C-588-PROG-ST

AIR TRAINING COMMAND

COMPUTER SYSTEMS DEPARTMENT

STUDENT TEXT

OZR0123
OZR0123-1

PROGRAM CODING

13 June 1966

RESPONSIBLE AGENCY OF ATC
Keesler Technical Training Center
Keesler AFB, Mississippi

DESIGNED FOR ATC COURSE USE ONLY

Computer Systems Department
Keesler Technical Training Center
Courses OZR0123 and OZR0123-1

Student Text
C-588-PROG-ST
13 June 1966

PROGRAM CODING

CONTENTS

CHAPTER 1

EVOLUTION OF THE SAGE SYSTEM

The Air Defense Problem

The problem of air defense is the protection of population, industrial areas, natural resources and retailatory forces from hostile air attack. To accomplish this, it is necessary to determine what means are available or should be developed to provide maximum security under the greatest possible threat.

Former Air Defense System and Its Limitations

Formerly, the task of defending the United States against hostile air attack was performed by a manual ground environment system, and the functions of detection, evaluation, and interception were completely dependent on human operation. However, due to the increased speed capabilities of aircraft since World War II and the rapid progress in the field of missile development, the manual system of air defense was recognized as being completely inadequate to cope with such formidable means of attack. Recognizing the potentialities of these weapons during the early stages of their development the authorities responsible for continental air defense saw the need for a rapid, accurate, automatic system of air defense.

Developing a New Air Defense System

The Air Force, in late 195Ø, enlisted the cooperation of various civilian organizations in its efforts to improve the capabilities of the United States air defense network. The overall program was known as the Continental Air Defense System (CADS) Project, under which civilian organizations helped to bring the national air defense system up to the best possible operating condition and made recommendations to ensure the system's continued effective operation. The air defense system was greatly improved by the CADS Project, but fell short of the Air Defense Command requirements for a vastly improved air defense system.

Simultaneously, studies were made on the combined use of digital computers and radar-data transmission equipment for application to air defense. The testing of a high-speed digital computer was recommended to the Air Force to provide information on the capabilities of such equipment to solve the ever-growing problem of air defense. The fundings of this program led to many new concepts for solving the problem and resulted in the establishment of an experimental project which gave rise to the SAGE System. This project was developed in three major phases: the 1953 Cape Cod System, the 1954 Cape Cod System, and the experimental SAGE subsector.

1953 Cape Cod System

The 1953 Cape Cod System was composed of a computer known as Whirlwind I (WWI) and a Direction Center, along with associated radar equipment. The purpose of this arrangement was to gather preliminary test data which would substantiate the concepts of the SAGE System then being planned. Emphasis was directed toward singling out obvious problem areas and attempting to correct whatever difficulties were encountered, rather than toward gathering complete statistical data on system operation. Consequently, there was very little modification of equipment.

1954 Cape Cod System

The 1954 Cape Cod System was the same as the 1953 system except that radar network and mapping facilities were increased. Several minor improvements were incorporated in the operating positions within the Direction Center. The primary objective was to supply statistical results on system capacity and accuracy.

Experimental Sage Subsector XD-1

The experimental SAGE Subsector, located in Lexington, Massachussetts was completed in 1955. It is equipped with a prototype AN/FSQ-7 Combat Direction Central known as XD-1. A radar system provides a variety of inputs similar in number and type to those used in the SAGE System. An Air Force ground-to-air data link is connected to the output for experiments with data-link-equipped aircraft.

The experimental SAGE Subsector provides experimental data on electronic reliability, computer programs, and operating procedures. It is organized to support the regular functions of a Direction Center and is used to obtain operational approval and to determine required equipment modifications.

Organizational Considerations

To ensure adequate air defense for continental United States, air defense activities were established at various echelons of command and responsibility. The overall function was delegated by the Joint Chiefs of Staff to the Continental Air Defense Command (CONAD). The primary concern of CONAD is the protection of retaliatory forces, population, natural resources, and industrial potential during initial or sustained attacks by hostile forces. This defense must ensure successful counter-attack by this country and must also ensure the eventual successful conclusion of hostilities.

To carry out its assigned tasks, CONAD delegated certain portions of the air defense mission to subordinate echelons. This delegation has geographical as well as operational applications.

The Continental Air Defense Command is presently organized into three Defense Forces, and this structure is reflected in the SAGE System. Each Air Defense Force is composed of a number of Air Defense Divisions. The Air Defense Division is relieved of direct supervision of groups and squadrons. However, its area of responsibility is large, making mandatory the minimizing of administrative and logistic responsibilities at this level. These responsibilities are delegated to the Air Defense Wing.

The area for which an Air Defense Division is responsible is normally composed of three or more sectors. The headquarters for the division is the Combat Control Center. The Air Defense Division exercises operational control of units designated for air defense operations within the division and is the level at which co-ordination with adjacent divisions is achieved.

The Air Defense Wing, with headquarters at the Combat Direction Center, has subordinate units that are primarily weapons and radars. The area of responsibility

of the Air Defense Wing is called a sector. To perform the functions of air surveillance, identification, and weapons control, detailed information on all defense elements is continually maintained.

Description of the Sage System

General

The SAGE System is the portion of the air defense system of continental United States that provides the means for semi-automatic processing of data and weapons control. The SAGE System consists of the following (See Figure 1-1):

a. The facilities required to process and transmit air surveillance data from data-gathering sources to Combat Direction Centers.

b. Combat Direction Centers, where air surveillance data is processed, evaluated, and developed into air situations at a sector level from which threat evaluation, weapons assignment, and appropriate weapons guidance orders are generated.

c. The facilities required to transmit situation data from Combat Direction Centers to Combat Control Centers and other Combat Direction Centers.

d. Combat Control Centers, where situation data from the Combat Direction Centers is processed, and from which the utilization of weapons resources can be monitored and directed.

e. The facilities required to transmit instructions from Control Centers to Direction Centers and to forward divisional situation data to other Control Centers and higher echelons of command.

Direction Center

The Combat Direction Center is an installation which contains all the equipment (including the AN/FSQ-7) needed to execute the functions of air surveillance, identification, and weapons control.

The function of the Direction Center is to receive, process, and develop air surveillance data at the sector level. On the basis of the sector air situation, the threat is evaluated, weapons are assigned, and orders are given. Information of importance to an adjacent Direction Center is communicated (crosstold) to that Direction Center automatically via ground-to-ground data links. A summary of the sector air situation is passed (forwardtold) from the Direction Center to the Control Center which has jurisdiction over its activities.

Three specific operations are included in the overall function of the Direction Center in solving the air defense problem. These operations consist of detecting the approaching aircraft, identifying the aircraft, and exercising operational control of defense weapons.

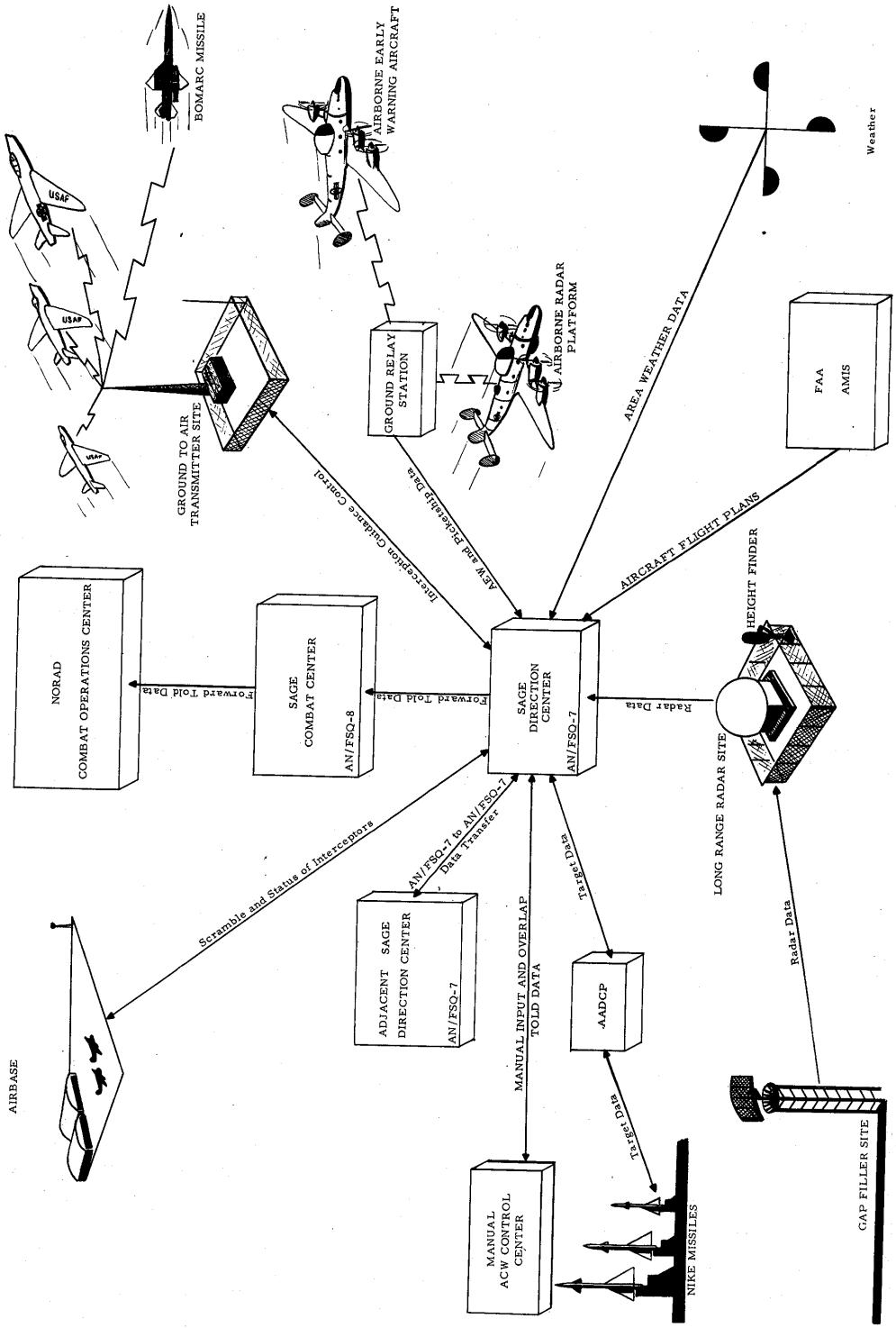


Figure 1-1. Relationship of SAGE to the Air Defense System

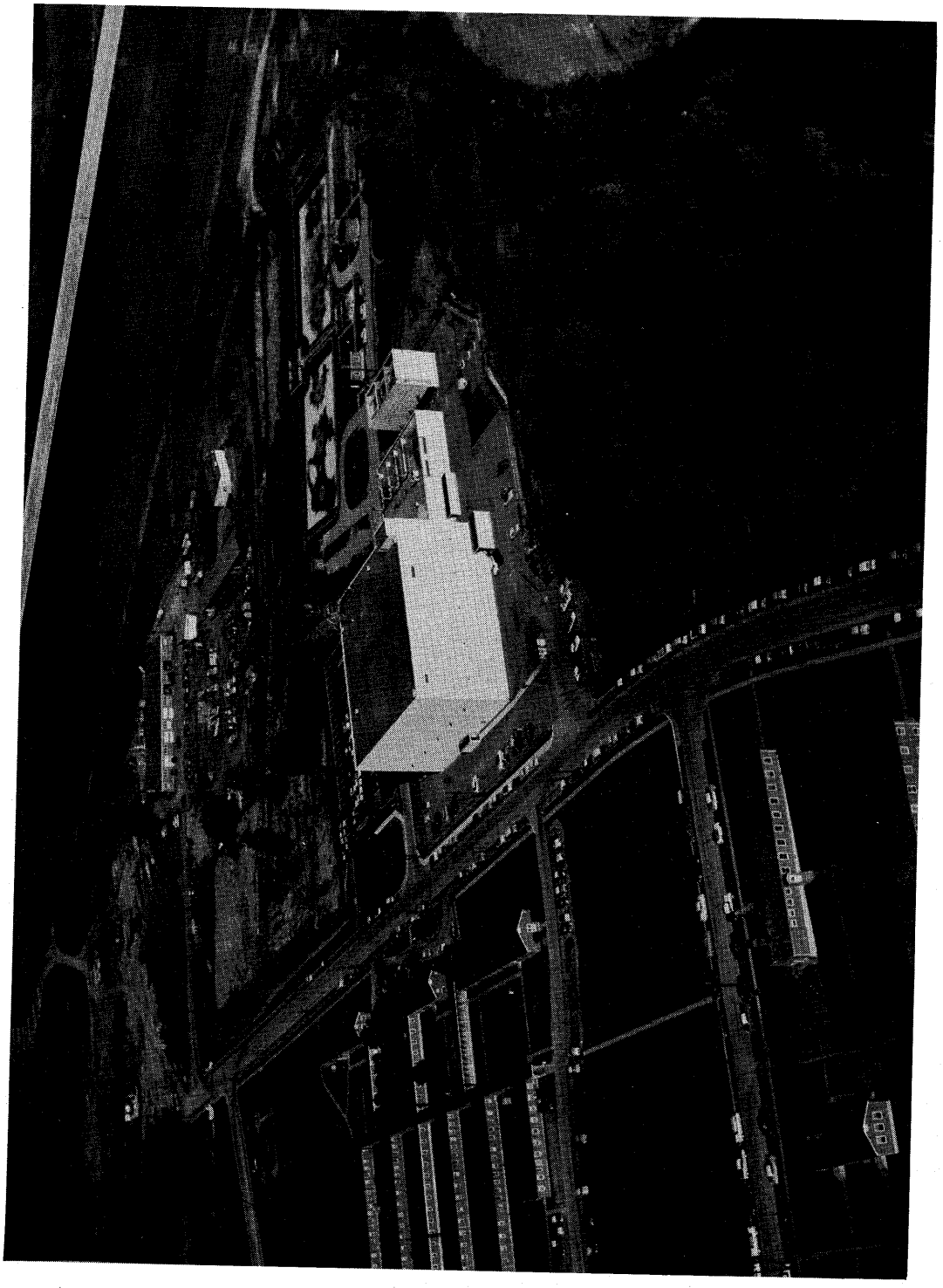


Figure 1-2. Typical SAGE Direction Center

(1) DETECTION

The detection of all aircraft approaching the United States is performed by a variety of units. Off-shore units, such as AEW aircraft, give early warning of approaching aircraft. Long-range radars and gap-filler radars spot and track all aircraft within or close to the United States. Reports of aircraft detected by the DEW lines in the northern parts of the continent are forwarded to the Combat Operations Center (COC) for dissemination to subordinate units of the SAGE System.

An important characteristic of the SAGE System is its flexibility to accept improvements in radar techniques and equipment as they evolve. The AN/FSQ-7 and the radar-data transmission equipments greatly reduce the time delays inherent in data processing and permit more effective use of data from many radars. Therefore, a large number of gap-filler radars can be used to provide an integrated, overall air picture.

Each Direction Center controls, and is connected with, all air defense radars that are geographically located within the sector. These radars provide adequate radar coverage. The air situation picture for the entire United States, therefore, is continuously maintained by both a perimeter radar screen and a radar umbrella. Any aircraft entering or originating in the United States can be detected and tracked continuously.

(2) IDENTIFICATION

There are three main forms of aircraft identification: Mark X (IFF/SIF) signals, the Multiple Corridor Identification System (MCIS), and the flight plan (military or commercial).

Aircraft equipped with Mark X equipment identify themselves automatically and do not require further action for identification by components of the SAGE System.

The MCIS requires all aircraft entering the coastal areas of the United States to identify themselves by means of prearranged check words and check maneuvers, as well as to use only predesignated corridors in approaching the Air Defense Identification Zone (ADIZ).

Flight plans are required of all military and commercial flights. These flight plans are inserted into the AN/FSQ-7 computers and are matched with radar returns, continuously and automatically, to aid in identifying the spotted aircraft. Computer programs record the number of times an aircraft deviates from its assigned course and notify appropriate personnel each time a deviation occurs.

(3) WEAPONS CONTROL

The SAGE System is designed to utilize not only existing forms of weapons but also new and improved equipment as it becomes available. Interceptors, anti-aircraft units, and guided missiles are presently the basic weapons used by the Direction Center in defense activities.

The air bases under the control of a Direction Center keep the center informed of their operational status, and computer programming indicates the number of weapons committed during any action. In this fashion, personnel at the center are kept up to date on the status of weapons available for assignment to targets.

Scrambled interceptors are given mid-course guidance by the computer and are aided in their return to base by computer-generated directions. Provisions are made for both data links and voice channels to control interceptors.

Point-defense weapons form the system of weapons that supply concentrated fire-power to protect small areas containing important targets or large, industrial, and heavily populated areas. The present, basic point-defense weapons are antiaircraft units and Nike missiles. As other types of weapons become operationally available, they may be incorporated into the SAGE System with only minor changes required in the programming of the Direction Centers.

CENTRAL COMPUTER SYSTEM

The Central Computer System (Figure 1-3) is made up primarily of a general-purpose, high-speed single-address, binary digital, stored-program computer with associated input-output (I/O) devices. The Central Computer is designed to accept large amounts of military tactical data and to process the data quickly. Generally speaking, the purpose of the Central Computer is to correlate and process data and to transfer the results to other systems of the Central.

The Central Computer is capable of adding, subtracting, multiplying, and dividing. More complicated operations such as extracting square roots and evaluating complex variables can be performed by combining the basic arithmetic operations. The computer is designed to respond to simple stereotype instructions, each of which causes the computer to perform a single arithmetic operation. An extended sequence of such instructions, resulting in the solution to a problem is called a program.

The AN/FSQ-7 Central Computer is composed of the following elements:

Instruction Control Element
Selection Control Element
Program Control Element
Arithmetic Element
Internal Storage Devices

Instruction Control Element

The instruction control element furnishes command pulses to the computer in correct sequence, enabling the computer to carry out the programmed instructions stored in the computer memory element. Functionally, the instruction control element decodes each instruction and generates the commands needed to carry out the processes indicated by the instruction. Physically, the instruction control element is divided into three sections: the instruction decoder, the pulse generator and control, and the command generators.

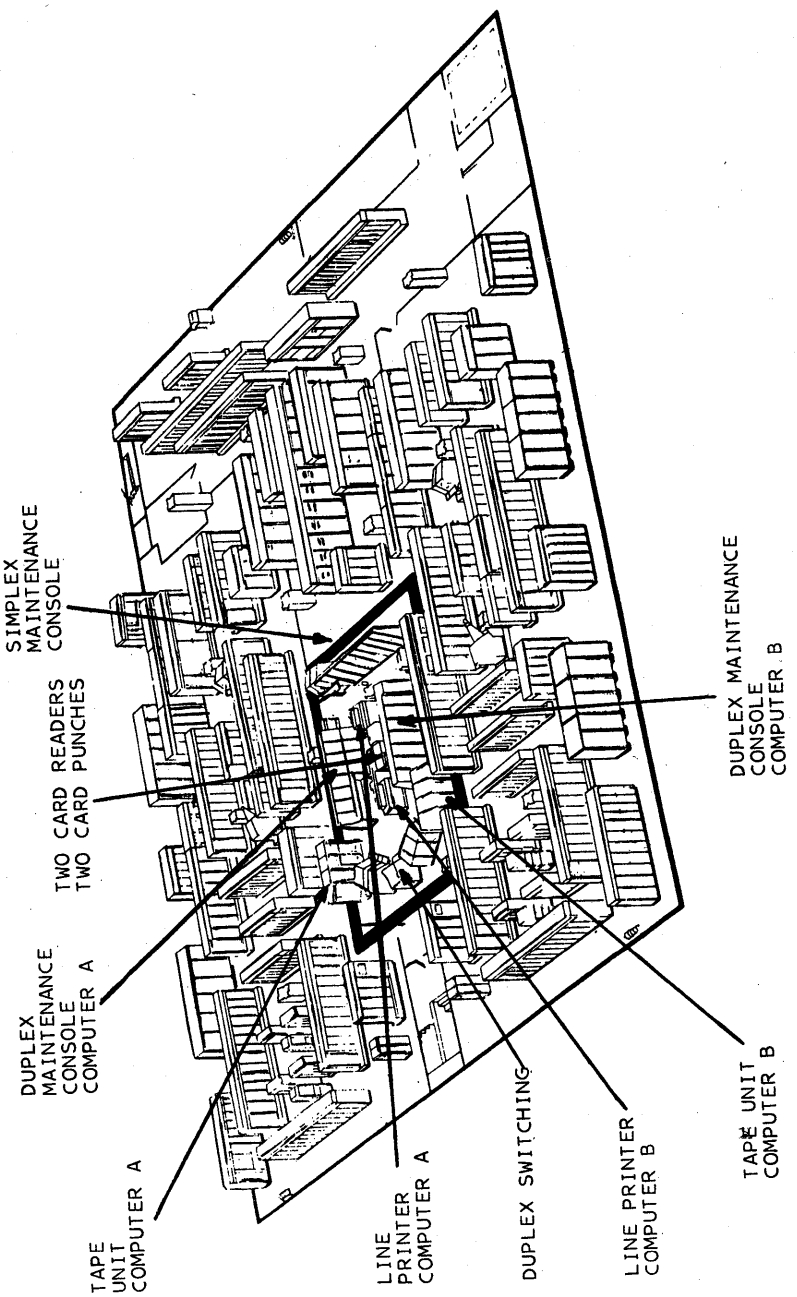


Figure 1-3. Central Computer System Overall View

Selection Control Element

The selection control element is divided into four sections: the index interval register, the Operate-Select-Sense (PERSELBSN) matrix, the break command generators, and the control circuits.

The selection control element synchronizes, controls, and directs data being transferred between the computer memory element and the several I/O units, including the Drum System. Prepared instructions, setting up control circuits in the selection control element, must be performed in advance of the actual transfer of information. This enables the information transfer to be properly initiated and processed.

The selection control element also incorporates circuits which permit the Central Computer, directed by a specific program, to perform certain operations affecting the electromechanical units allied with the I/O units and the several other electromechanical units in the system. In addition, the selection control element determines existing conditions in the Central and directs the operations of the computer accordingly.

Program Control Element

The program control element, in order to procure specific instructions, controls the selection of internal storage devices and such information as may be required by an instruction. In addition, the program element sequences the transfer of each instruction stepped out of the internal storage devices at the proper moment.

In guiding the processing of information, the program element co-operates with other elements of the computer. The instruction control and program elements together sequence and control the internal information-processing operations. The program and selection control elements together co-ordinate and effect the external transfer of information between the computer and other systems of the Central. The program element, therefore, contains circuits associated both with the internal control operations of the instruction control element and with the external control operations of the selection control element.

Arithmetic Element

The arithmetic element contains circuits and registers which perform mathematical computations on numerical data. The element is divided into two separate portions which are composed of identical groups of circuits and registers and are known as the left arithmetic unit and the right arithmetic unit.

Primarily, the arithmetic element employs the four basic arithmetic operations (addition, subtraction, multiplication, division) in the performance of its functions. Operands for the calculating processes are obtained from the computer memory element. The setting up of arithmetic operations is governed by add-class and multiply-class instructions which provide command pulses from the instruction control element. The result of the calculations is either retained in the arithmetic element for further processing or transferred to the memory element for storage until required by a subsequent program.

Internal Storage Devices

Three separate storage devices are contained within the computer: the core memory, the test memory, and the real-time clock. The core memory and the test memory furnish storage space for and permit quick access to instruction and information words. The real-time clock generates real-time information in the form of pulses.

In the AN/FSQ-7, the physical core memory units are referred to as memory 1 and memory 2. Memory 1, which is called BIG MEMORY contains $65{,}536_{10}$ or $200{,}000_8$ storage registers. Memory 2, which is called LIL MEMORY contains $4{,}096_{10}$ or $10{,}000_8$ storage 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 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 microseconds in the AN/FSQ-7. Thus, a minimum of 6 usec 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.

In addition to the core memories described previously, the AN/FSQ-7 contains another storage device referred to as TEST MEMORY. Test Memory consists of 16 plugboard registers, 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 test memory 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 usec. 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.

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.

INPUT-OUTPUT DEVICES

The I/O devices of the Central Computer System are composed of card machines, magnetic tape units, and the I/O register. In order to eliminate confusion brought about by similarity in terminology, a distinction between the Input and Output Systems and I/O devices should be made.

Figure 1-4. BIG MEMORY

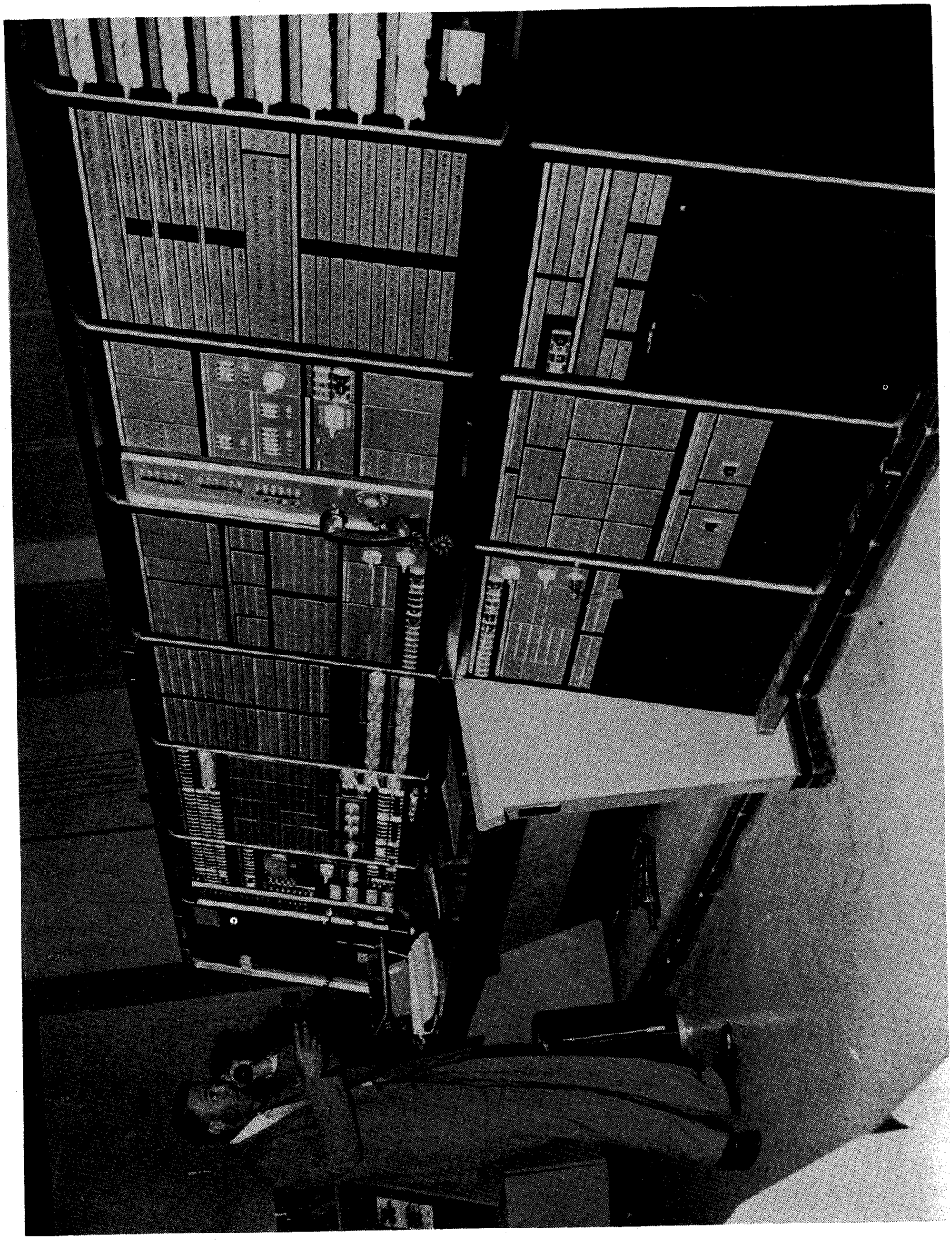


Figure 1-5. Duplex Maintenance Console - Primary Indicators and Controls

An input device associated with the computer may be defined as a unit which transmits data directly into the computer. An output device is a unit which receives data directly from the computer. Certain units, because of the nature of their functions, may be considered as being both input and output devices. The Input System and the Output System do not deal directly with the computer, as the data for these Systems must first pass through the Drum System.

Card Machines/Line Printers

The 713 Card Reader allows the insertion of information from punched cards directly into the computer system. The card reader reads a card one row at a time within a cycle of 4ØØ milliseconds with a maximum reading rate of 15Ø cards per minute.

The 718 Line Printer provides the means for the computer to prepare information in printed alpha-numeric form.

The 723 Card Punch allows the computer to present processed information in punched card form. The punch has an operating cycle of 6ØØ milliseconds duration, allowing a maximum punching rate of 1ØØ cards per minute.

Magnetic Tapes

The 728 Magnetic Tape Drive units which are used with the AN/FSQ-7 provide a 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 2,4ØØ feet, this means that one entire reel of tape can be written on or read from in about six minutes. The amount of information that can be recorded on the tape reel is approximately the same as that which could be contained in 2Ø,ØØØ standard IBM 8Ø-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.

Manual Controls and Computer Indicators

The manual controls and indicators of the computer are situated on the duplex maintenance console and on the duplex switching console, as are the controls and indicators for the other systems of the Central. These controls and indicators supply maintenance personnel at the Center with a means of manual control for loading initial operating programs, loading certain reliability and diagnostic test programs, and monitoring the operations of the major registers and circuits in the associated equipment.

The duplex maintenance console contains the majority of the controls and indicators for the manual operation of the Central Computer System. The alarms and neon indicators on the console show the status of the computer, and virtually all manual program and checking operations are affected by means of the console controls.

The duplex switching console is used by maintenance personnel to establish the status, active or standby, of the separate sets of duplex equipment. Each computer transfers status data to the duplex switching console. This data activates indicators and alarms, permitting maintenance personnel to determine the operating condition of each computer and to make quick decisions pertaining to duplex switching operations.

Display System

Information processed by the computer is sent via the display drum fields to the Display System where it is converted as required and presented on cathode-ray tube (CRT) displays to the operators of the Center.

The equipment contained in the Display System, for purposes of this discussion, is divided into the following major groups: the digital displays, the situation displays, and associated equipment. The digital and situation displays are made up of consoles housing CRT's which present a picture of the air situation to an observer. The associated equipment is made up primarily of various components connected with the operational functions of the displays.

The Display System provides the means by which the air situation of the area supervised by the Center is visually presented to human observers. This is accomplished by means of special CRT's mounted in specially constructed and equipped consoles.

Data is first accepted by the Input System and is subsequently transferred to the Display System via the Drum and Central Computer Systems. The tracks and raids representing the air situation are consolidated and converted into larger common coordinates for display at the consoles. Since all such data is received from the computer in binary form, the Display System converts the data from the binary form to a system of letters, numerals, vectors, and symbols. This conversion permits the presentation of a display which is more readily viewed and interpreted by the observer.

The console operator observing the air situation messages presented on the CRT may act on the information in several ways. The operator may communicate with the computer by requesting information from the computer, or by feeding back information in answer to a request from the computer. He may instruct the equipment to transfer a message to the console of another operator. These operations are possible through the use of manual-intervention switches and light guns located on the consoles. Telephone facilities located at the console are also available to the operator for communication with other personnel within the Center. He may also communicate beyond the area of the Center by wire and radio telephone circuits.

The Display System employs two basic types of CRT displays for presenting the visualized air situation to the human observer. They are the digital displays and the situation displays. The CRT's for these displays are mounted in various types of display consoles located at selected operating positions in the Center. Typical consoles are equipped with a situation display (SD) scope, a digital display (DD) scope, and various manual-intervention (MI) switches, alarms, and warning lights. Many consoles are also equipped with a light gun and telephone facilities.

In addition to the display consoles, the equipment of the Display System encompasses various types of associated equipment, including auxiliary consoles, which aid the Display System in the performance of its functions.

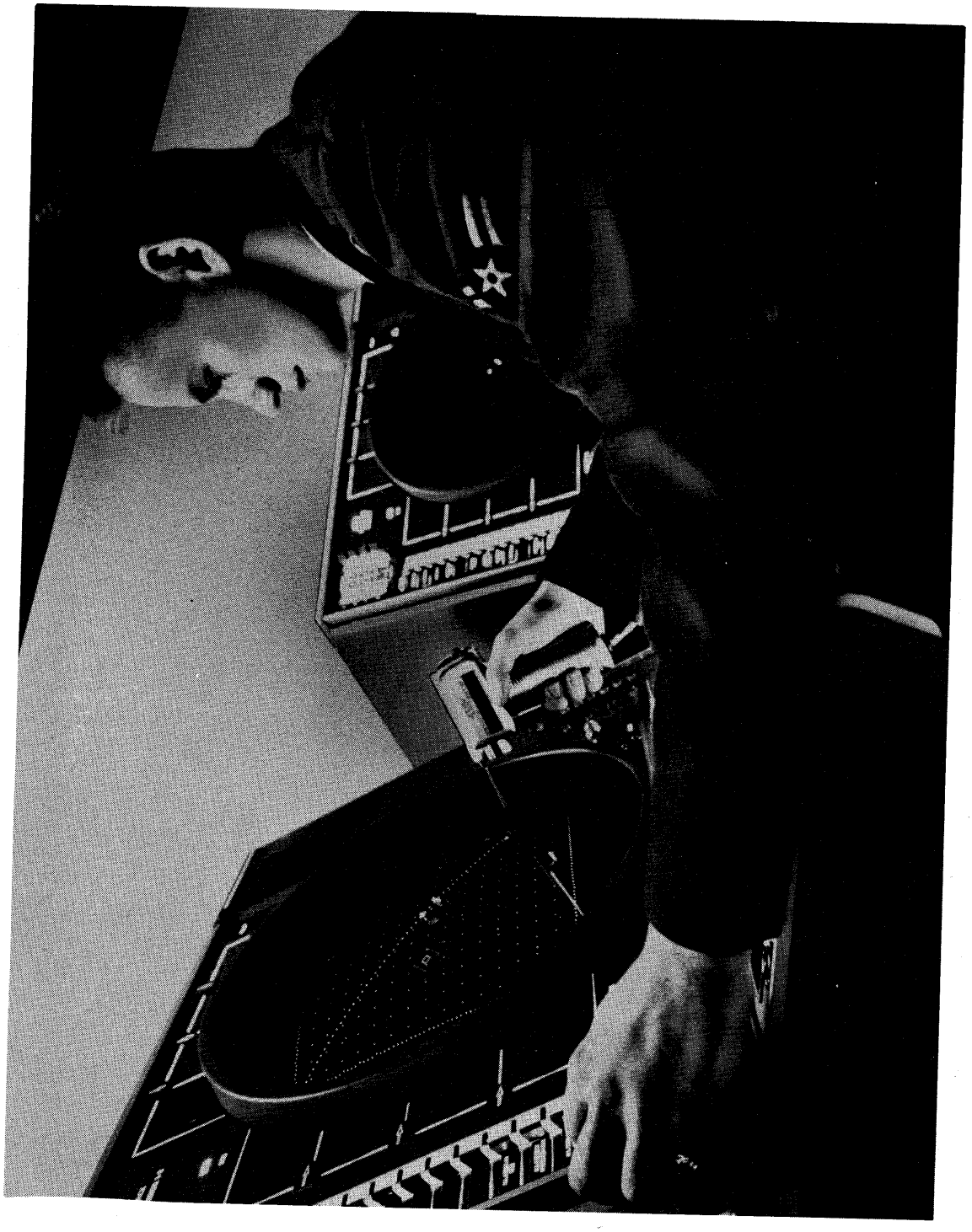


Figure 1-6. Typical Display Console

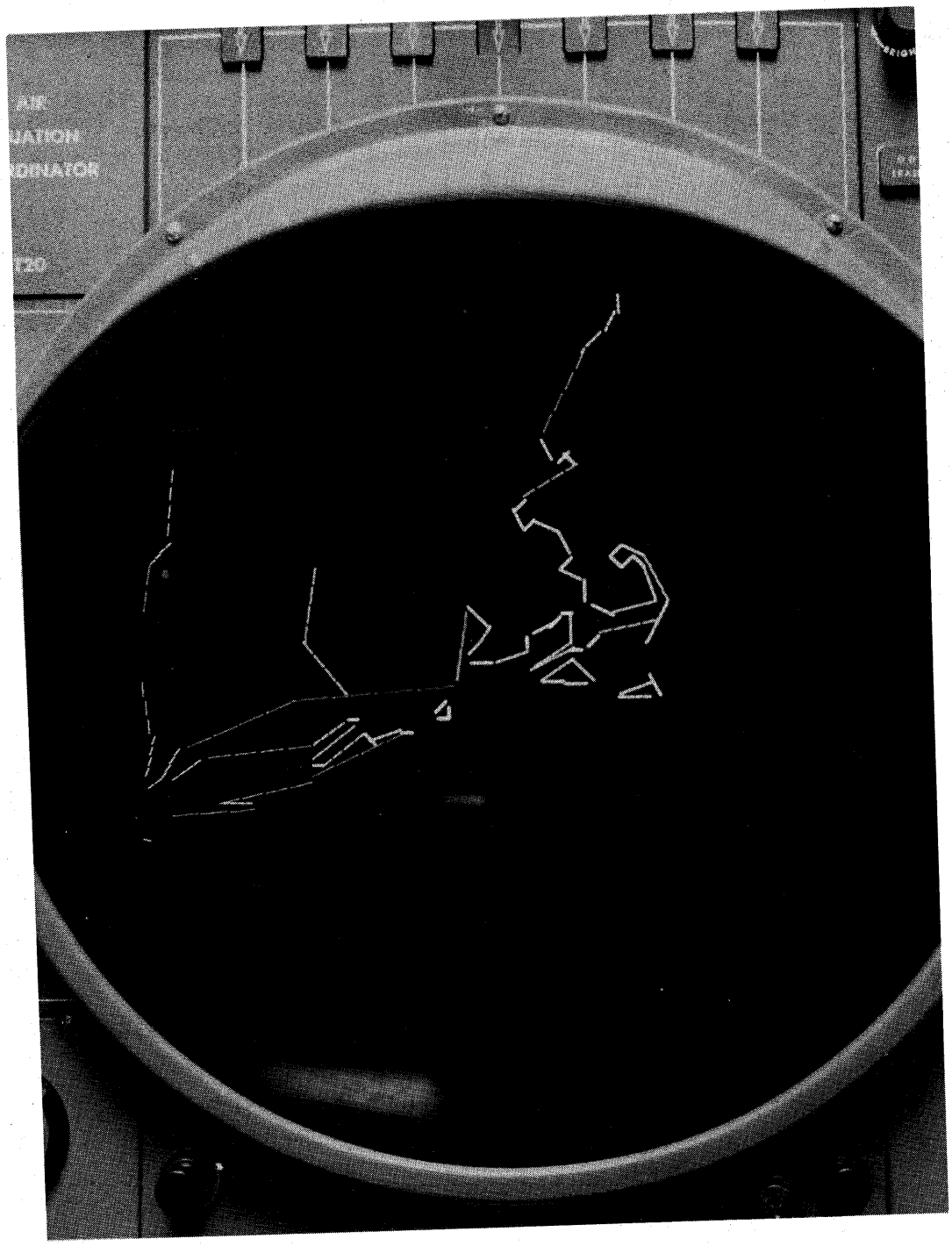


Figure 1-7. Computer-Constructed Picture of New England

Digital Displays

The DD scope is a 5-inch CRT mounted in the upper right portion of the display console. Supplementary data that is either too voluminous or too slow-changing to warrant its presentation on the larger pictorial SD screen is presented on the DD screen. The information appears as a tabular array of characters and symbols and, unlike the situation display, changes only when the computer orders a change due to, for example, the presentation of new information. Barring receipt of instructions from the computer to the contrary, the digital display remains indefinitely.

Digital display data is processed by the computer and transferred in binary form via the Drum System to the Display System. Since data in binary form is difficult for an observer to interpret, the binary information is converted to analog voltages which, in turn, are converted to the visual display presented on the face of the DD CRT.

Situation Displays

The situation display scope is a 19-inch CRT mounted in the center portion of the display console. A current plan-position map of the rapidly changing air situation which shows correct geographical relations between fixed points and moving targets is presented on this scope. Supplementary descriptive data in the form of vectors and special symbols is positioned next to specific points and targets for identification purposes. (See Figure 1-8).

The 19-inch CRT used for the situation display is not a conventional-type tube. It is a large, special-purpose tube which is capable of generating complete individual characters, drawing vectors, assembling the characters and vectors into message patterns, and positioning the entire message pattern in the proper place on the screen of the tube. The result viewed by the observer is an orderly arrangement of message patterns, each message consisting either of a single character or of a group of characters.

All SD messages are rewritten on the face of the CRT at short, regular intervals. If the basic data has not changed or been superseded, the display is continually repeated in the same position. If, on the other hand, the information has changed, the display is corrected. In this way, the console operator is presented with air-situation changes within an extremely short time after they occur.

