supplementing messages displayed on the situation display cathode-ray tubes (SD CRT's) and to summarize existing situations, such as weather data, assignment data, and other data of a comparatively stable nature. Unlike the situation display information which is redisplayed every 2.54 seconds, digital display information is stored in the DD CRT for any desired period and must be erased before new information can be written.

The distribution of digital display information differs from the distribution of situation display information. The Central Computer System writes the digital display data in slots of lengths varying from 4 to 32 registers on the digital display (DD) field of the MIXD drum. Each slot is assigned to a specific digital display indicator section. (In a few exceptional cases, one slot is assigned to two indicator sections.) In contrast, situation display messages are written by the Central Computer System in slots of equal length (eight registers for TD and one register for RD messages) on the respective drums, and these messages contain their own routing information.

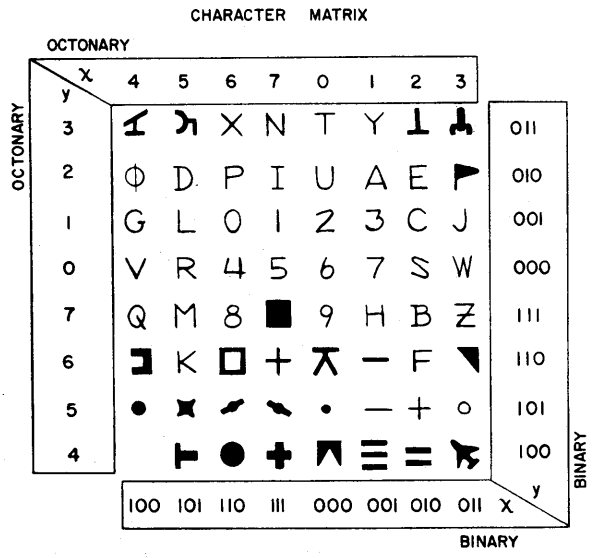
A typical digital display drum message format is shown in figure 4–24. The x and y co-ordinates referred to are the co-ordinates of characters contained in the character matrix of the DD cathode-ray tube. Six bits are required to completely select one character. For example, bits L1-L3 and R1-R3 will select the first character of a 5-character digital display message. The character matrix is shown in figure 4–25.

6.3.3 Manual Inputs

Information from the drum entry section is placed on the MDI field of the MIXD drum by status, with one item per drum register. The layout of information within each word is determined by the type of information. The drum layout of a track data word is shown in figure 4–26, part A. A radar data word layout is shown in figure 4–26, part B.

6.3.4 Warning Lights

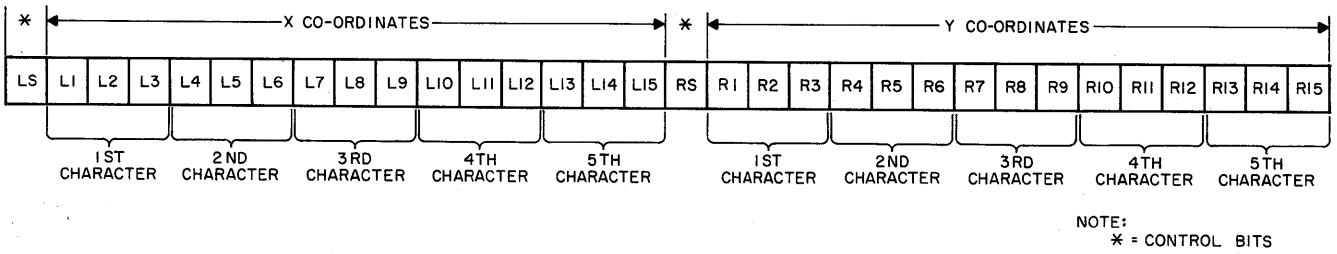
It is not possible to give any illustration of a warning light message, since only core memory registers are transferred to the warning light registers, and they may contain any value at any one point.



OCTONARY ADDRESSES OF CHARACTERS ON CHARACTER MATRIX

CHAR	x	y	CHAR	x	y	CHAR	x	y	CHAR	x	y
A	1	2	Q	4	7	7	1	0		7	4
B	2	7	R	5	0	8	6	7		0	4
C	2	1	S	2	0	9	0	7		1	4
D	5	2	T	0	3		0	6		5	5
E	2	2	U	0	2	—	1	6		4	5
F	2	6	V	4	0		4	6	+	2	5
G	4	1	W	3	0		3	6		3	5
H	1	7	X	6	3		5	4		7	7
I	7	2	Y	1	3	=	2	4		3	4
J	3	1	Z	3	7		6	5		4	3
K	5	6	1	7	1		7	5		5	3
L	5	1	2	0	1	•	0	5		2	3
M	5	7	3	1	1	—	1	5		3	3
N	7	3	4	6	0	●	6	4		4	2
O	6	1	5	7	0		6	6		3	2
P	6	2	6	0	0	+	7	6	BLANK	4	4

Figure 4–23. Situation Display Tube Character Matrix and Octonary Addresses



POSITION AND CONTENTS		MEANING
* LS	* RS	
0	1	START OF NEW MESSAGE
0	0	START THIS ROW IMMEDIATELY BELOW PRECEDING ROW DISPLAYED
1	1	SKIP A ROW BETWEEN THE PRECEDING ROW AND THE ROW ON WHICH THIS WORD IS TO BE DISPLAYED
1	0	DISPLAY THIS WORD ON THE TOP ROW ON THE OPPOSITE SIDE OF THE SAME INDICATOR SECTION

Figure 4–24. Digital Display Message Drum Word Layout

CHARACTER MATRIX

OCTONARY y \ x	4	5	6	7	0	1	2	3	BINARY y
3	u	f	X	N	T	Y	t	v	011
2	i	D	P	I	U	A	E	r	010
1	G	L	O	1	2	3	C	J	001
0	V	R	4	5	6	7	S	W	000
7	Q	M	8	●	9	H	B	Z	111
6	b	K	a	p	h	g	F	d	110
5	c	m	o	w	e	s	z	n	101
4		◣	▲	▼	—	↑	↓	x	100
BINARY x	100	101	110	111	000	001	010	011	

OCTONARY ADDRESS OF CHARACTERS ON CHARACTER MATRIX

CHAR	x	y	CHAR	x	y	CHAR	x	y	CHAR	x	y
A	1	2	Q	4	7	g	1	6	2	0	1
B	2	7	R	5	0	h	0	6	3	1	1
C	2	1	S	2	0	i	4	2	4	6	0
D	5	2	T	0	3	m	5	5	5	7	0
E	2	2	U	0	2	n	3	5	6	0	0
F	2	6	V	4	0	o	6	5	7	1	0
G	4	1	W	3	0	p	7	6	8	6	7
H	1	7	X	6	3	r	3	2	9	0	7
I	7	2	Y	1	3	s	1	5	●	7	7
J	3	1	Z	3	7	t	2	3	↓	2	4
K	5	6	a	6	6	u	4	3	↑	1	4
L	5	1	b	4	6	v	3	3	—	0	4
M	5	7	c	4	5	w	7	5	▼	7	4
N	7	3	d	3	6	x	3	4	▲	6	4
O	6	1	e	0	5	z	2	5	■	5	4
P	6	2	f	5	3	1	7	1	BLANK	4	4

Figure 4–25. Digital Display Tube Character Matrix and Octonary Addresses

6.4 PROGRAM INSTRUCTIONS

6.4.1 Situation Displays

6.4.1.1 RD and TD Drum Field Selection

The information presented to the SDGE consists of both RD and TD messages. The SDGE combines this information for presentation to the display consoles by reading all the radar data fields, then all the track data fields, etc. There are six track data fields and nine radar data fields, so the situation display system is capable of reading and automatically switching between 14 drum fields (one of the RD fields cannot be read during a reading cycle; this is the one which is being written on by the Central Computer System). However, when writing on the TD and RD fields, there is no automatic switching between fields, and a separate SDR code exists for every drum field of the RD and TD drums. The execution of these codes will select the specified drum field for a writing operation only; as explained, the reading is done automatically by the Display System. The six TD drum fields are selected by execution of an *SDR (41) – (46)* which will select TD fields 1-6, respectively. The RD fields are selected by execution of an *SDR (60) – (67), (70)* which will select RD fields 1-9, respectively.

6.4.1.2 RD Scan Counter

One feature of the RD drums is that we cannot read the field on which we are placing new information. This results in reading only eight RD fields at one time. The method used to pinpoint the skipped (not read) RD field is determined by the setting of a counter, called the scan counter. When set to 0, the scan counter will prohibit reading on field 1 and will cause fields 2 through 8 to be read. If the scan counter is set to 5, fields 6 through 9, and then fields 1 through 4, etc., will be read. Stepping of the scan counter is under program control. When it is desired to step the scan counter by 1 (causing a different field to be skipped and thus allowing writing on a new field), we execute a *PER (77)* instruction. When we wish to reset the scan counter, a *PER (76)* instruction is executed. The stepping of the scan counter when it is prohibiting reading of field 9 would result in its being reset; however, resetting the counter by executing a *PER (76)* instruction is more desirable because it is a more reliable operation.

6.4.1.3 Situation Display Test

When the Central Computer System wishes to compare information placed on the RD and TD drums, an *SDR (47)* instruction may be executed. This instruction will allow the Central Computer System to test-read (by identity only, of bits R5-R10) the entire 14 drum fields which make up the situation display information. Thus, the Central Computer System is simulating the operation of the Display System, when an IO transfer involving the use of *SDR (47)* is made. The normal testing sequence is for the Central Computer System to place a fixed pattern on the TD and RD fields and then to execute an *SDR (47)* and read this pattern back into memory to compare it.