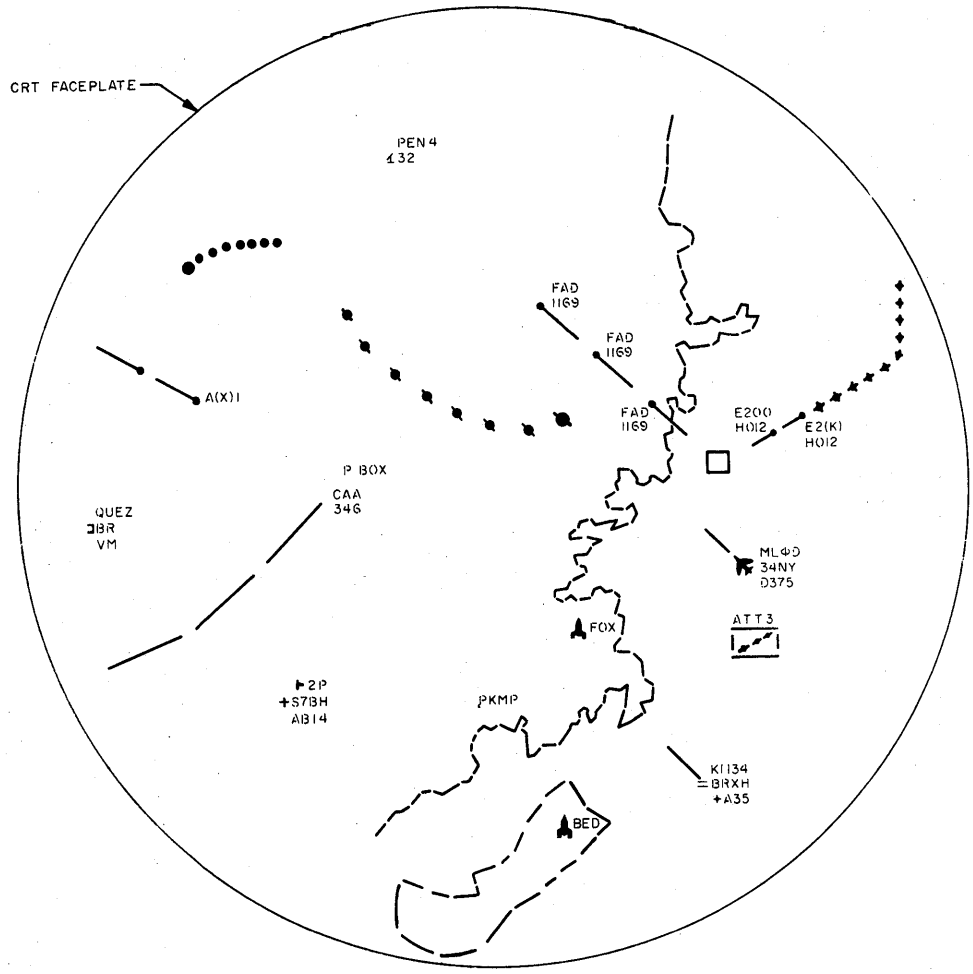


Figure 1-8. Typical Situation Display

CHAPTER 2

OPERATION OF THE CENTRAL COMPUTER SYSTEM

GENERAL

This Chapter presents in greater detail some of the concepts and element descriptions mentioned in the previous chapters, including machine language, machine timing, and the interactions of the memory, arithmetic, program control, instruction control, magnetic tape and card machine, and selection-control elements.

MACHINE LANGUAGE

The methods by which the circuits of the Central Computer accomplish their data-processing tasks is entirely compatible with the methods that a human being would use to perform the same task in binary arithmetic. For this reason, the Central Computer System language (Machine Language) is designated as binary. All operations performed by the Central Computer System, including computer word transfers, are based upon the two-stage nature of the binary digit, and relate directly to the binary number system.

Computer Words

The basic unit of Central Computer System intelligence is the thirty-two bit computer word, stored primarily in the memory element. Both instructions and data are stored in the memory element as thirty-two bit computer words. Instructions are known as instruction words, data as data words. The Central Computer System distinguishes between instruction and data words only as a function of timing.

Data Words

Due to the dual arithmetic characteristic of the Central Computer System, a data word is expressive of two distinct operands, used in an identical manner, on a simultaneous basis. These two operands exist in each of the two half-words of a computer word. Since two different configurations of half-words in a computer word are possible, a separate discussion of each follows. The discussion includes the relationship of non-quantative or logical data to data words (See Figure 2-1).

a. SIXTEEN BIT HALF-WORD CONFIGURATION. By dividing a thirty-two bit computer word into two sixteen bit half-words, each half-word can express a discrete value ranging from -32,767 to +32,767. The leftmost bit in each half-word represents the sign of the binary quantity expressed in the remaining 15 bits of the same half-word. The leftmost bit is called the sign bit; the remaining 15 bits are magnitude bits. Since there are 15 magnitude bit positions, each capable of containing the two binary digits, it is evident that there are 2^{15} possible variations in bit position content, or, a range of from $\emptyset$ through $2^{15}-1$, that is, 32767_{10}, in the magnitudes that they can express. The Central Computer System recognizes the sign bit as constant, that is, a $\emptyset$ (zero) in the sign bit indicates that the expression is positive, and the quantity expressed in the magnitude bits are precisely expressed in binary. When the sign bit is 1 (one) the quantity

expressed in the remaining 15 bit positions is not only defined to the Central Computer System as negative in sign, but also as expressed in the complement form. In the complement form, the value of each binary digit is reversed, so that all one bits in the magnitude become zeros, and all Ø's in the magnitude become 1's. For example, a binary quantity written ØØØ 1Ø1 1Ø1 Ø11 Ø1Ø will appear in a half-word as 1 111 Ø1Ø Ø1Ø 1ØØ 1Ø1. In notating binary expressions, the sign bit position of a half-word is separated from the magnitude bit position by a triangle placed between the first (sign) and second bits of the expression (for example, Ø$_{\Delta}$ØØØ 1Ø1 1Ø1 Ø11 Ø1Ø). A computer word may, of course, contain half-words opposite in sign.

b. FIFTEEN AND SEVENTEEN BIT HALF-WORD CONFIGURATIONS. Division of a thirty-two bit computer word into a fifteen and a seventeen bit half-word is accomplished by using the leftmost bit and the rightmost sixteen bits as the right half-word, and the remaining fifteen bits as the left half-word. The sign bit position of the right half-word is the leftmost of all right half-word bit positions, and is also the leftmost bit position of the computer word. In the seventeen bit right half-word, it is possible to express values ranging from -65,535 through +65,535, since there are 2^{16} possible variations of bit position content. The fifteen bit left half-word has no sign bit; it has only 15 magnitude bits. Thus, a value from Ø through 32,767 can be expressed in the left half-word. Dual arithmetic operations using the left half-word have results that are unsigned.

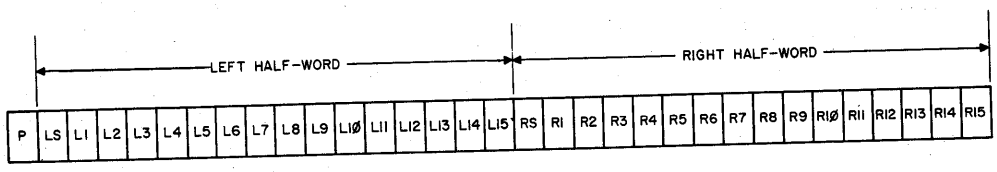


Figure 2-1. Computer Word Layout

c. LOGICAL CONFIGURATION. A computer word may express logical, rather than quantitiative, information. Any one, any number, or all of the bits in the computer word may be set to a binary value as a means of identification rather than quantitization. The identification is established by the person or program creating the computer word, and by the program or person interpreting their values into the logical terms they represent. Thus, if a given bit position in a word is set to "1", it might indicate the existence of a condition within a Central Computer System. If the same bit position is set at "Ø", it might indicate the absence of this same condition. The data word is divided into as many bit-position groups as may be required to cope with an equal number of logical settings.

Instruction Words

Computer words expressive of instructions are similar to a data word having fifteen- and seventeen-bit left and right half-words, except that the left half-word of an instruction word designates the operation to be performed, and the right half-word designates a specific register in the memory element the contents of which are required in order to perform the operation. Although variations do exist, the fundamental instruction-word interpretation is based on the fifteen and seventeen bit configuration. (No consideration will be given to the variations in instruction word configuration at this point).

MACHINE TIMING

Central Computer System operations are logically consistent, and are effected in a consistent and orderly fashion by means of a basic timing scheme. Since the Central Computer System is designed to handle real-time problem solutions, and consequently, must be as time-saving as possible, the basic time unit used is significant. This unit is determined by the length of time required for the slowest functioning element the use of which is essential to Central Computer System operations. The memory element of the Central Computer System establishes the time base, which is six microseconds (six millionths of a second). During the execution of a program, memory-element operation is required for each instruction. Subsequent demands for memory-element operations as a function of a particular instruction may be one or two in number or may not exist at all. Each memory-element operation is referred to as a memory cycle. The memory cycle controls, in part, the timing of the balance of the Central Computer System's operations, referred to as machine cycles. Operations of the Central Computer System fall into two basic categories--that is, internal operations, and input/output (I/O) operations. All information transfers in internal operations are accomplished within five of the Central Computer System elements, and exclude information transfers between these five elements and the magnetic-tape and card-machine element. I/O operations are exclusively concerned with information transfers between five of the Central Computer System elements and other systems of the SAGE Computer, and/or transfers between the five elements and the magnetic-tape and card-machine element.

Internal Operation Timing

Included in internal operations of the Central Computer System are the acquisition of instructions and data from the memory element, performance of arithmetic and logical data-processing operations, and the return of data-processing results to the memory element. Also included in internal operations are Central Computer Systems operations involving the issuance of control impulses to other SAGE Computer Systems, and to the magnetic-tape and card-machine element, and the sensing of circuit conditions within them. Uninterrupted internal operation takes place with the issuance of control impulses and sensing of circuit conditions, since no information transfers are made.

Program Time (PT)

Instructions executed during internal operations that required only six microseconds (one memory cycle) to complete, are accomplished in a program time (PT) cycle. Within this cycle, the instruction word is brought from the memory element, decoded, and completely executed. The following cycle is also a PT cycle, and applies to the next instruction, which is brought from the memory element and executed. A PT cycle is therefore one in which an instruction word is brought from the memory element in order to be executed.

Operate Time (OT)

Instructions executed during internal operations that require 12 microseconds (2 memory cycles) to complete placement or recovery of a computer word from the memory element are accomplished in an Operate Time (OT) cycle. The OT cycle is further defined into an OTA and an OTB cycle. In an OTA cycle the computer word is

obtained from the memory element; in an OTB cycle the computer word is placed in the memory element. The determination of the OTA or OTB cycle is a function of the instruction obtained from the memory element during the preceding PT cycle.

Those instructions requiring 18 microseconds (3 memory cycles) for execution, require the use of both an OTA and an OTB cycle. In these instructions, a word is obtained from the memory element, used (and perhaps changed), and placed back in the memory element. This OT cycle requires 12 microseconds; the time required to obtain the instruction itself is 6 microseconds. Immediately following the execution of the last memory cycle of either a 12- or 18-microsecond instruction, the PT cycle of the next instruction to be executed begins.

Some instructions require a longer time for their execution than a single PT cycle. In these cases, an instruction's execution will use an OTA cycle to ensure adequate time for the completion of the instruction before entering the PT cycle of the next instruction. The word obtained from the memory element during the OTA cycle is held in the memory-buffer register, where the word will be destroyed during the next instruction sequence.

Arithmetic Pause

Instructions, the execution of which involves the accomplishment of some repetitious process in the arithmetic element of the Central Computer System, generally interrupt the consecutiveness of memory cycles, and intiate an arithmetic pause. Such a pause occurs only between memory cycles and lasts as long as necessary for completing the performance of the repetitious process. When the process is completed, the arithmetic pause is discontinued and the next PT cycle begins.

I/O Operation Timing

The Central Computer System operations involved in the transfer of information to or from the magnetic-tape and card-machine element and/or the other systems of the SAGE Computer are referred to as I/O (Input-Output) operations. These operations are usually accomplished within the Central Computer System in the following sequence:

1. Designation of an address in the memory element of the first location the contents of which are involved in the transfer;

2. Designation of a magnetic-tape and card-machine element unit or a unit in one of the other systems of the SAGE Computer to or from which the information is to be transferred.

3. Initiation of the transfer of a given number of computer words to or from the designated unit.

Since the memory element is far more rapid than most of the I/O units, to avoid delay time in its operation during I/O operation, I/O transfers are accomplished by the Central Computer so that internal operations can be continued between transfers of consecutive words. This permits the time that would otherwise be lost in waiting for the I/O device to accept or transmit another word to be used for internal operations.

Whenever a word in the set of words being transferred can be routed to or from the memory element, the internal operations are interrupted long enough to make the transfer. The interruption takes place at the end of a memory cycle, except when an arithmetic pause is in effect, in which case the transfer is immediate. During an arithmetic pause, no interruption is required, since both this internal operation and the I/O operation can take place simultaneously. If, however, the arithmetic pause should terminate during a transfer to or from an I/O device, subsequent internal operations are delayed until the transfer is completed.

The Central Computer System is informed that the I/O device is prepared to handle a word by a break request from the I/O device. On receipt of a break request, the Central Computer System signals internal operations to cease at the end of the current machine cycle; and, at the end of the machine cycle, initiates a break cycle. The break cycle begins immediately when an arithmetic pause is taking place. A break cycle may be either a break-in cycle, or break-out cycle, depending upon the direction of transfer.

Break-In Cycle

When an I/O device is prepared to transmit one of the set of computer words it has been instructed to transmit, it sends a break request to the Central Computer System. The Central System initiates a break-in cycle as soon as internal operations permit it to do so. The break-in cycle results in the word being placed in a location within the memory element (memory cycle); the break-in cycle has a duration of six microseconds.

Break-Out Cycle

When an I/O device is prepared to receive one of the set of computer words it has been instructed to receive, it sends a break request to the Central Computer System. The Central Computer System initiates a break-out cycle as soon as internal operations permit it to do so. The break-out cycle results in the word being routed from the memory element to a register in the storage media of the I/O device. A memory cycle is required; this defines the duration of the break-out cycle as six microseconds.

I/O Pause

The actions preceding and including initiation of an I/O transfer of information are part of the internal operations, since the execution of the instructions do not usually transfer information when they are being executed. However, the instructions cannot be executed until any I/O transfer in progress has been terminated. This may cause a delay in internal operations while awaiting the completion of one I/O transfer before initiating another. Such a delay is called an I/O pause.

Timing Summary

The basic rate of operation (machine cycle) of the Central Computer System is the same as the basic rate of operation of its memory element (memory cycle)--six microseconds per cycle. During internal operations, a PT cycle is required to obtain each instruction from a specific location in the memory element. A given instruction will require (1) no further machine cycles; (2) an OTA cycle if a computer word must be

brought from the memory element, or if additional time is required to complete the instruction; (3) an OTB cycle if a computer word must be placed in the memory element; (4) both an OTA and an OTB cycle if both types of operations are required. When an instruction requires repetitious operations to be performed in the arithmetic element, an arithmetic pause is initiated. The arithmetic pause interrupts the machine cycles long enough to complete the operations required by the instruction.

I/O operations occur when an I/O transfer has been initiated and a word can be transferred between the memory element of the Central Computer System and an I/O unit in either some other system of the SAGE Computer, or in the magnetic-tape and card-machine element of the Central Computer System. Prior to the I/O operation, the designated I/O device sends a break request to the Central Computer System, indicating readiness for transfer. Subsequently, between machine cycles involving internal operations, or during an arithmetic pause, the Central Computer System initiates a break-in or break-out cycle that lasts six microseconds. A break-in cycle allows a word to be transmitted by the I/O device and to be stored in the memory element. A break-out cycle allows a word to leave the memory element and to be stored in the I/O device. Initiation of I/O transfers are a part of internal operation. The instructions initiating I/O transfers can also initiate an I/O pause, thereby delaying their execution until the I/O transfer in progress is terminated.

THE MEMORY ELEMENT

The responsibilities of the memory element of the Central Computer System are:

1. To provide a media whereby each of a large number of computer words may be stored and recovered with equal facility;

2. To provide a means for designating a given register for content-transferral action;

3. To provide a time-measuring device by means of an accessible register the contents of which normally reflect continual incrementation;

4. To provide a means by which computer word transfers between the memory-element and external-storage devices are checked for transmission inaccuracies or are prepared for subsequent checking.

Components Of The Memory Element

The information upon which the Central Computer System operates, and the instructions that direct these operations (from the program and instruction control elements) are stored as computer words in the memory element. The memory element consists of:

1. Two magnetic core memory units. One unit has a storage capacity of 65,536 computer words. The other unit has a storage capacity of 4,Ø96 computer words, in addition to a test memory unit where 19 computer words can be stored.

2. Three registers, each related to one of the memory units which permit reference to a given register within a given unit.

3. One register, the contents of which are accessible, and reflect, due to consistent incrementation, the passage of time.

4. A buffer-type register through which all computer words entering the memory element for storage, and all computer words being transferred out of the memory element must pass.

Memory Timing

The speed of the Central Computer System is largely determined by the speed of operation of the memory element. Successive words can be obtained from core memory at intervals of six microseconds. Therefore, core memory operates in cycles of six microseconds. Two types of cycles--read or write--are possible.

Read Cycle

During a read cycle, a word is read from core memory and placed into the buffer-type register (memory buffer register). From the memory buffer register, the word can be made available to other registers in the SAGE Computer. Within the same read cycle, the word is written back into the memory-element storage register from which it came, thus restoring the original contents to that register. Reading from a memory-element storage register is considered non-destructive reading.

Write Cycle

During a write cycle, a core memory register is cleared, and a word previously placed in the memory-buffer register is written into the cleared core memory register.

Memory Addressing

Every memory register has an address (optional for test memory registers). This means that each word contained in a memory-element storage register is referred to by a number relating directly to its location in the memory element. A memory-register address is always used when a computer word is to be taken out of storage or when a computer word is sent to that memory register for storage. Since core memory has a storage capacity of 69,632 computer words, there are a total of 69,632 registers, and an equal number of different addresses. Since binary expressions do not lend themselves to clear and concise notation, the octal number system is used for addresses, although addresses within the computer are expressed in binary. The first register in core memory has been arbitrarily specified as having address zero. Succeeding registers have octal addresses of 1, 2, 3, 4, 5, 6, 7, 10, 11, 12, etc. The last register in core memory (the 65,536th) has an address of $20\emptyset,777_8$ which, in the computer, is expressed in binary as 1Ø,ØØØ,111,111,111,111. Including the address zero, there are $21\emptyset,\emptyset\emptyset\emptyset_8$ registers (the next number from $20\emptyset,777_8$). Accordingly, $21\emptyset,\emptyset\emptyset\emptyset_8$ is equal to $69,632_{1\emptyset}$, exactly the number of registers in core memory.

There are sixteen addresses associated with the nineteen test-memory storage registers. These addresses range in value from $377,76\emptyset_8$ to $377,777_8$. Sixteen of the nineteen registers are contained in a plugboard located in and removable from the Duplex Maintenance Console (Figure 2-2). Each bit position of these registers consists of two holes on the plugboard. By connecting these two holes with a small conductive device known as a jackplug, a circuit is established, which defines this bit position's content of one to the memory element. An unplugged set of holes defines that bit position's contents as zero. There are thirty-two bit positions within each register, each capable of being set to one or zero, depending on whether the corresponding holes are plugged or unplugged.

Each of the test-memory plugboard registers have associated with it four additional two-hole sets, labelled "A", "B", "L", or "P". Only when the associated two-hole set labelled "P" is plugged with a jackplug are the plugged contents of that register addressable by the appropriate test-memory address. Even though a test-memory plugboard register may have jackplugs in the bit position holes (hubs), the accessable contents of that register will be zero in all bit positions, unless the associated two hole set labelled "P" is plugged. When all sixteen registers on the plugboard have their associated "P"-hubs connected by a jackplug, the plugged contents of the first register on the plugboard would be addressable by the first test memory address, $377,76\emptyset_8$, the content of the second register by the next higher address, etc. Thus, the contents of all sixteen of these registers are available through designation of the appropriate test-memory addresses. The only means by which the contents of a test-memory plugboard register may be altered is through manual alteration of the jackplugged connections on the plugboard. Any attempt to program a change to the contents of a register by an instruction that stores a computer word into a memory-element storage register, and that designates a test-memory address, results in placement, in the test register, of the word that is to be stored.

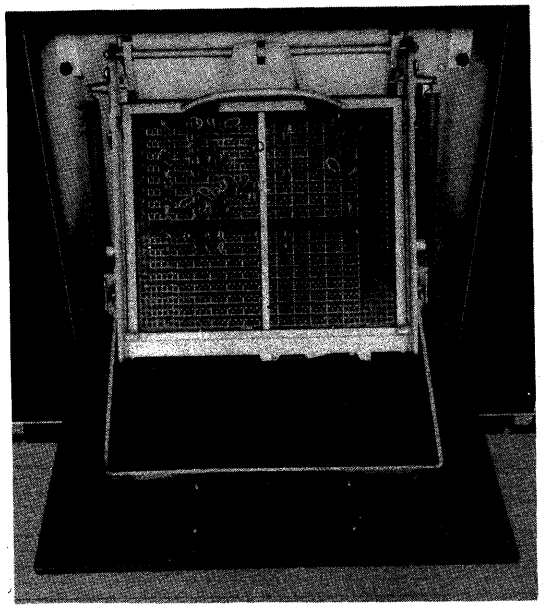


Figure 2-2. Test Memory Plugboard Installed In Duplex Maintenance Console

The test register is a thirty-two bit flip-flop register. Its contents can be obtained only by addressing a test-memory plugboard register the hubs of which are plugged in the "L" position only. Any or all of the plugboard registers may be plugged in the "L" position only. Therefore, the test register's contents may be made accessible through designation of a number of test-memory addresses. However, when only the "L" hubs, of a given register are plugged, the plugged contents of that register are ignored when the register is addressed; only the test register's contents are available at that address.

Two banks of two-position toggle switches are mounted on the Duplex Maintenance Console. Each bank has 32 switches. One bank of switches is referred to as Test memory switch register "A"; the other is test memory switch register "B". The contents of either register are established and/or altered by manual action only. The contents of test-memory switch register "A" are made available by means of addressing a test memory plugboard register. The register must have only its associated "A" hub plugged. Any bit-position jackplugging in the plugboard register is ignored, since the associated "P" hub is not jackplugged. Test-memory switch register "B" is made available by means of a test-memory plugboard register plugged in its associated "B" hubs only. An attempt to store a computer word into a test-memory plugboard plugged in the "A" of "B" hub position only, results in the word being placed in the test register, even though the associated "L" hubs are not plugged.

> NOTE: Should a given register be plugged in two or more of the associated "A", "B", "L" or "P" hubs, the computer word obtained as a result of addressing that register is the logical sum of the contents of the registers designated by the plugging of these hubs.

A two-position, control-type toggle switch labelled, in its two positions, Test Memory ASSIGNED, and Test Memory UNASSIGNED, is located on the Duplex Maintenance Console. All statements previously made in the description of test memory operation are true only when this switch is in the Test Memory UNASSIGNED position. If the switch is in the Test Memory ASSIGNED position, the following differences exist:

1. Referral to address $377,760_8$ will effect action as though the first test memory plugboard register were plugged in the "A" position only.

2. Referral to address $377,761_8$ will effect action as though the second test memory plugboard register were plugged in the "B" position only.

3. Referral to address $377,777_8$ will effect action as though the last test memory plugboard register were plugged in the "L" position only.

All other test-memory plugboard-register plugging in the "A", "B", "L", and "P" hubs effects actions in the same way as discussed when the Test Memory - ASSIGNED/UNASSIGNED switch is in the UNASSIGNED position. Any attempt to store a computer word into a test-memory storage register other than the test register, results in that word's being placed into the test register regardless of the plugging of that register, and regardless of the position of the Test Memory - ASSIGNED/UNASSIGNED switch on the Maintenance Console.

The addressing scheme may be stated as follows:

Addresses of core-memory registers range from $\emptyset\emptyset\emptyset,\emptyset\emptyset\emptyset_8$ to $2\emptyset7,777_8$.

Addresses of test-memory registers range from $377,76\emptyset_8$ to $377,777_8$.

The memory-address register relating to each memory unit enables the memory element to find the location within itself of the particular register the computer-word contents of which are currently needed, and to effect access to the contents of that register.

Random Access

The ability of the memory element to refer to the location of any given register at any time is called random access. This means that any core-memory address can be designated at random, and the memory element will locate the address directly, without having to search through all core-memory locations until the associated register is found. For example, if we call for the information contained in core-memory location 7777_8 (the 4,Ø96th register) the memory element will not search through 4,Ø96 registers to find the information requested. It will locate the register directly within six microseconds. Random access provides the Central Computer System with the speed of operation needed for real-time data processing.

The Clock Register

The arithmetic element contains a register, having sixteen bits, that provides a means of real-time determination. Once every one-thirty-second of a second, this register's contents are increased by one. By requesting the contents of this register, a program can, during its execution, determine precisely the amount of time elapsed since the previous operation in which the clock register contents were obtained and stored. The contents of the clock register are, like the contents of any other memory register, routed through the memory-buffer register.

Arithmetic Element

The arithmetic element (Figure 2-3) consists of one special circuit and three flip-flop registers that enable it to handle all arithmetic and logical data-processing operations performed by the Central Computer System and the SAGE Computer. The three registers are: the A-register, the accumulator, and the B-register. The special circuit is the adder circuit and is associated with the A-register and the accumulator.

The A-Register

The A-register is a simple flip-flop register that serves primarily as a temporary storage register for computer words entering the arithmetic element from the memory element. When so instructed, it will receive the computer-word contents of the memory-buffer register and retain them until their use in the arithmetic or logical data-processing operation in process is required. A second and very important function of the A-register is to contain, during an arithmetic or logical operation, one of the two operands involved and to supply this operand as often as is necessary during the operation.

The Accumulator

The accumulator register performs the function of containing the second of two operands during the execution of logical or arithmetical operations, supplying the operand for use during the operation, and finally, containing in part or in whole, the results of the operation. The results may be stored in a memory register as a function of instruction execution, but this type of transfer will not destroy the accumulator's contents. The contents of the accumulator remain unaltered until cleared through the execution of a subsequent instruction that requires an empty accumulator for its execution. The rightmost sixteen-bit positions of the accumulator (right accumulator) are logically shared with and constitute a register in the program-control element.

The B-Register

The B-register is an auxiliary register that augments certain arithmetical and logical operations by either extending the capacity of the accumulator or by serving as an intermediate storage center for quantities obtained and/or used during the course of the operation. In those logical operations requiring three operands, the B-register serves as the storage register for the third operand, the first operand being stored in the A-register, and the second in the accumulator.

The Adder Circuitry

The adder circuitry links the accumulator and the A-register when the contents of the registers are used jointly in an arithmetical or logical operation, and places the results of this operation in the accumulator. The B-register in certain arithmetical and logical operations, when used as an extension of the accumulator, also receives results of these operations from the adder circuitry.

Word Configuration in the Arithmetic Element

The selection of a half-word configuration to be used with computer-word operands is exclusively a function of the instruction using the computer words as operands. For instance, if two computer words, to be arithmetically or logically used as operands, are present in the arithmetic element, the instruction specifying the operation determines whether a balanced (sixteen-bit left and right half-word) or an unbalanced (fifteen-bit left half-word and seventeen-bit right half-word) configuration is to be used in the following dual-arithmetic operation. The instruction uses the adder circuitry to obtain the appropriate half-word bit positions from each computer word and to produce results corresponding to the selected configuration of each half-word. The dual arithmetic performed in the arithmetic element can be directed to use either computer-word configuration.

THE PROGRAM CONTROL ELEMENT

The program-control element is responsible for the control of four important aspects of Central Computer System operation.

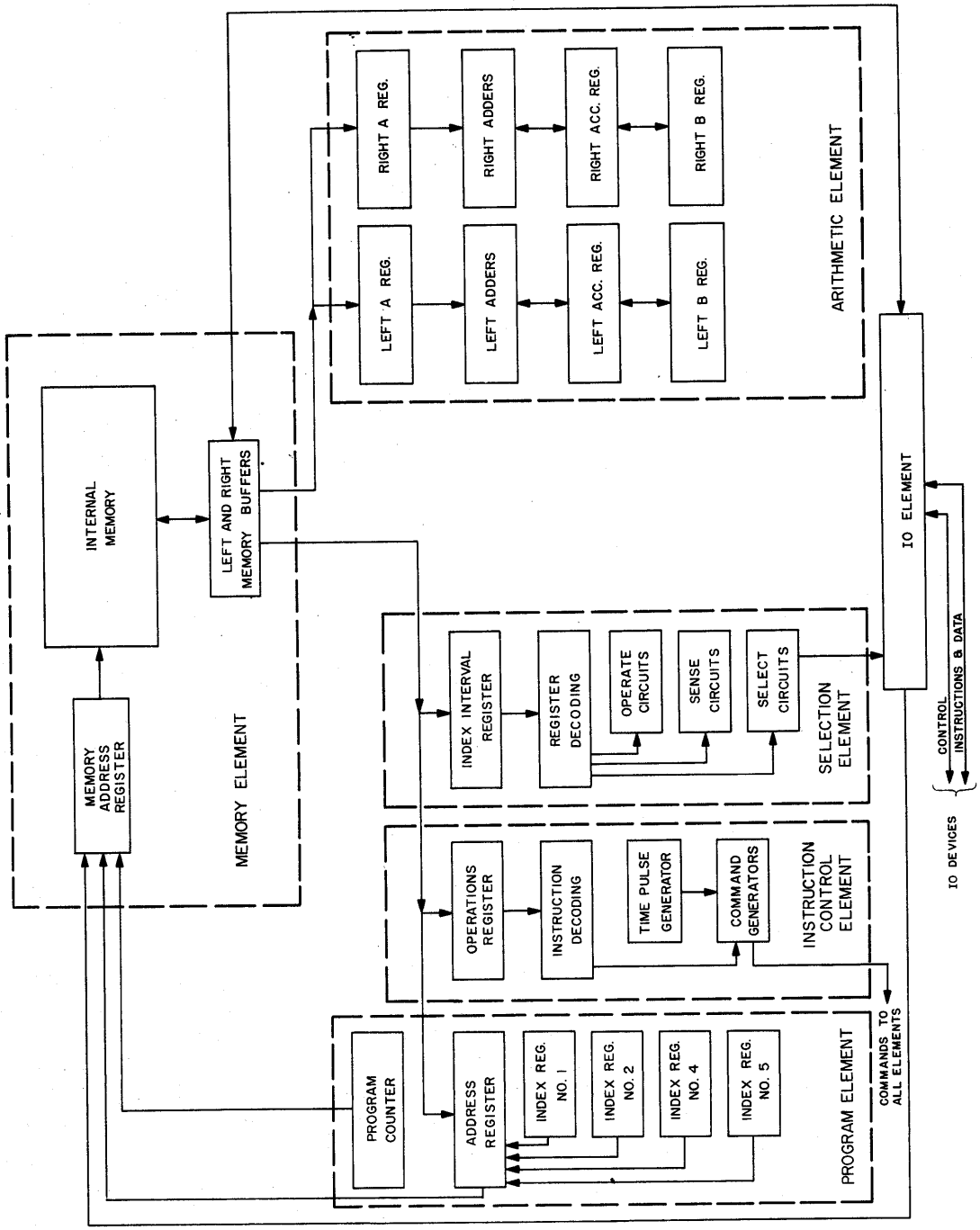


Figure 2-3. Overall Information Flow in the Central Computer

1. All information (addresses) routed to the memory-address registers of the memory element;

2. The counting aspects of arithmetic and logical operations in which a given number of repetitious operations are involved;

3. The number of computer words transferred between the memory element of the Central Computer System and the Selection Control Element of the Central Computer System;

4. The acceptance-determination and/or routing of information to the selection-control element and subsequently to the memory element. To fulfill these responsibilities the program-control element has one special circuit and twelve registers. The special circuit is called an index adder. One of the twelve registers is shared in logical function with the arithmetic element (the sixteen bit right accumulator). The twelve registers are:

1. The Program Counter
2. The Address Register
3. The Index Registers (four)
4. The Right Accumulator
5. The Step Counter
6. The I/O Address counter
7. The I/O Word counter
8. The Drum Control Register
9. The I/O Buffer Register.

} Considered with input/output programming

The Program Counter

The program counter controls the sequence of instruction execution. At the initiation of an instruction, one of the first actions in the instruction's executions, is to increase the contents of the program counter by one. Further changes of the program-counter register may take place, again, as a result of the actions of the instruction. The contents of the program counter immediately following the completion of an instruction will always express the address of the location (register) in the memory element in which the computer word containing the next instruction is to be found. Most instructions that change the program-counter content, in addition to the initial increase by one, do so only if some given condition of the SAGE Computer is present. In fact, these instructions afford a decision-making capability within the computer programs where their execution is accomplished. Programs are stored in the memory element, with the first instruction

to be executed placed in a given register, the next instruction to be executed in the next higher memory register, etc. The only exception to this is when an instruction that changes the program-counter contents other than by adding one to the initial count, appears in the program. In this case, the next instruction to be executed may or may not be located in the next higher addressed register, depending on whether the execution of the decision-making instructions changes the program counter.

When a given instruction has been completed, the memory-address register to which the current contents of the program counter refer, receives the contents, and subsequently makes the computer word, designated as the next instruction, available for processing.

The Address Register

The seventeen-bit right half-word of an instruction word is sometimes concerned with expressing the address of a register in the memory element the contents of which are pertinent to that instruction's execution. At other times, these seventeen bits are used for different purposes. Regardless of the purpose, these instruction-word bit positions are routed to the address register at the initiation of an instruction. When the instruction requires a computer word operand or result to be obtained from or placed into some register of the memory element, the contents of the address register are routed to the appropriate memory-address register, initiating the transfer. In a decision-making instruction, the seventeen-bit right half-word is used to express the address of the register containing the next instruction to be executed. If a decision-making condition is present in the SAGE Computer, the right half-word is routed to the program counter. Otherwise no such transfer is made. The transfer of address-register contents to other registers within the Central Computer System is described with the appropriate register.

The Index Registers and the Right Accumulator

The four index registers in the program-control element of the Central Computer System provide on request the value stored within them to be added to the contents of the address register. An instruction word that addresses one of the four index registers, adds the contents of the designated index register to its own right half-word. The result of this addition is placed into the address register, destroying the existing contents (the instruction right half-word). The addition is performed early in the instruction's execution so that the sum obtained, rather than the instruction right half-word, is used during the remaining instruction time. Not all instructions are capable of using index-register contents in this manner; instructions that do use index registers are indexable; those that do not are non-indexable. The contents of the index registers are established by means of specific instructions. In one of these instructions the value at which the index register is to be set is expressed in the instruction word, along with the address of a particular index register. One particular decision-making instruction is capable of reducing a given index register. (This instruction also bases its decision on the amount expressed within the index register.)

The program-control element can be instructed to use the contents of the right accumulator as though the accumulator were a standard index register. Thus, the right accumulator can be spoken of as sharing in logical function with both the program control and the arithmetic elements. Use of the right accumulator's contents to increase

the address register count is the same in all respects as an index register. The instructions that establish values in the standard index registers, however, cannot be used to establish a value in the right accumulator; this is strictly a function of the arithmetic element's operations. The instruction, mentioned previously, that reduces index register contents cannot be used to change the contents of the right accumulator. Only the content of the right accumulator may be used within the program-control element operations.

The Index Adder Circuitry

The index-adder circuitry is used in the addition of the index register or right-accumulator contents with the contents of the address register, reduction of index-register contents, and the establishment of index-register contents.

The Step Counter

Certain instructions require that a repetitious operation be initiated within the arithmetic element. The number of times this operation is to be performed is either specifically designated in the right half-word of the instruction word, or is always automatically the same for a given instruction, and is made available regardless of the context of the instruction word. In either case, the step counter is, at the start of the instruction, made to contain the quantity expressive of the number of repetitions involved. As the operation is repeated, the step counter is correspondingly reduced by one, until the step counter indicates that the number of operations performed in adequate. At this point, the step counter signals the completion of these operations.

THE INSTRUCTION CONTROL ELEMENT

The prime responsibility of the instruction-control element is that of translating the binary expression found in the fifteen bit left half-word of an instruction word into appropriate SAGE Computer actions. (The seventeen bit right half-word of the instruction word is handled by the program-control element.) The instruction-control element consists of two registers--the operation register and the index interval register; four decoding matrices, and one special circuit--the command generator circuit. Several other miscellaneous circuits each contributing to the element's control functions are also present in this element.

The Operation Register

The operation register in the instruction-control element is responsible for receiving those bit positions in the left half of the instruction word that express the following:

1. The class of instructions to which a particular instruction belongs;

2. The variation within the class which sets apart the instruction from all other instructions within the same class;

3. The index register the contents of which are to be associated with the execution of the instruction if such association is valid for the instruction.

On receipt of these bits, the operation register makes its contents available to three of the four decoding matrices. The functions of the decoding matrices are described separately.

The Index Interval Register

The index-interval register receives those bit positions in the left half of the instruction word the contents of which may augment the actions caused by execution of the instruction by designating equipment or optional actions, or by specifying values necessary in obtaining all of the computer actions desired. On receipt of these bits, the index-interval register makes its contents available to one of the four decoding matrices.

Decoding Matrices

The functions of the four decoding matrices of the Instruction Control Element of the Central Computer System is translating a binary expression into SAGE Computer actions are described below:

Class Matrix

The class matrix is responsible for decoding the bit configuration of that portion of the operation-register content indicating the class in which the instruction to be executed belongs, and for transmitting this decoded information to the command generator.

Variation Matrix

The variation matrix is responsible for decoding the bit configuration of that portion of the operation-register content expressive of the variation within an instruction class that sets apart the instructions to be executed from the other instructions within the same class. The decoded information is routed to the command generator.

Index Interval (PERSELBSN) Matrix

The PERSELBSN Matrix can, as a function of the execution of certain instructions, decode the content of the index-interval register and route the decoded information to the selection-control element for translation into SAGE Computer actions. Index-interval register contents not routed to the PERSELBSN matrix for decoding are either not used or are decoded by other elements of the Central Computer System.

Index (Register) Selection Matrix

The index-selection matrix receives information from those bit positions in the operation register the contents of which designate an index register for use in the following operation. The output from this matrix is routed to the command generator.

The Command Generator

The Command Generator, on receipt of information from the class, variation, and index selection decoding matrices issues timed-control impulses to the various circuits throughout the Central Computer System. The timed control impulses effect the coordinated functioning of the various circuits in performing the actions required to accomplish the instruction being executed. Certain of these timed control impulses result from information received from the class decoder matrix. Once these have been determined, the information from the variation decoder matrix causes the additional timed

control impulses necessary to effect computer actions pertinent to the particular variation of an instruction class that defines the instruction being executed. The information from the index selection matrix is, if required by the instruction, used to generate timed control impulses to use a given index register in the program control element. When all of the timed impulses have been sent from the command generator, the instruction's execution has been accomplished.

Miscellaneous Circuits

The instruction-control element has various circuits that serve to control the cycle timing operations of the Central Computer System. These circuits are controlled by the command generator and include the following:

The PT-OT Flip-Flop. In one stable state, the circuit defines the current (or next) machine cycle as a program time cycle. In its other stable state, the circuit defines the current (or next) machine cycle as an operate time cycle.

The A-B Flip-Flop. The A-B flip-flop operates in conjunction with the OT setting of the PT-OT flip-flop to further define the operate time as OTA (in one stable state) or OTB (in the other stable state).

The I/O Interlock Flip-Flop. In one stable state, the I/O interlock flip-flop defines an I/O operation as being in process; in the other, no I/O operation is currently being performed.

CHAPTER 3

BASIC PROGRAMMING

INTRODUCTION

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.

Simultaneous with our consideration of the AN/FSQ-7 instruction repertoire will be a presentation of the coding for these instructions. It will be very important to maintain a clear distinction between rules and limitations imposed by the operation of the computer and those imposed by coding conventions.

ADD CLASS INSTRUCTIONS

The ten instructions grouped within this class involve adding the contents of a specified memory location or register to the contents of the accumulator. Because subtraction in the Central Computer System is accomplished by the addition of complements, those instructions which involve subtractions are also included in the add class. All the instructions in this class require a PT and an OT cycle for execution. The indexing control circuits are conditioned during the execution of any add class instruction. However, not all add class instructions are indexable. Similarly, only five of the ten instructions can cause "overflow". The add class includes six relatively simple instructions. At this time, only these six instructions--CAD, ADD, TAD, CSU, SUB, and TSU--will be described. Although not in the add class, the "halt" instruction, by necessity, will be included in this discussion.

CAD Instruction

The "clear and add" (CAD) instruction will clear the accumulators (L/acc, R/acc) and transfer the contents of the memory location specified by the director portion of the instruction word into the accumulators. The contents of the specified memory location and the B registers remain unchanged. The following table is a summary of the affected registers before and after the execution of the CAD instruction.

		BEFORE OPERATION		AFTER OPERATION	
word in memory		L acc	R acc	L acc	R acc
ø.12345	ø.12345	1.23671	ø.41211	ø.12345	ø.12345
1.54321	1.54321	1.77777	1.77777	1.54321	1.54321
1.77777	ø.øøøøø	anything	anything	1.77777	ø.øøøøø

The CAD instruction is used to enter some quantity (X) into the accumulators. To generate the sum, S = X + Y, the first step would be to enter "X" into the accumulators by programming the instruction CAD the memory location containing "X".

The CAD instruction obviously cannot generate an overflow. The octal operation code for CAD is .01000, and the instruction may be indexed. These three pieces of information can be obtained for any Q-7 instruction from the Programmer's Coding Card. A copy of this card is appended to this text.

From the above, the instruction CAD 1øøø will be seen to have the following action:

a. The accumulators will be cleared (set to +ø, +ø).

b. A copy of the contents of core memory location 1øøø will be brought into the accumulators.

c. The B registers and core memory will remain unchanged.

Coding for the CAD instruction

When the instruction CAD is desired, it would be inconvenient to have to write on a coding sheet the actual binary number which will be in core when the program is operated:

	Left Half Word (LHW)	Right Half Word (RHW)
Binary	ø. øøø øø1 øøø øøø øøø (Operational Code for CAD)	ø. øøø øø1 øøø øøø øøø
Octal	ø. ø 1 ø ø ø	ø. ø 1 ø ø ø

Even the "octal equivalent" expression is unwieldy, but it deserves consideration. The first digit in front of the binary point in each half-word is a binary zero. Then there follows in each half-word five octal digits. It is in this way that we represent the 16 binary bits of each half-word. Bits L1 through L15 inclusive represent the operation-specifying portion of the instruction. Within these 15 bits, the central 9 actually specify the instruction CAD. Its octal code is 1øø, as the programmers card shows. This value has been placed into the middle three octal positions of the LHW.

The Director of the instruction (bits LS and RS through R15) represent the octal address 1ØØØ.

Binary: Ø. --- --- --- --- --- Ø. ØØØ ØØ1 ØØØ ØØØ ØØØ

or $Ø_2$ $Ø_2$ ØØØ ØØ1 ØØØ ØØØ ØØØ

or $Ø_8$ 1_8 $Ø_8$ $Ø_8$ $Ø_8$

or Octal: $ØØ1ØØØ_8$

The following are other sample conversions from augmented 17 bit configuration to octal:

CAD 12345	Ø.Ø1ØØØ	Ø.12345
CAD 1Ø7732	Ø.Ø1ØØØ	1.Ø7732
CAD 177777	Ø.Ø1ØØØ	1.77777
CAD 2ØØØØØ	1.Ø1ØØØ	Ø.ØØØØØ
CAD 2Ø7777	1.Ø1ØØØ	Ø.Ø7777
CAD 377777	1.Ø1ØØØ	1.77777

(17 bit director portion of the instruction is underlined).