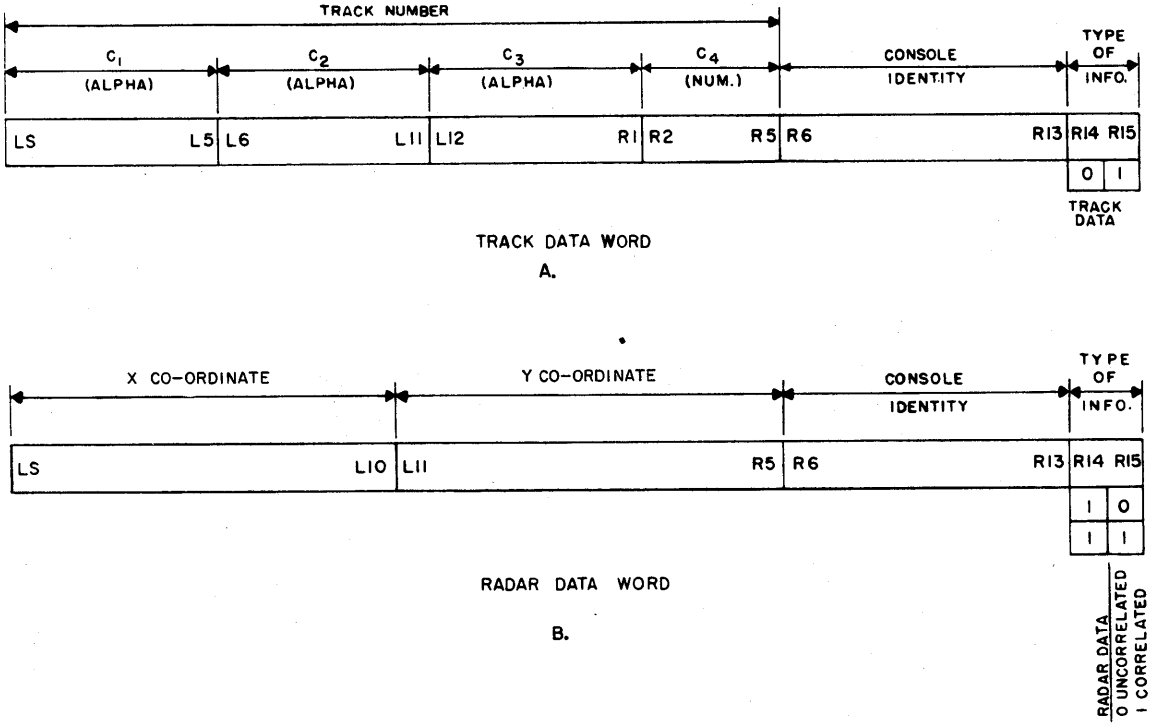


Figure 4–26. MDI Message Drum Field Layout

6.4.1.4 SD Camera Modes

There are cameras utilized with the AN/FSQ-7 and AN/FSQ-8 which may take pictures of a situation display for use in training and evaluation. These cameras may be operated manually or by program control. In addition, there are two modes of display available for the camera with a different mode taking a different type of picture. When we desire to operate the camera in mode I under program control, we can execute a *PER (31)*. If the desired mode of operation is mode II, we can execute a *PER (32)*. Execution of either of these two instructions will cause the camera to take one picture of the situation display. In addition, two codes are available for the use of more cameras, if so desired. These codes are *PER (33)* and *PER (34)* and are presently reserved for cameras 3 and 4. At the present time, these cameras are not in use in the AN/FSQ-7 or AN/FSQ-8. If we wish to take some action when the SD camera is being operated, we can sense its status by execution of a *BSN (35)* instruction. This instruction senses a flip-flop which is set when the camera is taking a picture. If the flip-flop is set, the *BSN (35)* will cause the program to branch to the location specified in the address portion of the instruction. Execution of a *BSN (35)* instruction does not clear the flip-flop.

6.4.1.5 Sense Display

If the Display System is displaying track data, a flip-flop is set, indicating this condition. We can examine this flip-flop by execution of a *BSN (37)* instruction. If track data is being displayed at the time a *BSN (37)* instruction is given, the program will branch to the memory location specified by the instruction. However, a *BSN (37)* instruction does not clear the flip-flop.

6.4.2 Digital Displays

6.4.2.1 DD Drum Field Selection

When the Central Computer System wishes to write information on the DD drum field, it selects this field by execution of an *SDR (27)* instruction. This instruction is valid for writing operations only.

6.4.2.2 DD Drum Field Test Selection

When the Central Computer System desires to test-read the DD drum field, it may select the DD field for this purpose by execution of an *SDR (17)* instruction. Execution of this instruction will prepare the DD drum field for reading into the Central Computer System by identity of bits R14-R15 only. This instruction enables the Central Computer System to simulate the action of the Display System.

6.4.2.3 Start Digital Display Sections (1 and 2)

Unlike the situation displays, which present a picture of the air defense situation every 2.54 seconds, the DD display presents information only upon request of an operator. Thus, it is necessary for the program to tell the digital display element when to generate a display. There are provisions for dividing the digital displays into two sections: the DD 1 section for rapidly changing information; and the DD 2 section for slower changing of the information. This would enable part of the digital displays to be changed, instead of changing all of them. At the present time, the DD 2 section is not in use in the AN/FSQ-7 or AN/FSQ-8 and is a spare. The code used to start generation of a digital display in section 1 is *PER (35)*, while the DD 2 section is started by execution of a *PER (36)* instruction.

6.4.3 Manual Inputs

6.4.3.1 MI Drum Field Selection

The MI drum field can be selected for reading by the Central Computer System by one of two methods. If the MI drum field is to be selected for reading by status, an *SDR (22)* instruction is given. On the other hand, if we wish to read the MI drum field by identity (bits R14-R15), an *SDR (23)* instruction is executed.

6.4.3.2 Area Discriminator Operation

The area discriminator may be operated in one of two modes, both selectable by program control. At the present time, only one mode of operation is in use with the area discriminator. When an area discriminator is operated by a *PER (20)* instruction, it initiates tracks detected by it over a period of 2.54 seconds (one display cycle). Then, it is turned off and is not operated again until another *PER (20)* instruction is executed. A *PER (17)* instruction is provided for operation of the area discriminator in another mode, although this code is not used at present.

6.4.3.3 Manual Input Matrix Selection

The MIM is selected for reading by the Central Computer System by execution of a *SEL (06)* instruction. A conventional IO transfer between the MIM and the Central Computer System may then take place. Since the MIM consists of 128_{10} computer words, the entire MIM is usually read by execution of an *RDS* 200_8 instruction. If any less than 200_8 words are read, the remainder of the information contained in the MIM is lost, since it is usually replaced with new information before another *SEL (06)* instruction is given.

6.4.4 Warning Lights

The warning light registers are replaced periodically by the contents of the core memory image. To do this, a standard IO transfer program must be executed, requiring that the warning light registers be selected. A *SEL (00)* instruction performs the selection, and selects the registers in the storage element for a writing operation only. As a rule, all 8_{10} warning light registers are replaced with the core memory image during an IO transfer, necessitating the execution of a *WRT* 10_8 instruction.

CHAPTER 7
PROGRAMMING THE MARGINAL CHECKING SYSTEM

7.1 INTRODUCTION

The marginal checking facilities built into AN/FSQ-7 and AN/FSQ-8 are used to provide a high degree of equipment reliability. Marginal checking is performed by varying a supply voltage to a group of circuits. The variation of the supply voltage is called an excursion. The magnitude of an excursion necessary to produce operational circuit failure is called a margin.

The circuit margin can be determined by varying an applied voltage at the same time that a test program is being run. In order to derive the maximum benefit from this type of testing, the program used must be designed to fully test the equipment covered by the marginal checking test program.

The probability of circuit malfunction during normal operation is very high when the circuit margin decreases beyond a prescribed value. To improve reliability, those circuits that have decreased margins should be replaced. The probability of trouble-free performance until the next test program is very high if the circuit functions normally during the marginal checking program. There are three marginal checking elements in the AN/FSQ-7 and AN/FSQ-8. Two identical marginal checking elements are provided for duplex equipment, one for computer A and one for computer B. The third marginal checking element is provided for the simplex equipment. Marginal checking programs are normally performed on that section of the duplex equipment that is in the standby status. Marginal checking operations can be performed only on simplex units that are in the standby status.

Marginal checking is accomplished by varying an applied voltage while a test program is being run. If a voltage is varied, all circuits supplied by this voltage have the same excursion applied. Assume the voltage variation causes the gain of each circuit to which it is applied to decrease. Further, suppose that several of these circuits are combined to perform one operation. Although each circuit functions properly, the slight decrease of amplification through cascaded stages could produce an error indication. Thus, the test program would give an erroneous test indication.

In order to overcome the inherent disadvantages which arise when a system-wide excursion is applied, only one supply voltage to a single portion of the equipment is varied at any one time. Of the five d-c service voltages used for marginal checking, only one voltage can be varied at any one time. In each computer, the varied d-c voltage is applied to one of the eight marginal checking groups. Each of the equipment groups may be further subdivided into six circuit groups, and each of the circuit groups may be divided into six lines. The varied voltage may be applied to any combination of circuit groups and lines within the equipment group.

7.2 PROGRAMMED MARGINAL CHECKING

7.2.1 General

When the marginal checking system is being operated under program control, each marginal checking sequence is automatically initiated, as required by the program. The program also selects the marginal checking group, circuit group, line voltage, and the amplitude and polarity of the excursion. These selections are made by means of a marginal checking control word which is transferred from core memory to the live register of test memory. The bit assignments and the corresponding selections for the AN/FSQ-7 are shown in table 4–24. It should be noted that the marginal checking control word layout is only slightly different for the AN/FSQ-8. A binary code of 10 in the L1 and L2 of an AN/FSQ-8 control word will cause the program to continue from location 20000_8, rather than 37760_8. Also, a binary code of 0111 in bits RS through R3 will select only the XTL common element for checking, as the LRI and GFI elements are not included in the AN/FSQ-8.

7.2.2 Excursion Application

Once the control word for a marginal checking routine is placed in the live register by the marginal checking executive program, the excursion is initiated by the execution of the *PER (21)* instruction. This instruction delivers the control word to the marginal checking controls, removes any previously applied excursion, and generates a *control clear* command just prior to the program restart called for by the marginal checking control word. The delay between the execution of the *PER (21)* instruction and the generation of the *control clear* command varies between 300 and 500 ms. This interval could be used for further calculation but, since the delay time is variable, the *PER (21)* instruction is usually followed by a *Program Stop (HLT)* instruction, leaving the Central Computer System idle until the program restart occurs.

TABLE 4–24. MARGINAL CHECKING CONTROL WORD LAYOUT

BIT	SELECTION	CODE	ACTION
LS	Change or start excursion	1	Start excursion
		0	Change excursion
L1-L2	Restart after excursion applied	00	Load from drums
		01	Continue from 00000_8
		10	Continue from 20000_8
		11	Load from card reader
L3-L4	Restart after excursion removed	Same as L1-L2	
L5-L6	Time duration of excursion	00	Infinite
		01	3 seconds
		10	7 seconds
		11	30 seconds
L7	Polarity of excursion	0	Positive
		1	Negative
L8	Spare		Used by programmers for control decoding (safe limit). Does not affect MC system.
L9-L12	Excursion magnitude	0000	0 volts
		0001	10
		0010	12
		0011	14
		0100	16
		0101	18
		0110	20
		0111	25
		1000	30
		1001	35
		1010	40
		1011	50
		1100	60
		1101	70
		1110	85
		1111	100
L13-L15	Voltage group selection	001	+250 volts
		010	+150
		011	+90

TABLE 4–24. MARGINAL CHECKING CONTROL WORD LAYOUT (cont'd)

BIT	SELECTION	CODE	ACTION	
		100	—150	
		101	—300	
RS-R3	Marginal checking group	0001	1 Memory	
		0010	2 Arithmetic	
		0011	3 Program and control	
		0100	4 IO control	
		0101	5 Drums	
		0110	6 Displays	
		0111	7 LRI, GFI, XTL common	
		1000	8 Outputs	
		1001	Simplex	
R4-R9	Circuit group	100000	A	
		010000	B	
		001000	C	
		000100	D	
		000010	E	
		000001	F	
R10-R15	Lines	100000	1	Used for duplex marginal checking
		010000	2	
		001000	3	
		000100	4	
		000010	5	
		000001	6	
R10-R15	Simplex (RS-R3 = 1001)	100000	G	Used for simplex marginal checking
		010000	H	
		001000	J	
		000100	K	
		000010	L	
		000001	M	

The *control clear* command generated just before restart clears all computing and control flip-flops in the Central Computer System, allowing the restart program to begin from a known condition. Since the index registers are also cleared by this command (if they are to be used after the restart), their contents must be stored temporarily in core memory and then reset by the restart program itself.

7.2.3 Excursion Removal

Two separate *Operate* instructions are available for the removal of excursions. *PER (22)* removes excursions applied to duplex equipment groups; *PER (23)*

removes excursions applied to simplex circuit groups. After excursion removal is called for in the marginal checking word, each instruction generates a *control clear* command just prior to the restart. Since the delay between execution of either of these instructions and the generation of the *control clear* command is the same as for *PER (21),* the same cautions apply in continuing calculation after execution of either instruction; calculation may be continued for 300 ms after any of these instructions and then be terminated by an *HLT* instruction.

It is conceivable that some marginal checking programs will require margins on duplex equipment for some phases and on simplex equipment for other phases of the program. The application of these margins is accomplished by proper selection coding of the marginal checking word in conjunction with the *PER (21)* instruction. However, removal requires that a *PER (22)* instruction be used for duplex and a *PER (23)* instruction for simplex. This necessitates the placing of both of these instructions in the same program. When programming this type of routine, *PER (22)* and *(23)* should never be given in successive order since this causes two consecutive restarts, resulting in unpredictable operation. A method of discriminating between duplex and simplex margins should be included in this type of program, after which the applicable *PER* instruction to remove that margin may be given.

7.2.4 Excursion Detection

One of the functions of the *PER (21)* instruction is to turn on the appropriate marginal checking sense unit, indicating that either duplex or simplex excursions are applied. The *PER (22)* and *(23)* instructions turn off the duplex and simplex sense units, indicating that excursions have been removed.

The *BSN (20)* instruction provides a means of sensing the duplex marginal checking sense unit to determine if duplex excursions are on or off. The branch will be executed, only if the sense unit is on, and will not affect the setting of the sense unit.

The *BSN (27)* instruction operates in the same manner for simplex equipment and has its own marginal checking sense unit. Again, the branch will be executed only when the sense unit is on. The *BSN (20)* and *(27)* instructions provide a means of discriminating between margins applied to duplex and simplex equipment.

7.2.5 Use of the LS Bit

The presence of a 1 in the left sign of a marginal checking word indicates that it is a start excursion word. When the *PER (21)* instruction is executed, any previous excursion and voltage, equipment group, circuit group, and line selection will be removed. The new selection, as coded in the marginal checking word, will be made. In this type of word, all information pertinent to the margins to be applied must be specified.

Having made the desired selections by means of a start word, it is possible to alter the excursion magnitude, polarity, and the mode of restart by using a change word after the excursion is applied. A change word is specified by a 0 in the LS bit position. The change word does not have provisions for selecting equipment groups, voltage groups, circuit groups, or lines, nor does it cause former selections to be removed. Therefore, any polarity or magnitude changes which have been made affect the previously selected equipment and can only be made after a start word has been used to specify the desired selections.

The duration timer is restarted from time 0 by a change word, but selection of duration is not affected. The same timer setting that was specified by the preceding start word will remain selected and will start retiming the new excursion from time 0. If the time duration, specified by the start word, expires, causing a *control clear* command, the next change word or words will be useless. This is also true following a programmed *PER (22)* or *(23)* instruction; a change word can be used only if excursions are still applied to the selected equipment.

When using a change word, the excursion magnitude, polarity, and restart (L1 and L2) must be specified, with all other bits given as 0. This is true even when some information affected by a change word is to remain the same. Since a change word requires approximately half the time required for a start word, the change word may prove advantageous when time is a major problem.

7.2.6 Restarts

Bits L1 and L2 designate restart after excursion applied, and bits L3 and L4 designate restart after excursion removed. These restarts are necessary to maintain program control. They are initiated after margins have been applied or removed, and the *control clear* has been generated. The restart after excursion applied is initiated by the *PER (21)* instruction. Restart after removal is initiated by the *PER (22)* or *(23)* instruction, or by the duration timer running down to 0. In either case, there are four possible modes of restart.

The first of these is the load-from-drums mode in which the first 30_8 words from auxiliary memory 1 are read into core memory locations 0.00000 through 0.00027. The program will then continue from location 0.00000_8 with the first instruction loaded from the drum field. The continue from 0.00000 mode causes the program to start from core memory location 0.00000 after the excursion has been applied or removed. Continue from 3.77760_8 causes the program counter to be set to 3.77760_8, the first location in test memory from which

the Central Computer System will obtain its first instruction after the restart. It should be remembered that an address of 0.20000 is used to designate the first test memory address in the AN/FSQ-8.

The load-from-card-reader mode is equivalent to depressing the LOAD FROM CARD READER pushbutton on the duplex maintenance console. A break-in operation is initiated, and one card of 30_8 binary words is loaded into core memory location 0.00000 through 0.00027_8. The program then continues from memory location 0.00000. Since these four modes are available for both restarts, neither of which is dependent on the other, a great deal of versatility is obtainable through careful programming.

7.2.7 Time Duration

Bits L5 and L6 of the marginal checking word are devoted to selecting a desired time for an excursion to remain applied to selected equipment. There are four preset times available: infinite; 3 seconds; 7 seconds; and 30 seconds. The duration timers initiate a *control clear* command in the event that instructions *PER (22)* or *PER (23)* have not been reached by the end of the desired time. These timers provide a means of retaining program control by initiating an automatic restart after excursions are removed. The *PER (21)* instruction resets all timers to 0 prior to the selection of a timer by the marginal checking word.

7.2.8 Polarity

Marginal checking excursions may be applied with positive or negative polarity. Bit L7 specifies the polarity of the excursion to be applied as selected by the marginal checking control word. If this bit is 0, a positive excursion is called for; if the bit is a 1, a negative excursion is called for.

For certain selections of equipment, voltage circuit group, and line, only one excursion polarity can be applied. The safe limit information in marginal checking breakdown charts indicated whether an excursion of a particular polarity is permitted on a given line; if a positive excursion is not permitted, for example, the positive safe limit is indicated as 0V. Only if both safe limits have values other than 0 can the line tolerate excursions of either polarity.

7.2.9 Safe Limit

Due to component considerations, certain selections may not have their supply voltages subjected to excursion beyond definite limits. There are five of these limits, 0, 25, 50, 75, and 100V, referred to as safe limits. Although the duplex marginal checking word has provisions for coding in bit L8 for one of two of these limits (100 or 25V), the code will not directly affect the marginal checking system. The safe limit bit is for decoding purposes only and is used to indicate at what magnitude the excursion should be removed. This is primarily a time-saving feature since the system is interlocked to prevent the safe limit from being exceeded, regardless of the coding or mode of operation.

Assume, for example, that excursions are to be applied from 0 magnitude to failure to a line whose safe limit is 25V, and that no safe limit is coded in the marginal checking word or that the control program has no provision for recognizing safe limits. If no failure is incurred between 0 and a 25V excursion, the control program will continue to apply an excursion to that line. However, the magnitude of the excursion applied will not exceed 25V because of the safe limit feature of the marginal checking system. The 25V excursion will be applied eight more times until the 100V word is reached. Since the time required to apply an excursion is in the hundreds of milliseconds, a large amount of time is wasted in this type of operation. This time can be saved by providing the control program with a means of recognizing the safe limit and removing excursions upon reaching that limit. The safe limits may be obtained from the marginal checking breakdown charts.

7.2.10 Excursion Magnitude

There are 16 possible excursion magnitudes, ranging from 0 to 100V, which are coded in bits L9 through L12 of the marginal checking word. These bits are decoded by the marginal checking system and the proper magnitude of excursion is applied to the selected line. If, for some reason, the coded magnitude exceeds the safe limit for the selected line, only the safe limit will be applied. Control programs may be designed to increase the magnitude on consecutive passes. In this manner, margins may be applied from a prescribed point to failure or until the safe limit is reached.

7.3 INSTRUCTION SUMMARY

A summary of the instructions related to the various IO systems is given in table 4–25.