ASSEMBLING AN INSTRUCTION

Though the assembly process will be considered in detail at a later point, the symbolic coding for each instruction will be stressed from the beginning. Below is shown the instruction CAD $1ØØØ_8$ coded in COSEAL assembly format:

	LOCATION							R	OP CODE			AUX.		RC WORD						ADDRESS						+ − INCR.			
17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39	40	41	42	43	44	45
								C	A	D											1	Ø	Ø	Ø				

Notice that the symbolic mnemonic for the "Clear and Add" instruction, that is CAD, has been coded in the "Op Code" columns, 25, 26, and 27. The octal address 1ØØØ has been put into the address field, columns 36 through 41, right justified.

Throughout our discussion of the Q-7 instructions, we will first show the action of the instruction, and this will be followed by an explanation of the pertinent coding rules.

ADD Instruction

The ADD 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. The octal code for this instruction is 1Ø4. 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.

The arithmetic capacity of the AN/FSQ-7 is limited to those numbers ranging between ± 1; consequently, the execution of the ADD instruction could cause an overflow. Overflow can occur only when either two positive numbers or two negative numbers are added. It can never occur, however, as the result of adding a positive number to a negative number. Two examples of an overflow condition are given below:

```
   ADD
   Ø.77776
+  Ø.ØØØØ2
   1.ØØØØØ
```

Notice that the addition of two positive quantities has generated a sum which is the most negative number the computer can handle.

```
   ADD
   1.ØØØØØ
+  1.ØØØØ1
  (1) Ø.ØØØØ1
carry ↘ 1
   Ø.ØØØØ2
```

Notice that the addition of two negative numbers (-.77777 and -.77776 has resulted in a positive sum +.ØØØØ2).

Whenever the sum of two quantities is logically greater than $\pm$.77777, an overflow is generated, and the result is therefore meaningless. Circuits are built into the computer which note the fact that an overflow has occurred. Furthermore, the programmer, by program action, can check for an overflow; and if an overflow occurs, he can include appropriate program instructions to remedy the situation.

LOCATION	R	OP CODE	AUX.	RC WORD	ADDRESS	± INCR.
		CAD			5	
		ADD			6	
		ADD			7	
		ADD			1Ø	
		HLT				

The preceding program will first clear the accumulators (set them to +Ø, +Ø), then obtain the sum of the contents of memory registers 5, 6, 7 and 1Ø. (Remember that these are octal addresses.) Note that another instruction has been added.

HLT (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 15Ø punched cards, all 15Ø 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 ØØØ (bits L4-L1Ø). 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.

Notice that in the coding for the instruction HLT, we put no address in the "address" field on the coding sheet. If the field is left blank, the COSEAL assembler will automatically insert +Ø in the right half word.

TAD 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, the TAD instruction is similar in execution to the ADD instruction. The following table is a summary of the affected registers before and after the execution of a TAD instruction:

	BEFORE OPERATION		AFTER OPERATION	
word in memory	L acc	R acc	L acc	R acc
1.11111 1.22222	Ø.ØØØØØ	Ø.ØØØØØ	1.11111	1.11111
Ø.ØØØØ1 1.23456	Ø.ØØØØ3	Ø.ØØØ24	Ø.ØØØØ4	Ø.ØØØ25

The octal operation code for the TAD instruction is .Ø11ØØ. Since the same value is added to the contents of both accumulators, overflow may occur in either or both accumulators, depending on their original contents.

LAD Instruction

The "left add" (LAD) instruction causes the left half portion of the contents of the specified register to be added to the contents of the left accumulator. The right half of the memory register and the right accumulator are not used. Otherwise, the LAD instruction is similar in execution to the ADD instruction. The following Table is a summary of the affected registers before and after the execution of a LAD instruction.

		BEFORE OPERATION		AFTER OPERATION	
word in memory		L acc	R acc	L acc	R acc
1.76432	ø.41356	ø.13124	1.46532	ø.11557	1.46532
ø.ØØ635	1.42171	1.17631	ø.41522	1.2ø466	ø.41522

The octal operation code for the LAD instruction is .ø12ØØ. Overflow is possible in the left accumulator only.

CSU 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 first cleared to positive zero, and then the complement of the contents of the register specified in the address portion of the instruction are transferred to the accumulators. The contents of the memory location specified and the B registers remain unchanged. The CSU instruction will not cause a computer overflow. An octal operation code of .ø13ØØ designates the CSU instruction. The following table illustrates the conditions of the accumulators prior to and following the execution of a CSU instruction.

		BEFORE OPERATION		AFTER OPERATION	
word in memory		L acc	R acc	L acc	R acc
ø.12367	1.325ø4	anything	anything	1.6541ø	ø.45273
1.ØØØØØ	ø.77777	1.77777	1.77777	ø.77777	1.ØØØØØ

SUB Instruction

A "subtract" (SUB) instruction is used to subtract the contents of the selected memory register from the accumulators. The accumulators are not cleared. Consequently, the result which appears in the accumulators will be the difference between the original contents of the accumulators and the contents of the specified memory location. The contents of the memory location and the B registers remain unchanged. The octal operation code to identify the SUB instruction is .ø134ø, and it should be noted that the instruction may cause an overflow. The following table summarizes the action of an SUB instruction:

		BEFORE OPERATION		AFTER OPERATION	
word in memory		L acc	R acc	L acc	R acc
ø.12345	ø.76542	ø.12345	ø.76542	1.77777	1.77777
ø.63152	ø.321ø5	ø.76543	ø.54321	ø.13371	ø.22214
ø.641ø2	ø.76543	ø.5321ø	ø.62431	1.671ø5	1.63665

TSU Instruction

The "twin and subtract" (TSU) instruction is used to subtract the left half portion of the specified memory register's contents from both the left and right accumulators. This instruction is similar in execution to the SUB instruction--the difference, as usual, appears in the accumulators. The contents of both the memory location and the B registers remain unchanged. Overflow, as a result of using the TSU instruction, may appear in either or both accumulators. The octal operation code that designates a TSU instruction is .ØØ14ØØ.

Other ADD Class Instructions

The remaining add class instructions--ADB, CAM, DIM, and CAC--are more complex. Consequently, they will be described after you have become proficient in the use of the six simpler instructions.

CHAPTER 4

THE ASSEMBLY PROCESS

GENERAL

In the preceding chapter we coded the instructions on COSEAL coding sheets without giving clearcut reasons. In this chapter we will discuss what is done with these sheets and how an instruction gets from them into the computer.

Each line of the coding sheet corresponds to one EAM punched card, columns 1 through 8Ø. The Q-7 cannot read columns 1 through 16, so on any card to be fed into the computer, these columns may contain any information desired by the programmer (e.g., his name). Symbolic coding information is punched by the programmer in columns 17 through 8Ø. Coding for these columns becomes the primary emphasis in learning machine language programming.

There exists a utility system for the Q-7 known as COSEAL (COMPASS System Extensively Altered). This is a system of programs which perform various functions for programmers which are constantly required. This includes programs for tape handling, giving quick dumps of core or drums, and for constructing basic tables of data required by other programs. There are programs to perform massive data reduction, and libraries of subroutines to do everyday jobs, subroutines that a programmer can incorporate into a program of his own.

The primary program in the COSEAL utility system is the assembler, called "Translator". It is this program which takes symbolic cards (coded by the programmer), reads them in, decodes them, and translates them into binary instructions. This binary program is then put out onto binary tapes or cards. These binary cards or tapes can then be put back into core memory and run. They are the final program as written by the programmer. This two step operation is called the assembly process. To review:

1. Symbolic cards are read in by "Translator", a program in the COSEAL Utility System.

2. Symbolic coding is translated to binary instructions by "Translator".

3. A binary program is put out onto binary tapes or cards by "Translator".

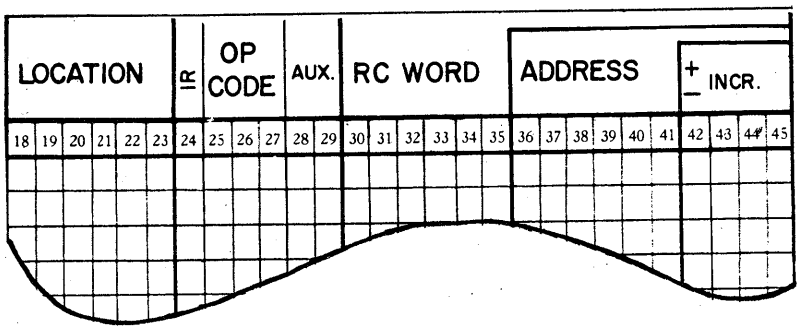
4. The binary program is read back into core memory from tape or cards.

5. The binary program is operated.

CODING

On the 8Ø column coding sheet, columns 1 through 16 are not interrogated by the computer, as stated above. Therefore, we will concern ourselves with columns 17 through 8Ø. In normal COSEAL assemblies, columns 17 and columns 72 through 8Ø are not used. Columns 46 through 71 are used for programmer comments and may contain anything that the programmer wishes. (It should be noted from the outset that the more

comments the programmer uses, the easier will be the subsequent debugging.) Our primary concern, however, will be columns with 18 through 45.



In the accompanying drawing of a Q-7 coding sheet, you will observe the major breakdown of columns 18 through 45:

Columns 18 - 23

Symbolic names to be given to a particular location within a program. These names may be from one to five alphanumeric symbols long in columns 18 to 22. The only restriction on the names to be used is that they must contain at least one letter.

Examples:

```
START
DONE
COUNT
Ø1A
7736X
 END
```

Columns 24 - 29

These columns delineate the left half word of the memory location. They will usually contain the mnemonic code for an instruction in columns 25 - 27. When this is the case, column 24 would specify an index register, and columns 28, 29 would give certain other pertinent information relevant to the particular instruction specified.

Columns 3Ø - 35

Columns 3Ø through 45 are often used in their entirety to specify an "RC word." These words will be considered at a later point.

Columns 36 - 41

These columns specify an address. It may be an octal address; e.g., 1ØØØ, Ø, 2Ø7777, 377777, etc. It might be a symbolic address which was originally defined elsewhere in your program in columns 18 - 22. Numeric addresses are right justified, symbolic are left justified. Although one need not refer to all addresses he defines, one must define all addresses to which he refers.

Examples:

ADDRESS					
36	37	38	39	40	41
S	T	A	R	T	
Ø	1	X			
				5	7
1	7	7	3	4	6

Certain types of tags are used for naming common types of entities:

Item names - LLLL (Letter-letter-letter-letter)

Table names - LLLD (Letter-letter-letter-digit)

Air Defense Programs - LLL (Letter-letter -letter)

Columns 42 - 45

These columns, known as the "increment" field always modify the address, symbolic or numeric, in cols. 36-41. A plus or minus sign is coded in column 42, and a decimal increment, right justified, in columns 43-45. Typical would be:

```
+  2
+3Ø4
-999
-  38
```

Such increments imply that the address in columns 36-41 is not the address actually desired, but rather, the programmer wants that address plus or minus the increment in cols. 43-45.

```
e.g.,   CAD     ØlA   -  4
        SUB     START + 55
        TSU     177345-9ØØ
```

If the programmer wanted to add the contents of the tenth register of table TPYØ to whatever was already in the accumulators, he would, code

```
ADD     TPYØ + 9
```

(The "9" is not a mistake; the first register of table TPYØ is obviously TPYØ+Ø, or, more simply, TPYØ itself.)

Other Forms of Addressing

Another useful symbol which "Translator" can recognize is the "self-reference" symbol, a dollar sign in column 36. Without bothering to figure out where an instruction will eventually be in core, the programmer may wish to refer to an instruction or data in a core memory register five past where the present instruction is located. This may be accomplished by coding:

```
ADD     $  +  5
```

which can be interpreted as, "add the contents of 'self' plus five". When this program is eventually operated, and if the above ADD instruction happened to be at location 1ØØ1, for example, the contents of location 1ØØ6 (that is, 1ØØ1 + 5) would be added to whatever was already in the accumulators.

The letter "T" in column 41 is interpreted by Translator as a request for the first register in Test Memory. A positive decimal increment of from 1 to 15 in columns 42 through 45 will address any other Test Memory register.

Examples:

T.	is equivalent to.	37776Ø
T + 1. . . .	" ".	377761
T + 7. . . .	" ".	377767
T + 1Ø . . .	" ".	377772
T + 15 . . .	" ".	377777

PSEUDO INSTRUCTIONS

The programmer may use certain instructions that appear, on the coding sheet, similar to a AN/FSQ-7 instruction, but for which there is no equivalent instruction on the Q-7 Programmer's Coding Card. Such instructions are known as pseudo instructions. These instructions are of two types:

1. Generative Pseudo Instruction - This type of pseudo instruction generates a binary output which will appear, in the register for which it was coded, within the programmer's binary output program.

2. Non-Generative Pseudo Instructions - Such instructions generate no binary output but are, however, recognizable to the Translator Assembler Program.

The three non-generative pseudo instructions will be discussed here. Once again, these are instructions which will yield no binary output but are, however, instructions to the program Translator, giving certain information to that program--that is, telling it how to assemble your program.

IDT Instruction

The first instruction is the Program Identification Instruction, or more simply IDT. The letters IDT would be coded in columns 25 through 27, inclusive, and a Director would be coded in columns 36 through 41. This Director can be any combination of alphanumeric characters including blanks and may consist, indeed, of all numbers. It is this Director, in columns 36 through 41, which will become the name of the program. The name is arbitrary, and may be anything meaningful to the programmer. If this is to be a program used in Air Defense, the IDT should begin with the numeral "5" in column 36. Inasmuch as your student programs are not to be a part of the Air Defense program, this is the one thing which you, as a student programmer, should avoid. Coding a five in column 36 would make your program compool-dependent. Another way to achieve compool dependency is coding the letters COMPOL in columns 3Ø through 35. Both of these methods of invoking the Compool should be avoided in your student programs. If your program requests that the binary output program be punched onto cards rather than be put onto a binary output tape, the binary cards will contain this IDENT in columns 1 through 6. This would merely be to identify your deck, inasmuch as the Q-7 would not interrogate these columns when the binary program is read back into the computer. In short, therefore, you need only think of the IDT card as a "Program Namer." Its primary significance in your program will be to tell the "Translator" program that here is where your card deck begins; here is where assembly begins.

LOC Instruction

The next pseudo instruction, likewise non-generative, is the LOC pseudo instruction. This instruction tells the program Translator where in core memory that your program is to be finally located. LOC is coded in columns 25 through 27, and an octal number is coded, right justified, in columns 36 through 41. This octal number defines the address of the first actual binary output constant or instruction. LOC itself yields no binary output. A program assembled without a LOC card is automatically begun at octal location 3ØØ. Unless there are overriding reasons to the contrary, therefore, you, the student programmer, need not include a LOC card with your program under most situations.

END Instruction

The third pseudo instruction, likewise non-generative, is the END card. The primary purpose of this pseudo instruction is to tell the program Translator that this is the end of your deck of cards, and at this point assembly may begin. There is one other function for the END card, of which you may wish to avail yourself. You may, in columns 36 through 41, include an octal or symbolic address. This address will be for the purpose of telling the program Translator where you wish the program to begin. If you do not specify an address on the END card, program Translator will automatically assume that you wish to start your program at the first generated binary output constant or instruction, at the top of the program. There are occasions where the programmer would like to start his program at some other location, possibly down in the middle of

the program. The END card provides the programmer with the means to accomplish this objective. The symbolic or octal location of the starting place of his program could be inserted on the END card.

All programs which the student programmer submits must include an IDT and an END card, minimally. Without these cards, program Translator has no way of knowing where your deck begins and ends. The choice of a name for your program is entirely arbitrary. It is very helpful to include the programmer's name in full in the comment field of the IDT card. It would be well for you to follow this practice.

RC WORDS

One of the most useful and common Directors is the "RC Word". "RC" means "Register Containing". Suppose that you wanted to increase the left half of the accumulator by one. This would require an instruction of the form ADD X, and a register, location X, with the contents Ø.ØØØØ1, Ø.ØØØØØ. For the convenience of the programmer, the Translator will allow an RC Word Director. Instead of some address X, the programmer fills in the desired word in columns 3Ø through 41. The Translator will reserve a register at the end of the program, after the END card, to contain this number. It will supply the number in the register, and fill in the address of the register in the instruction. The instruction to add one to the left half of the accumulator would look like: ADD ØØØØØ1ØØØØØØ, where the first zero appears in column 3Ø, the last zero in column 41. Note that the machine point is omitted on the card, and that there is no space between the left and right half of the RC Word. All of the zeros coded above are not really necessary. The instruction could have been coded as follows: ADD 1 Ø, with ADD in columns 25-27, 1 in column 35, and Ø in column 41. In the above example, leading zeros have been omitted, but program Translator will automatically insert them.

ASSEMBLY DLO OUTPUT

In addition to a binary output program, the program Translator will produce a symbolic listing on tape for the programmer. This tape may be taken from the computer to an off-line facility located outside the computer room, and printed out on a printer located there. The student programmer must become familiar with the meaning of this DLO listing. Certain errors can only be detected by analyzing the binary output which will be listed on the DLO output. Several sample listings have been appended to this study guide, and should be studied by the student with care, and in detail. One example program however, will be considered in detail at this point. A sample coding sheet, Figure 4-1, has been prepared. This program was then punched on symbolic cards in columns 17 through 8Ø and assembled by program Translator. The DLO listing appears as Figure 4-2.

Let us give a careful analysis of the program together with its DLO listing. As you can see, the first punched card in the program was an IDT. The name "JOEØ1" was given to the program. You will see that program Translator adopted the name JOE for this program, and inserted it in the heading at the top of the DLO page. The "Ø1" in columns 4Ø and 41 was adopted as the MOD, or model, of the program. The next card in the program was a LOC card. Program Translator was told to start assembling this program for octal location 1ØØØ. The next instruction, the Clear and Add, was indeed assembled and inserted at location 1ØØØ. Note that there is no binary output generated for the IDT and LOC cards. The next instruction (Clear and Add) shows us the

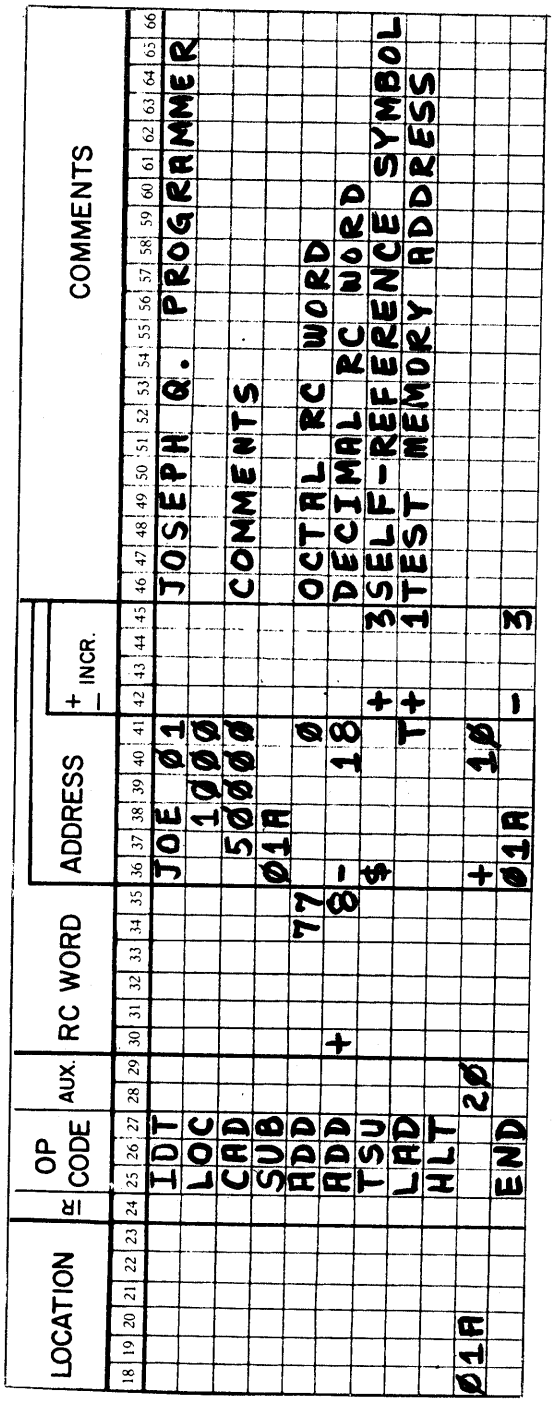


Figure 4-1. Program Illustrating Various Forms of Addressing

```
                                        PROGRAM JOE  MOD 01          COMPOOL  04D 00   PAGE      1

        IDT                 JOE 01
        LOC                   1000
        CAD                  50000          001000  001000 050000  JOSEPH Q. PROGRAMMER        0000.01*T00
        SUB                  01A            001001  001343 001007                             0000.02*T01
        ADD          77        0            001002  001043 001010  COMMENTS                   0001.00*T02
        ADD     +     8  -    18            001003  001043 001011                             0002.00*T03
        TSU                  $     +  3     001004  001403 001007  OCTAL RC WORD              0003.00*T04
        LAD                        T+  1     001005  101203 177761  DECIMAL RC WORD            0004.00*T05
        HLT                                  001006  000000 000000  SELF-REFERENCE SYMBOL      0005.00*T06
01A        20                +    10        001007  000020 000012  TEST MEMORY ADDRESS        0006.00*T07
        END                  01A   -  3                                                       0007.00*T08
                                             001010  000077 000000                             0008.00*+00
                                             001011  000010 177755                             0008.01*T01
```

Figure 4-2. Delayed Output (DLO) Listing of Figure 4-1

first full line of DLO. Note to the left the contents of the symbolic coding sheet, from columns 18 through 45, have been relisted on the DLO. The next three columns are fields of six octal numbers each. The first column, whose first entry is 1ØØØ, indicates the octal address for which that instruction was assembled. Indeed, we had requested that the program be assembled for location 1ØØØ, and program Translator put the first instruction at that location. The next two columns are the left and right half words, respectively, which will occupy the indicated core location. In the left half word we see an octal 1ØØØ. This is the octal code, 1ØØ, for Clear and Add, inserted in the left half-word, in bit positions L4-L12. In the right half-word, we see the octal address 5ØØØØ, which was the director requested by the programmer. The next field, moving rightward, on the DLO output, is a reproduction of the comments originally coded by the programmer. Next comes a series of numbers supplied by program Translator indicating card numbers. In our work, we may largely ignore these numbers. On the extreme right of the DLO listing appears error printouts produced by program Translator during the assembly process. You will note that there is only one entry in this field, that of "Ill Use RC". The meaning of this phrase will be brought out below.

Returning to the Clear and Add instruction, we find the next instruction in line is Subtract. Here a symbolic address has been used, that of Ø1A. You will note that Ø1A appears farther down in the program, and indeed, during assembly it was determined that Ø1A would fall at octal address 1Ø1Ø. When assembling, program Translator inserted this octal address in the right half-word of the Subtract instruction. During the last phase of program development (actual program run) the contents of Ø1A, that is 1Ø1Ø, will be subtracted from whatever is in the accumulators. The next instruction specifies an RC Word. Specifically, the programmer has requested that at this point in the program, the contents of a register which contains "77" in the left half word and "Ø" in the right half-word be added to whatever is already in the accumulators. These are, of course, octal constants. Inasmuch as these constants are to be added into the accumulators, the programmer did not really care where they appeared in core. He therefore availed himself of the RC word capability of program Translator. You will note that program Translator created such a constant and inserted it at the first available memory location after the END card. This turned out to be location $1Ø11_{(8)}$. A register was created with those contents and the address of that register inserted as the director of the Add instruction. Note that 1Ø11 appears as the right half-word of the Add instruction, in the binary output.

The next instruction, Add +8 and -18, illustrates the use of decimal constants. When a plus or a minus sign is inserted in columns 3Ø or 36, whatever number fills the remaining five registers of that particular field will be interpreted as a decimal number. Note the use of 8, and 18, which are certainly not octal numbers. Again, program Translator has created an RC word at the end of the program, which contains the requested constants, translated into octal. It is also worth noting that should the same RC word be requested twice in one program, Translator will only create one such register and use it whenever needed.

The next instruction, the Twin and Subtract, illustrates the use of the self-reference symbol. In this case the instruction says "Twin and Subtract register self plus three".

Inasmuch as it worked out that the Twin and Subtract instruction occurs at location 1ØØ4, a "1ØØ7" has been inserted as the Ddirector of the Twin and Subtract instruction. (That is, $1ØØ7_{(8)}$ plus $3_{(8)}$.)

Moving on, we find the next instruction to be a Left Add. Use has been made of the Test Memory addressing capability of program Translator. A "T" has been inserted in column 41 and an increment in columns 42 through 45 has been utilized. Since the programmer has requested address T + 1, program Translator has inserted address 377761.

The next instruction, the Clear and Add, has been used to illustrate a typical error printout from program Translator. An RC word was requested, but you will note that none has been created to correspond with the zero and nine. This is because the nine is obviously a decimal integer, but without a plus or minus sign preceding it, program Translator attempts to decode it as an octal number. In so doing, an obvious error situation exists. Therefore program Translator flags this location as "Ill Use RC".

The next instruction is a Halt instruction and has been assembled as "Ø" in the left half word. You will note that the programmer inserted nothing in columns 36 through 41, and program Translator therefore automatically inserted a zero. This capability of automatically assuming zero, should the programmer code nothing, is built into the assembly program. The next location, tagged Ø1A, contains a constant inserted in the left and right half words respectively. In the left half-word the programmer has requested an octal 2Ø, and in the right half-word he has requested a +1Ø, namely a decimal number. You will note that program Translator inserted such a constant at that point in the program performing translation where called upon to convert the decimal 1Ø to an octal 12.

We next encounter the END card. Note that the programmer has inserted a symbolic address in columns 36 through 45 inclusive. Address Ø1A -3 has been requested, which is the address of the Left Add instruction. For some reason the programmer wished for his program to actually begin operating with the Left Add instruction. Had he not inserted any address in columns 36 through 45, symbolic or octal, the program would have automatically begun operation, when finally in binary form, with the Clear and Add 5ØØØØ instruction.

It is obvious that the assembled program is little more than nonsense coding. However, it does illustrate many important principles and capabilities of program Translator. Throughout your further progress in this study guide, you should make frequent reference to the example programs, which we have mentioned were appended to this study guide. Examples of almost all coding situations which are referenced in this block of instruction will be found in the Appendices. Although specific reference may not be made to all such programs in the back of this book, nonetheless the student should avail himself of the opportunity to study sample DLO listings. Coding sheets for these programs have not been shown; however, the contents of the original coding sheet can be reconstructed from the DLO listing of columns 18 through 45 and the comment fields in columns 46 through 71, inclusive.

CHAPTER 5

SIMPLE STORE CLASS INSTRUCTIONS

INSTRUCTION DEFINITIONS

In this chapter we will discuss five simple Store Class instructions in the AN/FSQ-7 repertoire, and go into some further capabilities of program Translator. As you encounter these instructions you should consult your Programmer's Coding Card, so as to discover certain pertinent facts listed there. Things to note would be the octal code for the instruction, its indexability, and whether it can cause overflow. It should be noted that only one of the instructions that we will discuss in this chapter is capable of causing overflow.

FST Instruction

The Full Store instruction takes a copy of the contents of the accumulators and stores them into the specified memory location, destroying any previous contents of that memory register. The contents of the accumulators and other arithmetic registers are unchanged. This is our first instruction offering us the capability of taking results we have obtained in the accumulators, and storing those results into memory for later reference. A sample Store Class program is shown in Figure 5-1.

LOCATION	IR	OP CODE	AUX.	RC WORD	ADDRESS	± INCR.
18 19 20 21 22 23	24	25 26 27	28 29	30 31 32 33 34 35	36 37 38 39 40 41	42 43 44 45
		IDT			STORE	
		CAD			$	+ 3
		TAD			ALPHA	
		FST			BETA	
		HLT				
		END				

Figure 5-1. Simple Store Class Program

You will note that it contains an IDT card and an END card to define the limits of the program. The first instruction in this program is a Clear and Add $ + 3. This instruction will bring the contents of the Halt instruction into the accumulators. Since the octal code for Halt is ØØØ, and nothing has been coded in the right half-word, therefore the net effect of the Clear and Add instruction will be to initially clear the accumulators. Next, the contents of location ALPHA have been twinned and added into the accumulators. This will have the effect of taking the left half-word of location ALPHA and placing it into both the left and right accumulators. Next, the results of this twinning and

adding have been stored into location BETA. This program, therefore, has the effect of storing a copy of the left half word of location ALPHA into both the left and right half words of location BETA.

Another way of indicating the contents of a given memory or arithmetic register is to use the following notation:

C(ALPHA).

Indicated by the above notation are the "Contents of Location ALPHA". This is a useful way to refer to the contents of a memory register and/or an arithmetic register, and will be frequently employed throughout this text.

Left Store Instruction (LST)

The Left Store instruction places a copy of the contents of the left accumulator into the left half-word of the memory location specified by the instruction director. The original left half-word of the memory location specified is thereby destroyed. The original contents of the right half-word of the memory location specified, and all arithmetic registers, are unchanged.

Right Store Instruction (RST)

The action of the Right Store Instruction is similar to that of the Left Store. A copy of the contents of the right accumulator is stored into the right half-word of the specified memory location. The previous contents of the right half-word of the specified memory location are thereby destroyed. The original contents of the left half-word of the specified memory location, and the contents of both accumulators, are left unchanged.

Both the Left Store and Right Store Instructions, similar in action, enable us to have greater control over the data we store in memory. They perform actions similar to the Full Store Instruction, but enable us to gain control over half-words.

Store Zeros Instruction

The STZ (Store Zeros) Instruction sets the memory location specified by the instruction director to Positive Zero. Both left and right half-words will be fully cleared. Referring to your Programmer's Card, you will note that this instruction is one of the "illegal" instructions. Further explanation of the illegal instruction concept will be made at a later point in this volume, so that at this point you may consider such instructions to be normally available and entirely legal within Q-7 programming. The Store Zeros Instruction enables the programmer to clear out selected memory locations prior to using such locations for the storage of data.

Add One Right Instruction

The AOR (Add One Right) instruction enables the programmer to increment the contents of the right half-word of a memory location by one. Specifically, the instruction brings a copy of the contents of the right half-word of the specified memory location

into the right accumulator, erasing any prior contents of the RHW of that register. "One" is then added to the right accumulator. The results of this addition are then stored back into the right half-word of the specified memory location. The programmer should view this instruction in its primary purpose. This is to add one to the right half-word of the specified memory location. However, he should not lose sight of the fact that since the addition is accomplished in the right accumulator, the previous contents of the right accumulator will be destroyed. It is this effect, which in hurried programming is often forgotten, which causes endless problems in debugging. Might a word to the wise be sufficient?

The AOR Instruction finds frequent use in two primary areas. The first is keeping track of the count of certain characteristics of data. The following program offers an example of such a use:

```
          STZ       CNTR
          . . . . . . . . . . . . . . .
          AOR       CNTR
          . . . . . . . . . . . . . . .
          AOR       CNTR
          . . . . . . . . . . . . . . .
CNTR      HLT
```

Early in the program location CNTR (counter) is cleared with an STZ instruction. This will initialize our counter. Later on in the program, after certain characteristics of our data have been ascertained, we determined that a certain piece of data has indeed the characteristic for which we have been searching. We wish to add one to the counter to note this fact. The Add One Right instruction is employed at this point. If we later encounter another piece of data with the desired characteristic, we again will add one to the counter. At the conclusion of the program, the counter will contain the numerical count of the number of pieces of data which contain the desired characteristic. It is notable that the Halt instruction itself has been used as the counter. This use of the Halt instruction is made possible for two reasons. Firstly, the Add One Right instruction does not affect the left half-word of the specified memory location. Therefore, the instruction operation code in L1 through L15 of the Halt instruction will not be changed by adding one right to the right half-word of the halt instruction. Secondly, an HLT instruction does not need its right half-word. When the computer detects that Left 1 through Left 15 contains zeros, it knows, without any reference to the right half-word, that it should halt. Therefore, the right half-word may be used, at the discretion of the programmer, for any purposes he might wish.

Another use of the Add One Right instruction, commonly found in operational programming, is instruction modification. Since such a usage will require the branch class of instructions, we will defer illustration of this aspect of the Add One Right instruction until a later chapter.

PRE-STORING OF CONSTANTS

Throughout your programming, it will be necessary to employ the use of constants. Such constants may be employed as RC Words, as shown in the preceding chapter, or they may be found directly in the main body of your program. We will here discuss three types of constants, and the ways they may be specified in the body of your program.

Octal Constants

Columns 24 through 29 inclusive and columns 36 through 41 inclusive are used to specify the octal contents of the left and right half-words, respectively, of the memory location within your program where the octal constant will fall. For example, the sign bit would appear in column 24, be it one or zero, and the remaining five octal digits would be specified in columns 25 through 29 inclusive. A similar use of columns 36 through 41 would be employed. Location Ø1A in Figure 4-1, provides us an example of such an octal constant in the left half-word. We see that an octal 2Ø was requested, and in Figure 4-2 we see that program Translator indeed inserted such an octal constant in the specified memory location.

Decimal Constants

Decimal constants may be inserted in your program in one of two forms, both integer and fractional.

Interger Decimal Constants. Columns 24 through 29 and 36 through 41 may again be employed for the storing of decimal constants. An example would be as follows: +2Ø48. The plus sign would be encoded in column 24 and the 2048 would be coded, right justified, in columns 25 through 29. The plus sign indicates to program Translator that this is indeed a decimal constant, rather than octal. A minus sign could be used to indicate the same. You must remember, therefore, that to indicate decimal constants a plus or minus sign is necessary. This will be true of decimal fractional constants as well. The plus or minus sign is program Translator's only way of knowing that this constant is not to be octal.

Decimal Fractional Constant. These constants are very similar in form to decimal integral constants. The only difference is that immediately following the plus or minus sign, a decimal point must be coded. This would be in columns 25 or 37. Program Translator will automatically convert the four digit decimal fractional constant which follows into its appropriate binary form.

The three types of constants described above may be intermixed as the programmer likes. The left half-word of a given memory location could contain an octal constant, while the right half contains a decimal fractional constant. It should further be noted that the same types of constants may be used to specify RC Words. Therefore, although all above references were to columns 24 through 29 and 36 through 41, the same type of coding could be used in the six columns from 3Ø through 35 inclusive. It is also worth noting that these constants must never be greater than octal 177777. Were you to request a constant in octal greater than this number, or were your decimal constant to convert to a number larger than that, it would not fit in the half-word provided. Such overflow would create an error condition within the assembly process.

Exchange Instruction (ECH) code .03500

The ECH instruction exchanges the contents of the specified memory location with the contents of the accumulators.

CHAPTER 6

BRANCH CLASS INSTRUCTIONS

INTRODUCTION

Thus far in our programming we have been dealing with programs that are known as "strictly straight-line". That is, all of our programs have executed instructions from consecutively higher numbered locations in memory. We have had no facility for skipping instructions or branching back to prior ones. The Branch Class of instructions will offer us this capability, and thereby greatly expand the class of programming problems with which we may deal.

BRANCH INSTRUCTIONS

In this chapter we will explore five of the simpler Branch Class instructions. In the most general sense, they are divided into two types: conditional and unconditional. An unconditional branch instruction will always alter the path of program flow. A conditional branch instruction, on the other hand, may or may not alter the path of program flow, depending upon conditions it finds within the computer. It may make its decision to branch or not to branch on the basis of the contents of certain index registers, or it may choose the condition of the data in the accumulators to be the factor which will determine whether or not it will branch.

Unconditional BPX Instruction

The Unconditional BPX instruction will always branch to the location specified in its 17-bit director, that is, LS and RS through R15. On your coding sheets, the 17-bit director may be specified in columns 36 through 41, in either a decimal or octal form, or as a symbolic location tag. If it is coded as a symbolic location tag, it must of course be coded in columns 36 through 4Ø, in the same relative position as it appears in the five columns 18 through 22.

BFZ Instruction

The BFZ (Branch Full Zero) instruction will branch to the location specified by its director if, and only if, both accumulators contain zero. This may be any form of zero, i.e., the left accumulator may have negative zero and the right accumulator positive zero, or both accumulators contain a negative zero, et cetera. As you can see, the BFZ instruction makes its decision as to whether or not to branch on the basis of the data which is currently in the accumulators. We may there alter the course of our program flow depending on the conditions we find in the data tables.

BFM Instruction

The BFM (Branch Full Minus) instruction will branch to the location specified by its director if, and only if, both accumulators contain a negative number.

BLM Instruction

The BLM (Branch Left Minus) instruction branches if the left accumulator contains a negative number.

BRM Instruction

The BRM (Branch Right Minus) instruction will branch if, and only if, the right accumulator contains a negative number. It is obviously similar in function and purpose to the Branch Left Minus instruction.

The above five Branch instructions offer us the facility to alter the course of program flow depending on data. In Figure 6-1, we see illustrated the principle of data dependency in our program. Program "LARGER" compares the algebraic quantities stored in the right half-words of location 22B and location 33B. The program is designed to find the larger number and store it in the right half-word in location 44B.

LOCATION						R	OP CODE			AUX.		RC WORD						ADDRESS						± INCR.			
18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39	40	41	42	43	44	45
							I	D	T									L	A	R	G	E	R				
							C	A	D									2	2	B							
							S	U	B									3	3	B							
							B	R	M									$						+			4
							C	A	D									2	2	B							
							R	S	T									4	4	B							
							H	L	T																		
							C	A	D									3	3	B							
							F	S	T									4	4	B							
							H	L	T																		
							E	N	D																		

Figure 6-1. Simple Branching Program

Let us analyze this program in some detail. The first instruction brings the contents of location 22B into the accumulators, having first cleared this register. Next, the contents of location 33B are subtracted from the previous contents of the accumulators. Thus, in the accumulators we have the difference of the contents of the two locations in question. If the content of 33B is the larger, we will branch to location SELF+4--that is, we will branch to the Clear and Add instruction four steps beyond. This would be called a "successful branch". If the content of 22B is the larger, the subtraction will leave a positive number in the accumulators. We will therefore not find a negative number in the right accumulator, and we will therefore not branch to the specified location. We will indeed "fall through" the branch instruction. When we fall through, the accumulators will be cleared, a fresh copy of the contents of 22B will be brought in, and this value will be stored into location 44B. The program will then halt. If we branch, the accumulators would be cleared, the contents of 33B would be brought in and stored in location 44B. Thus, it is obvious that we are not necessarily sure, when our program is operated, where we will halt. It may be at either one of the two halts that are located within the program. The data will determine the course of program flow.

CLOSED SUBROUTINES

If in a program there are several branches to location Ø1A, it is not necessarily certain when the program arrives at that location and how it got there. If the program must take alternative actions at a point after Ø1A, depending on the route the program took to get to Ø1A, it will be necessary at that location to determine from whence the program came. After each successful branch in the AN/FSQ-7, the address of the instruction immediately following the branch instruction will be left in the right A register. That is, had the branch not occurred, the program would have moved on in a straight line fashion to the next consecutively higher numbered address. It is this address which will be left in the right A register. More particularly, the program counter always contains the address of the instruction which immediately follows the one currently being executed. For example, if there is a Clear and Add instruction stored at location 3Ø1 currently being executed, the program counter will contain address 3Ø2 during the execution of that Clear and Add. Therefore, during the operation of a branch class instruction located at address 6Ø3, for example, the program counter contains octal 6Ø4. If the branch occurs to another location in core, that address, i.e., 6Ø4, will be put into the right A register. The new address to which we will branch will then be put into the program counter. If we "fall through" a branch class instruction, the right A register is not affected. We may therefore say that if indeed we do branch, the address of the instruction which would have been executed had we not branched, will be left in the right A register.

If in the preceding mentioned example, the case of a successful branch to location Ø1A, we need to determine when we arrive there, where we have come from, we can interrogate the right A register. Access to the right A register can be obtained by using the STA instruction.

STA Instruction

The STA (Store A) instruction stores the contents of the right A register into the right half-word of the specified memory location. The previous contents of the right half-word of the specified location are erased, and the left half-word remains unchanged. Therefore, at Ø1A we could have an STA instruction if we wished to determine where we came from. The address of the instruction which would have been executed had we not branched will have been left in the right A register. The STA instruction can store this address into memory for later reference. One cannot wait until a later point in the program to exercise the STA instruction, for almost all instructions in the Q-7 repertoire destroy the preceding contents of the right A register. It is therefore imperative to save the contents of this register as soon as information is placed there, i.e., immediately after a branch.

Figure 6-2 illustrates the typical usage of an STA instruction, which provides us with the means to perform closed subroutines outside the main body of our program. Four instructions are shown in the body of the main program - two branches and two instructions from the Add class. Also shown are the first and last instructions of a closed subroutine located underneath the main body of the program. Let us analyze the operation of the illustrated portions of this program in some detail.