TABLE 4–25. SUMMARY OF IO INSTRUCTIONS

INSTRUCTION NAME	MNEMONIC NAME	OCTAL CODE	EXECUTION TIME	INDEXABLE	CAUSE OVERFLOW
Select	*SEL*	62-	12 μsec	No	No
Select Drums	*SDR*	61-	12 μsec	Yes	No

TABLE 4–25. SUMMARY OF IO INSTRUCTIONS (cont'd)

INSTRUCTION NAME	MNEMONIC NAME	OCTAL CODE	EXECUTION TIME	INDEXABLE	CAUSE OVERFLOW
Load IO					
Address Counter	*LDC*	600	6 μsec	Yes	No
Read	*RDS*	670	6 μsec	No	No
Write	*WRT*	674	6 μsec	No	No
Clear and Subtract					
Word Counter	*CSW*	020	6 μsec	No	No

PART 5
ADVANCED PROGRAMMING METHODS

CHAPTER 1
UTILITY SYSTEMS

1.1 GENERAL

Utility systems are groups of programs and subroutines which perform some function for other programs. A utility system program is generally loaded into core memory and left there while the programs it serves are loaded and executed. Thus, the utility programs and subroutines can be easily referenced to by any program that desires to make use of them.

An example of a utility program is one which translates information from cards punched in Hollerith code into binary code and which then stores this information in core memory. Another type of utility program would be one which examined a program and assigned an absolute location to each instruction in the program. It should be remembered that both of these programs are only part of a utility system; they can be operated independently of one another but cannot perform the entire service that a particular program may need.

1.2 LINCOLN UTILITY SYSTEM

Instead of discussing a utility system not closely related to the AN/FSQ-7 or AN/FSQ-8 operation, or a hypothetical system, it was decided to describe, in general, a working utility system. The system chosen is known as the Lincoln Utility System because it was written by personnel from the Lincoln Laboratory of the Massachusetts Institute of Technology. The reader should keep in mind that the Lincoln Utility System is described and discussed in the following paragraphs only as a typical example of a large utility system. The discussion is not intended to provide the theory of operation of the Lincoln Utility System, in particular, but rather is designed to acquaint interested personnel with a general idea of the concepts involved in the application of any large utility system.

1.2.1 Description and Purpose

The Lincoln Utility System is a group of programs comprising about 60,000 separate instructions. Its main purpose is to aid programmers responsible for the writing and testing of the air defense programs used in the Direction Center Active (DCA) and Control Center Active (CCA) master air defense programs. These programs are run in the active computers at the various AN/FSQ-7 and AN/FSQ-8 sites.

Some of the functions performed by the various utility programs are as follows:

1. The Lincoln Compiler will translate a deck of punched cards containing Hollerith-coded instruction words into standard binary form used by the AN/FSQ-7 and AN/FSQ-8. The compiler will produce either a tape record or a binary deck which can later be read back into the computer and operated. Thus, the compiler serves mainly to translate information from a language used by the programmer into one which can be interpreted by the computer.

2. The Read-In program and the Table Simulation program are used to place on the drums a program and the test data it may need. This program may then be transferred from the drum to core memory and operated in a manner similar to the way it would be used in the air defense program.

3. The Storage Print program will provide a printout of various drum areas for analysis. The Checker will operate and interpret a program step by step, printing out errors which may cause program malfunction. Minor changes to a program already loaded on the drums may be made by a program known as the Octal Load program. These three programs are used mainly to troubleshoot or correct a program already loaded into the computer.

4. Several programs which constitute part of the utility system are used to assist in the maintenance and modification of the utility system. These programs also assist in the loading and processing of tapes.

5. The Utility Control program is the program which controls the overall operation of the utility system and performs functions which aid computer operations. This program is of the "executive" type; it does no actual data processing but, instead, controls the sequencing and correct execution of the other programs in the Lincoln Utility System. Naturally, not all of the programs which comprise the overall Lincoln Utility System have been covered; examples have been given of only some of the things the Lincoln Utility System can do.

1.2.2 Operation

All of the programs which make up the Lincoln Utility System are stored on one logical tape. A "logical" tape is one which contains a series of consecutive, related programs or files which may be loaded on the same physical tape reel with other "logical" tapes. To operate the utility system, the Utility Control program is first brought into core memory from the logical tape. Then, the programmer communicates his requests to the Utility Control program either by means of push-buttons on a console, known as the Utility Control console, or by placing coded words into punched cards which can be read in through the card reader. When the Utility Control program is ready to act on the request it has been given, it reads the contents of the manual input matrix which has been set by depression of various buttons on the Utility Control console or has been set by reading in the coded cards from the card reader. The Utility Control program then initiates the requested action by reading in another utility program from the tape and branching to it or by performing the requested function itself.

After a utility program has finished its task, control is returned to the Utility Control program which then prepares to accept the next operator request. This method allows continuous, sequential operation of the Lincoln Utility System.

To ensure the most rapid and efficient operation, it is desirable for us to keep at least a part of the Utility Control program permanently in memory. This portion of the Utility Control program is contained in memory locations 0.00000 through 0.00225 and contains storage areas for vital information as well as program instructions. Each of the other utility programs which may be called into core memory to perform some task is constructed so as not to destroy the permanent portion of the Utility Control program. By the same reasoning, whenever we operate one of the air defense programs under control of the utility system, care must be taken to see that this program will not damage the permanent portion of the Utility Control program and will not interrupt operation of the overall utility system. One other feature of the utility system which plays an important part in the successful operation of the overall system is core image. As we have indicated, the prime purpose of the Lincoln Utility System is to test an air defense program. However, a single utility program which is to perform some operation of a given air defense program may occupy much of core memory, and the air defense program may be equally large. It is then necessary for the utility system to time-share its own program with the program being tested. To allow this, the utility system combines four auxiliary drums to form what is known as Core Image. Each drum register corresponds to a core memory location and serves the purpose of storing either an air defense program or the results of that program for inspection and modification by the Utility Control program. (Core Image was designed into the Lincoln Utility System at a time when an expanded memory unit was not available for use. Today, Core Image would not be necessary for proper operation of the Lincoln Utility System; however, it illustrates how the problem of limited memory space might be dealt with in another utility system.)

This, basically, is the way the overall operation of the Lincoln Utility System is performed. Most utility systems operate in similar fashion by using a control program which selects the proper utility programs, performs the required tasks, and proceeds as rapidly as possible to the next request.

One of the outstanding features of the Lincoln Utility System is the large number of automatic coding devices used. Automatic coding may be defined as the interpretation of a program written in a language which is convenient for the programmer. This automatic coding is performed by another program called a Compiler or Assembly program whose function is to translate symbolic expressions into binary form. The use of symbolic expressions and how they serve as a means of communication within a program and between other programs is discussed in the following paragraphs.

1.2.3 Internal Program Communication

1.2.3.1 Internal Tags

The most simple application of a symbolic expression in a computer is its use in representing an absolute address. It is often much easier to write a symbolic address than it is to determine the exact numerical address of the register involved. As a rule, the programmer need not be concerned with specific addresses of words except those to which he makes direct reference in his program. For example, if he wished to branch to a certain address or to store a result in a designated register, or to use a specific word as operand, he could express the addresses of all of these things by a system of symbols.

Specifically, a symbolic address might be used in the following manner. Assume a situation in which we wish to write an instruction directing the program to *BFZ* to 02A. It does not matter what actual address the term 02A represents, as long as we know what instruction to branch to and designate its address as 02A. For instance, if we wished to *CAD* 065, provided the branch condition of the *BFZ* instruction was satisfied, we would simply give the instruction *CAD* 065 an address of 02A. The instructions would thus appear as follows:

BFZ 02A
02A *CAD* 065

When the program containing these instructions is loaded into the computer, the Compiler program will determine what absolute address the instruction at 02A has. Then, wherever the compiler encounters a symbolic address of 02A in the program, it will automatically substitute the actual memory location that 02A was determined to be. Constants as well as instructions may be referred to by these symbolic expressions or "tags" as they are commonly called. One of the primary advantages of using symbolic expressions is that insertions and deletions may be made to a program without rearranging the entire program, as would be the case if it has been written using actual numerical addresses.

Symbolic expressions can also give directions to a compiler program such as: "generate a storage block of 100 registers with all registers being cleared to positive 0", or "skip the next 20 registers."

Using the medium of a compiling program it is possible to translate into binary form any number of symbolic terms that we desire. By choosing a vocabulary of symbolic expressions which make the programmer's task of expressing his program as simple and as easy as possible, we do two things:

1. Reduce the number of programming mistakes
2. Make more time available to the programmer for concentration on his specific program by reducing the amount of time needed to express it.

1.2.3.2 Pseudo Instructions

Pseudo instructions are those instructions which cannot be interpreted directly by the computer but can be interpreted by another program such as the compiler. These instructions are usually used to manipulate data so that it may be directly stored or processed by a following legal instruction which the Computer program would recognize. For example, if we wished to place an *FCL* instruction into a program at a certain point but did not know exactly how many cycling positions should be specified, we could insert a pseudo instruction which would direct the compiler to insert an *FCL* instruction in place of the pseudo instruction and calculate the correct number of cycles required in the right half-word of the *FCL* instruction by analyzing what the program had done up to the point at which it encountered the pseudo instruction.

1.2.3.3 Location Tags

Location tags are defined as symbolic addresses by which the programmer can refer to words within his own program. Location tags used with the Lincoln Utility System are expressed by a combination of two decimal digits plus any letter. For example, the tag 02A, which we used earlier, is a location tag. In this case, the location tag referred to a core memory address, but it can also act as the right half-word of a constant and can serve various other purposes.

The structure of the location tag is intended to allow for the organization of a program into logical sections. For instance, the two digits can be used to denote a region or block of related instructions, while the letter can be used to differentiate among instructions within the block. Only those words which are referred to by other words need to be tagged; however, it is permissible to tag a word which is not referred to elsewhere in the program. This is especially helpful when it is desired to locate and identify various logical blocks within a program by tagging the first word of a block.

A particular location tag can be used to signify the location of only one word in any one program, but it can be used as the right half-word of as many instructions as necessary. If a location tag is used to specify an address in the right half-word of an instruction but does not appear next to the instruction or constant that it should, it is known as an unassigned tag and will be printed out by the compiler, as such. In this way, the programmer can check to make sure that every location tag used does actually refer to something.

1.2.3.4 RC Tags

RC tags are symbolic expressions by which the programmer can refer to a word without assigning that word any particular location in a program. It is sometimes more convenient to designate the word we desire rather than to indicate a location for the word. For example, if we wish to *CAD* a constant of negative 0, we might use an RC tag as follows: This is read as *"Clear and Add"* a register containing 1.77777, 1.77777. Thus, RC is simply an abbreviation for the words "register containing." When the compiler encounters an RC tag, it will assign the word to a location at the end of the program and will later substitute this location for the RC tag.

1.2.3.5 Temporary Storage Tags

Temporary storage tags are symbolic expressions by which we can refer to a temporary storage register

without assigning it a specific location in our program. A temporary storage tag is composed of the letter T followed by three decimal digits. If the three-digit number following the T contains leading 0's, they may be omitted. For example, the expressions T004 and T4 refer to the same temporary storage location. In the Lincoln Utility System, the temporary storage tag containing the largest number is used to automatically determine how many temporary storage locations will be provided at the end of the program. Thus, the programmer should always use consecutively numbered T-numbers, beginning with 0.

1.2.4 Communication between Programs

Since the overall air defense program is so large that all of it plus the data it uses cannot be stored in core memory at one time, some means of communication must exist among the various individual programs or subroutines. This communication is achieved by storing the information generated by one program in tables for use by another program. These tables are given tags (called communication tags) to simplify the job of addressing them when they are being run in conjunction with the utility system.

1.2.4.1 System Tables

System Tables consist of registers containing one of the following types of data:

1. Data on which memory space is required during one cycle (frame) or less of the overall air defense problem.
2. Data to be communicated between individual programs within a subframe.
3. Data received from input equipment or data for output or display equipment.

System Tables are classified into categories depending on their use and function in the program. The different classifications and their definitions are as follows:

1. Central Tables – Tables used by two or more program groups.
2. Communication Registers – A special type of Central Table which, in general, provide one-way communication from one program to another.
3. Isolable Tables – Tables used by one program or program group only.
4. Peripheral Tables – Tables representing an image of an input, output, or display format.

1.2.4.2 Items

Any independent piece of data constitutes an item. Items may range from 1 to 32 bits in size and are of two types: tagged and tagless. Tagged items are specified by unique four-letter designations, called item tags. Tagless items are specified by name, table block, and word location. The location of tagless items within a particular table block is guaranteed, whereas the position of a tagged item is not guaranteed because the programmers cannot assume a guaranteed location and because the tagged item must be treated using the conventions of the Lincoln Utility System.

1.2.4.3 Table Tags

Each table is identified by the use of a unique designation consisting of three letters, known as the table tag. The table is addressed for programming purposes by means of the Table Block Tag which consists of the three-letter table tag just mentioned, followed by a digit representing the block number.

1.2.5 Communications Tags

Up to this point, we have discussed symbolic expressions which deal with one program and symbolic expressions which deal with more than one program. While internal tags are used to specify locations within one given program, a communications tag is used to refer to other programs or tables. This distinction between internal and communications tags is useful because of the way in which the tags are treated in the input process. Internal tags are translated by the compiler into numerical locations related to the first location occupied by the program and are adjusted at the time of read-in. Communication tags, on the other hand, are translated at the time of read-in by reference to a tag pool common to all programs. This tag pool is known as the compool (Communications Tag Pool).

1.2.5.1 Compool

The compool is maintained by an external agency and contains all of the information needed by the Lincoln Utility System to process the tags used in the air defense program. Because the compool is not maintained by any internal program, any changes in the tag pool are passed on to all affected programs at the time of read-in. For example, suppose a table of data was reassigned to a new storage location. The revision of a single value in the tag pool (giving the new starting address of the table) will automatically correct all programs which have reference to the table, making it unnecessary to alter words in the individual programs. The use of a compool in any utility system reduces the amount of effort that is involved in rearranging tables and simplifies the writing of a program since the programmer need not know the exact location of tables within the overall air defense program in order to make reference to them.

1.2.5.2 Item Tags

Many tables in the air defense program will have blocks in which each register contains several items. The item tag provides for obtaining an item by code designation, positioning it as desired in the accumulators, mask-

ing out all bits except those in the selected item, and redepositing the item in its original location. It is not necessary for the programmer to know how many bits comprise the item, or what manipulation is required to get it into position, nor how to store the masks for extracting and depositing the item. An item tag is made up of four letters with the sole restriction that it can not start or end with the letter O.

1.2.5.3 Parameter Tags

Parameter tags are used to refer to constants which are used by several different programs. These tags are expressed as two letters plus a decimal digit. For any parameter thus indicated, the compiler will automatically assign an available register at the end of the program. A parameter value can later be obtained from a central source. An example of the use of a parameter tag is the expression:

CAD RB1

In this case, RB1 indicates an absolute memory location, but a parameter tag may also be used as the right halfword of a constant or as the right half of an RC word.

1.2.6 Summary of Symbolic Expressions

The above paragraphs describing different types of expressions and references which are coded in symbolic form for use with the Lincoln Utility System have attempted to show the versatility that can be achieved by the use of these symbolic expressions on tags. The reader should not try to familiarize himself with details of these tags but should gain an understanding of how tags are used both for expressions internal to one program (internal tags) and for expressions relating to several programs (communication tags). The use of tags within any given utility system is highly flexible: this usage permits rapid processing of programs with a minimum amount of coding by the programmer.

CHAPTER 2
SCALING

2.1 INTRODUCTION

2.1.1 General

Digital computers manipulate only pure numbers. Therefore, when numbers representing physical quantities are used as operands in a digital computer, the programmer must keep a record of the units in which the quantities have been measured. For example, in computing a velocity, the programmer must be aware that the quantities representing distance and time are expressed in miles and hours, respectively, in order to conclude that the result of the computation expresses velocity in miles per hour. Fixed-point digital computers, so called because the radix points of the numbers remain in arbitrarily fixed positions throughout the calculation, require that the programmer manipulate the actual magnitudes of the numbers. In contrast, floating-point digital computers reposition the point throughout the computations, thus automatically arriving at results of correct magnitude without any action by the programmer. This advantage of floating-point computation is offset by the disadvantage of using digits within the computer words to indicate the position of the point, thereby reducing the precision that can be achieved with a given word size.

The Central Computer System is designed to perform fixed-point computation with numbers having a binary point between the sign bit and the first magnitude bit. The magnitude of the number designated by a half-word is restricted to fractional values lying between $+1$ and -1. All numbers representing physical quantities must be transformed for use in the Central Computer System into equivalent fractional expressions by the technique of scaling. This technique involves separating the number into two parts. One part consists of the significant digits of the number reduced in magnitude to the range which can be handled by the Central Computer System. The other part consists of a factor, which must be handled by the programmer, indicating the magnitude of the original number. Thus, scaling is an operation performed by the programmer which allows insertion of the significant digits of a number into the Central Computer System and which leaves the magnitude (the true position of the radix point) and the units of measurement for manipulation by the programmer.

2.1.2 Pure Numbers and Physical Measurements

When a physical quantity is measured, the number generated represents the amount of specific units contained in this quantity. For example, the statement that the distance between New York and Chicago is 1,000 miles expresses the fact that the distance has been measured and found to contain approximately one thousand units of distance. However, the number 1,000 does not reveal how precisely the measurement has been made, because the three zeros might indicate magnitude and not precision. If the measurement is made to the nearest mile, then all four digits are significant digits. However, if the measurement is correct to the nearest hundred miles, only the first two digits are significant. The last two zeros are included to fill up the space to the decimal point whose position indicates the magnitude of the number. To avoid this ambiguity, numbers representing measurements of physical quantities may be written in scientific notation. This method of notation separates the number into two parts; one part consists of the significant digits, and the other part is a factor indicating the magnitude of the number. The point is arbitrarily positioned in the group of significant digits, usually after the first significant digit. For example, the velocity of light written in this manner is expressed as $2.998 \cdot 10^8$ meters per second. This notation distinguishes between those digits signifying the precision of the measurement (2998) and those signifying the magnitude of the number. The latter, 10^8, is called the scale factor. The scale factor remains unchanged once the position of the point is fixed regardless of how many significant figures the original measurement provides.

2.1.3 Scaling for Central Computer System

2.1.3.1 Principle of Scaling

When numbers representing physical quantities are scaled for use in the Central Computer System, the technique of scientific notation is applied, except that the binary point is placed to the left of the first significant digit. The scale factor is always manipulated by the programmer outside the machine. In general, the relation between x (the number representing the original quantity), x (its fractional representation within the Central Computer System), and 2^q (the scale factor), is given by the equation:

$$x = 2^q \cdot x$$

The smallest power of 2 which is just greater than x is normally chosen as the scale factor (2^q). The exponent q is the number of places that the true value x must be shifted to the right (if x is larger than 1) in order to obtain a fractional representation x with no leading zeros (i.e., x representing x with maximum precision).

2.1.3.2 Scaling a Constant

To scale the number representing the velocity of light for use in the Central Computer System, the scaling equation may be set up as follows:

$$2.998 \cdot 10^8 = .2998 \cdot 10^9 = 2^q \cdot x$$

If 10^9 were equal to an integral power of 2, it would be possible to use the decimal scale factor or its binary equivalent for manipulation by the programmer, and to convert only the significant figures into the fractional binary representation acceptable by the Central Computer System. Since 10^9 cannot be expressed as an integral power of 2, one of two methods using the entire number must be chosen to obtain both the correct scale factor and the fractional representation of the number. The first method consists of direct conversion of the decimal number into binary form.

The resulting binary number is:

10 001 110 111 101 001 010 111 000 000

When scaling this number, the binary point must be moved 29 places to obtain the fraction:

0.100 011 101 111 010 010 101 110 000 00

In order to indicate the original magnitude of this number, it must be multiplied by a scale factor of 2^{29}. Note that this power is equal to 536,870,912 which meets the requirement that the scale factor is that power of 2 which is just greater than the original quantity x. Because the length of the scaled binary number exceeds the length of a half word, the number is rounded off to 15 bits. The final scaling equation thus takes this form:

$$2.998 \cdot 10^8 = 2^{29} \cdot 0.100\ 001\ 101\ 111\ 010$$

The second method converts the decimal number into an octonary number which in turn is converted into the equivalent binary number. Although an additional step is required in this method, the actual arithmetic operations are considerably reduced, thus lessening the possibilities of arithmetical mistakes during the direct conversion to binary form. The scaling equation is set up as follows:

$$2.998 \cdot 10^8 = 2998 \cdot 10^5 = 2^q \cdot x$$

Writing the original quantity in this manner yields:

$$2998_{10} = 5666_8$$
$$10^5 = 303{,}240_8$$

Multiplication of these octonary terms gives the following result:

```
   303240
     5666
  -------
  2223700
 2223700
2223700
1720440
-------
2167512700
```

This number is converted by inspection to:

010 001 110 111 101 001 010 111 000 000

This result is identical to the direct binary conversion.