The first instruction shown in the program is a BRM. If the BRM is successful, it will branch to location SBR, the first location in our closed subroutine. If the branch

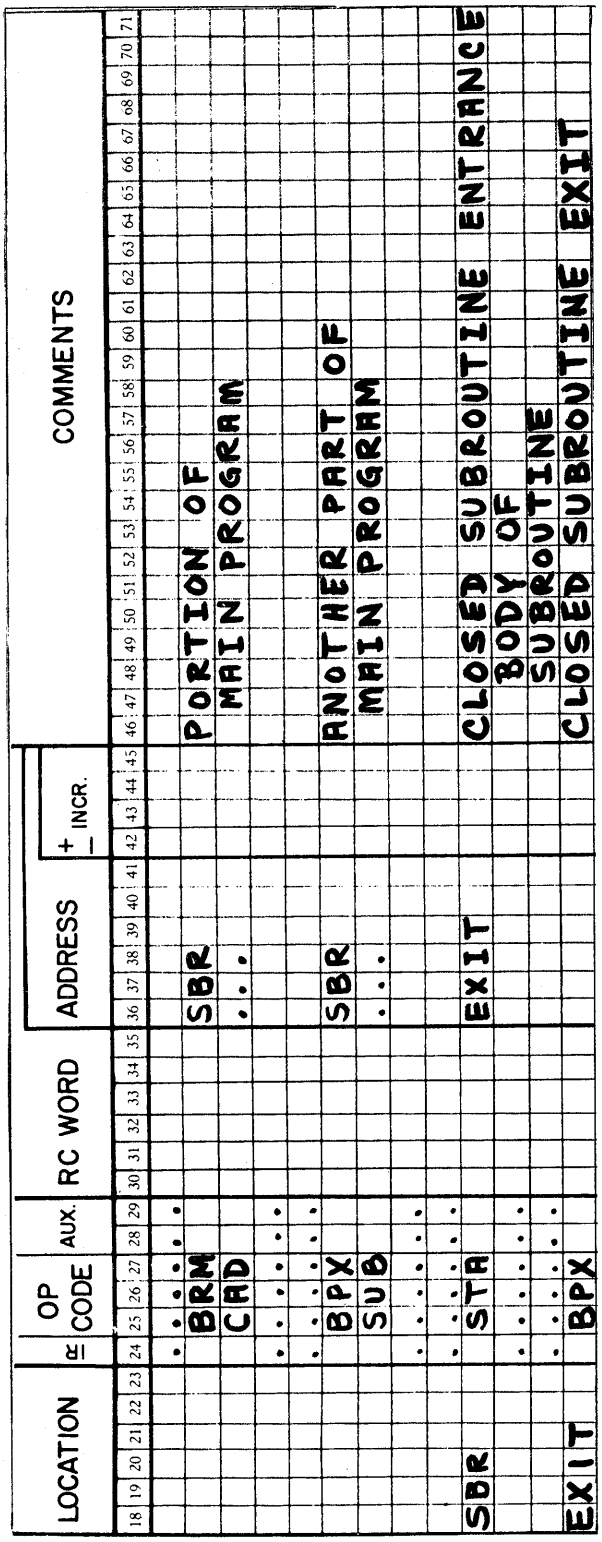


Figure 6-2. Example of Placement of a Closed Subroutine in a Program

is successful, the address of the CAD instruction will be left in the right A register. Immediately upon our arrival at SBR, that address is stored out of the right A register to the right half-word of location EXIT. Next follow several instructions which would constitute the body of the closed subroutine. At location EXIT, there is a BPX, apparently to nowhere. Recall, however, that an address has been put into the director portion of the BPX instruction. Specifically, the address of the CAD instruction was put into the right half-word of the BPX. Therefore, when we arrive at the branch, we will unconditionally go back to the CAD instruction. The program will then proceed down to the BPX to SBR. Again we go to the closed subroutine, this time unconditionally. The address of the SUB instruction will be left in the right A register. Immediately upon our arrival at location SBR, this address will be stored into the right half-word of location SBR, this address will be stored into the right half-word of location EXIT, erasing any previous addresses stored there. The subroutine will then be executed and we will finally arrive at location EXIT. At that point we will be BPX to the SUB instruction, for the BPX instruction now has the address of the SUB instruction as its director. Thus we can see that the closed subroutine linkages shown above offer a method for continual branches down to the subroutine. Every time we will be transferred back to the instruction immediately following the branch instruction which took us to that routine. It is this method of operation which classifies the routine shown as a "closed subroutine". Although the above example may at first appear to be circuitous, if you will pay careful attention to the path of program flow, the uses and methods employed in constructing a closed subroutine will become apparent.

From the above, we can deduce that all that is necessary to construct a closed subroutine is to write a short program which will accomplish the intended purpose, and then to surround that routine with an STA and a BPX instruction. The STA instruction will be the first instruction within the closed subroutine and its director will be the address of the BPX instruction. Closed subroutines offer us an example of the principle of "Instruction Modification". When the program is initially assembled, the BPX does not have a director. Inasmuch as it is customarily left blank on a coding sheet, program Translator will insert a director of Ø. It is obvious that we do not actually wish to branch to location zero when our program is operated. The program itself will insert a director into the right half-word of the BPX instruction when we finally utilize the closed subroutine. Indeed, the STA instruction will place this director into the right half-word of the BPX. It can be said that the STA instruction, therefore, performs "instruction modification" upon the BPX instruction.

Figure 6-3 illustrates the concept of a closed subroutine, further capabilities of program Translator, and the instruction modification capabilities of the Add One Right instruction. It is obvious, therefore, that it deserves your careful attention. The STA instruction at location CLEAR, and the BPX instruction at location EXIT, are the two instructions which make this program a closed subroutine. Between these instruction, is a list of instructions designed to accomplish the task of clearing Little Memory. Since this is a closed subroutine, an IDT and END card have not been shown, inasmuch as this routine would be appended to a larger program. Whenever the larger program wished to clear Little Memory, it could branch to location CLEAR. This subroutine would accomplish that job, and then return control to the location immediately following the branch instruction which took the main program down to location CLEAR.

LOCATION	IR	OP CODE	AUX.	RC WORD	ADDRESS	+/− INCR.	COMMENTS
18 19 20 21 22 23	24	25 26 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 59 60 61 62 63 64 65 66 67 68 69 70 71
CLEAR		STA			EXIT		STORE RETURN ADDRESS
		STZ			2ØØØØØ		CLEAR FIRST LOCATION
		AOR			$	− 1	MODIFY CLR FOR NEXT PASS
		SUB		Ø	7777		ARE WE FINISHED
		BRM			CLEAR	+ 1	NO
		CAD		STZ	2ØØØØØ		HOUSEKEEPING
		FST			CLEAR	+ 1	
EXIT		BPX					RETURN TO MAIN PROGRAM

Figure 6-3. Closed Subroutine to Clear Little Memory

Analyzing the closed subroutine itself, we see that the first instruction clears location 2ØØØØØ, the first location in Little Memory. The next instruction adds "1" to the right half-word of the Store Zeros Instruction itself. This means that the instruction at location CLEAR+1 is no longer an STZ 2ØØØØØ instruction. Rather, it is now an STZ 2ØØØØØ1 instruction. If the subroutine branches back to location CLEAR+1, the second location in Little Memory will now be cleared. It must also be remembered that the Add One Right instruction leaves the new right half-word of location CLEAR+1 in the accumulators. You will recall that the addition performed during the Add One Right instruction is performed in the accumulators. Therefore, the new address will be left in the right accumulators. More specifically, the right half-word of location CLEAR+1 will be left there. Now, location CLEAR+1 contained the address 2ØØØØØ initially. This would consist of a one in Left Sign and all zeros in the right half-word. The one in Left Sign will remain in memory undisturbed, and will not be brought into the accumulators. Therefore, at the conclusion of the AOR instruction, a one will be left in the right accumulator. It is this result left in the right accumulator, which the program will use to determine how many times it should loop, so as to consecutively clear increasingly higher numbered locations in Little Memory.

Immediately following the AOR instruction is a Subtract instruction. Specifically, it says that the program is to subtract a register containing a zero in the left half-word and octal 7777 in the right half-word. This is the highest numbered address in Little Memory, with the one in Left Sign deleted. It is this subtraction which will determine how many times the program is to loop. Since there was a "1" left in the right accumulator at the conclusion of the AOR instruction, when we subtract an octal 7777, we will obtain a negative result. The next instruction says that if the right accumulator is negative, as we just determined it will be, we are to branch to location CLEAR+1, the STZ instruction. However, the STZ instruction now contains a director of 2ØØØØ1. Thus the second location in Little Memory will now be cleared. Again, we Add One Right to location CLEAR+1 thereby leaving an instruction in that location of STZ 2ØØØØ2. When we again subtract octal 7777 from the 2 which will be left in the right accumulator, we will come up with a negative result. We will therefore again branch to location CLEAR+1, and now clear the third location in Little Memory.

It is apparent that we will continue to loop until we have cleared all of the locations in Little Memory. Each time we subtract an octal 7777 from the results of the latest AOR instruction, we will come up with a negative result, and we will again loop back to the newly incremented STZ instruction. When we have cleared the last location in Little Memory, we will have in location CLEAR+1 an STZ 2Ø7777 instruction. We will then add one right to this instruction, and leave an octal 1ØØØØ in the right accumulator. When we subtract octal 7777 from this number we will now come up with a positive result. We will fall through the Branch Right Minus instruction and thereby come out of our cycling loop. At this point, the program requests that the accumulators be cleared and a register which contains an instruction STZ 2ØØØØØ be brought into the accumulators. We note here a new capability of program Translator, specifically that RC Words may be of the form of the instructions. You will note that the STZ is coded in columns 31 through 33. Whenever an instruction is coded in columns 3Ø through 35, the operational code and all auxiliary information is coded within those six columns in the same relative positions as it would be coded in columns 24 through 29. We have now brought a fresh, clean copy of the STZ 2ØØØØØ instruction into the accumulators. This fresh copy is now

stored into location CLEAR+1. The Add One Right instruction has continuously altered the director of the STZ instruction. Therefore, we no longer have a fresh version of this instruction at location CLEAR+1. Should this subroutine be used again, it would result in immediate error because the instructions are not now as they appear on your coding sheet. The closed subroutine must modify itself so that the next time it is used it will again appear as it does now in Figure 6-3. This is the reason for the two instructions at locations EXIT-1 and -2, which are noted as "housekeeping" instructions in the comments field.

While it might be said that Figure 6-3 was used to illustrate too many things at once, a careful perusal of this program will be invaluable to you.

SUDOR AND WECO SUBROUTINE LIBRARIES

There are two tapes of subroutine libraries designed for use with the COSEAL program Translator. These tapes both contain an index followed by many closed subroutines. The subroutines are in unassembled, "prestored", format. They have been designed to do most of the "busywork" programming that is commonly encountered. There are subroutines to handle tapes (e.g., rewinding, setting prepared, positioning), save important arithmetic registers and restore same prior to the operation of other subroutines, etc. Many of the subroutines on the SUDOR tape are particularly designed to do much of the work needed in SAGE-oriented programming: determining distances from sector-center, calculating stereographic projections within the sector's geographic bounds, etc.

The WECO (Western Electric Company) Tape contains all of the routines on the SUDOR tape plus many others that accomplish general mathematical functions: table averaging and median computation, 3Ø-bit dual precision arithmetic, simulated floating-point arithmetic, pseudo-random number generation, etc. One can call these subroutines using calling sequences defined in the COSEAL manual for the SUDOR tape and the WECO library manual for their tape.

One codes for these subroutines by using the pseudo instruction SBR in columns 25 through 27 and the name of the subroutine left-justified in columns 36 and following. Upon encountering an SBR (subroutine) instruction, COSEAL will generate an unconditional BPX at that point in the program to the location of the subroutine. Translator will search the subroutine library tape, finding the subroutine requested and any other subsidiary routines needed by it, and place these subroutines at the end of your program just before the end card. It is this location which will be used as the director of the BPX instruction which is inserted in your program at the location of the SBR pseudo instruction. All of the subroutines on the library tape have been written as closed subroutines with a STAA and a BPX at the end, and the programmer may use them as such.

17-BIT OPTION

Suppose you as the programmer were writing a program to be located in Little Memory. If this program were to have subroutines attached, the addresses in the main program to which the subroutines should branch back would require 17 bits. All address of 2ØØØØØ or greater require that many bits. The right A register is actually

a 17-bit register. Therefore, the return address would be in the right A register in its entirety. However, an STA instruction would not save all 17 bits. Only the 16 bits of the right half-word would be stored into the specified memory location. It is obvious, that the 17th bit must be saved. The logical place for this would be Left Sign of the specified memory location. It is for this reason that "17-bit option" has been built into the Q-7. Several instructions offer this 17-bit option. One of these is the STA instruction. When you wish to use the 17-bit option of a particular instruction, you request this option by coding an "A" in column 28 beside the Op Code.

STAA Instruction

The STAA instruction performs essentially the same function as the STA instruction. It stores the 17 bits in the right A register into Left Sign and Right Sign through Right 15 of the specified memory location. In almost all programming applications where the STA instruction is used, the extra A is usually added in column 28, thus invoking 17-bit option. When program Translator encounteres a call for 17-bit option, it will adjust the octal code of the binary output program accordingly. This is accomplished by adding a one in Left 12 of the specified instruction. A one in Left 12 of an instruction converts to an octal ten when viewed in the entire left half-word. It is for this reason that many documents relating to the Q-7 often refer to 17-bit option as, for example, STA1Ø. In all symbolic coding using COSEAL's Translator, however, an A in column 28 will suffice. With the 17-bit option, we see that it is now possible to save all 17 bits of any memory address. It will be for the purposes of Address Modification, that 17-bit option is usually used. Of the instructions we have encountered so far, four others have a possibility for 17-bit option.

AORA Instruction

The Add One Right A instruction is somewhat different in its operation from the ordinary AOR. When executed, the following actions are taken:

1. Both accumulators are cleared.

2. The 17-bit director of the specified memory location is brought into the accumulators. That is, specifically, Left Sign and Right Sign through Right 15. Bits L1 through L15 are not brought into the accumulator.

3. "One" is added to the right accumulator. Should overflow in the right accumulator occur, an end-around carry will not be generated. Rather, the carry will go into Left Sign and end-around carry out of Left Sign will be carried through Right 15. We thus see that we have created a 17-bit adder in the accumulators, and the Q-7 is wired for this possibility.

4. The 17-bit director in the accumulators is stored into the specified memory location in Left Sign and Right Sign through Right 15. Left 1 through Left 15 of the specified memory location are undisturbed throughout the operation of the AORA instruction.

It is important to note that the Add One Right A instruction leaves both accumulators changed from their preceding contents. This is a fact often times easy to overlook in

actual programming. The AORA instruction should be used whenever the programmer is manipulating addresses which either are already in the Little Memory or Test Memory range, or through additive processes are likely to get into those areas.

RSTA Instruction

The RSTA instruction stores Left Sign and Right Sign through Right 15, from the accumulators, into the corresponding bits of the specified memory location. Bits L1 through L15 of the specified memory location remain unchanged, and the accumulators are undisturbed. The RSTA instruction enables us to store an address into the director portion of an instruction. The address may range from Ø to 377777.

ADDA Instruction

The ADDA instruction enables us to add two addresses together in the accumulators, notwithstanding that they may be 17-bit addresses. Bits Left Sign and Right Sign through Right 15 of the specified memory location are added to the corresponding bits in the accumulator with a 17-bit addition, and a 15-bit addition occurs between Left 1 and Left 15 of the specified memory location and the corresponding accumulator bits. The specified memory location is left unchanged, and the results of the addition are left in the accumulators. We thus have a 17-bit and a 15-bit adder, rather than the normal configuration of two 16-bit adders in the accumulators. Should a carry occur out of Right Sign it will be added into Left Sign. Should end-around carry occur out of Left Sign, it will be added into Right 15. End-carry out of L1 in the left addition will be lost. It should be apparent that the results of the addition in Left 1 through Left 15 are therefore invalid. This is no impediment, however, because we are usually interested in the two 17-bit addresses which are being added together, and not Bits Left 1 through Left 15.

SUBA Instruction

The SUBA instruction performs exactly the same job as the ADDA instruction, with the exception that a subtraction is performed rather than an addition. The contents of the memory location are subtracted from the contents of the accumulators, with a 17-bit and a 15-bit subtraction occurring.

As we have seen, the above five instructions offer us the capability for a full handling of 17-bit addresses within the Q-7. We will, in fact, discover other instructions which enable us to handle these 17-bit addresses. Though these instructions may at first appear unwieldy, a careful study of them on the part of the student programmer will show them to be essentially simple extensions of the basic instructions from which they have been constructed.

CHAPTER 7

INDEXING AND TABLE CONSTRUCTION

INDEX REGISTERS

The Q-7 Computer contains four physical Index Registers numbered 1, 2, 4, and 5. These are 17-bit flip-flop registers, consisting of one control bit and sixteen magnitude bits. The usual use of these index registers is in the modification of addressing with the indexable instructions, noted by an asterisk on the AN/FSQ-7 Programmer's Card. Figure 7-1 illustrates the use of Index Register 1 with the Clear and Add instruction, which is indexable. The instruction basically requests that the accumulators be cleared and the contents of location 3ØØ be brought in. However, should the current contents of Index Register 1 be an octal 1ØØ, the instruction shown will actually clear the accumulators and bring in the contents of octal address 4ØØ. Thus we see that the index register specified in column 24 modifies the address portion of the currently operating instruction. The instruction's director is added to the current contents of the index register, giving a new net effective address when the instruction is actually operated. Similarly, the next instruction, Subtract 6ØØ, requests that the contents of location 6ØØ be subtracted from the current contents of the accumulators. When the program is operated, however, it will not be the contents of location 6ØØ, but rather the contents of that location whose address is 6ØØ plus the current contents of Index Register 4, the specified index register. We will in time discover other uses for these index registers, but the above examples show the most typical functions.

LOCATION						IR	OP CODE			AUX.		RC WORD						ADDRESS						+ − INCR.			
18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39	40	41	42	43	44	45
						1	C	R	D												3	Ø	Ø				
						4	S	U	B												6	Ø	Ø				

Figure 7-1. Examples of Index Register Address Modification

There is also an Index Register 3 in the Q-7. Index Register 3 actually consists of the right accumulator in the arithmetic element. If we specify Index Register 3, the current contents of the right accumulator will be used as if they were in a physical index register. However, the programmer should be aware that if the instruction being indexed alters the contents of the accumulator, then Index Register 3 will no longer contain the same value. We will likewise see later and fuller explanations of the usage and operation of Index Register 3.

Index Registers in the Q-7 have an unusual cleared state. When the index register is cleared, the control bit contains a 1, and all 16 magnitude bits must necessarily be zeros. There is no facility in the Q-7 for having a negative value in an index register.

It is electrically impossible for any of the magnitude bits to be other than zero and yet the control bit be a one. If there be any magnitude bits set to a one, the control bit will always be a zero. It is possible to have all 16 magnitude bits of the index register be zero and the control bit be zero. This is equivalent to having the index register set to zero, but not in the "cleared" condition. The "cleared" condition only occurs when the control bit is negative, and the magnitude bits necessarily zero.

RESET INSTRUCTIONS

The Reset Class of instructions offers facilities for loading index registers and accessing their contents. Although these instructions are listed on your Programmer's Card as "Non-Indexable," nonetheless you must specify an index register in column 24. Since these instructions manipulate index registers, they must be told which index register they are to alter or access.

XIN Instruction

The XIN instruction offers the facility to the programmer of loading his index registers. Specifically, the instruction loads the index register specified in column 24 with the value in the decimal director in columns 36 through 41. It is not necessary to have a plus sign in column 36, although the director is necessarily decimal. Since the Index Register will have 16 magnitude bits, you cannot load it with an octal number greater than 177777. This translates to a decimal value of 65535. Should the XIN instruction have a 1 in Left Sign, it will always clear the specified index register, no matter what the value in its right half-word. In Q-7 programs, therefore, you will often see the XIN instruction used with a T in column 41, specifying a Test Memory Address. The programmer is not attempting to load the specified index register with a Test Memory address. Rather he is using the T in column 41 in order to place a one into Left Sign of the XIN instruction. He would therefore know that this XIN instruction would clear the specified index register, and a T in column 41 is the most expeditious way of coding this desired effect. The XIN instruction can only be used with Index Registers 1, 2, 4, and 5.

XAC Instruction

The XAC instruction takes the contents of the right accumulator and puts it into the index register specified in column 24. This instruction has a 17-bit option. If the programmer specifies an XACA instruction, the index register will be loaded if, and only if, the Left Sign in the accumulators is a zero. Should LS be a one, the index register will be cleared. Note that this is the above mentioned "cleared" state, that is, a one in the control bit and all magnitude bits set to zero. The address portion of the XAC instruction, that is Left Sign and the entire right half-word, is not examined by the computer during the operation of the instruction. Since this is the case, the XAC Director could be used for temporary storage by the programmer. The XAC instruction may only be used with Index Registers 1, 2, 4, and 5. If the index register specified is Ø, 3, 6, or 7, the instruction will perform no operation.

ADX Instruction

The ADX instruction adds the 17 bits of its own director to the 16 magnitude bits of the specified index register and stores the 17-bit result into the Right A Register. This instruction offers our only means of accessing the contents of an index register, when those contents are unknown at that point in the program. Note, however, that only the magnitude bits of the index register are brought out by the ADX instruction. Should those magnitude bits have a value of anything other than a zero, we know that the control bit itself must be a zero. There is therefore no ambiguity. However, should the magnitude brought out of the index register be a zero, it is not necessarily sure whether or not the index control bit contains a zero or a one. One might have a "cleared" index register, in which case the control bit is a one and all the magnitude bits are zero; or one might have an index register set to zero, that is, all magnitude bits and the control bit set to zero. We will learn later a way to "get around" this possible source of difficulty. Once the ADX instruction has placed the contents of the index register plus its own director into the Right A Register, the STA instruction is usually used to put the contents of the Right A Register into a known location in memory. Almost invariably you will see this dual combination, ADX and STA, because the Right A Registers will soon be destroyed by the operation of any other instruction. It is therefore imperative to save the contents of the Right A Registers as soon as the ADX instruction puts something meaningful in them. Director of ADX is either unsigned decimal or symbolic tag.

Figure 7-2 offers several examples of the usage of the Reset Class of Instructions. The first instruction loads Index Register 1 with the value of $9_{(1\emptyset)}$. The second instruction loads Index Register 4 with a decimal 4ØØ. The next instruction brings a value of 5 and 27, octal, into the left and right accumulators. We see then a 2XAC instruction. This will load Index Register 2 with octal 27. That is, the right accumulator, which contains an octal value of 27, will be loaded into Index Register 2. At this point, we do not know the contents of Index Register 5. A 5ADX instruction is used to access this Index Register. Specifically, the ADX instruction will take the 16 magnitude bits of Index Register 5, add its own director (+Ø) to it, and leave the total in the Right A Register. Since nothing was added to the contents of the Index Register, the contents of Index Register 5 are therefore in the Right A Register. The contents of this register are then stored into location Ø1A by the action of the STA instruction.

NOP Instruction

The NOP, "No Operation", instruction performs exactly what it says, no operation. For six microseconds, the computer does nothing. This at first might appear to be a useless instruction, but we will see later usage of this facility. The instruction is mentioned at this point inasmuch as it is a part of the Reset Class of Instructions.

The Reset Class of Instructions cannot be used with index Register 3, the right accumulator. Reasons for this will be more apparent if you think to yourself what effects such instructions would possibly have. The XIN instruction is used to load index registers, but do we not have a more simple method of loading index register 3? Would it not be simpler to use a Clear and Add instruction to load Index Register 3? The XAC instruction loads the specified index register with the contents of the right accumulator.

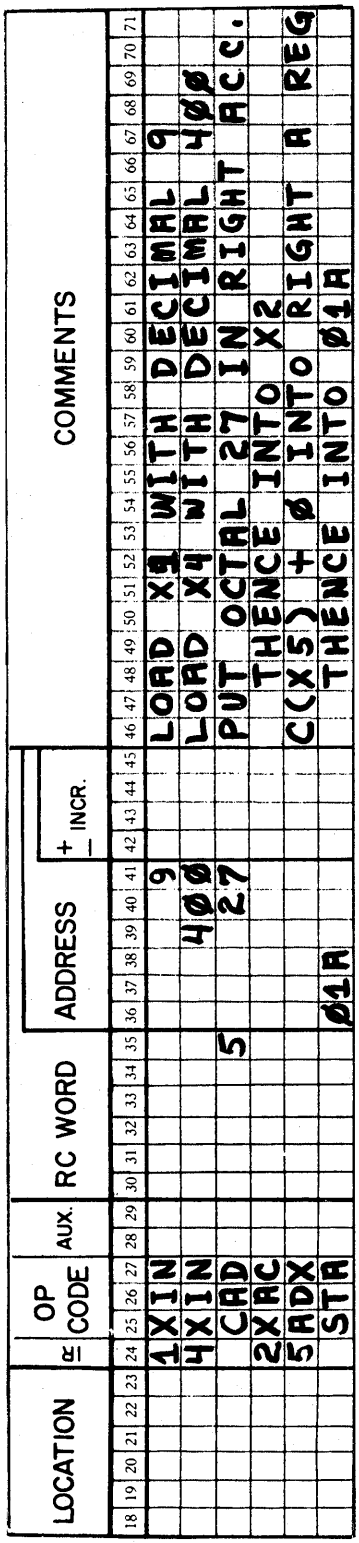


Figure 7-2. Examples of the Reset Class Instructions

It is therefore facetious to talk about "loading Index Register 3 with the contents of the right accumulator". The ADX instruction offers us the facility of accessing the specified index register. It is obvious that we have many methods of accessing the contents of the right accumulator. We see, therefore, that the Reset Class of instructions not only cannot be used with the right accumulator, Index Register 3, but, moreover, there is no real point in using these instructions with that index register.

CONDITIONAL BRANCHING

Instructions which enable the programmer to load or access the contents of index registers are almost worthless unless they are used in conjunction with instructions which will reduce the registers' contents, so that different operands may be obtained by using the same index registers. The instruction which will provide this facility is a second form of the BPX instruction of the Branch Class. Therefore, the last step in a given iterative loop will be a conditional type of BPX instruction which will sense the contents of the index register and possibly decrement same.

Conditional BPX Instruction

The Conditional BPX (Branch on Positive Index) instructions checks the control bit of the index register specified in column 24 (bits L1 through L3 of the instruction). If the control bit is a 1, that is, the index register is cleared, the instruction performs no further operation and we "fall through" the branch. If the control bit is not set, the index register is not cleared, that is, the index register specified will be decremented by the quantity specified in columns 28 and 29, and a branch will occur to a location specified in columns 36 through 41. Columns 28 and 29 are coded in decimal and may range in value from Ø to 63 inclusive. This corresponds to an octal coding, in bits L1Ø through L15, of 77. There are several important things to note concerning the BPX instruction. First, index registers in the Q-7 can only be decremented, and then only by a value from Ø to 63 decimally, inclusive. Secondly, when one is decrementing by "1", and a value of 1 resides in the index register, the index register will be decremented not to positive zero, but rather to the special "cleared" state. Thirdly, when a cleared index register is used for address modifications, it adds zero, not octal 2ØØØØØ. Fourthly, note especially that the decrementing is done after the index register is checked, not before. You will note from your previous experience in flow charting that this is crucial. This will imply that, in general, the Index Register is set to one less than the number of passes to be made through the program.

As an example, suppose we are given Table BOBØ, with 1ØØ decimal entries numbered Ø to 99, and we are to clear this table. That is, every register in this table is to be set to positive zero.

Figure 7-3 illustrates a flow chart of the proposed solution to this problem. Note that the flow chart has been drawn in such a way as to be easily transformed into a COSEAL problem. That is, the index register used has been decremented rather than incremented, and the testing was done prior to any decrementing. It is something of an art to flow chart your problems such that the solution will be easily coded in COSEAL. There are no strict rules for these procedures and only time and practice will show you this technique.

Figure 7-4 shows a typical solution to the problem given above. Program ERASE first loads Index Register 1 with a value of decimal 99. A register in BOBØ is then cleared, specifically, Register BOBØ +99. The conditional BPX then checks to see whether or not Index Register 1 is cleared. Determining that it is not, (it contains a 99), the BPX then determines that it will branch. Before branching, it decrements the specified index register by one (note cols. 28 and 29). A branch then occurs back to location AA, where BOBØ +98 is cleared.

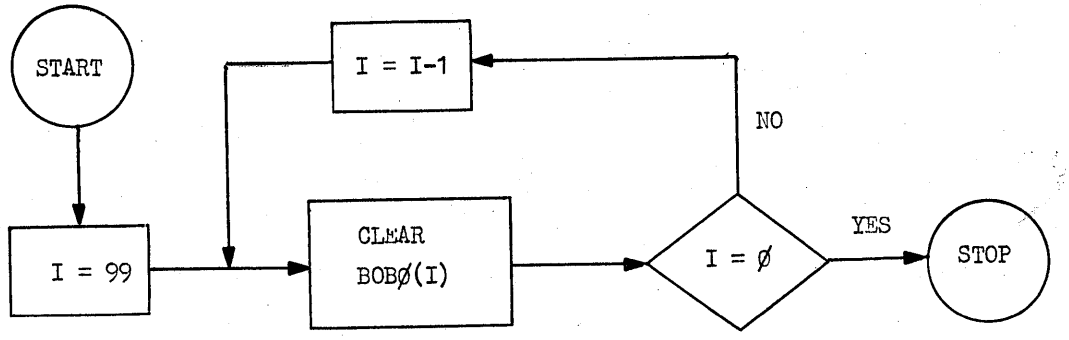


Figure 7-3. Program to Clear Table BOBØ

LOCATION	α	OP CODE	AUX.	RC WORD	ADDRESS	+/− INCR.
18 19 20 21 22 23	24	25 26 27	28 29	30 31 32 33 34 35	36 37 38 39 40 41	42 43 44 45
		IDT			ERASE	
		1XIN			99	
AA		1STZ			BOBØ	
		1BPX	Ø1		AA	
		HLT				
		END				

Figure 7-4. Flow Chart in Figure 7-3 Translated to COSEAL Coding

This will continue until Index Register 1 contains a value of 1. At this point, at location AA,BOBØ +1 is cleared. The conditional BPX again examines Index Register 1. It notes that it is not cleared and a decision is made to branch. Just before branching, Index Register 1 will be decremented by a value of 1. However, when decrementing, it will actually clear that index register. We then branch back to location AA where register BOBØ is cleared. The conditional BPX, on this pass, now determines that Index Register 1 is cleared. We therefore fall through the BPX instruction to the HLT instruction. This program illustrates a typical iterative loop, and a detailed analysis of this program should be made by the student.

TABLE AND ITEM DECLARATIONS IN COSEAL

In operational SAGE programming, tables and items used are listed in what is known as the Compool. This is nothing more than a reference guide listing the tables, items and programs in the SAGE System. It gives such information as scaling, length of tables, bit positions of items, and core locations. During assembly, program Translator has access to this information. Therefore, one may speak symbolically of one particular table, and program Translator will automatically search the Compool and insert core locations, etc. In your student programming, you will usually have little contact with a real Compool; rather, you will override the Compool with TCPO, ICPO, and TSKP Cards; also such facilities as the DIT card will be discussed. All of the above, and others which we will discover, are pseudo instructions.

You will recall that in SAGE programming, all tables, items, and programs have names of a specific format. Tables are given names with three letters and a digit, items are given names of four letters, and programs are three letters. You should further note that in SAGE, all tables are of parallel structure consisting of one or more blocks. All blocks would be of equal length. Since student programming will not invoke the Compool, we shall use CPO (Compool Override) cards. Several pseudo instructions will be used.

Compool Override Cards

There are two varieties of Compool Override Cards: one for defining tables and the other for items. We shall see illustrations and definitions for both types of cards.

TCPO Cards. By coding a TCPO in columns 24 through 27, one notifies program Translator that one is defining a table. Should the name happen to agree with the name of a table currently in the Compool, the definition in the Compool will be overridden. The name of the table should be coded in columns 18 through 21. Columns 28-29 equal the number of blocks in decimal (maximum 10). The starting address of the table would go in columns 36 through 41 in octal. Columns 32 through 35 will contain the decimal length of the block. If one is defining a table not to be used in a DCA program, any table name may be used in columns 18 through 22, left justified.

ICPO Cards. ICPO Cards are used to define items. In a fashion similar to the TCPO Cards, they override any definition in the Compool with an identical item name. The item name is coded in columns 18 through 21. This would be the case where one is using a 4-letter item name, as is customary in SAGE programming. One may, however, define item names of up to 5 letters, left justified, in columns 18 through 22. An "ICPO" is coded in columns 24 through 27. In columns 36 through 40, the name of the table which will contain the item is coded, left justified. One inserts the most significant bit

position in decimal in columns 34 and 35, and in columns 28 and 29 would go the decimal number of bits. The decimal bit locations correspond to the 32 bits of the Q-7 Computer Word, numbered from Ø to 31.

TCPO and ICPO Cards do not actually reserve that location in memory for the data. Merely defining a table does not clear out the area and prepare it for data, nor does it perform any data insertion action. These cards merely note to the program Translator where the tables and items will be located.

As was noted, Translator will now have the information available to it concerning table and item names, core locations, bit positions, et cetera. The uses to which program Translator will put this information are many and varied. One may say, for instance, CAD TABL. If TABL is a TCPO-defined table, COSEAL will insert the octal address of TABL. Item names may be used in a similar fashion, e.g., SUB ITEM. We will see other uses which program Translator can make of the information contained in TCPO Cards in a later chapter.

SKP Instruction

The SKP (Skip) instruction instructs program Translator to skip the decimal number of registers in the director field, columns 36 through 41. As with CPO cards, an SKP Card is a non-generative pseudo instruction. That is, when the program is being assembled, program Translator will leave an empty space between the two instructions surrounding the Skip instruction. However, nothing will be put into these locations. A location tag may be coded in columns 18 through 21, so that one may make symbolic reference to the first location which will be skipped. Should ten registers be skipped, one could therefore refer to Ø1A + 6. This would be the seventh register which was skipped, assuming that the Skip Card itself was given the location of Ø1A. The SKP instruction offers us a useful capability to reserve areas within our program for later data manipulation.

If, in the SKP pseudo instruction, one codes a T in column 24, the name given to the skipped register in columns 18 through 22 will become a table name. Items may then be defined within this table and it will acquire to itself all the normal characteristics of a table.

DIT Instruction

The DIT (Ditto) Instruction is a generative pseudo instruction which will reproduce the binary output of the last generated binary register the number of times specified in the decimal Director in columns 36 through 41. A location tag may be assigned to the first reproduced location by coding a name in columns 18 through 22. There is no "TDIT" instruction, and the programmer should not attempt to use name. One need not concern oneself with the possibility of endless registers appearing on the listing. Referring to the sample programs listed in the Appendix, you will see that when a large number of registers is either skipped or dittoed, program Translator shows only one of the skipped or reproduced registers.

TSYN Instruction

The TSYN (Synonymous) Instruction causes the symbolic table tag in the location field, column 18 through 21, to have the same octal address as the symbolic table tag

in the director field, columns 36 through 41, plus or minus the decimal value in the increment field.

All of the above instructions are designed to create data areas for later insertion of data which might be read into the computer or created by the program itself. These facilities enable us to create tables and items within them, and to make symbolic reference to such tables and items at a later point in our program. All reference to tables and items so defined must come after the definition of the particular table or item in question. Many of the advantages which will be offered by program Translator when we have so defined our tables and items will become apparent in later chapters.

Figure 7-5 gives an example of the instructions which we have just outlined. It shows their usage to reserve areas of core storage for various data. The program is not supposed to make any sense, so don't look for logic. It is merely shown as an example of the way in which the program Translator will assemble the references shown. It might also be noted that one reason this program will not work is that the programmer has not taken care to branch around his data areas. The programmer must be careful that his program does not march right into the data. Remember, the computer cannot distinguish between a data word and an instruction word. We note that after the first two operative instructions, the CAD and the SUB, the next ten locations have been reserved for data. However, this program will march right into that data area, and it will begin to use the data therein as instructions. Obviously, this will lead to program error.

```
                                PROGRAM TABL MOD ES          COMPOOL  04D 00   PAGE      1

          IDT                   TABLES
INFO      TCPO                   50000
NEWS      ICP005          0    INFO
          LOC                     1000
          CAD                  INFO          001000   001000 050000
          SUB                  NEWS   -  4   001001   001343 047774
                                   10        001002
01A       SKP                                001003
          CAD                  01A    +  1   001014   001000 001003
                                    9        001015
BOBO      TSKP
BUMS      ICP007         13    BOBO
          FST                  BOBO   +  2   001026   003240 001017
          ADD                  BUMS          001027   001043 001015
                                    3        001032   001043 001015
01B       DIT                                001033
          RST                  01B    +  1   001034   003340 001031
WASH      TSYN                 INFO   + 20   050024
SOAP      ICP021          3    WASH
          BFZ                  WASH   -  1   001035   005400 050023
          HLT                  SOAP          000000   050024
          END

THIS PROGRAM ILLUSTRATES                                0000.01*T00
 TABLE AND ITEM DEFINITION                              0000.02*T00
 AND A FEW OF THE USES                                  0000.03*T00
 WHICH TRANSLATOR CAN MAKE                              0000.04*T01
 OF SUCH DEFINITIONS.                                   0001.00*+01
                                                        0002.00*+02
                                                        0012.00*T00
                                                        0013.00*+01
                                                        0013.01*T00
                                                        0013.02*T00
                                                        0014.00*+01
                                                        0015.00*+02
                                                        0018.00*+00
                                                        0019.00*+01
                                                        0019.01*T00
                                                        0019.02*T00
                                                        0020.00*+01
                                                        0021.00*+02
                                                        0021.01*T03
```

Figure 7-5. Examples of Data Area Definitions

CHAPTER 8

TABLE AND ITEM MANIPULATIONS

In the preceding chapter we discussed methods of defining and reserving data areas. No discussion was made of the methods of putting data into these areas, nor was any mention made of the methods of accessing or manipulating such data. In this chapter we shall discuss some Q-7 instructions, some COSEAL pseudo instructions, and some capabilities of program Translator, all designed for item manipulation.

ITEM-ORIENTED PSEUDO INSTRUCTIONS

Two generative pseudo instructions have been incorporated into the vocabulary of program Translator specifically for the purposes of item manipulation. Although they are rarely used, they will make easier the understanding of several capabilities of program Translator to be explained at a later point in this chapter.

MSK Pseudo Instruction

The MSK (Mask) pseudo instruction generates a mask at the location where the MSK instruction appears in the program. The mask will contain 1 bits in the relative bit positions of the item specified in the director field, columns 36 through 4Ø. The item should be left justified in this field. Figure 8-1, which will be referred to extensively in the early portion of this chapter, illustrates two MSK instructions, and the generated binary output.

CON Instruction

The CON (Constant) pseudo instruction creates a register, at the point in the program where the CON instruction appears, with the value specified in the director portion of the instruction (columns 36-41), positioned to the bit positions of the item specified in the RC Word fields (columns 3Ø-34). The value specified in the address field may be a decimal or octal constant, and if a value greater than the item can hold is specified, garbage will be produced. Dual items, i.e., items of 32-bit length, can be loaded with this instruction. Once again, you should refer to Figure 8-1 where examples of the CON pseudo instruction are shown. In the middle of program ITEMS, shown in Figure 8-1, is a short four-step program for adding 2 to the current value in item TEST, and leaving the result in the accumulator. This program extends from location 1ØA to 1ØB, inclusive. The program initially clears the accumulators and brings in the register which contains, among other things, item TEST. The program then adds into the accumulators a register which contains 2 in the bit positions of item TEST. Having accomplished the desired end, the program then halts.

As was noted above, the CON pseudo instruction is rarely used. However, it illustrates the principle which will be used in the following area. Program Translator has the ability to accept RC Words of a form similar to those shown with the CON pseudo instruction. Note, for instance, the two instructions in Figure 8-1 at location 2ØB. The first instruction can be interpreted as follows: Clear the accumulators and add in a register which contains a decimal 3Ø in the bit positions of item WORK. The

```
        IDT                 ITEMS
TBL0    TCP0          100    25000
TEST    ICP007          0    TBL0
WORK    ICP016         16    TBL0

        MSK                  TEST          000300  177000 000000
        MSK                  WORK          000301  000000 177777

        CON     TEST     +    10           000302  012000 000000
        CON     TEST          10           000303  010000 000000
        CON     WORK     + 1000            000304  000000 001750
        CON     WORK       1000            000305  000000 001000

10A     CAD                  TEST          000306  001000 025000
        ADD                  10B           000307  001043 000311
        HLT                                000310  000000 000000
10B     CON     TEST              2        000311  002000 000000

20B     CAD     WORK     +    30           000312  001000 000326
        SUB     TEST          77           000313  001343 000327

30A     LDB                  10A           000314  000300 000306
        2LDB                    400        000315  020300 000400
        LDB           77        77         000316  000300 000330
        LDB     MSK          TEST          000317  000300 000331
        LDB                  TEST          000320  000300 000331

        ADB                                000321  001143 000000

40A     CAD     54626        176301        000322  001000 000332
        ETR     57003             0        000323  000040 000333
40B     ETR     MSK          WORK          000324  000040 000334
        ETR                  WORK          000325  000040 000334
        END

                                           000326  000000 000036
                                           000327  077000 000000
                                           000330  000077 000077
                                           000331  177000 000000
                                           000332  054626 176301
                                           000333  057003 000000
                                           000334  000000 177777
```

Figure 8-1. Examples of Item-Oriented Instructions

next instruction could be read: Subtract from the current contents of the accumulators a register which contains an octal 77 in the bit positions of TEST. This usage, similar to a CON instruction, is used widely in COSEAL programming. The student should become very familiar with the wording by which one can interpret such instructions, and you should note as well the RC Words which were generated.

B REGISTER INSTRUCTIONS

Two instructions deal exclusively with the B Registers, and we will find them of considerable use in item manipulation.

LDB Instruction

The LDB (Load B) instruction clears the B Registers and then places the contents of the specified memory location into the B Registers. The LDB instruction operates in a manner similar to that of the CAD instruction. Several examples of the LDB instruction and coding for same are shown in Figure 8-1. At location 3ØA, we see an instruction which tells the computer to clear the B Registers and then place therein the contents of location 1ØA. Next is an instruction which says load the B Registers with the contents of location 4ØØ as indexed by Index Register 2. The next instruction requests that the B Registers be loaded with the contents of a register which contains an octal 77 in both half-words. Next, we see an instruction which requests that the B Registers be loaded with a Mask for item TEST. That is, a data word will be loaded into the B Registers which contains ones in the bit positions of item TEST, and zeros elsewhere. The next line of coding illustrates a coding convention peculiar to only two instructions in the entire repertoire available to the Q-7 programmer. When one codes an item name in columns 36 through 4Ø with the LDB instruction, there is an implied MSK in columns 31 through 33. Thus, we see that the RC Word used as the director of both LDB instructions is the same. One may code the MSK in columns 31 through 33, although this is not necessary. The reasons for this coding convention will become apparent at a later point in this chapter.

ADB Instruction

The ADB (Add B) instruction adds the contents of the B Registers to the contents of the accumulators. The results are left in the accumulators, and the B registers are unchanged. This instruction has a 17-bit option, rarely used. The right half-word of the instruction is not examined by the computer during the operation of the ADB instruction. Therefore, the right half-word may be used by the programmer for temporary storage.

LOGICAL INSTRUCTIONS

Several instructions in the Q-7 repertoire may be classified as strictly logical operations. We will concern ourselves with two of these instructions at this point.

ETR Instruction

The ETR (Extract) instruction performs a logical multiplication between the contents of the specified memory location and the accumulator; results in the accumulator, memory unchanged. Another way of saying this is: where the specified memory location has a zero, clear that accumulator bit. Where the specified memory location has a one, leave that accumulator bit unchanged. As an example of the instruction ETR, note the instructions at location 4ØA in the program in Figure 8-1. The actions of this program are shown as follows:

C (accumulator)	ø.1ø1 1øø 11ø ø1ø 11ø	1.111 11ø ø11 øøø øø1
C (ø.øø2øø)	ø.111 111 øøø øøø ø11	ø.øøø øøø øøø øøø øøø
Final Result	ø.1ø1 1øø øøø øøø ø1ø	ø.øøø øøø øøø øøø øøø

We see that after the action of the listed instructions, the accumulators would contain 54øø2,øøøøø octal. If one considers the contents of memory during the operation of an ETR instruction as a Mask, it can be shown that the Mask operates as a "garbage clearer"; that is, in the bit positions of the Mask, the accumulators will be unchanged, elsewhere, the accumulators will be cleared. At location 4øB, we see the use of an ETR instruction with a Mask for item WORK. When the program was operated, the bit positions in the accumulator corresponding to the relative bit positions of item WORK would remain unchanged; all other bit positions in the accumulator would be cleared. Were one to execute the instruction CAD WORK, the entire register which contained item WORK would be brought into the accumulators. If there are other items in this register, these would likewise appear in the accumulators. If we wish to operate upon item WORK, the other items in that register can be considered garbage, at that point in the program. We therefore extract the accumulators with a mask for item WORK. This will clear the accumulators in all of those bit positions which are not a part of item WORK. Note carefully that the actual memory register which contains item WORK has been unchanged, and likewise the mask with which we extract will also be unchanged. Only the accumulators are altered during the operation of an ETR instruction.