2.1.3.3 Scaling of Variable Numbers

When scaling a constant number for use in the Central Computer System, the only consideration is maximum precision of representation. Therefore, the scale factor used is the one which eliminates leading zeros (zeros between the sign bit and the significant bits) in the fractional representation of the number. If each value of a variable number were scaled for maximum precision, it would be necessary to provide a different scale factor for each value, thus complicating the programmed manipulation of the variable. Since program simplification is a prime consideration of the programmer, one scale factor must be found to accommodate all values that the variable can assume. The only scale factor that satisfies this requirement is the one associated with the maximum value of the variable. If the scale factor for a value smaller than the maximum were chosen, representations greater than unity would result for those values of the variable greater than the selected value.

To illustrate the scaling of a variable, assume that a distance x can take any value between 2 and 4,000 miles. To scale this variable, the scaling equation is set up as follows:

$$2 \leqq x \leqq 4000 = 2^q \cdot x$$

Since the scale factor must be greater than the maximum value of the variable, 2^q must be greater than 4000. The integral power of 2 just greater than 4000 is 2^{12} which must be used as the scale factor for the variable or:

$$2^q > 4000$$
$$2^q = 2^{12} = 4096$$

2.2 ARITHMETIC REQUIREMENTS FOR SCALING

2.2.1 General

The scaling of constant and variable numbers discussed in the preceding paragraphs satisfies the requirements for insertion of the numbers into the Central Computer System but not necessarily the requirements for arithmetic manipulation. These manipulations place additional restrictions on scaling. The scaling restric-

tions, which depend largely on the type of arithmetic operation, have the primary objective of correctly scaling the result of the arithmetic manipulations, preferably to maximum precision.

2.2.2 Addition and Subtraction

2.2.2.1 Requirements

Just as in addition or subtraction of decimal numbers, the decimal points of the numbers must be aligned to obtain a valid result, so the binary points of two numbers to be added or subtracted in the Central Computer System must be positioned identically in each number. Because the binary point is positioned by the scale factor, it follows that, for addition or subtraction, the scale factors of the two numbers involved must be identical. A second requirement in scaling numbers for addition or subtraction is imposed by the inability of the Central Computer System to interpret correctly any number equal to or in excess of unity. Care must be taken that the result of an addition or subtraction does not produce this condition (overflow). Thus, the numbers involved in addition or subtraction must be scaled not only to fit into the machine but also to insure that the result of the operation is always fractional. On the other hand, the result should contain a minimum of leading zeros to provide maximum precision. In summary, then, two numbers being added or subtracted in the Central Computer System must have the same scale factor which is large enough to permit insertion of the original numbers in the machine and to prevent overflow, yet small enough to afford maximum precision.

2.2.2.2 Numerical Example

The following numerical example illustrates scaling of two binary numbers for addition in the Central Computer System. Assume that the numbers $111\ 101_2$ and $11\ 001_2$ are to be added. If each of these numbers were considered independently for insertion in the machine, the scale factors chosen would be 2^6 and 2^5, respectively. However, the two numbers are to be added. Therefore, both numbers must have the same scale factor. If 2^6 were used, the result would be greater than unity. Therefore, 2^7 is selected as the scale factor. The first number becomes:

$$0.011\ 110\ 1 \cdot 2^7$$

the second number becomes:

$$0.001\ 100\ 1 \cdot 2^7$$

Adding these numbers gives this result:

$$\begin{array}{r} 0.011\ 110\ 1 \\ +0.001\ 100\ 1 \\ \hline 0.101\ 011\ 0 \end{array}$$

The resultant number has the scale factor, 2^7, which provides the required maximum precision for this number. In this example, two numbers of like sign are added so that the magnitude of the result, which is greater than the original numbers, determines the scale factor. If two numbers of unlike sign are added or if a number is subtracted from a number of like sign, the result is smaller than either of the original numbers. In this case, the magnitude of the larger of the original numbers determines the scale factor. Although constants have been used in this example, the same considerations hold true for variables. The maximum absolute value (neglecting signs) of the variable must be taken into account. In general, the scale factor is determined by the upper bound on the absolute value of the largest variable in the calculation, regardless of whether it occurs in the original numbers or in the final result.

2.2.2.3 Other Considerations

If more than two numbers are to be added or subtracted in the Central Computer System, each partial sum or difference formed by two numbers can be treated as a new number to be added to or subtracted from a third number and scaled accordingly. This step by step procedure usually requires rescaling each partial sum or difference before performing the next operation. Rescaling of numbers already in the Central Computer System is accomplished by a shift instruction. To increase the scale factor, the number is shifted to the right; conversely, to decrease the scale factor, the number is shifted to the left. Although maximum precision of the result can be achieved by rescaling after each operation, the programmer must be aware that shifting takes up computer time. Therefore, an alternate method involves scaling all the numbers to be added or subtracted by the one scale factor which accommodates the largest upper bound encountered among the original numbers, the partial sums or differences, and the final result. The use of this constant scale factor results in a saving of computer time but a possible sacrifice of precision in the final result. Depending on the particular requirements of a problem, the programmer may compromise between saving computer time and precision by rescaling only those partial results which would produce the greatest loss of precision and by selecting the scale factors of the original numbers to minimize the amount of rescaling required.

2.2.3 Multiplication

2.2.3.1 Requirements

Multiplication in the Central Computer System of two numbers scaled to their fractional representations yields a sign bit plus a 31-bit fraction. Since the 31st bit duplicates the sign bit, this bit can always be considered as a true value zero and not as a significant bit. Generally, the product must be rounded off to a fraction of 15 significant bits, before the product is further manipulated in the Central Computer System. The

round-off operation is accomplished by an *SLR* instruction, which can also be used in scaling the product for further operations.

Multiplication of two binary numbers in the Central Computer System, unlike addition and subtraction, imposes no special restrictions on scaling the numbers. The numbers are scaled by finding the scale factor of the upper bound of each number. The resulting scale factor of the product can be found by adding the exponents of the scale factors of the two original numbers. Because the two numbers are scaled for maximum precision, the product is also represented with maximum precision. Therefore, the *SLR* instruction, which rounds off the product, is chosen so that no shift to the left is generated to prevent a possible loss of significant bits in the product.

2.2.3.2 Numerical Example

The following numerical example illustrates scaling for multiplication. Consider the product $x \cdot y = z$ with the following bounds on the absolute values of the terms:

$$|x| \leqq 250$$
$$|y| \leqq 50$$

Hence, the scale factors are:

$$x = x \cdot 2^8 \qquad (2^8 = 256)$$
$$y = y \cdot 2^6 \qquad (2^6 = 64)$$
$$z = x \cdot y = x \cdot y \cdot 2^{8+6} = z \cdot 2^{14}$$

Note that each scale factor chosen is the smallest integral power of 2 which is greater than the upper bound on the variable, resulting in a minimum of leading zeros in the fractional representation. The scale factor obtained for a z also satisfies this requirement if its upper bound is assumed to be the product of the upper bounds on x and y; i.e., $z < 12{,}500$ and $2^{14} = 16{,}384$.

It frequently occurs, however, that additional information is available which limits the upper bound on the product z to a value smaller than the product of the maximum bounds on x and y; i.e., the maximum values of the two original variables never occur simultaneously. Hence, the fractional representation z would contain leading zeros, resulting in a loss of precision. The *SLR* instruction can be used to rescale z by shifting the leading zeros out of the product in order to obtain maximum precision.

Assume that in the foregoing example where the absolute value of z is known never to exceed 4,000 ($|z| \leqq 4{,}000$). Therefore, the scale factor for z should be 2^{12} ($2^{12} = 4096$) for maximum precision. The scaling equation is written as follows:

$$x \cdot 2^8 \cdot y \cdot 2^6 = z \cdot 2^{12}$$

Division by 2^{12} gives:

$$x \cdot y \cdot 2^2 = z$$

This result indicates that the product of x and y must be rescaled by a shift of two places to the left by an *SLR* 2 instruction to obtain z with the desired scale factor of 2^{12}.

2.2.3.3 Other Considerations

If a certain scale factor of a product is desired, it is usually advisable to select the appropriate scale factors for the initial variables. However, in the event that the desired scale factor is smaller than the one obtained without rescaling but greater than the scale factor for maximum precision of the product, an *SLR* instruction can be used to rescale the product for the desired scale factor. If, in the preceding example, a scale factor of 2^{13} is desired, the scaling equation would be set up as follows:

$$x \cdot 2^8 \cdot y \cdot 2^6 = z \cdot 2^{13}$$

which reduces to:

$$x \cdot y \cdot 2^1 = z$$

indicating that a shift of one place to the left would be required. The appropriate instruction would be SLR 1 which would shift the product one place to the left before rounding off.

2.2.4 Division

2.2.4.1 Requirements

Although the only arithmetic restriction on division is the prohibition of dividing by zero, the design of the Central Computer System imposes another requirement which must be taken into account in scaling. This requirement demands that the divisor must always be greater in absolute magnitude than the dividend to prevent the quotient from becoming equal to or greater than unity. To satisfy this requirement, both the upper bound of the dividend and the lower bound of the divisor must be known by the programmer. This knowledge also allows the programmer to rescale the result for maximum precision.

The division operation places the sign bit for both the remainder and the quotient in the accumulator sign bit position, the 15 bits forming the remainder in the rest of the accumulator register, and the 16 bits of the quotient in the B register.

Comparison of the bit positions before and after the division reveals that the least significant bit of the remainder corresponds in magnitude to the 31st bit of the dividend, indicating that the remainder has been scaled up by 2^{16}. The first bit of the quotient corresponds in magnitude to the first bit of the dividend, indicating that the quotient has been scaled down by 2^{-15}. These changes in scale factor of the remainder and the quotient must be taken into account before further use is made of either one.

2.2.4.2 Numerical Example

The following example illustrates scaling for division. Given the equation $\frac{x}{y} = z$ with the following bounds.

$$|x| \leqq 1000$$

$$2 \leqq |y| \leqq 50$$

The upper bound on z is therefore:

$$\frac{1000}{2} = 500; \quad |z| \leqq 500$$

The scale factors for the numbers are determined as follows:

$$x = x \cdot 2^{10} \qquad (2^{10} = 1024)$$

$$y = y \cdot 2^{6} \qquad (2^{6} = 64)$$

$$z = z \cdot 2^{9} \qquad (2^{9} = 512)$$

The scaled equation is written as follows:

$$\frac{2^{10} \cdot x}{2^{6} \cdot y} = 2^{9} \cdot z$$

This equation reduces to:

$$\frac{x}{y} = 2^{5} \cdot z$$

This version of the scaling equation is not usable because the dividend is greater than the divisor; therefore, both sides of the equation are divided by 2^5:

$$\frac{2^{-5} \cdot x}{y} = z$$

Note that x is multiplied by 2^{-5} rather than y by 2^5 because multiplication of x by 2^5 is easily accomplished by a *DSR 5* instruction preceding the division operation. The *DSR 5* instruction shifts x five places to the right without loss of significant bits since 16 additional bit positions are available in the B register for the dividend. In addition, performance of this rescaling operation before the division operation satisfies the requirement that the divisor must be greater than the dividend.

2.2.4.3 Other Considerations

If the quotient z is to be used for further manipulations, the significant bits of z must be reunited with their sign bit by shifting them 15 places to the left, thereby cancelling the factor 2^{-15} introduced by the position of the quotient. The shift is accomplished by an *SLR* 17_8 instruction which also rounds off the quotient to 15 significant bits. The remainder, whose position z occupies after the shift, is thereby destroyed. The number of positions by which the quotient may be shifted left is sharply restricted. A shift greater than 17_8 places may lead to a loss of the most significant bits; a shift less than 17_8 places allows bits of the remainder to occupy the most significant bit positions of the quotient; either one is likely to produce an incorrect result. The restrictions on the use of the *SLR* instruction following a division operation render it relatively useless for rescaling, thus offering another reason for performing rescaling preceding the division operation.

If the remainder produced by a division operation rather than the quotient is to be used, no shifting is required, but the programmer must keep in mind that the scale factor of the remainder has been increased by 2^{16}.

As in multiplication, additional information may be available which fixes the upper bound on the quotient z to an absolute value smaller than the quotient of the upper bound of the dividend by the lower bound of the divisor. The existence of a smaller bound for z indicates the presence of leading zeros (equivalent to a loss in precision) in the fractional representation z. Unlike multiplication where a shift instruction can be used to rescale the fractional representation of the product, the restrictions on shifting the quotient bits make it advisable to take the smaller bound into account in the initial scaling equation. For example, if $|z| \leqq 100$ instead of 1,000 in the numerical example, the scaling equation reads:

$$\frac{2^{10} \cdot x}{2^{6} \cdot y} = 2^{7} \cdot z \qquad (2^{7} = 128)$$

This equation reduces to:

$$\frac{2^{-3} \cdot x}{y} = z$$

indicating that x has to be shifted to the right only 3 places instead of 5.

2.3 SCALING OF COMBINED OPERATIONS

2.3.1 General

The following numerical examples illustrate the application of scaling techniques to combined operations and the incorporation of these techniques into the programs for these operations. The first example shows scaling for maximum precision of a combined addition and subtraction of several terms. The same example is also discussed with constant scaling. In the second example, scaling for combined multiplication and division is demonstrated, and the final example shows an application of scaling to a combination of multiplication and addition.

2.3.2 Addition and Subtraction of Several Numbers

2.3.2.1 Maximum Precision Scaling

The following four variables are to be added and subtracted as specified by the equation:

$$T = u + v - w + z$$

The upper bounds on the variables are given as follows:

$$|u| \leqq 200$$

$$|v| \leqq 300$$

$$|w| \leqq 250$$

$$|z| \leqq 700$$

The Central Computer System performs this operation by forming t^1, the partial sum of the first two variables; t_2, the difference of this partial sum and the third variable; and T, the sum of the difference and the fourth variable. This statement is expressed mathematically as:

$$u + v = t_1$$

$$t_1 - w = t_2$$

$$t_2 + z = T$$

Scaling may now be determined according to the requirements for addition and subtraction of two numbers. The following scale factors for the individual numbers are supplied for reference but are not necessarily the final scale factors used in the operation:

$$u = 2^8 \cdot u \qquad (2^8 = 256)$$

$$v = 2^9 \cdot v \qquad (2^9 = 512)$$

$$t_1 = 2^9 \cdot t_1$$

$$w = 2^8 \cdot w$$

$$t_2 = 2^9 \cdot t^2$$

$$z = 2^{10} \cdot z \qquad (2^{10} = 1024)$$

$$T = 2^{10} \cdot T$$

For the first equation a choice must be made between the scale factors 2^8 and 2^9. Only 2^9 is large enough to permit insertion of u and v in the machine as well as to prevent overflow of t_1, yet small enough to permit maximum precision for t_1. The scaled equation is written as follows:

$$2^9 \cdot u + 2^9 \cdot v = 2^9 \cdot t_1$$

which reduces to:

$$u + v = t_1$$

Reasoning along the same lines, 2^9 must be used throughout the second operation, even through the minimum scale factor for w can be 2^8; therefore, the scaled operation is written as follows:

$$2^9 \cdot t_1 - 2^9 \cdot w = 2^9 \cdot t_2$$

which reduces to:

$$t_1 \cdot w = t_2$$

To perform the third operation, t^2 must be rescaled to conform to the scale factors of z and T. The scaled equation is written as follows:

$$2^9 \cdot t_2 + 2^{10} \cdot z = 2^{10} \cdot T$$

Dividing the equation by the desired scale factor 2^{10} reduces the equation to the following:

$$2^{-1} \cdot t_2 + z = T$$

This result indicates that t_2 must be reduced in magnitude by multiplication by 2^{-1}, increasing its scale factor from 2^9 to 2^{10}. The multiplication by 2^{-1}, is accomplished by an *ASR 1* instruction which shifts t_2 one place to the right. The preceding scaling analysis shows that the original numbers must be scaled as follows to arrive at the correctly scaled total T:

$$u = 2^9 \cdot u$$

$$v = 2^9 \cdot v$$

$$w = 2^9 \cdot w$$

$$z = 2^{10} \cdot z$$

$$T = 2^{10} \cdot T$$

The resulting program is written as follows:

CAD (u)

ADD (v)

SUB (w)

ASR 1

ADD (z)

FST (T)

2.3.2.2 Constant Scaling

If in the preceding operation it is more desirable to save computer time rather than to obtain a result with maximum precision, the shift operation can be eliminated by the choice of the constant scale factor 2^{10} throughout the entire operation. In general, the programmer must perform the scaling analysis as shown, and, depending upon the requirements of the individual problem, decide whether it is preferable to use a constant scale factor or to incorporate shifts for a result with maximum precision.

2.3.3 Multiplication and Division of Several Numbers

The following variables are to be multiplied and divided as specified by the equation:

$$T = \frac{x \cdot y \cdot z}{u \cdot w}$$

The absolute bounds of the variables are given as follows:

$$|x| \leqq 100$$
$$|y| \leqq 50$$
$$|z| \leqq 70$$
$$5 \leqq |u| \leqq 40$$
$$10 \leqq |w| \leqq 20$$

Hence, the implied upper bound for T is:

$$|T| \leqq 7000$$

The operation is performed by the Central Computer System by forming t_1, the quotient of x and u; then t_2, the product of t_1 and y; then t_3, the quotient of t_2 w; and finally T, the product of t_3 and z. (An alternate method is the formation of the products of u and w, and of x, y, and z followed by the division of the second by the first. However, the method presented offers advantages in scaling and requires approximately the same amount of computer time as the alternate method.) The operation is expressed mathematically as:

$$\frac{x}{u} = t_1 \qquad |t_1| \leqq 20$$
$$t_1 \cdot y = t_2 \qquad |t_2| \leqq 1000$$
$$\frac{t_2}{w} = t_3 \qquad |t_3| \leqq 100$$
$$t_3 \cdot z = T$$

The scaling analysis is performed to determine the amount of rescaling required, rather than to determine the scale factors for the original numbers as in addition and subtraction. The numbers are scaled for insertion into the Central Computer System with maximum precision by considering their upper bounds:

$$x = 2^7 \cdot x \qquad (2^7 = 128)$$
$$y = 2^6 \cdot y \qquad (2^6 = 64)$$
$$z = 2^7 \cdot z \qquad (2^7 = 128)$$
$$u = 2^6 \cdot u \qquad (2^6 = 64)$$
$$w = 2^5 \cdot w \qquad (2^5 = 32)$$
$$t_1 = 2^5 \cdot t_1 \qquad (2^5 = 32)$$
$$t_2 = 2^{10} \cdot t_2 \qquad (2^{10} = 1024)$$
$$t_3 = 2^7 \cdot t_3 \qquad (2^7 = 128)$$
$$T = 2^{13} \cdot T \qquad (2^{13} = 8192)$$

The first scaling equation is:

$$\frac{2^7 \cdot x}{2^6 \cdot u} = 2^5 \cdot t_1$$

which reduces to:

$$\frac{2^{-4} \cdot x}{u} = t_1$$

Therefore, the first rescaling operation is a shift of the dividend four places to the right. The second scaling equation is:

$$2^5 \cdot t_1 \cdot 2^6 \cdot y = 2^{10} \cdot t_2$$

In order to reconcile 2^{11}, the product of the scale factors on the left side of this equation, with 2^{10}, the scale factor required by the absolute bound of t_2, the product of t_1 and y must be multiplied by 2^1. This result is derived mathematically by dividing the equation by 2^{10}, yielding:

$$2^1 \cdot t_1 \cdot y = t_2$$

The rescaling multiplication can be performed by a shift of t_2 one place to the left. However, the division operation which follows may require a shift to the right. Therefore, the next scaling equation should be analyzed before a final decision about shifting t_2 is made. The third scaling equation is written as follows:

$$\frac{2^{10} \cdot t_2}{2^5 \cdot w} = 2^7 \cdot t_3$$

which reduces to the following:

$$\frac{2^{-2} \cdot t_2}{w} = t_3$$

The rescaling of the dividend requires a shift of two places to the right. However, this shift of two places to the right can be combined with the one-plate shift to the left from the previous step into a single shift of one place to the right; i.e., $2^1 \cdot 2^{-2} = 2^{-1}$.