One should also note that the ETR instruction has the same coding peculiarity as does the LDB instruction; that is, if an item is specified left justified in columns 36 through 4ø, there will be an implied MSK in columns 31 through 33. Although the use of this peculiarity with the LDB instruction is not at this point apparent, it should be evident that, for the extraction of items in the accumulator, this becomes a very useful facility with the ETR instruction. If we first CAD an item, and then we ETR the same item, we will be left with the value of that item, as it appears in memory, less any other items which might also be in that register. We may say, therefore, that we have brought the item into the accumulators and "gotten rid of the garbage".

At this point in a program, we could now do manipulation upon the value which is contained in the item being manipulated. When we have arrived at a result, we would usually wish to place this result back into that item. If however, we used an ordinary Store Class instruction, the entire contents of that memory register (or, at least, one of the half-words thereof) would be altered. This would most likely destroy the value currently within other items in that register. We need an instruction that puts the contents of the accumulators into only specified bit positions of a given memory register. The DEP instruction offers this facility.

DEP Instruction

The DEP (Deposit) instruction performs a selected load into Memory. Specifically, where the B registers contain a 1, it puts the corresponding accumulator bits into the specified memory location. Where the B registers contain a zero, memory is unchanged. The accumulators are then changed so that they appear exactly like the specified memory location. Throughout, the B registers are unchanged.

Another way to say all this is, the accumulators are put into memory in the masked bit positions, elsewhere memory is brought into the accumulators.

As noted above, the primary use of the DEP instruction is to provide for storage of less than one half-word in any bit position or combination of positions without destroying the remaining bits. This capability is especially useful when it is desired to update items within a word when only part of the word requires changing, and all the other bit positions are still valuable. As a simplified example of this use, let us consider the following situation: Assume that location 1ØØ contains 1.36625 in the LHW, and 1.76543 in the RHW. Bits L4 through L9 of this word are set aside for track identity. However, Bits L4 through L9 of location 2ØØ contain, among other items, a more recent identification. The problem is to update the item at 1ØØ without altering the remainder of the word, because the other bits indicate such things as track velocity, track number, etc. The program to update the information at location 1ØØ is shown in Figure 8-2.

The CAD instruction will bring the new identity into the accumulators:

C(acc) = 1.ØØ1 1Ø1 Ø1Ø ØØØ 11Ø 1.1ØØ Ø1Ø ØØØ ØØ1 1Ø1

The LDB instruction will place into the B registers a mask for saving bits L4 through L9:

C(B reg) - Ø.ØØØ 111 111 ØØØ ØØØ Ø.ØØØ ØØØ ØØØ ØØØ ØØØ

The DEP instruction will cause the masked bits to be deposited in the core memory:

Old C(Ø.ØØ1ØØ) = 1.Ø11 11Ø 11Ø Ø1Ø 1Ø1 1.111 11Ø 1Ø1 1ØØ Ø11

New C(0.00100) = 1.Ø11 1Ø1 01Ø 01Ø 1Ø1 1.111 11Ø 1Ø1 1ØØ Ø11

Notice that by varying the masks, the LDB and DEP instructions can be used to change any combination of stored bits without destroying the remaining bits.

LOCATION	R	OP CODE	AUX.	RC WORD	ADDRESS	+ − INCR.
18 19 20 21 22 23	24	25 26 27	28 29	30 31 32 33 34 35	36 37 38 39 40 41	42 43 44 45
		IDT			CHANGE	
TRKØ		TCPO		5Ø	1ØØ	
TIDY		ICPOØ6		4	TRKØ	
		CAD			2ØØ	
		LDB			TIDY	
		DEP			1ØØ	
		HLT				
		LOC			1ØØ	
		136625			176543	
		LOC			2ØØ	
		115206			142015	
		END				

Figure 8-2. Program To Update Track Identity

SHIFT INSTRUCTIONS

There are within the Q-7 repertoire, eight Shift Class instructions. Each of the instructions involves shifting the contents of one or both accumulators and sometimes also the B registers. In the binary instruction word, the number of shifts would be indicated to the computer in bits R1Ø through R15. This means that shifts are limited to an octal 77. On a coding sheet, the number of shifts is always indicated as a decimal director in columns 4Ø and 41. This decimal number should always range from Ø to 63 inclusive. Broadly speaking, there are two types of Shift instructions: Cycles and Shifts. The distinction is that in a Cycle, no bits are gained or lost. In a Shift, certain bits will be lost in the affected registers, and duplicated sign bits will be filled in to take up their place. Figure 8-3 gives a diagrammatic representation of the action of each Shift Class instruction. The name of the instruction and its octal code appears in the middle of each diagram. The mnemonics for each instruction stand for the following names:

Dual Shift Left	Left Shift Right
Dual Shift Right	Right Shift Right
Accumulators Shift Left	Dual Cycle Left
Accumulators Shift Right	Full Cycle Left

Following the maxim that a picture tells a thousand words, we will attempt to explain how to read the diagram rather than explain in words the action of each instruction. Consider first, the two cycle instructions depicted at the bottom of the illustration. As noted above, in a cycle instruction, no bits are gained or lost. Therefore what comes out of a register will go in the end of another register. The bits are cycled the number of times specified on the coding sheet in the decimal director. The Full Cycle Left instruction involves the accumulators only, and the Dual Cycle Left instruction involves both the accumulators and B registers.

Consider next the action of the ASR instruction further up the illustration. Considering just the right accumulator, you will note that the number of bits specified in the decimal director will be lost out of the least significant bit position of the accumulator. This will create vacancies in the most significant bit positions. In every diagram where there is an S, the S stands for "sign bit". In all shift instructions, as in opposition to cycle instructions, these sign bits are never moved, merely duplicated. It is these duplicated sign bits which fill up the vacated bit positions, as bits are lost out of the other end of the register. We will see in the next chapter the justification for duplicated sign bits. Suffice it to say at this point that they fill a definite programming need.

Shifting Pseudo Instructions

Program Translator has the capability of recognizing two pseudo instructions which will generate cycles in the binary program. These two instructions are POS and RES, standing for Position and Restore, respectively. Each of these pseudo instructions has two configurations: with one and with two item tags.

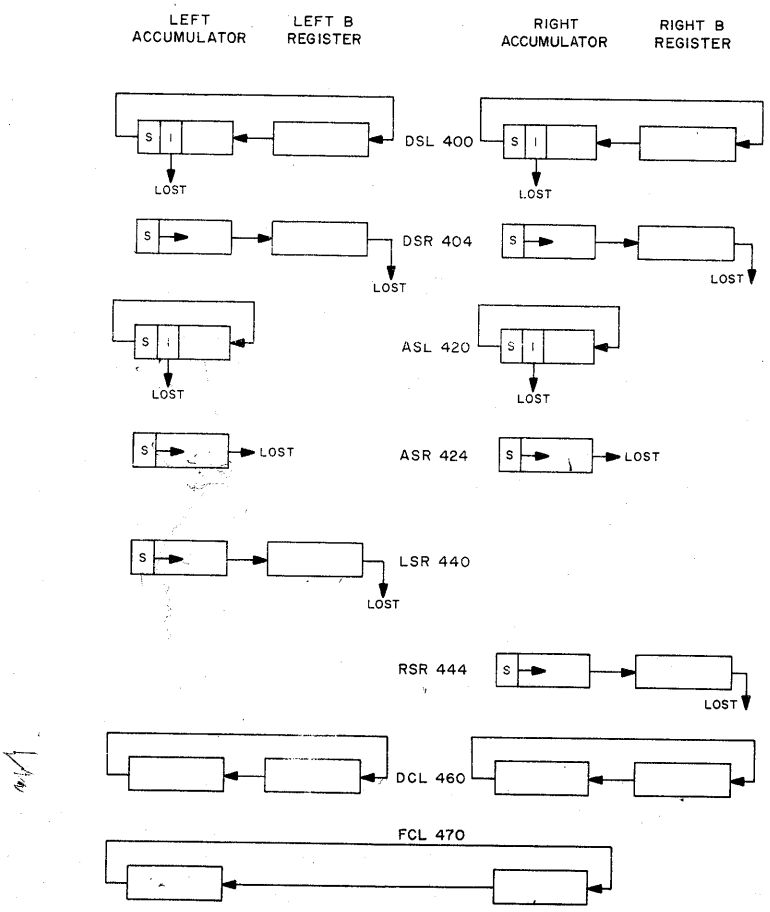


Figure 8-3. Shift Class Instructions

POS With One Item Tag. This pseudo instruction generates the appropriate Full Cycle Left such that the accumulators will be cycled so that the item specified in the director field will have its least significant bit moved to the bit position specified in the decimally-coded AUX columns.

POS With Two Item Tags. This pseudo instruction generates the appropriate Full Cycle Left such that the accumulators will be cycled as to bring the least significant bit of the item in the RC field to the least significant bit position of the item in the Address field.

RES With One Item Tag. This pseudo instruction generates the appropriate Full Cycle Left such that the accumulators will be cycled to move the bit position specified in the decimally-coded AUX columns to the least significant bit position of the item specified in the Address field.

RES With Two Item Tags. This pseudo instruction generates the appropriate Full Cycle Left such that the accumulators will be cycled so as to move the most significant bit position of the item specified in the RC field to the most significant bit positions of the item specified in the Address field.

Figure 8-4 illustrates the effects of various POS and RES instructions. In the case of each pseudo instruction, the appropriate Full Cycle Left will be generated such that the indicated actions will occur. At location AA, a Full Cycle Left will be generated such that the least significant bit position of item NAME will be cycled to bit position 31. This action would require a FCL 9. At location BB, the least significant bit position of item NAME would be cycled to the least significant bit position of item DATE. This action would require a FCL 14. At location CC, the accumulators would be cycled such that bit 15 will be restored to bit position 26, the least significant bit position of item DATE. An FCL21 would be generated. At location DD, the appropriate Full Cycle Left would be generated such that the most significant bit position of item NAME would be restored to the most significant bit position of item DATE. This would require a FCL 18.

A careful study of the above examples and the rules for generating such pseudo instructions should be made by the student. All of the work so far in this chapter has been leading up to a type of coding which will now be illustrated.

Compool Independency

Consider the instructions in the incomplete program in Figure 8-5. At AA, the entire register which contains, among other things, item DOGS is brought into the accumulators. In order that we may work only on the bit positions of item DOGS, we next extract the accumulators with a register containing a mask for DOGS. Thus the garbage will be removed from all bit positions surrounding DOGS. Next, at CC, the information in the bit positions of DOGS will be cycled such that the least significant bit will now be in the accumulator bit 31. Some arithmetic manipulation is now made on the accumulators. At the conclusion of these operations, the results will be restored from bit position 31 to item DOGS. The B Registers are then loaded with a mask for DOGS, and the accumulators will be deposited, in the masked bit positions, into the register which contains DOGS. No other items in that register in memory will be disturbed, because of the method by which the DEP instruction operates.

This sequence of six instructions is one of the most basic and fundamental in SAGE programming. It illustrates what System Development Corporation calls "Compool Independency". This is not the meaning mentioned earlier in our discussion of the IDT Card. There, we meant by this term that the program would not invoke the Compool. In the present context, we imply by Compool Independency that the program would work no matter where item DOGS is located in memory. Further, it does not matter where item DOGS is within that particular memory register. Therefore, as Compool versions come and go, and item DOGS is moved around in memory, this program need only be re-assembled using the new Compool. The instructions which deal with item DOGS will still perform the desired operations.

Figure 8-4. Effects of POS and RES

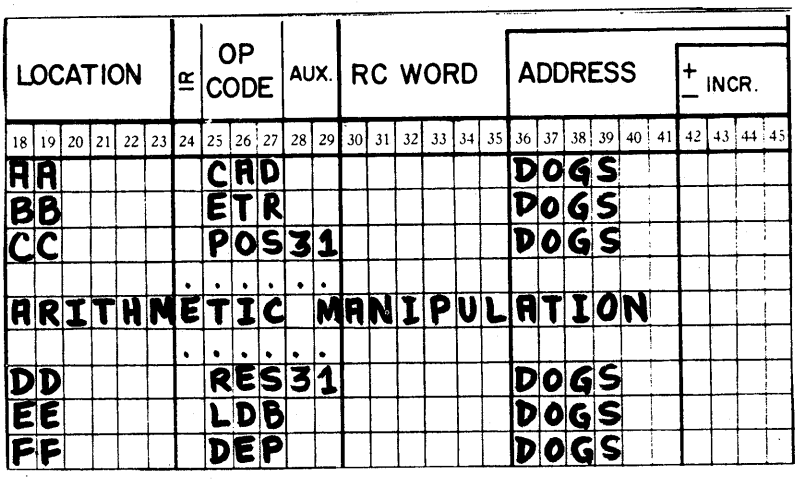


Figure 8-5. Compool Independent Item Manipulation

As student programmers, it has already been noted that you will likely not be using a Compool. Nonetheless, this method of handling items would be most useful in your programming. Once you have defined your items initially at the beginning of your program with ICPO Cards, you need never again concern yourself with bit positions and/or memory addresses. Program Translator will take care of all such busy work. Your coding can become extremely symbolic and not hamstrung by a rigorous and laborious concern with bit positions, et cetera.

CHAPTER 9

SCALING AND FURTHER ARITHMETIC OPERATIONS

BASIC PRINCIPLES; ADDITION AND SUBTRACTION

A number, as such, is of very little use, and must be related in some way to the physical world to be meaningful. In other words, it must represent units of some kind, such as miles, pounds, degrees (these need not be simple units; each integral increment can represent such things as 3/4 of a mile, 1/256 of a revolution, 2.4 pounds) etc., and must contain a radix point, to indicate which parts of the number are integral and which are fractional. The radix point in the decimal system is known as the decimal point, in the octal system as the octal point, and in the binary system as the binary point.

No automatic digital computer keeps track of the units represented by numbers within the machine; this is always up to the programmer. Some computers do keep a record of the position of the radix point within the machine; and since this point may move, according to the operation performed, these are called floating point computers. The AN/FSQ-7 however, keeps no record of the radix point, and treats all numbers as fractions; it is therefore called a fixed point computer.

It is possible to program by converting all numbers to fractional form before inserting them into the computer (by multiplying by a constant), and then when they have been processed, re-converting them to their original form. However, keeping track of the conversion factor is easiest if we think of the computer as actually containing a radix or real point; then we can put numbers into the machine in normal form and simply keep track of the real point. The technique of doing this is called "scaling".

In a fixed point computer, as the name implies, there is one immovable point recognized by the computer as a separator. This is known as the machine point, and is between the sign bit and the most significant magnitude bit position, in each half word. As explained above, we are assuming the existence of a real point, and also of an item point, which is immediately to the right of the least significant bit position in an item; both of these points are imaginary. Thus we can keep track of the position of the real point by noting its relationship to either the machine point, or to the item point (of which the relationship to the machine point must be known).

A complete half word item, in the AN/FSQ-7, will have 15 bit positions between the machine point and the item point. Therefore, in this case, the relationship of these two points need not be explicitly noted, and the position of the real point can be related independently to either.

SCALING NOTATION

It cannot be emphasized too strongly that any method we use for denoting scaling should be used only to speed up the solving of scaling problems, and to facilitate communication on the subject between programmers. To use a method without understanding what is actually happening to the effective position of the real point within the machine is most unwise.

The binary number Ø11Ø.1ØØ is equivalent to 6-1/2 decimal. If this number were to appear in a computer (one with a 7 bit half word, of which the first bit is the sign), the machine point would be between the sign bit and the next bit to the right, thus: Ø.11Ø1ØØ. If you were to multiply the number as it appears in the computer by 2^3 (1ØØØ in binary), you would get the actual value of the number as it appeared originally:

$$\begin{array}{r} \text{Ø.11Ø1ØØ} \\ \text{x} \quad \text{1ØØØ} \\ \hline \text{11Ø.1ØØØØØ} \end{array} \qquad \text{(6-1/2 decimal)}$$

Thus, this number, with the real point in the position originally shown, is said to be scaled 2^3 in relation to the machine point, or 2^3x_m, where the number itself is represented by "x". Thus, $x = 2^3x_m$. Note that the number of integral bit positions (not counting sign) in the item is 3; this same relationship always exists between the scaling exponent and the number of integral bit positions in an item.

The real point scaling can also be noted in reference to the item point. The above number, using the item point as the reference, would appear as Ø11Ø1ØØ.; if we multiply this by 2^{-3} (.ØØ1 binary), we get the correct value:

$$\begin{array}{r} \text{Ø11Ø1ØØ} \\ \text{x} \quad \text{.ØØ1} \\ \hline \text{Ø11Ø.1ØØ} \end{array} \qquad \text{(6-1/2 decimal)}$$

Therefore the item is said to be scaled 2^{-3} in relation to the item point. This may be written as either $2^{-3}x_i$, or $2^{-3}x_6$. The subscript number in this case indicates the number of bit positions between the machine point and the item point. If a number appears as a subscript, it will indicate item point scaling, and not machine point. Note that the item point scaling exponent is equal in magnitude to the number of fractional places in the number, and that item point scaling is easily converted to machine point by adding the subscript number and the scaling exponent. In this case: 6 + (-3) = 3, so $2^{-3}x_6 = 2^3x_m$. Thus, in the AN/FSQ-7, with its 15 magnitude bit half word, a full half word item scaled 2^{-5}_i would also be scaled $2^{1Ø}_m$.

In SAGE programming, the real point is almost never considered to be outside the word itself, so that negative scaling exponents indicate item point scaling, and positive ones indicate machine point scaling; so that the subscripts may be omitted, if so desired. In the DCA Compool item listing, the numbers that appear in the scaling column are the scaling exponents of 2 that apply to each particular item. These are usually in relation to the item point, except those thare are +ØØ, which are to the machine point. Whether you work with machine point or item point scaling depends on which is easier for a particular problem; in most cases, you will probably want to convert item point to machine point before using it.

The above forms of notation will be used within this document. Another form that is sometimes used is $x = 2^{1\emptyset}\bar{x}$, which is equivalent to $x = 2^{1\emptyset}x_m$; item point scaling in this system is shown in the same way, i.e., $2^{-5}\bar{x}$ is equivalent to $2^{-5}x_i$. A third system is to show the number of integral places in an item by using B (before the real point), so that xB1$\emptyset$ is effectively equivalent to $2^{1\emptyset}x_m$, in a half word item. The number of fractional places is shown by A (after the real point), so that xA5 is equivalent to $2^{-5}x_i$.

MAXIMUM PRECISION; INTRODUCTION TO SHIFTING

Once you know the length of an item, and the position of the real point within it, you can easily compute the range of values that can be expressed within the item. The item RPOS, in the DCA Compool, for instance, is scaled 2^{-5}_{i} ($2^{1\emptyset}_{m}$), and each integral unit represents one nautical mile. Thus the largest number that can be represented is 1111111111.11111, or 1$\emptyset$23 31/32 nautical miles. Note that $2^{1\emptyset}$ actually equals 1$\emptyset$24, so this is a convenient method of determining the size of an item that can be contained in a given number of bit positions.

This also works in reverse; you can determine the precision of an item from its scaling. The computer clock, for instance, is scaled 2^{-5}_{15}, and each integral unit represents 1 second. 2^{-5} actually equals 1/32 (and 5 fractional places can contain a number precise to 1/32); therefore you know that the clock is precise to 1/32 of a second. (In this particular case, there is no sign bit; the clock contains 11 integral magnitude bits. Therefore you can think of the machine point scaling as being 2^{11}_{m}. $2^{11} = 2\emptyset48$, so that the largest number that can be contained in the clock is 2$\emptyset$47 31/32 seconds, or 34.14 minutes).

Setting an item to maximum precision simply means determining how large an item can be, then placing it in the word scaled so that the least significant bit will represent the smallest possible quantity, without losing any of the most significant bits. For a single item, this would mean that the most significant bit position would contain a 1; for a group of items, it would mean that the largest item would have a 1 in the most significant bit position.

The placing of the real point in the computer may be done by shifting (other uses of shifting are discussed later in this document). We may think of the real point as being moved along with the item, so that it stays between the same two bits during the shift. However, since we are moving the item, the real point will not be between the same two bit positions in the register at the end of the shift. For example, if we take the binary number $\emptyset\emptyset\emptyset$1$\emptyset\emptyset$.1$\emptyset$, with the real point in the position shown, and shift it left two (assuming we are dealing with an 8 bit register), we will have $\emptyset$1$\emptyset\emptyset$.1$\emptyset\emptyset\emptyset$, which is still the same number, although it is now in a different position.

Thus, we can see that if we have a number in the accumulator scaled 2^{10}_{m}, and we want it to be scaled 2^{12}_{m}, we could shift right (either ASR or DSR, depending on our purpose) two and get what we wanted. If we want to change the scaling of an item, then, this is probably how we will do it.

ADDITION AND SUBTRACTION OF SCALED ITEMS

Since both addition and subtraction are accomplished in the computer by adding, the scaling problems are the same for both operations.

In adding two or more numbers on paper, you must first line up the radix points with each other; the same usually applies when adding in the computer (exceptions are discussed later in this document). You must shift, if necessary, one or more of the numbers to be added, so that all are scaled the same (e.g., have the radix point in the same place) before adding.

One additional restriction applies when adding in the computer; you must make sure there is room in the accumulator for the largest magnitude possible in the result. Therefore, you must determine how large your answer can be, figure out how many places are needed to contain this answer, then scale all the numbers so that their real points are in the same place as you want the real point to be in the answer.

Example:

Given: $A = 2^{-4}A_{15}$, stored in register 100

$B = 2^{-5}B_{15}$, stored in register 101

$C = 2^{-6}C_{15}$, stored in register 102

All of the above numbers are already scaled to maximum precision.

Required: Add the above three numbers and store them in register 103 in maximum precision.

Solution:

```
CAD   100
ASR     1
DCL    16
CAD   101
ASR     2
ADB
DCL    16
CAD   102
ASR     3
ADB
FST   103
HLT
```

Explanation: First you should convert the numbers to machine point scaling, and determine the possible size of the result. This is $2^{11}A_m$ (maximum value just under 2048); $2^{10}B_m$ (maximum value just under 1024); and $2^{9}C_m$ (maximum value just under 512). Adding the maximum values, we get a maximum value for our answer of just under 3584, which can be contained in 12 bit positions, and must therefore be scaled 2^{12}_m. Thus before adding, A must be shifted one position to the right to change its scaling from 2^{11}_m to 2^{12}_m; B must be shifted two to the right (2^{10}_m to 2^{12}_m); and C must be shifted three to the right. The shift instructions in the program accomplish this (the cycle instructions are not used to affect the scaling; they simply utilize the B register for temporary storage).

MULTIPLY CLASS AND ASSOCIATED INSTRUCTIONS

Multiplication in the binary system is accomplished exactly as it is in the decimal system. Since the only digits that can be used in the binary system are "1" and "0", all numbers used as operands or operators can be only combinations of "1" and "0". A binary multiplication table showing all possible products is as follows:

Multiplier X Multiplicand = Product

1	X	1	=	1
1	X	0	=	0
0	X	1	=	0
0	X	0	=	0

As an example, consider multiplication of the numbers 0.1101101_2 and 0.1001000_2.

```
  0.1101101
  0.1001000
  -------
    000000    1st partial product
   0000000    2nd partial product
  1101101     3rd partial product
 0000000      4th partial product
0000000       5th partial product
1101101       6th partial product
-----------
0.0111011101000   final product
```

The method shown is direct binary multiplication, and though it is not exactly how the AN/FSQ-7 performs multiplication, the method will serve as the basis for an understanding 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 identical to the multiplicand. In the above example, there were two "1" bits in the

multiplier; therefore, the multiplicand appeared twice as a partial product (partial products 3 and 6). The partial products are added together to obtain the final product. Thus, multiplication becomes a function of addition.

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 (where addition is normally performed) and the B register, which is an auxiliary register.

The instructions which involve division are also included in the multiply class. Division is accomplished by repetitive subtraction.

All four instructions in the multiply class are indexable. No instruction in the multiply class is capable of causing an overflow.

INSTRUCTIONS FOR MULTIPLICATION

MUL Instruction

The "multiply" (MUL) instruction (octal operation code .Ø25ØØ) is used to obtain the product of the contents of the accumulators and the contents of the specified memory location. The number previously placed in the accumulator acts as the multiplier, and the number from core memory specified by the address portion of the MUL instruction is the multiplicand. Execution of the instruction leaves a 3Ø bit product in the combined accumulator and B register. The least significant bits of the product appear in the B register. The least significant bit of the B register (bit 15) is identical to the sign of the multiplier (original contents of the accumulator) and is not considered a part of the product.

As you know, the AN/FSQ-7 computer treats all numbers as fractions, and when two fractions are multiplied, the product is always smaller than either of the two original factors. For instance, .1 X .1 = .Ø1, a product smaller than either factor. Thus, the possibility of an overflow is eliminated.

Notice that the number of places in the answer is equal to the sum of the places in the multiplier and the multiplicand. Therefore, if two fractions are multiplied, the product will appear in the correct position, but if two whole numbers (treated as fractions) are multiplied, the product will not appear in the correct position. Generally, the product in the AN/FSQ-7 must be rounded off to a fraction of 15 significant bits before it (the product) is further manipulated by the computer. The "round-off" operation is accomplished by the SLR instruction, which will be described later. The rounding off process may affect the precision of the computation.

"Accuracy" and "precision" are terms that have distinct meanings and application in relation to numerical data. "Accuracy" refers to the correctness of the expression, whereas "precision" is the degree of correctness. For instance, a person is sometimes described as being over 35 years old. This may be accurate, but it is not precise--the person may be precisely 35 years, 4 months, 12 days, 16 hours, 13 minutes, and 28 seconds of age. Similarly, "pi" may be expressed as 3.14; this is a representation of the value of "pi" accurate to three places, but it is not precise. Precision would require that the value be expressed to some greater number of decimal places.

SLR Instruction

The "shift left and round" instruction provides the means of manipulating a number within the arithmetic element and rounding off that number to 15 significant bits. You will recall that the result of a multiplication leaves a 31-bit product (including sign bit) in the combined accumulators and B registers. This 31-bit word is not compatible with the word length of the AN/FSQ-7--the computer deals with half words of 16 bits each, but the multiplication product is almost twice that length. Consequently, some method must be found to reduce these numbers to 15 significant bits and yet preserve the greatest precision possible. This is the function of the SLR instruction.

The SLR instruction shifts the data in the combined accumulators and B registers to the left by the number of places specified by the right half portion of the instruction and then rounds off the accumulator contents to 15 significant bits. Thus, a shift of two places to the left 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, and so forth through the entire register, except for the accumulator sign bit, which remains unchanged. The computer performs round-off in much the same manner as we perform round-off with pencil and paper. For instance, the decimal number "96.5" is rounded to "97", whereas the decimal number "94.4" is rounded to "96". In binary arithmetic, we are concerned only with "1's" and "Ø's". In the machine, the sign bit of the B register is sensed--that is, if it is a "1", the contents of the accumulator are increased by "1"; if it is a "Ø", the accumulator contents are left unchanged. (Remember that the sign bit of the B register does not indicate the sign of the contents of the B register when it is used with multiplication or division instructions. Rather, it is a magnitude bit.)

The SLR instruction is used after a multiply or a divide instruction when it is desired to round off the left and right half words to 15 most significant bits plus a sign bit. Each accumulator and its respective B register is formed into a 31 bit shifting register, with connections established so that the sign bit of each B register is connected to bit 15 of the corresponding accumulator register. As each shift is performed, the contents of accumulator bits L1 and R1 are lost, and the contents of B register bits L15 and R15 are replaced with the associated accumulator sign bit. The contents of the accumulator sign bits are not altered by the shift operation. The shift left operation is followed by the round-off operation (if required). For the round-off operation, the contents of the combined registers are first put into positive form, and then a "1" is added to the contents of the accumulator if the associated B register's sign bit contains a "1". The sign of the accumulator is then restored, and, finally, both B registers are cleared at the completion of the instruction.

Execution of the SLR instruction can cause an overflow. The instruction, which is actually in the miscellaneous class, is designated by an octal operation code of .ØØ24Ø.

TMU 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 accumulators. Thus, the TMU instruction causes the left half word obtained from the memory to be used as the multiplicand in both arithmetic elements. Execution of the instruction leaves

a 3ø-bit product plus a sign bit in each combined accumulator and B register. The least significant bits of the products appear in the B register, but the least significant bit of the B register (bit 15) is identical to the sign of the multiplier, and it is not considered a part of the product. The sign of each product is determined by the usual rule of algebraic signs.

INSTRUCTIONS FOR DIVISION

DVD Instruction

The "divide" (DVD) instruction is used to divide the number in the combined accumulator and B register by the number in the core memory location specified by the address portion of the DVD instruction. The left and right half words are divided into the contents of the 32-bit combined left accumulator and B register and the 32-bit combined right accumulator and B register, respectively. The sign bit of each quotient appears in the sign bit portion of the associated accumulator register. However, the two unsigned 16-bit quotients appear in the associated B registers, and the remainders appear in bits 1 through 15 of the associated accumulators. Generally, after each division is executed, the SLR 15 instruction is used to move the quotient from the B registers into the accumulators and to round off the result to 15 significant places plus a sign bit (the maximum half-word length we can store in memory). The instruction is indexable and is specified by an octal operation code of .ø26øø.

Up to this point, the problem of division in the AN/FSQ-7 has been described only from the standpoint of actual execution of the instruction. However, you will recall that the computer has been designed to handle only those numbers that fall between the limits of +1 and -1. In multiplication, no difficulty was encountered because the product of two fractional numbers is always another fractional number. But in division, the quotient of two fractions may be a number larger in magnitude than the computer can handle. For instance, the division of ".6" by ".2" yields a quotient of "3", a number that exceeds the -1 to +1 limits. Therefore, the programmer must make sure that the operator to be used as a divisor is larger in magnitude than the dividend. (This arrangement will always ensure a fractional answer). If this precaution is not taken prior to a division instruction, the results in the combined accumulator and B register may be meaningless. Furthermore, division by zero is obviously as undesirable in computer operations as it is in mathematics in general. The methods for scaling shown later will take care of all this automatically.

TDV 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 elements. Execution of the instruction leaves an unsigned quotient of 16 significant bits in each of the B registers and the remainder and sign bit of the quotient in each of the accumulators. Generally, the quotient must be brought into the accumulator and rounded off to 15 significant bits before it is further manipulated by the computer. This shifting and rounding operation is usually accomplished by a SLR 15 instruction.

The restrictions applicable to the DVD instruction also apply to the TDV instruction--dividing by zero is prohibited, and the divisor must always be greater in absolute magnitude than the dividend. Consequently, both the upper boundary of the dividend and the lower boundary of the divisor must be known by the programmer. The TDV instruction is designated by an octal operation code of .ØØ264Ø and is indexable.

MULTIPLICATION AND DIVISION

ADDITIONAL CONSIDERATIONS

Multiplication

In multiplying in the computer, you never have to worry about having room for your answer, as you always multiply two 15 magnitude-bit half-words by two other 15 magnitude bit half words, and you have two 3Ø magnitude bit half words to contain your answer (accumulator and B register combined). So the only problem is in getting the real point of the answer in the place you want it.

When you multiply two numbers on paper, you count the number of fractional (usually decimal) places in each of the two numbers, add these together, and mark off this number of fractional places in your answer. The computer works in much the same way, in effect - if you are multiplying two numbers, each scaled $2^{1Ø}_{m}$ (5 fractional places each) together, your answer will have 1Ø fractional places, which are marked off from the right end of the B register, not counting the rightmost bit as a position, because this is a sign bit. Therefore, your answer will consist of 2Ø whole number places and 1Ø fractional places, or the real point will be 5 places to the right of the left end of the B register. Therefore, if you want your answer to be scaled $2^{1Ø}_{m}$ also, you must SLR 1Ø, to bring the real point back into the accumulator, 5 places to the left of the right end. (Here you must be careful to make sure that your answer can be contained in 1Ø whole number places - not when you multiply, but when you shift left after multiplying.)

Because the computer always contains leading zeros, you can also count the number of integral places in each number being multiplied, add these, and have the number of whole number places in the answer. And because the number of integral places is equal to the scaling exponent, in relation to the machine point, you can simply add exponents and determine the scaling of your answer. In the above case: $2^{1Ø}_{m} \times 2^{1Ø}_{m} = 2^{2Ø}_{m}$; so that immediately after multiplying, your answer is scaled $2^{2Ø}_{m}$ (the real point is 5 places inside the B register, as above). Therefore, if this time you want the answer scaled 2^{15}_{m}, you would have to SLR 5.

Example:

Given: $A = 2^{8}A_{m} \leq 25Ø$, stored in register 1ØØ

$B = 2^{10}B_m \leq 24$, stored in register 101

Required: Multiply A x B and store the answer in register 102, in maximum precision.

Solution:

```
CAD   100
MUL   101
SLR     5
FST   102
HLT
```

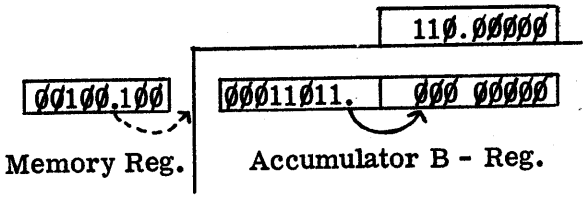
Explanation: Since we know the maximum of both numbers, we can multiply these together and get the maximum for the answer. Doing this, we get: 25 x 24 = 600. We can contain 600 in 13 integral bit positions, so our final answer must be scaled 2^{13}_m. Since in multiplying we can add scaling exponents to get the scaling of the result, the multiplication becomes:

$$2^8A_m \times 2^{10}B_m = 2^{18}AB_m$$

To change 2^{18}_m to 2^{13}_m, we must SLR 5.

Division

This also works much the same in the computer as it does on paper, as far as scaling problems go. When you are dividing, you first count the number of fractional places in the divisor, and then move the radix point that same number of places to the right in the dividend, before dividing. The computer also does this, in effect; therefore, if you were dividing a number scaled 2^{15}_m by one scaled 2^5_m (10 fractional places), the computer would (in effect - remember that the computer does not recognize the existence of the real point) move the real point 10 places to the right of its original position - in this case, to 10 places inside the B register. This can be thought of graphically as follows (using a different example, and 8 bit registers):



(There is of course only one B register: it is shown above to indicate its change of function during division: first it contains part of the dividend; then the quotient).

Note that during the course of the division, the computer has effectively moved the real point 3 places to the right of its original position in the accumulator, because the divisor has three fractional places. Note also that the dividend originally had 7 integral magnitude bits (the 8th bit is the sign), whereas the divisor had 4. The quotient

has 3 integral magnitude bits (there is no sign bit in the B register after a division), which is equal to the difference between 7 and 4. Since the number of integral magnitude bits is equivalent to the scaling exponent, we find we can subtract scaling exponents in division to determine the scaling of the quotient. Thus:

$$\frac{2^{7}x_{m}}{2^{4}y_{m}} = 2^{3}x/y_{m}$$

The above example also serves to point out the chief difficulty involved in scaling numbers for division within the computer: the quotient must be able to fit within the B register. In the above case, for instance, if the number in the accumulator happened to be placed one bit position to the left of its present position before division, there would not be room in the B register for the most significant bit of the quotient, and the answer would be anvalid. (In this connection, note that in addition, subtraction, or multiplication, if you do not allow sufficient room for your answer, you will merely lose the most significant bits; whereas in division, because of the manner in which the computer works, you will not actually lose any bits, but your answer will be completely invalid).

Therefore, before dividing in the computer, you must always determine the largest possible size of your result, and if necessary shift the dividend (or in some cases, the divisor) before dividing so that the quotient will fit within the B register. For instance, if you divide a number scaled $2^{1\emptyset}{}_{m}$ by another also scaled $2^{1\emptyset}{}_{m}$, without shifting first, your answer will be scaled $2^{\emptyset}{}_{m}$ (in the B register; e.g., the real point will be at the extreme left end of the B register, immediately after dividing). Therefore, unless you knew the largest possible answer was less than 1, the results would be invalid. In this case, if you knew your answer would require $1\emptyset$ integral bit positions ($2^{1\emptyset}{}_{m}$) you would have to shift the dividend right $1\emptyset$ before dividing.

In shifting before dividing, it is a good idea, unless you have just finished a multiplication operation, and therefore know what is in the B register, to DSR 16, then DSL x (x = 16 - the number you would normally DSR) before dividing, to make sure that the unused portion of the B register contains sign bits. In the above example, this would be DSR 16, then DSL 6 (which puts the real point in the same place that a DSR $1\emptyset$ would).

Because the quotient, after dividing, is contained wholly in the B register (and is scaled in relation to it), you will normally SLR 15 at the end of the operation to put it wholly in the accumulator.

Example:

Given:

$$A \leq 1\emptyset = 2^{4}A_{m}, \text{ stored in register } 1\emptyset\emptyset$$

$$1 \leq B \leq 2\emptyset = 2^{5}B_{m}, \text{ stored in register } 1\emptyset1$$

$$C \leq 3\emptyset = 2^{6}C_{m}, \text{ stored in register } 1\emptyset2$$

$$1 \leq D \leq 4\emptyset = 2^{7}D_{m}, \text{ stored in register } 1\emptyset3$$

Required: Compute $\frac{\frac{A}{B} \times C}{D}$, and store it in register 1Ø4 in maximum precision.

Solution:

```
CAD     1ØØ
DSR      16
DSL      11
DVD     1Ø1
SLR      15
MUL     1Ø2
DSR       6
DVD     1Ø3
SLR      15
FST     1Ø4
HLT
```

Explanation: The largest possible magnitude of A/B is 1Ø, which must be scaled 2^4_m. A straight division would give an answer scaled 2^{-1}_m (4 - 5 = -1), so before dividing we must shift right 5 (DSR 16, then DSL 11); now the answer will be scaled 2^4_m. We then bring the quotient back into the accumulator with a SLR 15. When we multiply this result by C, scaled 2^6_m, the answer will be scaled $2^{1Ø}_m$ (4 + 6 = 1Ø). Since the largest magnitude we could have is 3ØØ (3Ø x 1Ø), we could scale it 2^9_m. However, it would do us no good, because we would have to immediately shift it right again for the next division; therefore, for the time being, we leave it where it is, scaled $2^{1Ø}_m$. The largest number we can get from dividing this result by D is 3ØØ, which needs 9 whole number places. If we divided without shifting, the answer would have only 3 whole number places (1Ø - 7); therefore, we must first DSR 6, to insure that the answer can be contained within the B register. Note that this time it is unnecessary to DSR 16 and then DSL 1Ø, because we know what is in the B register, and in fact want to use part of it.

If you want to set up scaling equations for division, a good way is first to set them up as an inequality and then see what must be done to the dividend to make the inequality equal (or in occasional cases, to the divisor); because to start with you know the scaling of both the dividend and the divisor, and must also know or compute the necessary scaling of the quotient. The first division in the above problem, then, could be set up as follows:

$$\frac{2^4A_m}{2^5B_m} \neq 2^4A/B_m$$

We can see that to make this an equality, we will have to change the scaling of A to 2^9A_m, which may be done by a shift right of 5.

RELATIONSHIP BETWEEN ITEM, MACHINE, AND REAL POINTS

Of the three reference points in the AN/FSQ-7, only the machine point is always in the same place; both the item point and the real point may be moved by the programmer. The machine point is also the only actual point; both the others are imaginary and exist only in the mind of the programmer (and in his notes).

The real point, then, can be moved at will by the programmer (in an item he has control over) if he merely decides it is now in a different place, and changes his concept of the scaling and magnitude of the item accordingly. For instance; assume that the binary number Ø111Ø111 is in an 8 bit (1st bit = sign) register in a computer, and is scaled 2^{4}_{m}. The real point is therefore as shown: Ø111Ø.111, and the number represents 14.875 decimal. If you were to move the real point one place to the left, you would have Ø111.Ø111, which is now scaled 2^{3}_{m}, and represents 7.4375 decimal, even though the actual bits within the computer remain unchanged. Note that by moving the real point one position to the left, we effectively divided the number by 2 (and changed the scaling). (This change in magnitude is the reason we did not move the real point this way up to now.)

The item point, because it is always just to the right of the rightmost bit in an item, is a little less flexible. We can move it, but to do so we must either shorten or lengthen the item, or shift the actual binary bits of the item within the computer. This is best covered in the next section.

Shifting

So far, we have been using shifting only as a means of changing the machine point scaling, and have not thought about whether we were changing the position of a whole item along with that of the real point, or whether we were only shifting the bits of an item (and the real point) within the limits of the item. Actually, to change machine point scaling, we can do either of these; however, it is important to recognize the distinction.

Let us consider the case of a 5 bit item at the right end of the accumulator, scaled $2^{-Ø}_{15}$ (real point and item point in the same place): ØØØØØØØØØØØØ [ØØ1ØØ.] If we shift this left two, we can think of it as keeping the item (and item point) in the same place, and moving the bits and the real point, so we have now: ØØØØØØØØØØØØ [1ØØ.ØØ] , which is scaled 2^{-2}_{15}.

Another way to think of this same shift is that it changes the position of the item point, real point, binary bits, and the position of the item within the register all at once, so that our result is ØØØØØØØØØ [ØØ1ØØ.] ØØ. The scaling started as $2^{-Ø}_{15}$, and is now $2^{-Ø}_{13}$, so that the scaling of the item within itself (to the item point) has not changed.

It is important to realize that the choice of what this shift actually does is completely up to us, because the computer doesn't even know there is a real point or an item point within it.

Thus, we see two uses of shifting: to change the scaling of an item, or to simply move the item from one place to another. There is one more major use: to multiply the item by a power of two, without changing the scaling or position of the item.

Using the same item and the same shift as above, we this time think of it as keeping the item and real points stationary, so the word becomes: ØØØØØØØØØØØØ 1ØØØØ. We have effectively multiplied the number by 2^2 (a shift left of three would multiply it by 2^3; a shift right of three would multiply it by 2^{-3}, etc.). Originally we had $x = 2^{-Ø}x_{15}$; we have multiplied both sides of this equation by 2^2, giving us $2^2{}_x = 2^{-Ø}x_{15}$; (before shifting); we then change the right hand side of this by shifting left two, which gives us $2^2x = 2^{-Ø}x_{15}$; so the number has been multiplied by 4.

Example:

Given: $A = 2^8A_m$, in bit positions 1 - 14 in the accumulator

Required: Generate one shift instruction that will multiply A by 4, move it into bit positions 2 - 15, and change the scaling to $2^{11}A_m$.

Solution: ASR 1

Explanation: The real point in this item is between bit positions 8 and 9, as given in the scaling notation. We can multiply the item by 4 simply by moving the real point 2 places to the right, so it is now between bits 1Ø and 11. To move the item into positions 2 - 15 we must ASR 1, moving the real and item points at the same time, so as not to alter the magnitude of the item. The real point is now between bit positions 11 and 12, which is where we want it, because this represents a scaling of $2^{11}A_m$. This could be shown in scaling notation as follows:

Originally: $A = 2^{-6}A_{14}$

Multiply by 4: $4A = 2^{-4}A_{14}$

Change position (ASR 1): $4A = 2^{-4}A_{15}$

We could have also multiplied by 4 by shifting the item left two, keeping the real and item points stationary. However, by so doing we would have lost 2 bits out of the left end of the accumulator, and would still have had to shift right 3 to change the scaling to the desired value.

Adding Items with Different Scaling

It is possible to add items that are not scaled the same directly, without shifting. However, since the real points must still be lined up, we would have to move the real point in one or both of the items first, thus changing the magnitude of the items. For example, $2^{10}A_m + 2^8B_m$ could equal either $2^{10}(A + 4B)_m$, $2^8(A/4 + B)_m$, or even $2^9(A/2 + 2B)$. This may sometimes prove useful.

SUMMARY

There are many different valid concepts of scaling; we have presented only one or two of them. It is expected that once you become familiar with the subject, you will develop your own methods. However, whatever method you use, you should make sure that you understand exactly what happens to the effective position of the real point during each computer operation; otherwise the subject will never be completely clear to you.

CHAPTER 10

COMPARE AND TEST BITS INSTRUCTIONS

Many operations in programming depend upon logical comparisons. This chapter will present the aid instructions of the compare class, and two test bits instructions. With these instructions, the programmer will be enabled to make logical comparisons and thereby alter the course of program flow.

TEST BITS INSTRUCTIONS

The following two instructions ascertain the contents of one or two particular bits in memory. For these instructions, the contents of the accumulators and the B registers are irrelevant, and these registers are not altered.

TOB Instructions

The TOB (Test One Bit) instruction tests the memory word specified in the director portion of the instruction. It tests the particular bit specified in the aux columns, which are coded in decimal. The aux columns will therefore contain a number between Ø and 31_{10}. If the bit tested is a zero, the instruction takes no special action. If, on the other hand, it is a one, the next instruction will be skipped (i.e., an extra "one" will be added to the program counter). As noted above, neither the accumulators nor the B registers are tested or altered.

The TOB instruction has a special coding provision should the director specified on the coding sheet be an item. In this case, the bit specified in decimal in the aux columns will be the relative bit position within the item. Bit numbering within an item starts with Ø as does the numbering of the bit positions within the entire computer word.

TTB Instruction

The TTB (Test Two Bits) instruction tests two bits within the memory word specified in the director portion of the instruction. It checks the bit specified in the aux columns, which are coded in decimal, and the bit to its left. Depending upon the binary contents of the bits tested, an equivalent number of instructions will be skipped. For example, if the bits tested contain 11_2, three instructions will be skipped. Should the bits tested contain $ØØ_2$, no instructions will be skipped.

If the bits specified in the aux columns happen to be ØØ or 16 (that is, LS or RS) only the bit specified will be tested. The instruction will then have the action of a TOB instruction.

There are also special coding provisions for the TTB instruction. Should the director specified on the coding sheet be an item, the bits tested will be that relative bit position within the item specified in the aux columns and the bit to its right.

Assembling the Test Bits Instructions

You will note on your Programmer's Card that the octal codes for the test bits instructions are listed differently from those of any other instructions in the repertoire of the AN/FSQ-7. Specifically, the TOB (Test One Bit) instruction is shown as having octal codes ranging from "Ø5ØØ to Ø537". This means that the instruction's octal code is actually of the form Ø5, similar to that of the BPX instruction. Bits L1Ø through L15 of the instruction, which correspond to the two least significant octal digits of the LHW, are used to specify the bit which will be tested in the memory word. Obviously, the $ØØ_8$ through 37_8 correspond to the $ØØ_{1Ø}$ through $31_{1Ø}$ by which the programmer specifies, in the aux columns, the bit to be tested.

The TTB (Test Two Bits) instruction is shown on the Programming Card as having octal codes ranging from "Ø54Ø to Ø577". The least significant two octal digits again specify the bit to be tested. Of course, this instruction will actually test two bits, the bit specified and the bit to its left. Again, a correspondence exists between $4Ø_8$ through 77_8 and $ØØ_{1Ø}$ and $31_{1Ø}$. If one were manually assembling a COSEAL-coded program which employed a TTB instruction with an item director, several translation processes would have to be made. The aux columns would have been coded in decimal, and the bit specified would have been the leftmost of the two bits.

Examples of Test Bits Usage

Refer to Figure 10-1, where you will find program TEST. The first instruction in this program will bring a +.87, -428 into the accumulators. For tracing the course of subsequent program flow, this fact is irrelevant. The next instruction tests bit $15_{1Ø}$ of memory location Ø1A. Depending upon the contents of that bit, the program may or may not skip the first of the subsequent two HLT's. We see that the left half word of location Ø1A contains an octal positive zero. Therefore, the bit tested will be a zero, and the next instruction will not be skipped. This means that the program will come to a halt at the first of the two HLT instructions. Since the program does not contain a LOC card, it has been assembled for location 3ØØ, and therefore the HLT instruction at which the program halts will be at location 3Ø2. The program counter will thus contain 3Ø3 when the program halts.

LOCATION	IR	OP CODE	AUX.	RC WORD	ADDRESS	± INCR.
		IDT			TSTBIT	
		CAD		+.87	- 428	
		TOB	15		Ø1A	
		HLT				
		HLT				
Ø1A			Ø		21	
		END				

Figure 10-1. Example of TOB Instruction

Figure 10-2 contains an example of the action of a TTB instruction. The instruction has been coded using an item director, and therefore the bits tested will be the relative bit position specified within the item and the bit to its right. Specifically, we are testing bit $\emptyset 4_{1\emptyset}$ of item MESS in location MESS + 1. Since MESS is in table TRY$\emptyset$, location MESS + 1 will be memory address $1\emptyset\emptyset 1_8$. Figure 10-3 shows an expanded view of the left half word of that memory location, and the bit positions of item MESS have been specifically noted. We see that the bits tested contain a $\emptyset 1_2$, and therefore one instruction after the TTB instruction will be skipped. The program will therefore branch to location BB and halt. Since location BB will be assembled for memory address 3$\emptyset$7, the program counter will contain 31$\emptyset$ when the program halts.

LOCATION	R	OP CODE	AUX.	RC WORD	ADDRESS	± INCR.
		IDT			TST 2BT	
TRY$\emptyset$		TCPO		2	1$\emptyset\emptyset\emptyset$	
MESS		ICPO$\emptyset\emptyset$7		8	TRY$\emptyset$	
		CAD			MESS	
TEST		TTB$\emptyset$4			MESS	+ 1
		BPX			AA	
		BPX			BB	
		BPX			CC	
		HLT				
AA		HLT				
BB		HLT				
CC		HLT				
		LOC			1$\emptyset\emptyset\emptyset$	
		$\emptyset$61331			-	$\emptyset$
		$\emptyset$55544			1572	
		END				

Figure 10-2. Program Showing Example Actions of TTB Instruction

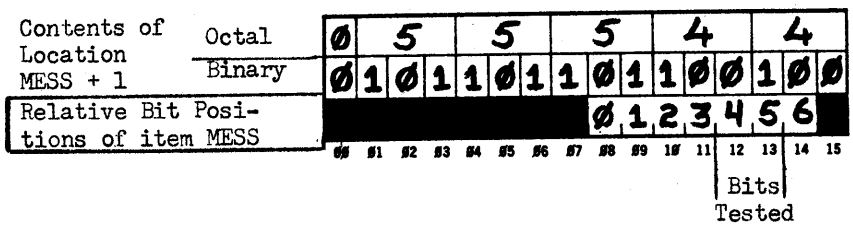


Figure 10-3. Expanded View of Left Half Word of Register MESS + 1

COMPARE INSTRUCTIONS

The eight instructions known as the "compare" instructions enable us to check various numbers of bits in the accumulators against the corresponding bit positions of the specified memory location. If the bit or bits checked in the accumulator do not compare exactly with those in memory, these instructions take no special action. Otherwise, should an exact comparison exist, the next instruction will be skipped.

CMF Instruction

The CMF (Compare Magnitude Full) instruction compares the entire contents of both accumulators with the contents of the entire specified memory location. As noted above, if a comparison is made, no special action is taken; otherwise, the next instruction will be skipped.

CML Instruction

The CML (Compare Magnitude Left) instruction performs a comparison on the left accumulator and the left half word of the specified memory location only. Right half words are not examined.

CMR Instruction

The CMR (Compare Magnitude Right) instruction compares the contents of the right accumulator with the right half word of the specified memory location.

These three compare instructions perform a comparison and do not alter the contents of either the accumulators or the specified memory location. The next three instructions not only perform comparisons but also execute a subtraction as well.

Compare and Difference Instructions

As stated previously, the actions of these instructions are twofold. First, the three compare and difference instructions perform a comparison of exactly the same sort as do the corresponding compare magnitude instructions. That is, for example, the CDR instruction first compares the contents of the right accumulator with the contents of the right half word of the specified memory location. If a comparison is made, no special action will be taken at the conclusion of the operation of the CDR instruction. If a comparison is not made, at the conclusion of the operation of this instruction the next instruction will be skipped. Another way of stating this is: if no comparison is made, a "one" will be added to the contents of the program counter. As with the execution of any instruction, the program counter already contains the address of the next sequential instruction. By adding a one to the program counter, the computer ensures that the next instruction will be skipped.

After the appropriate comparison is made, whether it be a CDF, CDL or CDR instruction, a subtraction will next be performed. The contents of both accumulators will be subtracted from the contents of both half words of the specified memory location.

Even if, for example, only the right half words were compared, as in the action of a CDR instruction, nonetheless the subsequent subtraction will be made in both accumulators. The results of the subtraction are left in the accumulators, and the specified memory location is left unchanged. It is very important to note that the subtractions made by the compare and difference instructions are a reversal of the normal action of subtraction in the Q-7. Normally, the contents of memory are subtracted from the contents of the accumulator. You will note from the above that this is not true in the case of the compare and difference instructions.

Maskable Compare Instructions

The CMM (compare masked magnitude bits) instruction permits us to perform comparisons on selected bit positions. By the action of a prior instruction (probably an LDB), the programmer will have loaded a mask into the B registers. The actions of a subsequent CMM instruction are as follows:

(1) The masked bit positions of the accumulators (that is, the bit positions where the equivalent B register bits contain a "1") will compare with the equivalent bit positions of the specified memory location. If a comparison is made, no special actions will be taken. If no comparison exists, the next instruction will be skipped.

(2) After the comparison is performed, the accumulator's unmasked bit positions will be filled with "one" bits.

(3) The specified memory location and the B registers are not altered by the CMM instruction.

CDM Instruction

The CDM (compare and difference masked magnitude bits) instruction initially performs the actions of a CMM instruction. It then attempts to perform a subtraction in the masked bit positions. However, because end carry circuits have only been built into LS and RS, the actions of this instruction are usually not useful to the programmer. Under special circumstances, this instruction could indeed be used; but when one considers the amount of time necessary to predict the actions of this instruction, it would seem wiser to use more straightforward programming methods. For the reader who enjoys trivia and would like to spend a considerable amount of time poring over Q-7 circuit diagrams, it can be shown that it would have actually required extra wiring for the Q-7 not to have this rather useless instruction in its repertoire.

CHAPTER 11

MACRO AND COMAND COMPILERS

COSEAL's program Translator contains two relatively advanced capabilities not often found in assembly programs. These are the MACRO and COMAND compilers. By invoking the use of these compilers during the assembly process, the usual assembly rule of one line of coding yielding one instruction output may be broken. MACRO compiler enables the programmer to get a repetition whenever and wherever in his program he may wish of any frequently used set of instructions. COMAND compiler enables the programmer to avoid entirely the busy-work coding involved in many frequently found operations.

MACRO COMPILER

Frequently, a programmer uses the same set of instructions many times in a program such as:

```
ADX     Loc 1
STA     Loc 2
```

varying only the parameters (loc 1, loc 2 in this example) with each use. Any such sequence may be defined by the programmer as macrocoding. To form a macro definition of the above example, one would do the following:

```
TOP     TAG1,TAG2$ (macro definition)
ADX         TAG1
STAA        TAG2
BOT
```

This macro definition consists of the following:

1. 1st Line

   Cols. 18-22 - Name of Macro containing up to 5 alphanumeric characters.

   Cols. 25-27 - TOP

   Cols. 3Ø-69 - TAG indication of 1st Col. where parameters are to be inserted.

2. 2nd Line - Next to Last Line (skeletal portion of macro definition)

   Cols. 18-45 - Use regular symbolic coding. Wherever the programmer desires to have the coding variable he should place a parameter tag in the 1st col. to be altered.

3. Last Line

   Cols. 25-27 - BOT - Not a part of the skeletal portion of a macro definition. Just serves as a termination indicator. Non-generative, yielding no assembly output.

Rules for the Naming of Macros

They must be one to five alphanumeric characters and must begin in column 18. They may not consist of all numbers. Macro names are kept in a special table. They will not conflict with internal program tags, compool overrides or compool tags. Macro names should not have as their first three letters any Q-7 instruction or COSEAL assembly instruction. (i.e., IDT, LOC, SKP, CAD.)

Rules for Parameter Tags

They may be one to five alphanumeric characters. They may not contain spaces or special characters such as dollar signs, commas, etc. If the parameter tag is only one character long, which we will see later on is a good policy to follow, it may not be A, B, C, D, E, 1, 2, 3, 4, or 5.

On the 1st line of a macro definition:

(a) 1st parameter tag must begin in Column 3Ø
(b) a comma must separate parameter tags.
(c) Parameter tags must be terminated by a $ - very important, if not terminated by a $ program won't be assembled properly.
(d) Parameters must be terminated by Column 69. They can't begin again in Column 3Ø of the next line.

Within the Skeleton:

(a) Must agree with a tag in Columns 3Ø-69 of 1st line of macro definition.
(b) Must fall completely within one of the following fields:

    (1) Cols. 18-22
    (2) Cols. 25-29
    (3) Cols. 3Ø-35
    (4) Cols. 36-41
    (5) Cols. 42-45

In the event Columns 25-29 do not agree with a parameter tag in Columns 3Ø-69 of 1st line of definition, this field is broken up further:

(a) Cols. 25-27
(b) Cols. 28-29

In the event Columns 42-45 do not agree with a parameter tag in Colums 3Ø-69 of 1st line of definition, this field is broken up further:

(a) Cols. 42
(b) Cols. 43-45

(An example of the importance of the above -- if you have 31, as a parameter tag in Columns 3Ø-69, and as a skeletal instruction you have a POS31, then this 31 will be considered as a parameter tag -- Translator will assume that this is what is desired.)

All macro definitions should appear at the top of the program, but in any event they must appear before any call is made on them, the definition of the macro is itself non-generative.

Macro Calling Instructions

Now if the name ADXS in columns 18 through 22 of our initial example were placed in columns 25 through 29:

```
ADXS     100A,100B$     as a macro calling instruction -- would cause the
                        following coding to be generated in its place:

ADX      1ØØA    *      ADXS     100A,100B$

STAA     1ØØB    *
```

An asterisk will appear in Column 46 of the listing to indicate generated coding. (This asterisk also appears in Column 46 for COMAND type pseudo instructions.) The original macro calling instruction will appear in the comments Columns of the listing. The programmer's own comments will also appear in the listing, although it may be necessary to place part of the comments on a 2nd line of generated listing. To be safe, the programmer should put any comments in a skeleton on the 3rd line. (These comments should be started in Column 47. Don't use Column 46.)

Rules for Substituted Values

Substituted values may be one to thirty characters in length. They may be larger than the parameter tags they replace, but they cannot be smaller. The first value must begin in column 3Ø, and the last value must be terminated by column 69. They may not begin again on the next line. The values must be separated by commas and terminated with a dollar sign. You don't have to consider the size of the values supplied the last time a particular macro was called. One always receives a fresh copy of the particular macro skeleton being called. If you don't furnish a value large enough to cover a parameter tag, that portion of the tag not covered will remain and will in most cases cause an error.

All calling macro instructions with any parameter values supplied will appear in the comments columns in the final listing. The programmer's comments will not be lost. In the final listing the programmer will have the original definition, his macro calling instruction, and the altered skeleton (in symbolic -- and octal).

Examples of MACRO Usage

```
MACSX       TOP         F,G,X,G$
            CAD         F
            SUB         G
            X           H  SUBTRACTION
            BOT
```

If the above macro definition had been included with macro definitions following the IDT card, then the following macro calling instruction:

```
MACSX       1ØA,3D,BPX,1ØØBC$      SUB THEN GO TO END
```

would cause these instructions to be generated.

```
CAD         1ØA         *          MACSX1ØA,3D,BPX,1ØØBC$  SUB
SUB         3D          *          THEN GO TO END
BPX         1ØØBC       *          SUBTRACTION
```

Another example of a macro definition:

```
TT          TOP         ITEM,I,YY,F,G,H,X,J$
            ITTBYY      ITEM
            BPX         F
            BPX         G
            BPX         H
            X           J
            BOT
```

This combination of instructions was used 30 times in one DCA program. If this programmer defined macro had been placed after the IDT card, then its sketetal portion (symbolic coding between TOP and BOT) could be called forth with but a single macro calling instruction each time it was necessary to use this set of instructions.

TT CMOD,3,Ø2,4ØG,4ØG,44A,BPX,4ØB$ (macro calling instruction) would cause to be generated:

```
3TTBØ2      CMOD        *          TT   CMOD,3,Ø2,4ØG,4ØG,44A
BPX         4ØG         *               BPX,4ØB$
BPX         4ØG         *
BPX         44A         *
BPX         4ØB         *
```

TT TPOS,1,31,37G,37K,37N,FST,1ØB$
(referring to same macro definition)

would cause to be generated:

```
1TTB31      TPOS     *      TT    TPOS,1,31,37G,37K,37N
 BPX        37G      *            FST,1ØB$
 BPX        37K      *
 BPX        37N      *
 FST        1ØB      *

 TT   ØØØ5ØØ,2BPX,Ø1,37G,37K,37N,HLT,ØØØ$
```

would cause to be generated:

```
2BPXØ1      ØØØ5ØØ   *      TT    ØØØ5ØØ,2BPX,Ø1,37G,37K,37N,
 BPX        37G      *            HLT,ØØØ$
 BPX        37K      *
 BPX        37N      *
 HLT        ØØØ      *
```

Now to show some errors with the same macro definitions.

```
 TT   1ØA,1  ,2,37G, 37N,HLT,ØØØ$
```

would cause to be generated:

```
1 TB2Y      1ØAM     *      TT    1ØA,1  ,2,37G,  37N,HLT,ØØØ$
 BPX        37G
 BPX         37N
 BPX        HLT
ØØØ
```

The errors in the above example are as follows:

Number 1. The first parameter value supplied didn't have enough characters to cover its parameter tag. (In this case "ITEM"). If the parameter tag is four characters in length, then the value supplied must be at least four characters long so that it will cover it. (It is unlikely that the programmer expected to come up with the tag 1ØAM.)

Number 2. The second parameter value supplied was followed with a space ("1"). This space erased the initial "T" of the TTB instruction.

Number 3. The third parameter value supplied ("2") didn't have enough characters to cover its parameter tag ("YY").

Number 4. The fourth parameter value for parameter tag G was left out. As a result Translator considered that:

37N - was the parameter value for 5th tag (G)
HLT- was the parameter value for 6th tag (H)
ØØØ - was the parameter value for 7th tag (X)

No parameter value given for 8th tag (J). Translator would insert columns after the $ (in this case, spaces).

A COMAND pseudo instruction generates a sequence of COSEAL-format Q-7 instructions which will:

1. SET (set) expression 1 to the value specified by expression 2.

2. TST (test) expression 1 for the value specified by expression 2, and if found branch to the location specified by expression 3. Expression 3 must be a DDL (digit, digit, letter) location tag.

3. SCH (search) expression 1 for the value specified by expression 2 and if found branch to the location specified by expression 3. Expression 3 must be a DDL (digit, digit, letter) location tag. Expression 1 must be indexed, and the search instruction will generate a conditional BPX to search through expression 1 on the basis of the cited index register. The programmer must have previously loaded the cited index register.

Remarks Format

1. Op Code Cols. 25-27

   The requested function (SET-TST-SCH) will occupy Columns 25-27. This is in accordance with COSEAL format.

2. Parameter Cols. 3Ø-8Ø

   a) Expression length is variable, and will occupy from 1 column minimum through 16 columns maximum.

   b) Each expression must be separated by one or more blank columns.

   c) The first expression (Expression 1) must begin in Column 30.

   d) A period or $ must immediately follow the final expression (Expression 2) for a SET function -- Expression 3 for a TST-SCH function.

The Five Expression Types

Expressions 1 and 2 may be of one of the following 5 varieties:

Constant (K)

A decimal value is specified by a + or - sign, followed immediately by 1 (min.) decimal digit through 5 (max.) decimal digits.

e.g., 1) +Ø
2) -Ø
3) -249
4) +5896
5) -32767

An octal value is specified by 1 (min.) octal digit through 6 (max.) octal digits.

e.g., 1) ø
2) 177777
3) 5246
4) 37
5) 431

NOTE: A constant (K) expression must not be indexed and/or incremented, nor may it designate any bit positions. If the K value is in error such as a (9) in an octal value the director will be left as (ø) without an error message bringing this to your attention.

Item (I)

The item specified may be indexed and/or incremented. (Don't refer to bit positions):

If indexed, a digit followed by a comma (the index register specified) must immediately precede the item.

If incremented, the item is immediately followed by a + or - sign and 1 (min.) decimal digit through 3 (max.) decimal digits (no leading zeros are necessary).

Do not refer to bit positions within an item.

e.g., 1) 2,TIDY+4
2) 5,TPOS+999
3) 4,MVEL-8
4) NAUK+29
5) NACJ
6) 1,TVEL

Location (L) in Expressions 1 or 2

The location specified may be indexed and/or incremented. It may also designate first and last bit positions (if the other expression, 1 or 2, is an item then one can't designate first and last bit positions of a smaller size than the item):

If indexed, a digit followed by a comma (the index register specified) must immediately precede the location.

If incremented, the location is immediately followed by a + or - sign and 1 (min.) decimal digit through 3 (max.) decimal digits (no leading zeros are necessary).

If first and last bit positions are designated, a comma immediately follows the location - or the final increment digit, if incremented. If the other expression, 1 or 2, is an item then bit positions can't be designated which are smaller than the item size.

Immediately following the comma, that part of the expression which designates bit positions may take one of two forms:

a. It may be composed of 4 letters which represent a legal compool item or,

b. It may be composed of 1 or 2 decimal digits (first bit position) followed immediately by a slash (/) which is immediately followed by 1 or 2 decimal digits (last bit position). No leading zeros are necessary. The location, itself, may be a tag (internal, table, or program) or in absolute form (octal or decimal). An absolute location cannot be distinguished from a constant; therefore:

The expression must be indexed, and/or,

The expression must designate bit positions.

No deviation from these requirements is permitted; however, (a) and (b) above do not apply to a branchpoint (Expression 3).

e.g.,
1) 1,1ØA,Ø/4
2) 1,1ØA,TIDY
3) TBLØ+19,16/31
4) TBLØ+19,NNML
5) 5,TPYØ+128,Ø/15
6) 5,TPYØ+128,STTN
7) SCX
8) 2,15ØØ,15/19
9) +512,SCAL
10) +512,22/31
11) 1,1ØØØØ
12) 1ØØØØ,8/15
13) 1ØØØØ,KUHK
14) 1,+125

Accumulator (A)

The accumulator is specified by the letter "A", and occupies 1 column, unless it designates bit positions.

If first and last bit positions are designated, a comma immediately follows the "A". If the other expression, 1 or 2, is an item then one can't use bit positions of smaller than the item size.

Immediately following the comma, that part of the expression which designates bit positions may take one of two forms:

a) It may be composed of 4 letters which represent a legal Compool item, or,

b) It may be composed of 1 or 2 decimal digits (first bit position) followed by a slash (/) which is immediately followed by 1 or 2 decimal digits (last bit position). No leading zeros are necessary.

e.g.,
1) A
2) A,Ø/Ø
3) A,HSAK
4) A,19/31
5) A,TRCL
6) A,Ø/5
7) A,FSFL
8) A,18/18

Index Register (X)

An index register is specified by the letter "X" followed immediately by a digit designating which index register is required.

This expression always occupies 2 columns.

e.g.,
1) X1
2) X2
3) X3
4) X4
5) X5

General Remarks

COMAND does not check for size of a value. If Item 1 is to be set with Item 2, Translator will obtain Item 2 and fit as many bits as will go into Item 1. If Item 2 is to be set into bit positions of a location the overall size of Item 2 will be used:

TST I K L$	If the constant (K) is too large for the Item (1) then just those bits within the Item size are tested.
SET I_1 I_2 $	If I_2 is larger than I_1, only those bits which can be placed within I will be deposited. A good rule of thumb here is that the 2nd expression is cycled to the position of the first expression and deposited using mask of 1st expression.
TST X2 K L$	JNKØ register is used to store contents of index registers in your program.
TST 1ØA,Ø/4 L L $	First expression here is cycled to position 31. (X, L, and A are considered as being positioned at bit 31).
TST I L L $	Cycles the 2nd expression to the position of the 1st expression and uses most of the item.
TST L A,5/9 L $	The 2nd expression is cycled to position 31 of the first expression and most of the 1st expression is used (Ø-31) therefore results will be a CMF instruction without any extraction.

General Rule	2nd expression will be cycled to position of the 1st expression. Mask of 1st expression will be used (unless 1st expression is not an item and 2nd expression is - then mask of item will be used).
TST I X L $	Position X to I and use UNKØ to store index register. In testing an item if the value K fills the item a (CSUØ) will result in order to save an RC word.

A location tag can be used with a COMAND pseudo instruction.

Example: 1ØA SET 2,TSTS -Ø $

It is also possible to use comments following COMAND expressions. (Up to Col. 67).

Example: 1ØA SET 2,TSTS -Ø $ TSTS NOT FINISHED.

Translator will move the COMAND pseudo instruction, COMAND expressions, and programmer comments into the comments field on final listing. If there isn't enough room on one line part of the comments will be placed on the next line (in comments field).

CHAPTER 12

MISCELLANEOUS INSTRUCTIONS AND COMPUTER INSTRUCTIONS

MISCELLANEOUS PSEUDO INSTRUCTIONS

HOL Instruction

The HOL (Hollerith) instruction generates inverted six bit Hollerith data words at the point in the program where the HOL pseudo instruction appears. The five alphanumeric characters appearing in columns 36 through 4Ø will be translated into inverted six bit Hollerith. Only six special characters may be used: the dollar sign, the plus sign, the minus sign, the comma, the asterisk and the period.

If one coded HOL ABCDE, then COSEAL would generate a register at that point in the program which contained 153514, 1713Ø4 for the left and right half-words respectively. If one looks up the inverted six bit Hollerith for ABCDE, inserts the code for "E" in bits Ø through 5 of the computer word, the code for "D" in bits 6 through 11, etc., one will arrive at the above octal contents for that memory location in the program.

PGM Instruction

The PGM (program) pseudo instruction will generate, in the register in which the PGM instruction appears, the 6-bit inverted Hollerith ident of the program as it appears in the IDT card. The format will not be the same as that of the HOL instruction, but will separate the program ident from the program mod. The format is as follows:

Bits 0-17 will contain the 6-bit (inverted) Hollerith coding for the three character ident (reading right to left).

Bits 18-29 will contain the 6-bit (inverted) Hollerith coding for the two character mod (reading from right to left).

For example, if the IDT card of the program was punched:

IDT ABC DE

the PGM instruction would generate the Hollerith code in this order:

CBAED bits 30 and 31 will contain 00.

PGM cannot be used in or with an RC word.

WRD Instruction

The WRD (word) pseudo instruction is used with two symbolic location tags. The address of the tag in the RC field is placed in the left half word and the address of the tag in the director field is placed in the right half word of the program location at which WRD appeared. Only 16 bit addresses will be handed correctly.

MISCELLANEOUS Q-7 INSTRUCTIONS

Absolute Value Instructions

The CAM (Clear and Add Magnitude) instruction replaces the current contents of the accumulators with the absolute value of the contents of the specified memory location. Memory and the B registers are unchanged.

DIM Instruction

The DIM (Difference in Magnitude) instruction performs two primary actions:

(1) A copy of the contents of the accumulators replaces the current contents of the B registers.

(2) The absolute value of the contents of the specified memory location is subtracted from the absolute value of the contents of the accumulators. Results are left in the accumulators, and the content of memory is unchanged.

The actions of the above two instructions are indicative of one of the ways in which the Q-7 shows itself to be other than a general purpose computer. Because air defense is intimately concerned with coordinate manipulations, and because the programmer is often concerned with differences in length between coordinates regardless of resultant direction, these instructions were built into the repertoire of the Q-7. It is obvious, however, that there are many instances in general mathematical manipulation where these instructions could be utilized.

CAC Instruction

The CAC (Clear and Add the Clock) instructions is one of only two instructions by which one can access or affect the real time clock register. The other instruction, PER14, is used for specialized test purposes and will not be considered.

First, we will discuss the configuration of the clock register itself. Essentially, it is a 16 bit binary adder. It is pulsed 32 times per second, each time the computer adding "one" to its least significant bit. Note, that all of the bits in the clock register are magnitude bits; there is no initial sign bit. Since the least significant bit position receives a "carry one" pulse 32 times per second, we may say that the scaling of the clock register is 2^{-5}, in units of seconds.

The CAC instruction initially clears both accumulators. It then brings into the right accumulator a copy of the current contents of the clock register. It should be noted that the most significant bit position of the clock register will be brought into RS of the accumulator. Manipulations upon this data word now in the right accumulator must be made carefully since most Q-7 instructions will manipulate the right accumulator's most significant bit as a sign bit.

From the above discussion, one can deduce that the contents of the clock register bear no necessary relationship with time in the outside world. However, if the computer is told, at any one particular time, what time it is on an external clock, it can

then measure differences in elapsed time between that time and any future time. That is, if the computer operator indicates to the Q-7, by external switch action, that the time is now Ø9ØØ hours ZULU, the computer would then take a reading of its clock register. If forty minutes later the program needed to ascertain what time it is in the outside world, it could again interrogate its clock register. By programmed action it could determine that forty minutes had elapsed since the last clock register reading. By adding this to the time initially given it by the computer operator, it would then know that the time in the outside world was Ø94Ø hours ZULU.

The director of the CAC instruction, while conveying no really essential information to the computer during the instruction's execution, is not totally meaningless. The director must contain a test memory address, for parity-check suppression. This test memory address may be obtained by coding a "T" in column 41. If, however, you mistakenly code anything that would yield other than a test memory address, program Translator will graciously change it to "$T+15_{1Ø}$".

PER Instruction

The PER (operate) instruction is used to effect many and varied actions, primarily on input/output peripheral equipment. When coding the PER instruction, the programmer inserts the appropriate octal code for the action desired into the aux columns. A list of these octal codes, and the actions they produce, is to be found on your programmer's card. In this volume we will draw attention to only four of the available PER codes.

There are on the face of the computer console four neons known as the condition lights. These lights have no intrinsic meaning to the operation of the computer itself, and only acquire such meaning as the programmer may ascribe to them. You will note that the PER codes for these lights are listed as Ø1 through Ø4, and that the action of the relevant PER is to turn the associated light on. For example, PER Ø3 will turn on condition light number 3. The programmer may use these lights as Boolean items, indications of various happenings meaningful to the computer operator, or in any other way that he might desire.

During the execution of the PER instruction, the computer does not interrogate its director. Therefore, the right half word may be said to be "meaningless," and may be used for any purposes the programmer might desire.

BSN Instruction

The BSN (Branch on Sense) instruction tests for various conditions to be true of input/output equipment. The octal codes for the various conditions to be tested are listed on your programmer's card, and these octal codes may be inserted directly into the aux columns. If, during program execution, the computer finds the indicated condition to be <u>true</u>, a branch will be effected to the location specified by the director portion of the instruction. As with the case of the PER instruction, we will consider only a few of the many BSN codes in this volume.

BSN codes Ø1 through Ø4 test the associated condition light and effect a program branch to the specified memory location if the light is on. You will note that if the light

happens to be on, the asterisk on the programmer's card beside the octal code indicates that this is one of the many BSN instructions which turns the associated piece of equipment off, in this case the tested condition light. If the light to be tested is already off, it will remain off and we will "fall through" the BSN instruction.

In addition to the above-noted condition lights, there are on the face of the duplex maintenance console four toggle switches known as the "sense switches." Once again, these switches have no meaning in and of themselves to the operation of the computer; they only acquire such meaning as the programmer may give to them. They are customarily used as a means by which the computer operator may effect changes in program action. BSN codes 21 through 24 sense the setting of these switches. If the associated switch is on (that is, depressed), a branch will be made to the specified memory location. Otherwise, we will "fall through" the branch instruction. You will note that there is no asterisk beside the octal codes for these BSN's. It should be obvious that the computer cannot reach out of itself and raise the switch to an off position.

TEST MEMORY

Test Memory provides storage for a relatively small number of words for various special purposes. (The primary use made of test memory is in testing and loading core-memory programs.) These words have addresses, as do words in core memory. Test Memory addresses range from 3.7776Ø to 3.77777 (a total of $16_{1Ø}$ addresses). Access time to words at these addresses is the same as core-memory access time - that is, 6 microseconds.

There are three types of test-memory registers that can be assigned to a given test-memory address. In order to understand the assignment of a specific type of test-memory register to a specific test memory address, it is important to distinguish between the various types of registers.

Test-Memory Registers

As you recall, test memory is made up of two toggle-switch registers, 16 plugboard registers, and one flip-flop register. Their characteristics are described below.

a. The two-toggle-switch registers are referred to as the A and B switches. There are 32 toggle switches for each register, and all are located on the Maintenance Console. The switches are in two horizontal rows, and the rows themselves are divided into equal halves. The block diagram in Figure 12-1 illustrates the layout of the switch registers on the Maintenance Console.

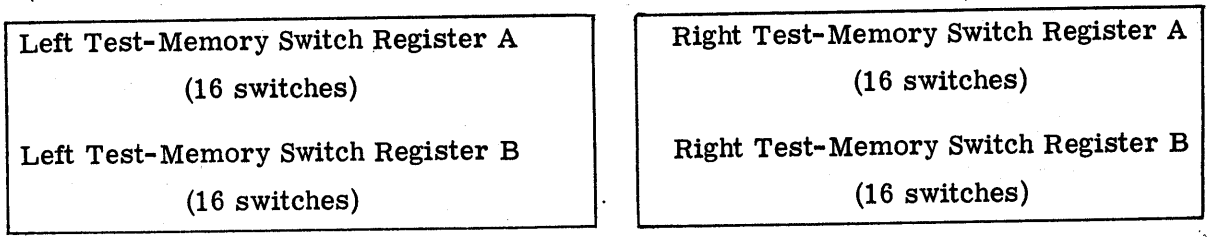


Figure 12-1. Block Diagram of Test-Memory Toggle-Switch Registers

Each switch corresponds to a bit-position in a binary word. Thus, the switches in each register can represent Left Sign through L15 and Right Sign through R15 of a computer word. When a toggle switch has been pressed down, a binary 1 has been placed in that bit-position of the test-memory switch register; when a toggle switch is up, a binary Ø is in that bit-position. The switches can be manually positioned by an operator at the Maintenance Console.

b. The contents of the 16 plugboard registers are set by inserting plug connections into a portable plugboard. By placing these plugs in appropriate holes (called "hubs") in the plugboard, binary 1's can be placed into the 32 bit-positions of each of the 16 plugboard registers. Once set up, the plugboard is inserted in the receptacle provided for it in the computer. When it is necessary to change the contents of test memory, the plugboard can be removed and another plugboard inserted in its place. The removed plugboard can be stored for future use without disturbing the plugged connections on it. Thus, with a number of plugboards, the contents of the plugboard registers of test memory can be changed rapidly and conveniently.

c. The Test-Memory flip-flop register is called the test register, or the live register, or register L. It is the only register in test memory that is functionally similar to a core-memory register, being the only register in test memory that can be written (stored into) as well as read. The live register is represented by a row of 32 lights on the Maintenance Console.

Test Memory Addressing

The assignment of a specific test-memory register to a specific address is controlled by two interrelated factors:

a. The positions of plugs in hubs in the test-memory plugboard.

b. The position of the test-memory UNASSIGN-ASSIGN switch on the Maintenance Console.

Four additional hubs are to the left of each of the 16 registers on the test-memory plugboard. These extra hubs are labeled:

A to designate switch register A
B to designate switch register B
L to designate the live register
P to designate a plugboard register

The 16 test-memory plugboard registers are numbered in octal from Ø to 17. Thus, the entire plugboard can be functionally diagrammed in Figure 12-2.

Each A, B, L, or P hub corresponds to the number aligned with it on the left. If a plug is inserted into any A hub, the contents of switch register A are assigned to the address specified by the number on the left. For example, assume the darkened holes in Figure 12-2 represent plugs. Since there is a plug in the A hub in line 6, the contents of switch register A have been assigned to test-memory register 6 (or address 3.77766). If a CAD 3.77766 instruction is now executed, the accumulators will receive the 32-bit binary number that is represented by the up-and-down switch positions of switch register A.

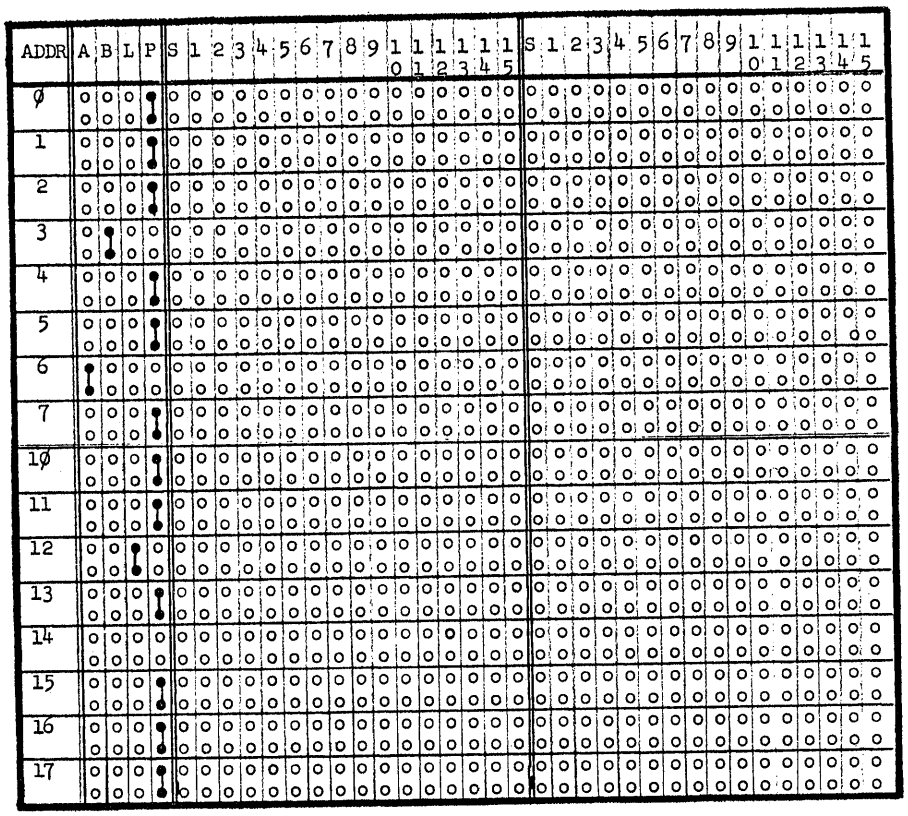


Figure 12-2. Function Diagram of Test-Memory Plugboard

If a plug is inserted into any B hub, the contents of switch register B are assigned to the address specified by the number to the left of that particular hub. Referring to Figure 12-2, the contents of switch register B have been assigned to test-memory address 3 (or address 3.77763). If all of the switches in switch register B are down, then a SUB 3.77763 instruction will subtract the octal number 1.77777, 1.77777 from the contents of the accumulator.

By plugging the L hub in a particular line, the live register is assigned the address of that line. In Figure 12-2, the address of the live register is 3.77772. If an ADD 3.77772 instruction is executed, the 32-bit number in the live register will be added to the accumulator's contents; if this 32-bit number consists of all 1's, then the value 1.77777, 1.77777 will be added to the accumulator's contents.

> NOTE: If any Test Memory address is in the address half of any store class instruction, the storage portion of the operation will be performed on the live register. Referring to Figure 12-2, for example, the execution of an FST 3.77765 instruction would full store a word into the live register and not into 3.77765. The contents of switch or plugboard registers in test-memory cannot be changed by any computer instruction; they must be changed manually, by the use of the toggle switches or plugs.

Each P hub corresponds to the number aligned with it on the left. Each P hub also corresponds to the plugboard register aligned with it on the right. By plugging a P hub in a particular line, the plugboard register in that line is assigned the address of that line. In Figure 12-2, for example, the P hub is plugged in line ø; test-memory address ø (or 3.7776ø) is therefore assigned to this plugboard register. Further, if this plugboard register is plugged in the leftmost 16 pluggable positions, and not plugged in the rightmost 16 pluggable positions, then an ADD 3.7776ø instruction will add the octal number 1.77777, ø.øøøøø to the accumulator.

In Figure 12-2, there are P plugs in other lines besides the 0 line. Wherever there is a P plug, the plugboard register in that line is assigned the address of that line.

If a Test Memory address is not assigned (i.e., if there is no A, B, L or P plug in a line), then referral to that address is as though a register containing +ø+ø were being referenced.