The fourth scaling equation is:

$$2^7 \cdot t_3 \cdot 2^7 \cdot z = 2^{13} \cdot T$$

Again the product of the scale factors on the left of the equation must be reconciled with the scale factor required to express T with maximum precision. Therefore, the product is shifted one place to the left, since:

$$2^1 \cdot t_3 \cdot z = T$$

The resulting program is written as follows:

```
CAD  (x)
DSR  4
DVD  (u)
SLR  17₈
MUL  (y)
DSR  1
DVD  (w)
SLR  17₈
MUL  (z)
SLR  1
FST  (T)
```

Although there are four multiplication and division operations in this routine, only three rescaling operations are required: *DSR 4, DSR 1* and *SLR 1.* (The *SLR* 17_8 instructions following the divisions are considered repositioning operations since they are required to reunite the sign bit with the quotient.) The *DSR* 1 instruction actually combines two shift operations into one. Note that the *SLR* 1 instruction includes the shift to the left required for rescaling as well as the round off normally required after a multiplication operation. This round off is not needed after the first multiplication because the subsequent division can utilize the 31-bit fraction produced by the multiplication operation.

2.3.4 Evaluation of a Function

The evaluation of a function such as $y = ax^2 + bx + c$ is another example of an operation requiring a careful scaling analysis for proper programming. The following bounds on the terms are given:

$$|a| < 2$$
$$|b| < 2$$
$$|c| < 2$$
$$|x| < 4$$
$$|y| < 28$$

The scaling analysis uses the bounds on a, b, and c rather than their values as constants to permit evaluation of the function for each value of x with a series of different values for a, b, and c. Note that the given bound on y is lower than the bound on y implied by the bounds on the other terms in the function.

To reduce the number of computational steps, the function is rewritten as:

$$y = (ax + b)\,x + c$$

Thus, the following terms will be generated:

$$ax,\ ax + b,\ (ax + b)\,x,\ (ax + b)\,x + c$$

In the first step, a and x are used directly and may therefore be scaled as:

$$a = 2^1 \cdot a$$
$$x = 2^2 \cdot x$$

In the second step, the sum $ax + b$ is to be formed. According to the rules of scaling for addition, the scale factors of the numbers taking part in the addition must be identical and are determined by the upper bound on the sum which is 10. Consequently, the scale factor used for all the terms in this addition must be 2^4. For this reason, the first scaling equation is written as

$$2^1 \cdot a \cdot 2^2 \cdot x = 2^4 \cdot ax$$

Dividing by the desired scale factor 2^4 reduces the equation to the following:

$$2^{-1} \cdot a \cdot x = ax$$

indicating that the product of a and x must be shifted one place to the right to obtain the desired scale factor 2^4.

The second scaling equation is written as:

$$2^4 \cdot ax + 2^4 \cdot b = 2^4\,(ax + b)$$

which reduces to:

$$ax + b = ax + b$$

The third step involves multiplication of two numbers which are therefore scaled according to their upper bounds; however, the upper bound on the result of the multiplication is limited by the given upper bound on the final result.

The original equation can be rewritten as:

$$(ax + b)\,x = y - c$$

Hence, the upper bound on the product in the third step is equal to the upper bound on the term $(y - c)$; viz:

$$|y - c| = |y| + |c| < 30$$

Consequently, the scale factor used for the product $(ax + b)\,x$ is 2^5 ($2^5 = 32$). The third scaling equation is written as:

$$2^4 \cdot (ax + b)\,2^2 \cdot x = 2^5 \cdot (ax + b)\,x$$

Dividing this equation by the desired scale factor 2^5 reduces the equation to:

$$2^1 \cdot (ax + b) \cdot x = (ax + b)\,x$$

indicating that the product must be shifted one place to the left to obtain the desired scale factor 2^5 in the result.

In the fourth step, c is added to the product obtained in the previous step. The scale factor of the numbers involved in the sum is determined by the given upper bound on y. The scaling equation is therefore written as:

$$2^5 \cdot (ax + b)\,x + 2^5 \cdot c = 2^5 \cdot y$$

which reduces to:

$$(ax + b)\,x + c = y$$

Summarizing the results of the scaling analysis, b and c are inserted with the scale factors of 2^4 and 2^5, respectively, to reduce the amount of rescaling required. All other quantities are scaled to their respective upper bounds. The final scaling equation, which includes all four steps, may be written as follows:

$$y = (2^{-1}\,ax + b) \cdot x \cdot 2^1 + c$$

from which the following routine may be coded:

```
CAD  (a)
MUL  (x)
DSR  1
SLR  0
ADD  (b)
MUL  (x)
SLR  1
ADD  (c)
FST  (y)
```

2.4 SUMMARY

Table 5–1 summarizes the scaling procedures to be followed for fixed-point computation.

TABLE 5–1. SCALING PROCEDURES FOR FIXED-POINT COMPUTATION

OPERATION	REQUIREMENTS	PROCEDURES
Addition or subtraction of two numbers	Binary points of both numbers must be aligned; overflow must be avoided; if possible, result should provide maximum precision.	Choose identical scale factors for both numbers; scale for largest term whether original number or result; scale factor should be the smallest that satisfies these requirements.
Multiplication of two numbers	No special requirements for insertion of original numbers; product must be rounded off for further use; rescaling may be necessary for maximum precision.	Scale both numbers for upper bounds; use an *SLR* instruction to round off and, if necessary, to rescale the product for maximum precision. To obtain a rescale factor, divide the scaling equation by the desired scale factor of the product.
Division of two numbers	Divisor must be greater than dividend; lower (as well as upper) bound on divisor must be known to determine upper bound quotient. If quotient is to be used, sign bit must be reassociated with significant bits; if remainder is to be used, it must be noted that remainder appears scaled up by 2^{16}.	Choose scale factors of both numbers for maximum precision; scale quotient for upper bound, then divide scaling equation by scale factor for quotient to obtain rescale factor for dividend; rescale dividend prior to division operation. Following division, program an *SLR* 17_8 instruction to reassociate sign bit with significant bits of quotient; if remainder is to be used, multiply its scale factor by 2^{-16}.
Combined operations	Same as for operation with two numbers.	Perform scaling analysis for each arithmetic operation; minimize rescaling by proper choice of initial scale factors; if possible, combine two rescaling steps into one. Use constant scale factors to save computer time if loss of precision is permissible. Note that round off operation after multiplication is omitted if followed by division.

APPENDIX A
ILLEGAL INSTRUCTIONS

A.1 SCOPE

This appendix contains a description of those instruction codes which at present are not legal codes for the AN/FSQ-7 or AN/FSQ-8. Because the execution of these codes can cause a variety of actions, they have been grouped together according to the instruction class into which they fall. It should be kept in mind that the codes listed as illegal at the present time may later become valid instructions which would not necessarily perform the same operations that they do now.

A.2 ILLEGAL MISCELLANEOUS CLASS INSTRUCTIONS

Octal operation codes 034, 060, 064, 070, and 074 are presently illegal instructions for this class. If any one of these codes is executed, the Central Computer System performs no operation for a period of 6 μsec.

A.3 ILLEGAL ADD CLASS INSTRUCTIONS

Octal operation codes 120, 124, 144, 150, 154, and 174 are presently illegal instructions for this class. If any of these codes is executed, the number in the left half of the specified memory location will be added to the left accumulator. The results remain in the left accumulator. The right accumulator is not affected by these codes; however, both A registers are cleared. Execution of these codes may cause overflow in the left accumulator. These codes may be indexed and require a total of 12 μsec in execution time.

A.4 ILLEGAL MULTIPLY CLASS INSTRUCTIONS

Octol operation codes 300, 304, 310, 314, 320, 354, 230, 234, 240, 244, 270, and 274 are presently illegal instructions for this class. If any of these codes is executed, the Central Computer System will become hung up.

A.5 ILLEGAL STORE CLASS INSTRUCTIONS

Octal operation codes 300, 304, 310, 314, 320, 354, 364, 370, and 374 are presently illegal instructions for this class. If any of these codes is executed, the contents of the specified memory location will be cleared to positive 0. These codes may be indexed and require a total of 18 μsec to execute.

To provide uniformity in writing programs, octal code 300 has been arbitrarily chosen to be the code used by programming personnel when they desire to clear one specific memory location. The code has been given a mnemonic name of *CLR,* and its function is described above. It should be remembered that while this code will perform the operations described above, it is not truly a legal instruction and could be used for some other purpose in the future.

A.6 ILLEGAL SHIFT CLASS INSTRUCTIONS

Octal operation codes 410, 414, 430, 434, 450, 454, 464, and 474 are presently illegal instructions for this class. Execution of any of these codes will cause the Central Computer System to perform no operation for varying periods of time. The amount of delay is determined by the value in the right half-portion of the instruction word. If the value is 6 or less, the execution time is 6 μsec. If the value is greater than 6, the execution time varies to a maximum of 35.5 μsec.

A.7 ILLEGAL BRANCH CLASS INSTRUCTIONS

Octal operation codes 500, 504, 530, 534, 560, 564, 570, and 574 are presently illegal instructions in this class. If any of these codes is executed, the Central Computer System will perform no operations for a period of 6 μsec. However, execution of one of these instructions will clear the A registers.

A.8 ILLEGAL IO CLASS INSTRUCTIONS

Octal operation codes 604, 630, 634, 640, 644, 650, 654, 660, and 664 are presently illegal instructions in this class. Execution of these codes, which cannot take place until the IO interlock is cleared, will cause the Central Computer System to perform no operations for a period of 6 μsec.

A.9 ILLEGAL RESET CLASS INSTRUCTIONS

Octal operation codes 700, 704, 710, 714, 720, 724, 730, 734, 740, 744, 750, 760, and 774 are presently illegal instructions in this class. Execution of these instructions will cause the Central Computer System to perform no operations for a period of 6 μsec.

To provide uniformity in writing programs, octal codes 700 and 740 have been arbitrarily chosen to be the codes used by programming personnel when they desire to introduce a 6-μsec delay into a program. These codes have been given a mnemonic name of *NOP,* and their function is as described above. It should be remembered that these codes may later be associated with legal instructions and might perform some action other than what they do at present.

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)