In Figure 12-2, for example, address 14 is not assigned. If a BPX 3.77774 instruction is executed, the computer will halt because of the ø's obtained by referencing location 3.77774 (the octal code for HLT is øøø).

By plugging two or more of the A, B, L, P hubs relative to a given address, any reference to that address will actually be a reference to all registers which are thus associated with that address. The resulting word transferred to the memory-buffer register is the logical sum of the contents of all the registers associated with this address. There will be a 1 in the bit positions where a 1 appears in any of the words, and ø where all words contain a ø. Using two octal half words to illustrate this, if the half words are ø.ø1ø1ø and ø.øø11ø, the resulting half word in the memory-buffer register is ø.ø111ø. When using one address for two or more registers, the operator must be aware that these registers will be effective at all times.

However, it is correct to assign two or more addresses for the selection of one register. For example, plugs may be inserted into the A hubs in test-memory addresses 6 and 7. A CAD 3.77766 instruction would then have the same effect as CAD 3.77767 and vice-versa; the contents of switch register A being placed in the accumulator by either instruction. Similarly, two or more B hubs or two or more L hubs may be plugged at two or more different addresses.

All of the above statements hold true only when the test-memory assignment switch is in the UNASSIGN position.

When the test-memory assignment switch is in the ASSIGN position, there are the three following important exceptions:

Address 3.7776ø is assigned to switch register A.
Address 3.77761 is assigned to switch register B.
Address 3.77777 is assigned to the live register.

(The A, B, L and P plugging corresponding to these three addresses is overridden.)

Addresses 3.77762 through 3.77776 remained assigned as plugged, (A, B, L or P).

Start From Test Memory Pushbutton

When the Central Computer System is halted, pushing the START FROM TEST MEMORY pushbutton starts the computer with the instruction located at the first test-memory register (address 3.77760). Thus, a program contained in test memory may be executed. This short program may direct the loading of a program from some other source. Or, the 16-word test-memory program may be used to test a computer program.

The START FROM TEST MEMORY pushbutton can also be used to enter another program at a selected point. For example, if a program has halted because of a programming error, and the operator wants to branch elsewhere in the program, the following procedure may be used:

a. Put the UNASSIGN-ASSIGN switch in the ASSIGN position.

b. Press the toggle switches of switch register A so that the left half-word of the register contains a BPX instruction, and the seventeen bit right half-word contains the address to be branched to.

c. Press the START FROM TEST MEMORY pushbutton.

This will cause the execution of the BPX instruction, and the desired branch will be accomplished.

OVERFLOW AND INTERRUPT PROGRAMMING

When overflow occurs in either accumulator, there are various things which can happen. Many of these occurrences depend upon the setting of bits L13 through L15 of the instruction which caused the overflow. In the Q-7 there are twelve instructions which can cause overflow. They are:

SLR	CDF	LAD
CDM	ADD	SUB
CDR	TAD	TSU
CDL	ADB	AOR

Overflow is one of the serious error conditions which can occur in the Q-7 and for which the computer can take very substantive remedial action through the use of interrupt programming methods. These actions are of two sorts:

(1) If bit L13 of the instruction causing the overflow is set to a zero, one or both of the two overflow indicators attached to each accumulator will be set. If, for instance, the execution of an ADD instruction induces overflow in the right accumulator, and if bit L13 of the instruction has been set to zero, the Right Overflow flip-flop will be set. This flip-flop could be sensed by a subsequent BSN13 instruction ay any later point in the program. Indeed, the flip-flop will remain on until it is so sensed. The flip-flop will remain on even if later additions are made, whether or not they produce overflow, until the BSN13 instruction is executed. In addition to setting the associated sensible flip-flop

(S), the computer will also sound an audible alarm which closely resembles a very loud foghorn. Needless to say, the computer will gain the operator's attention. This alarm will only be sounded if bit L13 is a zero.

(2) The computer can also effect an automatic machine-initiated branch to an error recovery routine. Such actions will only be taken, however, if many conditions are true. In order to understand these conditions, we will have to discuss three other parts of the Q-7 hardware.

Bits L14 and L15

Any branches to error recovery routines or program stops caused by accumulator overflow are controlled by these bits. If any of the subsequently detailed actions are to occur, one or both of these bits must be set. L14 is for the left accumulator and L15 is for the right. If overflow occurs in the right accumulator and the programmer wishes the below-detailed actions to occur, bit L15 of the instruction causing the overflow must be set to a "1". The same holds true for bit L14.

Auto-Branch Control Flip-Flop

No alarm branches or halts will occur unless this flip-flop is set. When the computer operator initiates a program, he usually first depresses the "MASTER RESET" or "RESET FLIP-FLOPS" push buttons. These switches set this flip-flop. One can change the setting of this flip-flop with the PER15 and PER16 instructions. Clearing this flip-flop by programmed action will prevent all alarm branches or halts from any cause whatsoever. Many of the other possible causes of interrupt program action will be detailed in other volumes.

Overflow Control Switch

There is a row of switches on the front of the duplex maintenance console, each one of which relates to one of the several conditions which are important enough to necessitate machine-initiated error recovery. One of these switches is known as the "overflow" switch and, if it is on, the computer will then be sensitive to conditions of overflow.

Let us now take an example of the actions which the computer will perform in the event of overflow:

(1) If L13 is set to zero and overflow occurs in the right accumulator, the right overflow flip-flop will be set and the audible alarm will be sounded. That flip-flop could, as we have noted above, be tested by a subsequent BSN13.

(2) If L15 is set to one, and if

(3) the auto-branch control flip-flop is set, and if

(4) the overflow control switch is active (that is, depressed),

(5) the computer will now interrogate another switch on the front of the duplex maintenance console.

> This is the "STOP-BRANCH" action. If it is in the stop position, the computer will immediately halt. If it is in the branch position, and immediate branch to location 37777Ø will occur. The address of the instruction which would have been next executed is left in the right A registers. At the above-referenced test memory location, the programmer would probably have plugged an STA instruction to save the return address and then a branch instruction to take the program to an error recovery routine.

At the time when the Q-7 was initially designed, interrupt programming was practically unknown. As you can see from the above discussion, a relatively sophisticated system has been built into the Q-7 to accomplish automatic error recovery.

Coding for Overflow Control

In the coding for those instructions which can cause overflow, we have not thus far utilized column 29. This column may be coded as follows for these instructions.

(1) Blank: assembled as an octal 3 in bits L13 through L15. Action will be to set the "BSNable" indicators, and allow branching and audible alarm sounding.

(2) "L": assembled as octal 1; set indicators, suppress left alarm and branching.

(3) "R": assembled as octal 2; set indicators, suppress right alarm and branching.

(4) "B": assembled as octal Ø; set indicators, suppress both alarm and branching.

(5) "E": assembled as octal 4; suppress all overflow alarms and indications; the "E" essentially stands for "everything".

"ILLEGAL INSTRUCTIONS"

Any instruction which is not a legal code (in L4 through L1Ø) for the Q-7 may nonetheless, through a fluke, be decoded as an instruction. If the programmer committed an error and branched into the data area, this possibility is quite likely. An illegal instruction code in any given class (14 through L6) will act exactly like any other illegal instruction in that class. Some of the illegal instructions have been found to do useful jobs and have been given names (LAD, STZ and NOP). An octal code has been arbitrarily chosen for these instructions from among the illegals available in that class. This is so that COSEAL will know what to put into the binary output.

Ø -- Misc. Class	No operation.
1 -- Add Class	Adds the contents of the LHW of the specified memory location to the contents of the Left Accumulator. Given the location to the contents of the Left Accumulator. Given the name LAD (12Ø chosen arbitrarily).
2 -- Mult. Class	Will cause the computer to "hang-up". (Infinite arithmetic pause.)
3 -- Store Class	Clears the specified memory location. Given the name STZ (3ØØ chosen arbitrarily).
4 -- Shift Class	No operation for a period between 6.Ø and 35.5 μsecs.
5 -- Branch Class	Clears the A registers.
6 -- I/O Class	Performs no operation, but will wait until the I/O interlock is cleared before going to the next instruction.
7 -- Reset Class	No operation. Given the name NOP (7ØØ chosen arbitrarily).

MISCELLANEOUS COSEAL INFORMATION

Program Operating Time

The amount of Q-7 computer time that is required to assemble a program is dependent, to a significant extent, upon what input and output options are requested. A permutation of the options possible, with corresponding operating times, is beyond the scope of this document. Listed below are approximate times required for input-output. A basic assembly time of 5Ø cards/sec. should be added to the times given below to determine total operating time.

1. Input

Combined Tags	2ØØ cards/sec.
Prestore	5Ø cards/sec.
Card	15Ø cards/min.
Symbolic Corrector Cards	15Ø cards/min.

2. Output

Combined Tape	2ØØ cards/sec.
Binary Tape	3ØØØ cards/sec.
Binary Cards	1ØØ cards/min.
DLO (Symbolic listing)	5Ø cards/sec.
Direct (Symbolic listing)	75 cards/min.

Error Printouts

Errors not causing the program to halt. Errors detected in the assembled program. All error lines of the listings are logged direct whatever the form of symbolic output requested.

1. "INDETR LHW": Indeterminate left-hand word is generally caused by either a special symbol in the instruction left-half word field, or a decimal integer such as 8 or 9 in an octal constant, or an op code punched in the wrong columns of the instruction left-half word field.
2. "INDETR RHW": Indeterminate right-half word is generally caused by a special symbol as an asterisk in the director field.
3. "UNDEFINED": This appears when Program Translator cannot find the symbolic tag contained in the director field in either the tag table, or the compool override table, and for Compool dependent programs, cannot find it in the Compool either.
4. "DUPL TAG": Duplicate tag appears on the line which contains the duplicate in the director. The address of the first duplicate is used for all the others.
5. "ILL USE RC": Illegal use of the RC word. This is caused by the use of an RC word with an instruction left-half word which does not take an RC word or an illegal instruction in the left half RC word field, such as DIT or CPO.
6. "INDETR RC": Indeterminate RC. This is caused by the same sort of error mentioned under indeterminate left-half word except the error would appear in the left-half word in the RC field.
7. "EXCESS RC": Execessive number of RC words. At present this will be caused by a program having more than 75Ø different RC words in it.
8. "EXCESSIVE CPO CARD": This is logged when the number of CPO cards exceeds 3ØØ. Assembly continues but all subsequent CPO card information is lost.
9. "XXXXX ILLEG TAG": Generally caused by an illegal symbol in the tag field of the instruction card. The X's will contain the illegal card.
10. "PLI ERROR - CCA COMPOOL NOT USED": The compool used was not assembled using the appropriate system control indicator. The "PLI" record will not be produced.
11. "PLI ERROR - TABLE NOT IN COMPOOL": The compool used does not contain the table PLI. The "PLI" record will not be produced.
12. "PLI ERROR - PROG NOT IN COMPOOL": The program being assembled is not defined in the compool used. The "PLI" record will not be produced.

Errors causing the program to halt. All error lines of listings are logged direct.

1. "NO IDT CARD": If symbolic input is from cards, attach IDT card to deck, put deck in card reader, and press (CONTINUE". If input is from tape, nothing can be done.

2. "READY I/O UNIT Ø1": Card reader not ready. Ready card reader and continue. This error printout also means there may not be an END card on the deck being assembled in the card reader. In this case, add the remainder of the deck with END card and press "continue". If there is no END card on the tape, no recovery is possible.

3. "READY I/O UNIT 1X": Tape X not ready. Ready tape X and press program continue.

4. "READY I/O UNIT Ø2": Punch not ready. Ready card punch and press program continue.

5. "TOO MANY TAGS": This occurs if the program being assembled has over 5ØØØ tags. No recovery possible.

6. "LOAD SUBR LIB ON 11": If the COSEAL master is on unit 11 when the subroutine library tape is needed, load subroutine library on 11, and continue.

Information Printouts

1. For all program assemblies, every IDT, LOC, DRM, and END card is logged.

2. For DCA program assemblies, every CPO card, in addition to those listed above, is also logged.

3. For DCA programs, the number of spare registers computed from COMPOOL information, is converted to a signed decimal integer, and logged as follows:

"$\pm$. . . . . EQ NUMBER OF SPARES"

The number of spares will not be logged if the program's core location in the COMPOOL is overriden by a LOC card.

4. If assembly is from tape and the symbolic output requested is not Direct, the number of cards prestored will be logged. If in addition, Sense Switch #1 is down the card numbers will be sequence checked. A card numbered less than or equal to the one immediately preceding it will be logged:

"CARD NO. XXXXXX OUT OF SEQUENCE"

The card sequence check can be discontinued at any time by raising Sense Switch #1. Once stopped the check cannot be reinitiated. If the sequence check is carried to completion, the following message is logged.

"SEQ CHECK COMPLETE"

5. If a program to be assembled for drums has no drum field assigned via either the COMPOOL or a DRM card, the Translator will log:

"NO DRUM ASSIGNED. ASSEMBLY WILL BE FOR CORE"

If the ALL feature has been used, assembly of any program following the one in question will be for drums.

Limits of COSEAL's Translator

1. Input Cards: Unlimited
2. CPO Cards: Maximum, 3ØØ
3. Location Tags: Maximum, 5ØØØ
4. If a program is assembled for location Ø, a DDL (digit, digit, letter) location tag at the first binary instruction will be labeled undefined.
5. The total number of all cards in macro definitions may not exceed 5ØØ.

CHAPTER 13

TRANSLATOR'S CORRECTION CAPABILITIES AND OTHER COSEAL PROGRAMS

SYMBOLIC CORRECTIONS WITHIN PROGRAM TRANSLATOR

A program being assembled from a combined or prestored tape has the option of being updated with "Symbolic Correction Cards" of the same format as that used for Symbolic Correction Program during DCA cycling.

This function of Translator makes it possible to change or delete cards contained on the Combined or Prestored tape, also to insert new cards as needed. An additional feature allows for changing the comment field only of a card.

Explanation of Terminology

The following definitions apply to "Symbolic Corrections Cards" which comprise an update deck:

Update Code - The symbol entered in Col. 17 (C, I, or D) of a "Symbolic Correction Card" which specifies the particular update action to be performed.

Update Identifier - The Symbolic Tag (five characters or less) entered in Cols. 72-76 which references the region in the program, on the combined or prestored tape, where the update action is to be performed.

Identifier Symbol - The symbol ("A", "B", "T" or "+") entered in Col. 77 of a "Symbolic Correction Card" which specifies the method of counting from the "Update Identifier" in the program on the Combined or Prestored tape.

Identifier Increment - The decimal number n ($\emptyset < n \leq 99$) entered in Cols. 78-79, which together with the "Identifier Symbol" specifies the number of counts, from the "Update Identifier", that the update action is to be performed.

Card Packet - A group of Symbolic Corrector Cards (one or more) which are associated with one Update Code, Update Identifier, Identifier Symbol, and Identifier Increment.

Card Block - A group of Card Packets totaling 3ØØ cards or less, which represent the number of update cards that are read and operated on at one time.

Update Deck - The group of Card Blocks which comprise all "Symbolic Correction Cards" for one program (NO limit).

Preparation of Card Packet

The first Symbolic Corrector Card in a Card Packet contains control information which applies to all cards in the Packet. Following is a description of the control information in this first card.

Col. 17 (Update Code)

I - Insertion before or after the instruction whose location is defined in Cols. 72-79. All cards in the Packet will be inserted in sequence, following this same location.

C - Change card whose location is defined in Cols. 72-79. Subsequent cards within the packet will cause sequential cards in the program to be changed.

D - Delete the card whose location is defined in Cols. 72-79. If more than one consecutive card is to be deleted one of the two procedures described below may be followed:

a. Enter the total number (decimal) of cards in Cols. 18-2Ø (right adjusted).

R - Comment insertion. May be used to insert one or more Comment Cards (instruction field, columns 18-45, blank). R Cards may be used as part of a normal insertion or change packet, or may be used alone. The normal rules for Col. 72-79 control apply. The R must appear on each and every comment insertion card used, whether there is Control Information in 72-79 or not.

A provision is provided to delete a block of cards by specifying the beginning and ending locations symbolically. If it is elected to use this provision, the delete card format is as described below.

Col. 17	D
Cols. 18-2Ø	"TAG"
Cols. 36-45	Symbolic location tag, with position increment if needed, which defines the end of the block of deletions. NOTE: The card whose location is defined in these cols. WILL NOT be deleted.
Cols. 72-79	Update Identifier Symbol and Identifier Increment as previously described. NOTE: The card whose location is defined in these Cols. will be the first deletion.

The location tag and increment in Cols. 36-45 should be coded as per usual COSEAL coding.

Cols. 72-76 (Update Identifier)

Symbolic location tag which specifies the program region where the update action is to occur.

Col. 77 (Identifier Symbol)

B - Insertion if BEFORE the specified instruction. Interpreted as "+" with an Update Code of "C" "D".

A - Insertion is AFTER the specified instruction. Interpreted as "+" with an Update Code of "C" "D".

T - With all Update codes, used to count all cards, not just which produce a binary instruction. Must be used to change assembly cards such as IDT, DRM, LOC, CPO, etc., if the card does not have a symbolic location tag.

"+" Use with update code "C" and "D" to increment the tagged location.

Cols. 78-79 (Identifier Increment)

A decimal increment (always positive) used in conjunction with the Update Identifier and Identifier Symbol to indicate the proper location within the program where the update action is to be performed.

Final Address Insertions

If the Identifier Increment is equal to the letters FA, the Card Packet is inserted immediately preceding the END card of the program. Card Packets for Final Address insertion may occur at any place within the Update Deck. A maximum of 300 Final Address Cards may be contained in an Update Deck.

Update of Comment Field Only

It is possible to change the Comment Field of an instruction card (Cols. 46-71) without changing the instruction field (Cols. 18-45). A change of this type must constitute a separate Card Packet. The first card of the Packet is coded as follows:

Col. 17	- Update code must be "C"
Cols. (25-27)	- "CMT"
Cols. (46-71)	- New Comment to be Inserted
Cols. (72-79)	- Coded as for a normal CHANGE card.

The second and subsequent cards in a packet need only contain the comment in Cols. (46-71).

Preparation of Card Block

Since the Card Block is only an arbitrary unit of 3ØØ cards, used by the Translator program, special concern need not be given to it by the programmer except when the update deck exceeds 3ØØ cards. Listed below are general rules which govern the preparation of the Card Block.

In determining what portion of an update deck constitutes a Block, consider the first 3ØØ cards as Card Block 1, the second 3ØØ cards as Card Block 2, etc.

(NOTE: FA cards, which may be placed anywhere in the update deck, should not be included in these counts.)

Within a Card Block (3ØØ cards group) Card Packets need NOT be arranged in the order of occurrence of the Update Identifiers in the program. However, all Update Identifiers contained in Card Block 1 MUST occur in the program prior to all Update Identifiers contained in Card Block 2, etc.

A Card Packet may be broken between two Card Blocks if the requirement above is satisfied.

Preparation of Update Deck

Card Packets are assembled into Card Blocks which in turn form the Update Deck. Preceding the first Card Block of the Update Deck MUST be an "IDT Card" containing the Ident. of the program to be updated.

Following the last Card Block of the Update Deck MUST be an "END Card". Below are the card formats:

IDT Card

Cols. 25-27 - "IDT"

Cols. 36-39 - Program Identification exactly as it appears on the Combined or Prestored Tape. The program Mod is optional and may be entered in Cols. 4Ø-41.

END Card

Col. 17 - "F", if this is the last Update Deck which follows one OPTION Card; otherwise, blank.

Cols. 25-27 - "END"

Sample Program Corrections

Figures 13-1, 13-2 and 13-3 illustrate symbolic corrections utilizing program Translator. The first of these figures shows a sample deck which has been assembled as shown. The second figure shows the corrector deck, and the last figure shows the updated program.

Error Printouts

Below are comments which are logged direct, along with the first card of a Card Packet, when an error is encountered while processing an Update Deck.

Symbolic Corrector Card - Error 1

More than one Card Packet which changes the program "IDT" card was encountered in the Update deck. The second and succeeding such Card Packets are skipped.

```
                                PROGRAM 3333 MOD 01          COMPOOL   04D 00   PAGE      1

          IDT                333301                                                0000.01*T00
          LOC                   100                                                0000.02*T01
          STAA               10F          000100  003410 000115                  0001.00*T02
        1XIN                      2       000101  017540 000002                  0002.00*T03
10A     1CAD               10D          000102  011000 000111                  0003.00*+00
        1FST               10C          000103  013240 000106                  0004.00*+01
          BPX              10F          000104  005100 000115                  0005.00*+02
10B       HLT                           000105  000000 000000                  0006.00*+00
10C            4                 1      000106  000004 000001                  0007.00*+00
          DIT                       2    000110  000004 000001                  0009.00*+02
10D     TSKP                        3    000111                                 0009.01*T00
        1CAD               10G          000114  011000 000000                  0010.00*+01      UNDEFINED
10F       BPX                           000115  005100 000000                  0011.00*+00
TBL1      CPO                       0                                          0011.01*T00
          END                   20000                                          0011.02*T01
                                                                    +   15  EQ NUMBER OF CARDS
```

Figure 13-1. Sample Program To Be Corrected

```
                    CORRECTOR CARDS FOR  PROGRAM                PAGE      1

C         IDT          333302                                        T
C         LOC              20                                        T01
C       1XIN                1                                        T03
I         FCL              16                                 10A    +01
D001                                                         10B
I         HLT               1                                       FA
          HLT               2
C10G    TCPO            1000                                 TBL1 T
F         END
```

Figure 13-2. Corrector Cards For Program In Figure 13-1

```
                                  PROGRAM 3333 MOD 02          COMPOOL  04D 00   PAGE      1

                 IDT      333302
                 LOC          20
                 STAA     10F          000020  003410 000035                    0000.01*T00
                                                                                0000.02*T01
                1XIN            1     000021  017540 000001                    0001.00*T02
          10A   1CAD     10D          000022  011000 000031                    0002.00*T03
                 FCL          16     000023  004700 000020                    0003.00*+00
                1FST     10C          000024  013240 000026                    0004.00*+01
                 BPX      10F         000025  005100 000035                    0005.00*+02
          10C          4              000026  000004 000001                    0006.00*+03
                 DIT             1    000030  000004 000001                    0007.00*+00
          10D   TSKP             2     000031                                  0009.00*+02
                1CAD     10G          000034  011000 001000                    0009.01*T00
          10F    BPX                  000035  005100 000000                    0010.00*+01
          10G   TCPO          1000                                             0011.00*+00
                 HLT             1    000036  000000 000001                    0011.01*T00
                 HLT             2    000037  000000 000002                    0011.02*+01
                 END          20000                                            0012.00*+02
                                                                               0013.00*T03
                                                                    +    15  EQ NUMBER OF CARDS
```

Figure 13-3. Corrected Program Originally Shown In Figure 13-1, After Corrections of Figure 13-2

Symbolic Corrector Card - Error 2

On the first card of a Card Packet, Column 17 is illegal (Not "C", "D", or "I"). The Card Packet is skipped.

Symbolic Corrector Card - Error 3

The Identifier Increment (Columns 78-79) is too large; the increment counts past the next location tag in the program. The Card Packet is skipped.

Symbolic Corrector Card - Error 4

The Update Identifier was not found on tape. The Card Packet is skipped. This comment is logged when the program END Card is reached.

Symbolic Corrector Card - Error 5

This comment is logged after Error 4 above occurs in one block of a multiple Card Block Corrector Deck, and the "END" card for the Corrector Deck has not been read. The first card of each remaining Card Block is logged. All Corrector Cards, except FA insertions, (beginning with the first one logged as an Error 5) remaining in the Update deck are skipped.

Symbolic Corrector Card - Error 6

The Update Identifier together with Identifier Increment specify a location in the program which has been altered (changed or deleted) by a previous update action. The card packet is skipped.

Symbolic Corrector Card - Error 7

Corrector Card(s) without control information punched in Columns 72-79 follow a change "IDT" card packet or a Delete Card packet. The first such Corrector Card in a group is logged and all cards, until the next one with control information in Columns 72-79 is encountered, are skipped.

EXCESS FA Cards

More than 3ØØ Final Address Cards are in the Update deck. All FA cards over 3ØØ are skipped.

Special Points

When preparing symbolic corrector decks, the following special points should be considered:

(a) Only one "IDT" card is needed for a program, regardless of how many changes are made. A Card Packet is needed for each non-consecutive location affected, or for each change of control. For example, if it is desired to change five consecutive cards beginning at 1ØA, only one "C" card would be needed. The "C" card would contain the tag, 1ØA punched in Columns 72-79, and the first of the five new instructions coded in Columns 18-71. This would be followed immediately by the remaining four instruction cards. However, if a card is to be inserted before 1ØA + 5, another control card with an "I" in Column 17 and Columns 72-79 coded as previously described is needed.

(b) To change an "IDT" card, the following cards are necessary.

1) An "IDT" card like the one on tape. This must be the first card of any corrector deck.

2) A card with "C" in Column 16, "IDT" in Columns 25-27, the new Ident in Columns 36-41, and Columns 72-79 left blank with the exception of Column 77 = T. This new "IDT" card must always constitute a separate card packet.

In the same way, cards may be inserted, deleted, or changed before the first location tag by leaving columns 72-76 blank on the control card, but putting an increment in columns 78-79 and "T" in Column 77. The increment in this instance, is the number of cards from the "IDT" card.

(c) To change a program "END" card the Identifier Symbol must change to "T". The Update Identifier and Identifier Increment are coded as previously described.

(d) If a T is used in Column 77, to compute the increment for Columns 78-79 every card after the tag must be counted. This included any "ASSEMBLY" or "COMMENT" cards that may be in the region being updated.

(e) If a T is not used in Column 77, to compute the increment for Columns 78-79, only count those cards which produce a binary instruction upon assembly.

(f) The last Correction Deck to be processed should have the letter "F" punched in Column 17 of its Hollerith END card. This tells the Translator that all corrections are now finished.

(g) As a general rule, the letter "T" should be used, where possible, as an "Identifier Symbol" when incrementing from a tag in the program on tape. The rules for determining the "Identifier Increment" are simpler, in that all-cards are counter, thereby reducing the change for programmer error.

(h) Special care should be taken when using the option of deleting cards from one symbolic location to another. If an error is made in specifying the ending location (Columns 36-45) this could cause all cards in the program from the initial location (Columns 72-79) to the "END" card to be deleted.

OTHER COSEAL PROGRAMS

There exist in the COSEAL Utility system three programs designed as diagnostic aids during program assembly add execution. We will give below brief explanations of the actions of these programs and show illustrative listings.

TAREF/FLOA Program

A TAREF (tag reference) listing or FLOA (flow analysis) may be obtained at time of program assembly. If TAREF is requested, a symbolic listing (direct or DLO) will be produced showing all tags used by a program along with all references made to each tag within the program. This will be followed by an alphabetic listing of all of the tags used by the program together with their card numbers, provided the deck is sequenced.

If FLOA is requested, a symbolic listing (direct or DLO) will be produced showing all instructions within an object program together with all references made to each instruction. An alphabetic tag listing will be produced as with TAREF. The primary difference between TAREF and FLOA is that a flow analysis is made on the program listing itself as will later be illustrated by the appended examples.

References to all instructions, whether TAREF or FLOA, will be indicated by MB (modified by) for store class instructions, EF (entered from) for branch class instructions, and USED for miscellaneous instructions.

A careful perusal of Figure 13-4 will illustrate an assembly of a sample program and its TAREF and FLOA listings.

Memory Print Program

High speed memory print enables programmers or computer operators to dump Q-7 storage (core, drums or tape), at high speed. The program will dump storage media in octal-constant or instruction format, either direct or DLO.

Figure 13-5 shows an area of core dumped in instruction format, and Figure 13-6 shows the same area dumped in octal-constant format.

Interpretive Trap and Trace Program

This program is designed to execute any other program's instructions and print out the contents of selected machine registers after each instruction. Contiguous areas of core memory may also be dumped at specified points during the interpretive execution of the program being checked.

Figure 13-7 illustrates a typical interpretive execution and shows the machine registers which are always dumped.

```
                                        PROGRAM XYZ  MOD              COMPOOL  04D 00   PAGE      1

          IDT              XYZ                                                              0000.01*T00
TDR0      TCPO        90      100                                                           0000.02*T00
TPOS      ICPO04       9   TDR0                                                            0000.03*T00
02A       CAD              TPOS          000300  001000 000100                            0001.00*+00
          POS31            TPOS          000301  004700 000015                            0002.00*+01
          RST              10A           000302  003340 000310                            0003.00*+02
01A       AOR              02A           000303  003443 000300                            0004.00*+00
          CMR          0   TPOS  + 20     000304  000420 000311                            0005.00*+01
          BPX                1000        000305  005100 001000                            0006.00*+02
          AOR              02A   +  2     000306  003443 000302                            0007.00*+03
          BPX              02A           000307  005100 000300                            0008.00*+04
10A                                       000310  000000 000000                            0009.00*+00
          END                                                                              0009.01*T01
                                          000311  000000 000124

*** *** *** *** *** *** *** *** *** *** *** LAST PAGE *** *** *** *** *** *** *** *** ***

   TAREF                          IDT              XYZ

                TAG TABLE REFERENCE LISTING

 ARD NO.  TAG-INCR.   MODIFIED BY  ENTERED FROM  TAG-INCR.  CARD NO.

 001.00   02A            MB                      01A        0004.00
               +002      MB                      01A  +003  0007.00
                                    EF           01A  +004  0008.00


0004.00   01A


0009.00   10A             MB                     02A  +002  0003.00


    TPOS       US  02A        0001.00   US  02A  +001  0002.00
    TPOS +020  US  01A  +001  0005.00

                ALPHABETIZED TAG LISTING

    01A   0004.00
    02A   0001.00
    10A   0009.00


     FLOA                           IDT              XYZ
                                TDR0   TCPO        90      100
                                TPOS   ICPO04       9   TDR0
01A  +004                       02A    CAD              TPOS       MB 01A        MB 01A  +003
                                       POS31            TPOS
                                       RST              10A
                                01A    AOR              02A
                                       CMR          0   TPOS  + 20
                                       BPX                1000
                                       AOR              02A   +  2
                                       BPX              02A
                                10A                                MB 02A  +002
                                       END

    TPOS       US  02A        0001.00   US  02A  +001  0002.00
    TPOS +020  US  01A  +001  0005.00

                ALPHABETIZED TAG LISTING

    01A   0004.00
    02A   0001.00
    10A   0009.00
```

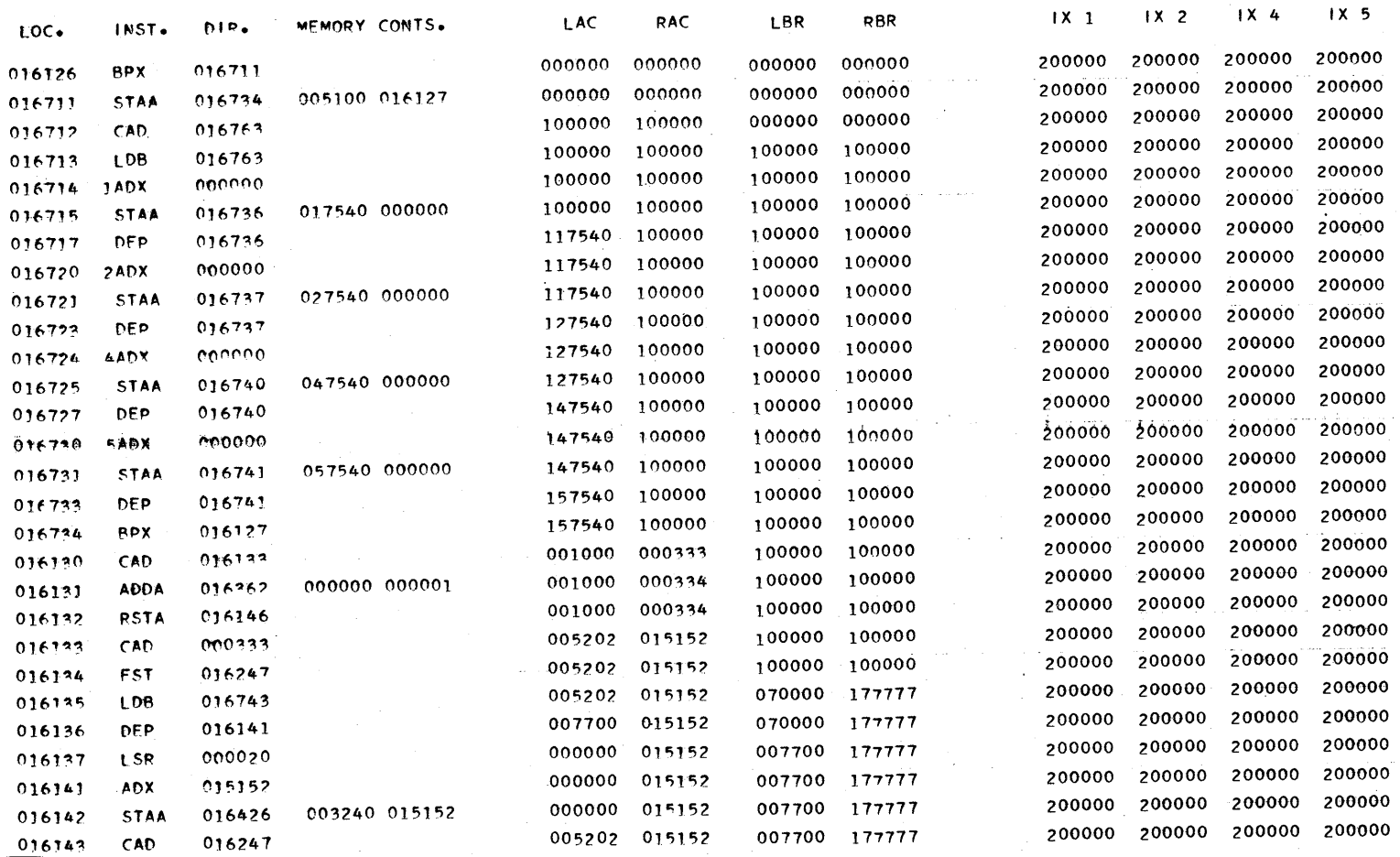
Figure 13-4. Program XYZ with its TAREF and FLOA Listings

```
CORE 017000 TO 017501 INSTRUCTIONS                    CON. LIGHTS ON ....   SENSE SWITCHES ON ....

017000   STAA  017111  STAA  017143  BPX   017064  HLT   000000  HLT   000000  HLT   000000  HLT   000000  HLT   000000
017060   HLT   000000  HLT   000000  HLT   000000  HLT   200000  FST   017735  CAD   017752  LDB   017752  1ADX  000000
017070   STAA  017740  1BPX  017073  DEP   017740  2ADX  000000  STAA  017741  2BPX  017077  DEP   017741  4ADX  000000
017100   STAA  017742  4BPX  017103  DEP   017742  5ADX  000000  STAA  017743  5BPX  017107  DEP   017743  STZ   017734
017110   BPX   017737  CAD   015240  FST   017732  LDB   017753  DEP   017116  CAD   017735  ADX   000000  STAA  017726
017120   STAA  017736  CAD   017726  BFZ   017211  1XIN  000013  CAD   017732  ETR   017754  1DIM  017216  RSR   000020
017130   BFZ   017144  1BPX01 017124  CAD   017734  BLM   017142  AORA  017143  AORA  017143  RSTA  017111  CAD   017063
017140   FST   017734  BPX   017110  BPX   017737  BPX   015240  FST   017734  1CAD  017216  RSTA  017207  CAD   017732
017150   ETR   017755  FCL   000020  RST   017730  ADD   017756  RST   017202  AORA  017143  BPX   017745  FST   017732
017160   ETR   017757  RST   017176  SUB   017756  RST   017175  FCL   000020  RST   017731  CAD   017732  ETR   017760
017170   FCL   000010  SUB   017756  FST   017727  BFZ   017214  CAD   017063  LSR   177776  FCL   000000  FST   017732
017200   CAD   017726  BPX   017745  FCL   000036  ETR   017732  FST   017733  AORA  017143  RSTA  017111  BPX   017324
017210   BPX   017110  CAD   017064  RSTA  017726  BPX   017123  STZ   017727  BPX   017174  NOP   017250  007040-017274
017220   007140-017276  007100-017307  007200-017314  007240-017324  007340-017366  NOP   017404  007300-017444  007440-017460
017230   007500-017457  007600-017477  STA   017243  1XIN  000004  CAM   017733  BFZ   017244  DCL   000020  CAD   017761
017240   DVD   017762  1FST  017056  1BPX01 017237  BPX   017252  CAD   017712  1FST  017056  CAD   017761  1BPX01 017236
017250   BPX   017510  BPX   017232  CAD   017056  BFZ   017262  BPX   017673  AORA  017252  BPX   017745  BLM   017271
017260   BFZ   017267  BPX   017254  BPX   017673  AORA  017252  BPX   017745  BLM   017271  BPX   017252  CAD   017712
017270   BPX   017254  CAD   017245  RSTA  017252  BPX   017110  BPX   017520  BPX   017251  CAD   017733  BPX   017531
017300   CAD   017055  BFZ   017305  BPX   017673  CAD   017056  BPX   017260  CAD   017712  BPX   017302  BPX   017520
017310   CAD   017733  BPX   017531  CAD   017252  BPX   017256  CAD   017730  FCL   000020  LDB   017763  DEP   017320
017320   SEL10 000000  LDC   017003  WRT   000030  BPX   017110  CAD   017736  BFZ   017327  BPX   017542  BPX   017552
017330   SEL03 000000  LDC   017643  WRT   000030  1XIN  000067  CAD   017764  RSTA  017614  CAD   017561  RSTA  017637
017340   SEL04 000000  LDC   017643  RDS   000030  CAD   017063  FST   017642  2BPX01 017626 1XIN  000027  CAD   017761
017350   1ADD  017643  1BPX01 017350  BFZ   017357  SEL03 000000  LDC   017643  WRT   000030  PER51 000040  CAD   017736
017360   BFZ   017110  1XIN  000027  1CAD  017033  1FST  017003  1BPX01 017362  BPX   017110  CAD   017731  SUB   017765
017370   BFZ   017400  ADD   017765  1XAC  000000  1BPX01 017374  CAD   017761  BPX   017673  1BPX01 017374  BPX   017110
017400   SEL04 000000  LDC   017003  RDS   000030  BPX   017110  STA   017443  CAD   017730  ADD   017756  RST   017430
017410   CAD   017730  SUB   017766  BRM   017417  FST   017732  AOR   017417  CAD   017732  BPX   017411  1XIN  000001
017420   CAD   017756  RST   017417  CAD   017731  2XAC  000000  CAD   017726  BPX   017745  1BPX01 017427  2BPX01 017430
017430   FCL   0C0036  FST   017732  BPX   017673  CAD   017732  FCL   000032  1BPX01 017442  AORA  017726  BPX   017745
017440   FCL   000036  1XIN  000004  2BPX01 017431  BPX   017551  CAD   017736  BFZ   017447  BPX   017542  BPX   017552
017450   SEL02 000000  LDC   017643  WRT   000030  BSN03 017455  BPX   017357  PER73 000000  BPX   017357  BPX   017510
017460   CAD   017755  BPX   017673  CAM   017733  1XIN  000003  MUL   017767  FST   017733  ASR   000013  BFZ   017475
017470   BPX   017673  CAD   017733  DSL   000004  1BPX01 017464  BPX   017110  CAD   017712  BPX   017470  CAD   017733
017500   BPX   017531
DUMP COMPLETE.
```

Figure 13-5. Core Dump In Instruction Format

```
CORE 017000 TO 017501 CONSTANTS                    CON. LIGHTS ON ....   SENSE SWITCHES ON ....

017000  003410-017111 003410-017143 005100-017064 000000-000000 000000-000000 000000-000000 000000-000000 000000-000000
017060  000000-000000 000000-000000 000000-000000 100000-000000 003240-017735 001000-017752 000300-017752 017700-000000
017070  003410-017740 015100-017073 003600-017740 027700-000000 003410-017741 025100-017077 003600-017741 047700-000000
017100  003410-017742 045100-017103 003600-017742 057700-000000 003410-017743 055100-017107 003600-017743 003000-017734
017110  005100-017737 001000-015240 003240-017732 000300-017753 003600-017116 001000-017735 007700-000000 003410-017726
017120  003410-017736 001000-017726 005400-017211 017540-000013 001000-017732 000040-017754 011640-017216 004440-000020
017130  005400-017144 015101-017124 001000-017734 005500-017142 003453-017143 003453-017143 003350-017111 001000-017063
017140  003240-017734 005100-017110 005100-017737 005100-015240 003240-017734 011000-017216 003350-017207 001000-017732
017150  000040-017755 004700-000020 003340-017730 001043-017756 003340-017202 003453-017143 005100-017745 003240-017732
017160  000040-017757 003340-017176 001343-017756 003340-017175 004700-000020 003340-017731 001000-017732 000040-017760
017170  004700-000010 001343-017756 003240-017727 005400-017214 001000-017063 004400-177776 004700-000000 003240-017732
017200  001000-017726 005100-017745 004700-000036 000040-017732 003240-017733 003453-017143 003350-017111 005100-017324
017210  005100-017110 001000-017064 003350-017726 005100-017123 003000-017727 005100-017174 007000-017250 007040-017274
017220  007140-017276 007100-017307 007200-017314 007240-017324 007340-017366 007400-017404 007300-017444 007440-017460
017230  007500-017457 007600-017477 003400-017243 017540-000004 001600-017733 005400-017244 004600-000020 001000-017761
017240  002600-017762 013240-017056 015101-017237 005100-017252 001000-017712 013240-017056 001000-017761 015101-017236
017250  005100-017510 005100-017232 001000-017056 005400-017262 005100-017673 003453-017252 005100-017745 005500-017271
017260  005400-017267 005100-017254 005100-017673 003453-017252 005100-017745 005500-017271 005100-017252 001000-017712
017270  005100-017254 001000-017245 003350-017252 005100-017110 005100-017520 005100-017251 001000-017733 005100-017531
017300  001000-017055 005400-017305 005100-017673 001000-017056 005100-017260 001000-017712 005100-017302 005100-017520
017310  001000-017733 005100-017531 001000-017252 005100-017256 001000-017730 004700-000020 000300-017763 003600-017320
017320  006210-000000 006000-017003 006740-000030 005100-017110 001000-017736 005400-017327 005100-017542 005100-017552
017330  006203-000000 006000-017643 006740-000030 017540-000067 001000-017764 003350-017614 001000-017561 003350-017637
017340  006204-000000 006000-017643 006700-000030 001000-017063 003240-017642 025101-017626 017540-000027 001000-017761
017350  011043-017643 015101-017350 005400-017357 006203-000000 006000-017643 006740-000030 000151-000040 001000-017736
017360  005400-017110 017540-000027 011000-017033 013240-017003 015101-017362 005100-017110 001000-017731 001343-017765
017370  005400-017400 001043-017765 017640-000000 015101-017374 001000-017761 005100-017673 015101-017374 005100-017110
017400  006204-000000 006000-017003 006700-000030 005100-017110 003400-017443 001000-017730 001043-017756 003340-017430
017410  001000-017730 001343-017766 005540-017417 003240-017732 003443-017417 001000-017732 005100-017411 017540-000001
017420  001000-017756 003340-017417 001000-017731 027640-000000 001000-017726 005100-017745 015101-017427 025101-017430
017430  004700-000036 003240-017732 005100-017673 001000-017732 004700-000032 015101-017442 003453-017726 005100-017745
017440  004700-000036 017540-000004 025101-017431 005100-017551 001000-017736 005400-017447 005100-017542 005100-017552
017450  006202-000000 006000-017643 006740-000030 005203-017455 005100-017357 000173-000000 005100-017357 005100-017510
017460  001000-017755 005100-017673 001600-017733 017540-000003 002500-017767 003240-017733 004240-000013 005400-017475
017470  005100-017673 001000-017733 004000-000004 015101-017464 005100-017110 001000-017712 005100-017470 001000-017733
017500  005100-017531
DUMP COMPLETE.
```

Figure 13-6. Core Dump In Constant Format



LOC.	INST.	DIR.	MEMORY CONTS.	LAC	RAC	LBR	RBR	IX 1	IX 2	IX 4	IX 5
016126	BPX	016711		000000	000000	000000	000000	200000	200000	200000	200000
016711	STAA	016734	005100 016127	000000	000000	000000	000000	200000	200000	200000	200000
016712	CAD	016763		100000	100000	000000	000000	200000	200000	200000	200000
016713	LDB	016763		100000	100000	100000	100000	200000	200000	200000	200000
016714	JADX	000000		100000	100000	100000	100000	200000	200000	200000	200000
016715	STAA	016736	017540 000000	100000	100000	100000	100000	200000	200000	200000	200000
016717	DEP	016736		117540	100000	100000	100000	200000	200000	200000	200000
016720	2ADX	000000		117540	100000	100000	100000	200000	200000	200000	200000
016721	STAA	016737	027540 000000	117540	100000	100000	100000	200000	200000	200000	200000
016723	DEP	016737		127540	100000	100000	100000	200000	200000	200000	200000
016724	4ADX	000000		127540	100000	100000	100000	200000	200000	200000	200000
016725	STAA	016740	047540 000000	127540	100000	100000	100000	200000	200000	200000	200000
016727	DEP	016740		147540	100000	100000	100000	200000	200000	200000	200000
016730	5ADX	000000		147540	100000	100000	100000	200000	200000	200000	200000
016731	STAA	016741	057540 000000	147540	100000	100000	100000	200000	200000	200000	200000
016733	DEP	016741		157540	100000	100000	100000	200000	200000	200000	200000
016734	BPX	016127		157540	100000	100000	100000	200000	200000	200000	200000
016130	CAD	016133		001000	000333	100000	100000	200000	200000	200000	200000
016131	ADDA	016262	000000 000001	001000	000334	100000	100000	200000	200000	200000	200000
016132	RSTA	016146		001000	000334	100000	100000	200000	200000	200000	200000
016133	CAD	000333		005202	015152	100000	100000	200000	200000	200000	200000
016134	FST	016247		005202	015152	100000	100000	200000	200000	200000	200000
016135	LDB	016743		005202	015152	070000	177777	200000	200000	200000	200000
016136	DEP	016141		007700	015152	070000	177777	200000	200000	200000	200000
016137	LSR	000020		000000	015152	007700	177777	200000	200000	200000	200000
016141	ADX	015152		000000	015152	007700	177777	200000	200000	200000	200000
016142	STAA	016426	003240 015152	000000	015152	007700	177777	200000	200000	200000	200000
016143	CAD	016247		005202	015152	007700	177777	200000	200000	200000	200000

Figure 13-7. Typical Interpretive Trap and Trace Output

Appendix I

Program Sample Using General Illustrative Coding

```
                                        PROGRAM SAMP MOD LE          COMPOOL  04D 00   PAGE     1

                IDT                SAMPLE                                COMMENTS IN COLS. 46 TO 710000.01*T00
          INFO  TCPO          90     1000                                     ILLUSTRATIVE       0000.02*T00
          WORD  TCP005         6   INFO                                    ODDBALL CODING       0000.03*T00
          MAC   TOP     ...F,GG$ MACRO DEFINITION WITH COMMENTS TO 71                           0000.04*T00
                1XIN             F                                                              0001.00*T01
                 LDB   GG                                                                       0002.00*T02
                 BOT                                                                            0003.00*T03
                 LOC             140000                                                         0003.01*T04
                 DRM73            41111                                AUX DRUM 73             0003.02*T05
173 1111 10A     CAD             AA          140000  001000 140001                             0004.00*+00
173 1112 AA      RSTA    +    9  +    9      140001  003350 000000  STORE CLASS DOESNT TAKE RC0005.00*+00   ILL USE RC
           01A   MAC     100,170000170000$ MACRO CALL WITH COMMENTS                             0005.01*T00
173 1113   01A   1XIN              100       140002  017540 000144  *   MAC  100,17000017000   0006.00
173 1114         LDB     170000  170000      140003  000300 140054  *   0$ MACRO CALL WITH COM0007.00
173 1115         POS31           WORD        140004  004700 000013                             0008.00*+03
                 SKP             INFO                               MUST HAVE DECIMAL DIRECTOR0009.00*T02   INDETR RHW
173 1116         ETR             WORD        140005  000040 140055  IMPLIED MSK IN COLS. 31-330010.00*+04
173 1117 BB                                  140006  000000 000000  LOCATION TAG WITH COMMENTS0011.00*+00
173 1120 CC                                  140007  000000 000000                             0012.00*+00
                                                                    COMMENTS ONLY             0012.01*T01
                                                                                              0012.02*T02
173 1121         SUB               01A       140010  001343 140002  COLS. 28,29  UNADORNED    0013.00*+01
173 1122         SUBA            177777      140011  001353 177777              17 BIT OPTION0014.00*+02
173 1123         SUB L           $      +999  140012  001341 141761              OVERFLOW     0015.00*+03   INCR. GR 4
173 1124         SUBAL                       140013  001351 000000              BOTH         0016.00*+04
                 SET     2,INF0+237,WORD 1,INF0-343,16/27$ COMMENTS                             0016.01*T07
173 1125         1CAD            INF0  -343  140014  011000 000251  * SET  2,INF0+237,WORD 1,I0017.00
173 1126         FCL                  17     140015  004700 000021  *   3,16/27$ COMMENTS      0018.00
173 1127         LDB     001740  000000      140016  000300 140055  *                         0019.00
173 1130         2DEP            INF0  +237  140017  023600 001355  *                         0020.00
                 TST     A,0/7 2,+37 10A$ READ AS LOC 45, INDEXED                               0020.01*T08
173 1131         FCL                  8      140020  004700 000010  *                         0021.00
173 1132         2CMF            +   37      140021  020460 000045  *   LOC 45, INDEXED       0022.00
173 1133         BPX             10A         140022  005100 140000  *                         0023.00
                 TST     2,A,0/7 37 10A$ ACCUM INDEXED - NO GOOD                                0023.01*T09
173 1134         NOP                         140023  007000 000000  *                         0024.00
                 SCH     2,WORD A,WORD 10A$ FIND ITEM EQ TO ACC BIT                             0024.01*T10
173 1135         LDB     001740  000000      140024  000300 140055  *                         0025.00
173 1136         2CMM            WORD        140025  020400 001000  *   TEM EQ TO ACC BIT     0026.00
173 1137         BPX             10A         140026  005100 140000  *                         0027.00
173 1140         2BPX01          $     -  2  140027  025101 140025  *                         0028.00
```

```
173 1141          SBR              SU123        140030  005100 140036  LIBRARY SUBROUTINE CALL   0029.00*+17
173 1142          CAD      WORD    +    9       140031  001000 140056  +9 IN BIT POSITIONS OF WD.0030.00*+18
173 1143          HOL              ABCDE        140032  153514 171304                           0031.00*+19
173 1144              7            103333       140033  000007 103333  OCTAL CONSTANTS           0032.00*+20
173 1145        +     9            -    2       140034  000011 177775  DECIMAL CONSTANTS,INTEGER 0033.00*+21
173 1146        +.5                -.0625       140035  040000 173777                FRACTION0034.00*+22
                                                                            LIBRARY TAPE ROUTINES
173 1147 SU123    STA                 2AM        140036  003400 140042  SU123   READ ONE CARD DIRE0035.00*+00
173 1150          STA                 4AM        140037  003400 140051  SU123   SET EXIT          0036.00*+01
173 1151          AOR                 4AM        140040  003443 140051  SU123                     0037.00*+02
173 1152   1AM    SEL01               -    0     140041  006201 177777  SU123   CARD READER       0038.00*+00
173 1153   2AM    CAD                             140042  001000 000000  SU123     +/- , LOC OF IMA0039.00*+00
173 1154         3LDC                              140043  036000 000000  SU123   SET LOC OF IMAGE  0040.00*+01
173 1155          BSN11               5AM        140044  005211 140052  SU123   TEST READY CONDITI0041.00*+02
173 1156          RDS                   2        140045  006700 000030  SU123                     0042.00*+03
173 1157          BLM                 3AM        140046  005500 140050  SU123   TEST FOR I/O DELAY0043.00*+04
173 1160          LDC                            140047  006000 000000  SU123   WAIT FOR I/O      0044.00*+05
173 1161   3AM    CAD      +    0  +    0        140050  001000 140057  SU123   CARD READ OK      0045.00*+00
173 1162   4AM    BPX                            140051  005100 000000  SU123   EXIT              0046.00*+00
173 1163   5AM    CAD                 1AM        140052  001000 140041  SU123   I/O UNIT NOT READY0047.00*+00
173 1164          BPX                 4AM        140053  005100 140051  SU123   LAST CARD         0048.00*+01
                  END               AA                                 PROGRAM STARTING ADDRESS  0049.00*T02
173 1165                                         140054  170000 170000
173 1166                                         140055  001740 000000
173 1167                                         140056  000440 000000
173 1170                                         140057  000000 000000
```

*** *** *** *** *** *** *** *** *** *** LAST PAGE *** *** *** *** *** *** *** *** *** *** ***

Appendix II

Sample Printout Of A COSEAL Translation

Errors And Examples

```
                IDT     COMPOL  WAKER                                                                   0000.01*T00
                LOC             10100                                                                   0000.02*T01
                                                                THE FOLLOWING IS A COMPEND0000.03*T02
                                                                IUM OF INCORRECT AND      0000.04*T03
                                                                ODDBALL CODING, WITH A FEW0000.05*T04
                                                                CORRECT EXAMPLES FOR COMP-0000.06*T05
                                                                ARISON.                   0000.07*T06
                                                                                          0000.08*T07
                                                                                          0000.09*T08
                                                                TIDY IS AN ITEM OCCUPYING 0000.10*T09
                                                                BITS 5 THROUGH 8 OF TDRO. 0000.11*T10
                                                                TPOS IA AN ITEM OCCUPYING 0000.12*T11
                                                                BITS 0 THROUGH 31 OF TDR1.0000.13*T12
                                                                SAFO IS A TABLE STARTING  0000.14*T13
                                                                IN CORE LOCATION 100062   0000.15*T14
                                                                TDRO IS A TABLE STARTING  0000.16*T15
                                                                IN CORE LOCATION 116017   0000.17*T16
                                                                                          0000.18*T17
                                                                                          0000.19*T18
                                                                                          0000.20*T19
                                                                RC WORDS                  0000.21*T20
                                                                                          0000.22*T21
000 0000 10A    CAD             $       010100  001000 010100                            0001.00*+00
                CAD     TIDY    +  15   010101  001000 010613  INSERTS DECIMAL 15        0002.00*+01
                CAD     TIDY       15   010102  001000 010614  INSERTS OCTAL 15          0003.00*+02
                CAD     TIDY    -  15   010103  001000 010615  INSERTS NONSENSE          0004.00*+03
                CAD     TIDY    +  77   010104  001000 010616  +77 IN TIDY + BITS BEFORE 0005.00*+04
                CAD     TIDY    + 1000  010105  001000 010617                            0006.00*+05
                CAD     TPOS    -  15   010106  001000 010620                            0007.00*+06
               2CAD     +   2   +   2   010107  021000 010621  INDEXED RC WORD           0008.00*+07
                CAD     +   6   +  6+002 010110  001000 010622  INCREMENTED RC WORD       0009.00*+08
                ADD     +  12      12.  010111  001043 010623  MIXED RC WORD ALL RIGHT   0010.00*+09
                CAD     +   5   +  U    010112  001000 000000  LETTER O USED FOR  NUMBER 0011.00*+10  UNDEFINED
                ADD     +   9   +  9    010113  001043 010624                            0012.00*+11
                ADD         9      9    010114  001043 000000  OCTAL 9 IN DIRECTOR       0013.00*+12  INDETR RC
                RST     +   9   +  9    010115  003340 000000  DOES NOT TAKE RC WORD     0014.00*+13  ILL USE RC
                CAD        +9     +9    010116  001000 000000  SIGNS IN WRONG COLUMNS    0015.00*+14  INDETR RC
                CAD     +9         +9   010117  001000 010625  NUMBERS IN WRONG COLUMNS  0016.00*+15
                CAD     SAFO    +  15   010120  001000 000000  DOES NOT WORK WITH A TABLE0017.00*+16  INDETR RC
                CML             010121  000440 000000  NOTHING TO COMPARE WITH   0018.00*+17
                CML     +   7           010122  000440 010626  INTERPRETS 7 AS A CONSTANT0019.00*+18
```

```
CML          +    7    010123 000440 000007 INTERPRETS 7 AS AN ADDRESS0020.00*+19
CML          -    7    010124 000440 177770                         0021.00*+20
CML    +  7  +    7    010125 000440 010627 COMPLETE CONSTANT        0022.00*+21
CAD    CAD   10A       010126 001000 000000                         0023.00*+22  INDETR RC
CAD    IDT     2222    010127 001000 000000 IDT, END, DIT NOT ALLOWED 0024.00*+23  INDETR RC
CAD    SKP        3    010130 001000 000000 NUR ARE SKP, LOC, DRM, CPO0025.00*+24  INDETR RC
                                                                      0025.01*T25
                                                                      0025.02*T26
                                                                      0025.03*T27
                                             A OPTION AND OVERFLOW    0025.04*T28
                                                                      0025.05*T29
ADD          10A     010131 001043 010100                         0026.00*+25
ADDA         10A     010132 001053 010100 SETS A OPTION            0027.00*+26
ADD R        10A     010133 001042 010100 SETS OVERFLOW INDICATION 0028.00*+27
ADDR         10A     010134 001053 010100 SETS A OPTION INSTEAD    0029.00*+28
ADD A        10A     010135 001042 010100 A IN WRONG COLUMN        0030.00*+29
ADD C        10A     010136 001041 010100 ILLEGAL OVERFLOW INDICATOR0031.00*+30
CADA         10A     010137 001000 010100 CAD DOES NOT HAVE A OPTION0032.00*+31
CAD R        10A     010140 001000 010100 CAD CANNOT OVERFLOW      0033.00*+32
RST          10A     010141 003340 010100                         0034.00*+33
FST          10A     010142 003240 010100                         0035.00*+34
RSTA         10A     010143 003350 010100 GIVES A OPTION           0036.00*+35
FSTA         10A     010144 003240 010100 DOES NOT HAVE A OPTION   0037.00*+36
                                                                      0037.01*T42
                                                                      0037.02*T43
                                                                      0037.03*T44
                                             PSEUDO INSTRUCTIONS      0037.04*T45
                                                                      0037.05*T46
POS07        TIDY    010145 004700 000002                         0038.00*+37
POS38        TIDY    010146 004700 000003 POS IS MOD 31            0039.00*+38
POS30        TIDY    010147 004700 000013 INSERTS CORRECT FCL      0040.00*+39
POS30        TIDY +002 010150 004700 000015 INCREMENTED ITEM       0041.00*+40
POS30  0     TIDY    010151 004700 010630                         0042.00*+41
POS30        10A     010152 004700 010100 DIRECTOR IS A TAG        0043.00*+42
RES    TIDY  TPOS    010153 004700 000006 USE OF TWO ITEMS         0044.00*+43
RES15  TIDY  TPOS    010154 004700 000006 IGNORES THE 15           0045.00*+44
RES    TIDY  SAFO    010155 004700 100062 INSERTS SAFO TABLE ADDRESS0046.00*+45
RES    SAFO  TIDY    010156 004700 000000                         0047.00*+46  INDETR RC
POS31        SAFO    010157 004700 100062 INSERTS SAFO TABLE ADDRESS0048.00*+47
HOL          ABCDE   010160 153514 171304 HOL USES THESE 5 POSITIONS0049.00*+48
HOL          ABCDEF  010161 153514 171304 HOL TAKES ONLY FIRST 5   0050.00*+49
```

```
4HOL          ABCDE          010162 153514 171304                          0051.00*+50
 HOL          ABCDE +001     010163 153514 171304                          0052.00*+51
 HOL   ABCDE                 010164 000000 000000 LETTERS IN WRONG POSITIONS0053.00*+52
 CHR          VWXYZ          010165 043613 000337                          0054.00*+53
 CHR          VWXY$          010166 043610 000330 NO SPECIAL SYMBOLS FOR CHR0055.00*+54   INDETR RHW
 TOB31        10A            010167 000537 010100                          0056.00*+55
 TOB02        10A            010170 000502 010100                          0057.00*+56
 TOB34        10A            010171 000502 010100 TOB IS MOD 31            0058.00*+57
 TOB06        TIDY           010172 000514 116017 TIDY IS A 4 BIT ITEM     0059.00*+58
 TOB06        TDRO           010173 000506 131205                          0060.00*+59
 CON   TIDY        15        010174 001500 000000 INSERTS OCTAL 15         0061.00*+60
 CON   TIDY   +    15        010175 001700 000000 INSERTS DECIMAL 15       0062.00*+61
 CON   TIDY   -    15        010176 176000 000077 INSERTS NONSENSE         0063.00*+62   ILL VALUE
 CON   TIDY                  010177 000000 000000 NO AMOUNT GIVEN          0064.00*+63
 CON   TIDY   +    16        010200 002000 000000 LARGER THAN CAPACITY     0065.00*+64   ILL VALUE
 CON   TIDY   +66666         010201 015200 000001                          0066.00*+65   ILL VALUE
 CON   TIDY       +15        010202 015200 000000 SIGN IN WRONG COLUMN     0067.00*+66   UNDEFINED
 CON   TPOS         1        010203 000000 000000 TPOS IS MORE THAN 16 BITS 0068.00*+67   ILL ITEM
 CON   SAFO        15        010204 000000 000000 SAFO IS A TABLE          0069.00*+68   INDETR RC
 CON   AAAA        15        010205 000000 000000 NOT IN COMPOOL           0070.00*+69   INDETR RC
 MSK   TIDY                  010206 000000 000000 TIDY IN WRONG COLUMNS    0071.00*+70   UNDEFINED
 MSK          TIDY           010207 001700 000000 GENERATES MASK FOR TIDY  0072.00*+71
                                                                           0072.01*T82
                                                                           0072.02*T83
                                                                           0072.03*T84
                                                   MISSING ZEROES          0072.04*T85
                                                                           0072.05*T86
2BPX 1       10A            010210 025101 010100 0 MISSING IN COLUMN 28   0073.00*+72
 TOB3        10A            010211 000536 010100 TRAILING ZERO MISSING    0074.00*+73
 TOB 3       10A            010212 000503 010100 LEADING ZERO MISSING     0075.00*+74
 CAD   +  3  +  3           010213 001000 010631 WORKS WITH NO TRAILING O'S0076.00*+75
 CAD         10A    +  5    010214 001000 010105 NO LEADING ZEROS IN INCR 0077.00*+76   INCR. GR 4
 ETR   177777 0              010215 000040 010632 TRAILING ZEROES MISSING  0078.00*+77
 ETR   177777       0        010216 000040 010632 LEADING ZEROES MISSING   0079.00*+78
 CAD          1 A            010217 001000 000000 0 OF 10A MISSING         0080.00*+79   UNDEFINED
                                                                           0080.01*T95
                                                                           0080.02*T96
                                                                           0080.03*T97
                                                   BRANCH AND CYCLE        0080.04*T98
                                                                           0080.05*T99
 BPX                +300     010220 005100 000454 ADDRESS IN THE INCREMEMNT 0081.00*+80   INCR. GR 4
```

```
          BPX          - 300     010221  005100 177323                                0082.00**+81
          1BPX63       10A       010222  015177 010100                                0083.00**+82
          1BPX64       10A       010223  015100 010100  64 IS ABOVE CAPACITY           0084.00**+83
          3BPX01       10A       010224  035101 010100  CANNOT DECREMENT ACC           0085.00**+84
          6BPX01       10A       010225  065101 010100  NO INDEX 6                     0086.00**+85
          9BPX01       10A       010226  015101 010100  NO INDEX 9                     0087.00**+86
           FCL         +   11    010227  004700 000013                                0088.00**+87
           FCL             11    010230  004700 000013                                0089.00**+88
           FCL11                 010231  004700 000000  11 IS IN WRONG POSITIONS       0090.00**+89
           FCL         177777    010232  004700 133161                                0091.00**+90
           FCL             44    010233  004700 000054                                0092.00**+91
           FCL         -    3    010234  004700 177774                                0093.00**+92
                                                                                       0093.01*
                                                                                       0093.02*
                                                                                       0093.03*
                                                                 TAGS                  0093.04*
                                                                                       0093.05*
           CAD         AAAA      010235  001000 000000  UNDEFINED TAG                  0094.00**+93    UNDEFINED
          CAD          10A       010236  001000 010100                                0095.00**+00
11A       ADD          10A       010237  001043 010100  DUPLICATE TAG                 0096.00**+00
          CAD          11A       010240  001000 010236                                0097.00**+01    DUPL TAG
20A*      CAD          10A       010241  001000 010100    IS ILLEGAL IN TAG           0098.00**+00
A         CAD          10A       010242  001000 010100  LEGAL TAG                     0099.00**+00
 A        CAD          10A       010243  001000 010100  LEGAL TAG                     0100.00**+00
  A       CAD          10A       010244  001000 010100  LEGAL TAG                     0101.00**+00
   A      CAD          10A       010245  001000 010100  LEGAL TAG                     0102.00**+00
    A     CAD          10A       010246  001000 010100  LEGAL TAG                     0103.00**+00
                                                                                       0103.01*T01
                                                                                       0103.02*T02
                                                                                       0103.03*T03
                                                                 INDEX REGISTERS       0103.04*T04
                                                                                       0103.05*T05
          1XIN         -    1    010247  017540 177776  PUTS VERY LARGE NO. IN IX     0104.00**+01
          1XIN         -    0    010250  017540 177777  PUTS LARGEST NO IN INDEX 1    0105.00**+02
          3XIN              5    010251  037540 000005  ASSEMBLES, BUT ACTS AS NOP    0106.00**+03
           XIN              7    010252  007540 000007  NO INDEX GIVEN                0107.00**+04
          1XIN         100000    010253  017540 103240                                0108.00**+05
          1XIN         177777    010254  017540 133161                                0109.00**+06
          1XIN         200000    010255  117540 000000  SETS BRANCH BIT               0110.00**+07
          1XIN         300000    010256  117540 100000  ALSO SETS BRANCH BIT          0111.00**+08
          1XIN         400000    010257  017540 000000                                0112.00**+09    UNDEFINED
```

```
2XIN              7      010260  000000 000000  USE NUMBER 1 FOR LETTER I 0113.00*+10     INDETR LHW
1XIN          65536      010261  017540 000001  CAPACITY IS 65535         0114.00*+11
1XIN          1,000      010262  017540 000000  NO COMMAS ALLOWED         0115.00*+12     UNDEFINED
1XIN           TIDY      010263  017540 116017                           0116.00*+13
1ADX           SAFO      010264  017700 100062  SAFO IS A TABLE           0117.00*+14
1ADX           TIDY      010265  017700 116017  TIDY IS AN ITEM           0118.00*+15
1XAC                     010266  017640 000000                           0119.00*+16
1XAC              5      010267  017640 000005                           0120.00*+17
                                                                         0120.01*T23
                                                                         0120.02*T24
                                                                         0120.03*T25
                                                COMAND PSEUDO INSTRUCTIONS0120.04*T26
                                                                         0120.05*T27
 TST    A,0/7 37 10A$ CORRECT FORMAT                                     0120.06*T28
 LDB    177400  000000      010270  000300 010633  * TST  A,0/7 37 10A$ CORRE0121.00
 CMM    017400  000000      010271  000400 010634  *   MAT                  0122.00
 BPX            10A         010272  005100 010100  *                        0123.00
 TST    A,0/7 37 10A. ALSO CORRECT FORMAT                                 0123.01*T29
 LDB    177400  000000      010273  000300 010633  * TST  A,0/7 37 10A. ALSO 0124.00
 CMM    017400  000000      010274  000400 010634  *   T FORMAT             0125.00
 BPX            10A         010275  005100 010100  *                        0126.00
 TST    A,0/7 37 10A $SPACE BEFORE $                                      0126.01*T30
 NOP                        010276  007000 000000  *                        0127.00
 TST    A,0/7 37 10A   SPACE BEFORE PERIOD                                0127.01*T31
 NOP                        010277  007000 000000  *                        0128.00
 TST    A,0/7 37$ NO THIRD EXPRESSION                                     0128.01*T32
 NOP                        010300  007000 000000  *                        0129.00
 TST    A, 0/7 37 10A$ SPACE AFTER COMMA                                  0129.01*T33
 NOP                        010301  007000 000000  *                        0130.00
 TST    A,7/0 37 10A$ BIT POSITIONS REVERSED                              0130.01*T34
 NOP                        010302  007000 000000  *                        0131.00
 TST    A, 37 10A$ NO BIT POSITIONS GIVEN AFTER CO                        0131.01*T35
 NOP                        010303  007000 000000  *                        0132.00
 TST    A 37 10A$ TESTS FULL ACCUMATOR                                    0132.01*T36
 CMF    000000  000037      010304  000460 010635  * TST  A 37 10A$ TESTS FUL0133.00
 BPX            10A         010305  005100 010100  *   MATOR                0134.00
 TST    A,0/7 37,0/31 10A$ TESTS CONTENTS OF LOC 3                        0134.01*T37
 FCL                8       010306  004700 000010  *                        0135.00
 CMF               37       010307  000460 000037  *   CONTENTS OF LOC 3    0136.00
 BPX            10A         010310  005100 010100  *                        0137.00
 TST    A,0/7 37+0 10A$ CONSTANT INCREMENT ILLEGAL                        0137.01*T38
```

```
 NOP                          010311  007000 000000  *                          0138.00
 TST     A,0/7 2,37 10A$ READ AS LOC 37, INDEXED                                 0138.01*T39
 FCL                8         010312  004700 000010  *                          0139.00
2CMF              37          010313  020460 000037  *   OC 37, INDEXED         0140.00
 BPX          10A              010314  005100 010100  *                          0141.00
 TST     A,0/7 2,+37 10A$ READ AS LOC 45, INDEXED                                0141.01*T40
 FCL                8         010315  004700 000010  *                          0142.00
2CMF            +   37         010316  020460 000045  *   LOC 45, INDEXED        0143.00
 BPX          10A              010317  005100 010100  *                          0144.00
 TST     A,0/7 39 10A$ ILLEGAL OCTAL CONSTANT                                    0144.01*T41
 LDB     177400  000000       010320  000300 010633  * TST  A,0/7 39 10A$ ILLEG0145.00
 CMM     000000  000000       010321  000400 010636  *   AL CONSTANT            0146.00
 BPX          10A              010322  005100 010100  *                          0147.00
 TST     A 1000000 10A$ 1000000 TOO LARGE                                        0147.01*T42
 NOP                          010323  007000 000000  *                          0148.00
 TST     TIDY+1000 37 10A$ INCREMEMNT OVER 999 ILLE                              0148.01*T43
 NOP                          010324  007000 000000  *                          0149.00
 TST     TIDY,0/1 37 10A$ NO BIT POSITION IN ITEM A                              0149.01*T44
 NOP                          010325  007000 000000  *                          0150.00
 TST     A,0/7 37 1000$ EXPRESSION 3 MUST BE DDL                                 0150.01*T45
 LDB     177400  000000       010326  000300 010633  * TST  A,0/7 37 1000$ EXPR0151.00
 CMM     017400  000000       010327  000400 010634  *    3 MUST BE DDL         0152.00
 BPX           2              010330  005100 020000  *                          0153.00
 TST     2,A,0/7 37 10A$ ACCUM INDEXED                                           0153.01*T46
 NOP                          010331  007000 000000  *                          0154.00
                                                     SET EXPRESSION ONE EQUAL   0154.01*T47
                                                     TO EXPRESSION TWO          0154.02*T48
                                                                                0154.03*T49
 SET     2,TIDY 1,TIDY$  BOTH EXPRESSIONS INDEXED
1CAD           TIDY           010332  011000 116017  * SET  2,TIDY 1,TIDY$  BOT0155.00
 LDB     001700  000000       010333  000300 010613  *   ESSIONS INDEXED        0156.00
2DEP           TIDY           010334  023600 116017  *                          0157.00
 SET     2,TDR0+999,0/7 1,TDR0+998,16/22$ EACH                                   0157.01*T50
1CAD           TDR0  +998     010335  011000 133153  * SET  2,TDR0+999,0/7 1,TD0158.00
 FCL                15         010336  004700 000017  *   ,16/22$ EACH           0159.00
 LDB     17?400  000000       010337  000300 010633  *                          0160.00
2DEP           TDR0  +999     010340  023600 133154  *                          0161.00
                                                     EXPRESSION MAY HAVE        0161.01*T51
                                                                                0161.02*T52
 SET     TIDY +49$       1 TO 16 CHARACTERS
 CAD     000100  000000       010341  001000 010637  * SET  TIDY +49$       1 T0162.00
 LDB     001700  000000       010342  000300 010613  *   HARACTERS              0163.00
 DEP            TIDY           010343  003600 116017  *                          0164.00
```

```
      SET   TIDY 16$          OCTAL VALUE                                                    0164.01*T53
      CAD   001600  000000        010344  001000 010640  * SET  TIDY 16$          OCT0165.00
      LDB   001700  000000        010345  000300 010613  *   UE                      0166.00
      DEP           TIDY          010346  003600 116017  *                           0167.00
      SET   TIDY 99$          ERROR NOT FLAGGED                                              0167.01*T54
      CAD   000000  000000        010347  001000 010636  * SET  TIDY 99$           ERR0168.00
      LDB   001700  000000        010350  000300 010613  *    FLAGGED              0169.00
      DEP           TIDY          010351  003600 116017  *                           0170.00
      SET   2,TIDY+4 10$      INDEXED AND INCREMENTED                                        0170.01*T55
      CAD   001000  000000        010352  001000 010641  * SET  2,TIDY+4 10$      IND0171.00
      LDB   001700  000000        010353  000300 010613  *   ND INCREMENTED        0172.00
     2DEP           TIDY  +  4   010354  023600 116023  *                           0173.00
      SET   10A,5/8 5$        LOCATION AND BIT POSITIONS                                     0173.01*T56
      CAD   001200  000000        010355  001000 010642  * SET  10A,5/8 5$        LOC0174.00
      LDB   003600  000000        010356  000300 010643  *   AND BIT POSITIONS     0175.00
      DEP           10A           010357  003600 010100  *                           0176.00
      SET   10A,TIDY 5$       LOCATION AND ITEM                                              0176.01*T57
      CAD   000500  000000        010360  001000 010644  * SET  10A,TIDY 5$       LOC0177.00
      LDB   001700  000000        010361  000300 010613  *   AND ITEM              0178.00
      DEP           10A           010362  003600 010100  *                           0179.00
      SET   A,TIDY 5$         ACCUMULATOR AND ITEM                                           0179.01*T58
      CAD   000500  000000        010363  001000 010644  * SET  A,TIDY 5$         ACC0180.00
                                                         *SETOR AND ITEM         ACC00180.01
      SET   X1 +99$           SET XR1 EQUAL TO 99                                            0180.02*T59
     1XIN           000143        010364  017540 000217  * SET  X1 +99$           SET0181.00
                                                         *SETQUAL TO 99          SET00181.01
      SET   A,5/8 5$          ACCUMULATOR AND BITS                                           0181.02*T60
      CAD   001200  000000        010365  001000 010642  * SET  A,5/8 5$          ACC0182.00
                                                         *SETOR AND BITS         ACC00182.01
10B   SET   1,TIDY 5$          INTERNAL TAG LEGAL                                             0182.02*T00
10B   CAD   000500  000000        010366  001000 010644  * SET  1,TIDY 5$         INT0183.00
      LDB   001700  000000        010367  000300 010613  *   TAG LEGAL             0184.00
     1DEP           TIDY          010370  013600 116017  *                           0185.00
                                                         SEARCH WITH SPECIFIED XR    0185.01*T01
                                                         THE TABLE FOR THE VALUE     0185.02*T02
                                                         SPECIFIED IN THE ITEM OR    0185.03*T03
                                                         BIT POSITIONS SPECIFIED     0185.04*T04
                                                                                     0185.05*T05
      SCH   2,TIDY +10 10A$ WITH XR2 FIND TIDY EQ 10
      CAD   001200  000000        010371  001000 010642  * SCH  2,TIDY +10 10A$ WIT0186.00
      LDB   001700  000000        010372  000300 010613  *   FIND TIDY EQ 10       0187.00
     2CMM           TIDY          010373  020400 116017  *                           0188.00
```

```
 BPX          10A          010374  005100 010100  *                          0189.00
2BPX01        $     ' 2     010375  025101 010373  *                          0190.00
 SCH  2,TDRO,5/8 +10 10A$  SAME WITH BITS                                     0190.01*T06
 CAD  002400  000000       010376  001000 010645  * SCH  2,TDRO,5/8 +10 10A$0191.00
 LDB  003600  000000       010377  000300 010643  *   WITH BITS              0192.00
2CMM          TDRO         010400  020400 131205  *                          0193.00
 BPX          10A          010401  005100 010100  *                          0194.00
2BPX01        $     ' 2     010402  025101 010400  *                          0195.00
 SCH  2,TIDY A,5/8 10A$  FIND TIDY EQ TO ACC BIT                              0195.01*T07
 FCL            31         010403  004700 000037  *                          0196.00
 LDB  001700  000000       010404  000300 010613  *   IDY EQ TO ACC BIT      0197.00
2CMM          TIDY         010405  020400 116017  *                          0198.00
 BPX          10A          010406  005100 010100  *                          0199.00
2BPX01        $     ' 2     010407  025101 010405  *                          0200.00
 SCH  2,TIDY A,TIDY 10A$ SAME WITH ITEM                                       0200.01*T08
 LDB  001700  000000       010410  000300 010613  *                          0201.00
2CMM          TIDY         010411  020400 116017  *   ITH ITEM               0202.00
 BPX          10A          010412  005100 010100  *                          0203.00
2BPX01        $     ' 2     010413  025101 010411  *                          0204.00
 LOC           10500                                                         0204.01*T09
                                                                             0204.02*T10
                                                                             0204.03*T11
                                                                             0204.04*T12
                                                   CALLING SEQUENCE          0204.05*T13
                                                                             0204.06*T14
 SDR18                     010500  000000 000000  ILLEGAL DRUM               0205.00*+23   INDETR LHW
 LDC   -   4  -    1       010501  006000 010646                             0206.00*+24
 RDS   -   4  -    1       010502  006700 000000                             0207.00*+25   ILL USE RC
 SDR          KIN          010503  006100 000000  READS PROGRAM              0208.00*+26   UNDEFINED
 LDC          KIN          010504  006000 000000  KIN FROM                   0209.00*+27   UNDEFINED
 RDS          KIN          010505  006700 000000  DRUM TO CORE               0210.00*+28   UNDEFINED
 SDR          TIDY         010506  006100 000000  WILL NOT WORK WITH ITEM    0211.00*+29
 LDC          TIDY         010507  006000 116017                             0212.00*+30
 RDS          TIDY         010510  006700 001046                             0213.00*+31
 SDR10                     010511  006110 000000                             0214.00*+32
 LDC             100       010512  006000 000100                             0215.00*+33
 RDS03            50       010513  006703 000062  MEANINGLESS INTERLEAVE     0216.00*+34
 BPX          GI           010514  005100 000000  DOES NOT ASSEMBLE          0217.00*+35   UNDEFINED
 BPX           16050       010515  005100 016050  WILL BRANCH TO GI          0218.00*+36
 SDI          10A          010516  007000 010100                             0219.00*+37
 CAD          10A          010517  000000 000000  SHOULD BE SECOND WORD OF   0220.00*+38   INDETR LHW
```

```
                                                                CALLING SEQUENCE            0220.01*T31
 BPX               GO                010520  005100 000000                                  0221.00**39     UNDEFINED
 SDF85             ACC               010521  007525 000000  LEAST SIG BIT IS MOD 32         0222.00**40
140 16                               010522  106020 000000  CHAR POSITION IS MOD 128        0223.00**41
                                                                                            0223.01*T35
                                                                                            0223.02*T36
                                                                                            0223.03*T37
                                                            MISCELLANEOUS ERRORS            0223.04*T38
                                                                                            0223.05*T39
 CAD                                 010523  001000 000000  NO DIRECTOR                     0224.00**42
 CAD               SAFO              010524  001000 100062  SAFO IS A TABLE AT 100056       0225.00**43
 CAD               SAFO    +005      010525  001000 100067  WITHIN TABLE                    0226.00**44
 CAD               SAFO    +010      010526  001000 100074  BEYOND SAFO TABLE               0227.00**45
 CAD                 1000            010527  001000 001000  CORE LOCATION 1000              0228.00**46
 CAD               1000              010530  001000 100000  1000 MISPLACED                  0229.00**47
 CAD                  10A            010531  001000 000000  10A MISPLACED                   0230.00**48     UNDEFINED
 CAD                 1000+001        010532  001000 001001  INCREMENT WITH ABS ADDRESS0231.00**49
 CAD               + 1000            010533  001000 001750  NOT LOC 1000,1/2 RC WORD        0232.00**50
 CAD                       +001      010534  001000 000001  INCREMENT ONLY DIRECTOR         0233.00**51
 100               10A               010535  010000 010100  100 IS OCTAL CODE FOR CAD 0234.00**52
  100              10A               010536  001000 010100  BUT IT MUST BE IN 26 TO 280235.00**53
 CAD               10A     +         010537  001000 010100  NO VALUE IN INCREMENT           0236.00**54
 CAD               10A      005      010540  001000 010100  IGNORES UNSIGNED INCREMENT0237.00**55
 CAD       +     0 377777            010541  001000 010647  GIVES 16 BIT RHW IN RC WD.0238.00**56
 CAD               377777            010542  101000 177777  GIVES 17 BIT ADDRESS            0239.00**57
 BSN14             10A               010543  005214 010100                                  0240.00**58
 BSN               10A               010544  005200 010100  NO  SENSE CODE                  0241.00**59
 PER75                               010545  000175 000000                                  0242.00**60
 PER99                               010546  000000 000000  PER TAKES OCTAL NUMBERS         0243.00**61     INDETR LHW
 PER                      75         010547  000100 000075  75 IN WRONG COLUMNS             0244.00**62
 PER42                               010550  000142 000000  MEANINGLESS INSTRUCTION         0245.00**63
 DIT               -     3           010553  000142 000000                                  0248.00**66
 CAD      TPOS     +   1.5           010554  001000 000000                                  0249.00**67     UNDEFINED
 DIT                     3           010557  001000 000000                                  0252.00**70
 CAD      TPOS     +    10           010560  001000 010650                                  0253.00**71
 DIT               +     3           010563  001000 010650                                  0256.00**74
 SKP                     3                                                                  0259.00*T67
 ETR      277777   000000            010567  000040 010651  2 IS IGNORED                    0260.00**77
 SKP               -     3                                  NEGATIVE DIRECTOR               0263.00*T69
 ETR      000000   277777            010573  000040 010652  2 IS IGNORED                    0264.00**80
 SKP               +     3                                  POSITIVE DIRECTOR               0267.00*T71
```

```
ETR     177777          010577  000040 010632  RIGHT HALF OF MASK MISSING0268.00**+83
SKP             TIDY                           ITEM IN DIRECTOR        0269.00*T73     INDETR RHW
ETR     TIDY    +   5   010600  000040 010644                          0270.00**+84
SKP     +   3   +   3                          USED AS RC WORD         0273.00*T75
ABC             10A     010604  000000 000000  ILLEGAL INSTRUCTION      0274.00**+87    INDETR LHW
1SKP                3                          INDEXED                 0277.00*T77
CAD                 T   010610  101000 177760  T PUNCHED CORRECTLY      0278.00**+90
SKP                   + 3                      DIRECTOR IN INCREMENT   0278.01*T79
LOC             10900                          0278.02*T80     INDETR RHW
DRM02             100                          0278.03*T81
CAD             T       010611  001000 000000  T SHOULD BE IN COL 41    0279.00**+91    UNDEFINED
LOC             80000                          0279.01*T83     INDETR RHW
DRM55             100                          0279.02*T84
CAD                 T+016 010612  001000 000001  BEYOND TEST MEMORY      0280.00**+92    INCR. GR 4
END             10B                            UNDEFINED TAG           0280.01*T86
                          010613  001700 000000
                          010614  001500 000000
                          010615  176000 000077
                          010616  011500 000000
                          010617  014400 000000
                          010620  000000 177760
                          010621  000002 000002
                          010622  000006 000010
                          010623  000014 000012
                          010624  000011 000011
                          010625  057620 057620
                          010626  000007 000000
                          010627  000007 000007
                          010630  000000 116017
                          010631  000454 000454
                          010632  177777 000000
                          010633  177400 000000
                          010634  017400 000000
                          010635  000000 000037
                          010636  000000 000000
                          010637  000100 000000
                          010640  001600 000000
                          010641  001000 000000
                          010642  001200 000000
                          010643  003600 000000
                          010644  000500 000000
```

```
010645  002400 000000
010646  177773 177776
010647  000000 177777
010650  000000 000012
010651  077777 000000
010652  000000 077777
```

*** *** *** *** *** *** *** *** *** *** *** LAST PAGE *** *** *** *** *** *** *** *** *** *** ***

Appendix III

Duplex Maintenance Console