C690-416L-ST

AIR TRAINING COMMAND

COMPUTER SYSTEMS DEPARTMENT

STUDENT TEXT

ABR30533-1

CENTRAL COMPUTER CONTROL

17 January 1966

RESPONSIBLE AGENCY OF ATC

Keesler Technical Training Center
Keesler AFB, Mississippi

Computer Systems Department
KTTC, Mississippi

Student Text
ABR30533-1
17 January 1966

CONTENTS

This Student Text consists of three parts, divided into fifteen chapters.

CHAPTER 1 - BASIC CENTER OPERATION AND EQUIPMENT

DUPLEXING

The Air Defense of the United States is the primary concern of the SAGE system. Because Air Defense is effective only if it can be maintained with the least amount of interruption, it is of the utmost importance that the FSQ-7 be able to run continually. To enable this continuous operation, the FSQ-7 was developed in the duplex concept. Within the Direction Center, the equipment is duplicated, in whole or in part, to attain the greatest possible reliability. This "Duplex" equipment is contained in two groups designated Computer A and Computer B (see Figure 1-1). Computer A and Computer B each contain an entire Central Computer System, Drum System, Output System, and those elements of the other three systems whose failure would render all systems useless if not replaced or repaired. For example, since the failure of a generator element in the Display System would render the entire Display System inoperative and limit the usefulness of the entire equipment, the generator elements are duplexed, (duplicated in Computer A and Computer B). On the other hand, the failure of one display console will not render the entire Display System inoperative. Therefore, display consoles are not duplex, instead; spare display consoles are available as substitutes in case of failure of a console. Non-duplexed units are known as "Simplex" equipment and are not shown in Figure 1-1.

It should be noted that only one set of duplex equipment is processing raw air defense data at any one time. This set, called the active computer, may be either Computer A or Computer B. The nonactive computer, called the standby computer, may be undergoing maintenance or, if operative, may be performing supplementary data-processing operations. The intercommunication (IC) fields of the two Drum Systems allow for interchange of information between the active and the standby computers.

SAGE CENTER

As previously stated the overall problem assigned to the SAGE System is the Air Defense of North America. This problem is three-fold; detection, identification, and if necessary, destruction. In order to solve Air Defense problems, special physical components were developed. Detailed specifications were prepared covering the physical components: radars, computers, weapons, etc., and how they were to interact. The purpose of the sophisticated equipment that was developed is to supply information, process this data, present a visual observation of the data that is processed, and present an option of type of weapons to be used. This sophisticated equipment does not make the decision, this is the function of man in the SAGE system.

The Inputs Equipment of the AN/FSQ-7 receives information from several agencies. Some of the agencies which send inputs to the AN/FSQ-7 are:

1. Long Range Radar (LRI): LRI sites present the azimuth and height of the detected object. It also supplies the Mark X IFF information if available. Devices used to provide this information are the land based radar squadrons, airborne early warning aircraft and picket ships.

2. Other AN/FSQ-7's (XTEL): provide information concerning the Air Defense situation in adjacent sectors.

DCS Operation

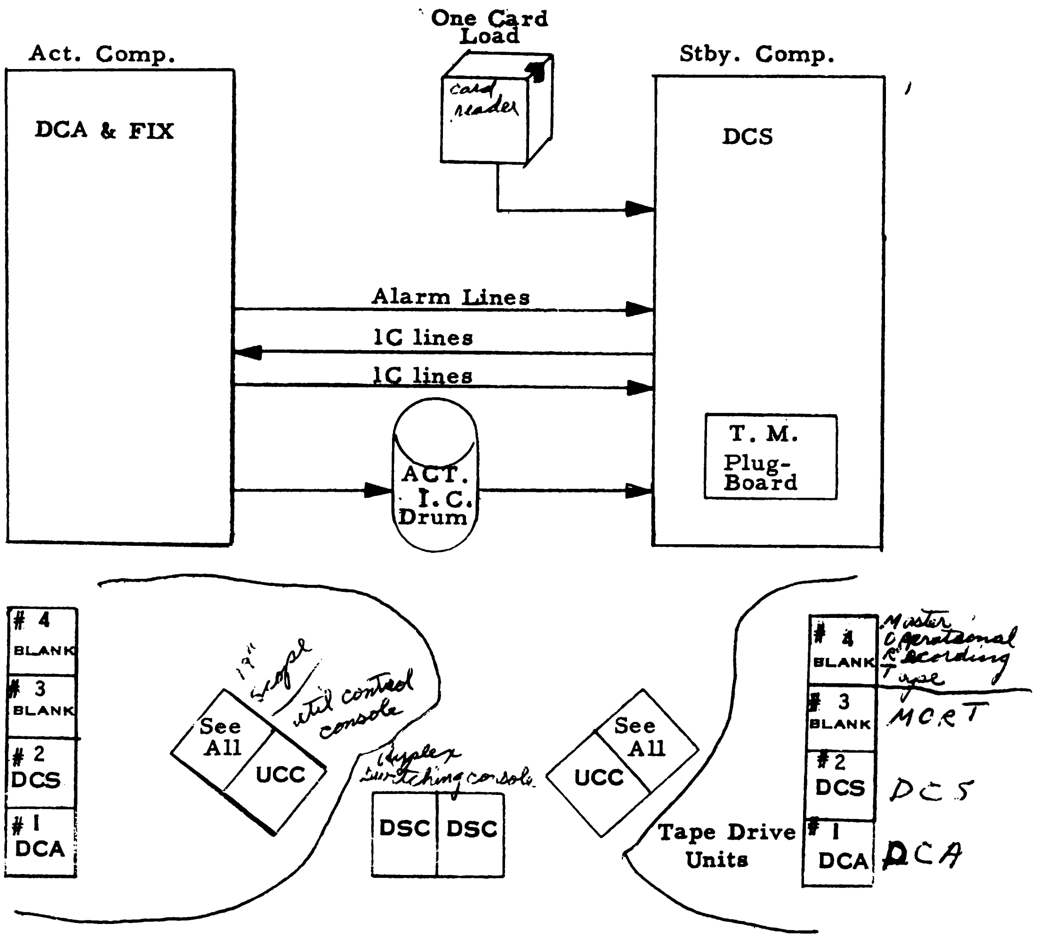


Figure 1-1

3. Air Route Traffic Control Centers (ARTCC): provide data concerning the flow scheduled air traffic in the area, i.e.; commercial airlines, military flights, etc.

4. Manual Inputs, Weather (WX), Bomarcs: present current picture of status of weapons systems, personnel, etc.

The information provided by these various agencies is transformed into digital data which can be used by other portions of the computer. This information is processed, analyzed, and is presented in visual form to the controllers by the Central Computer System.

The Display System provides for visual observation of the data processed by the Central Computer System. Situation Display handles air movements information, which requires the display to change as the information changes. Digital Display handles statistical information or information summaries that need not be changed as rapidly as situation displays.

The Outputs Systém provides information to various agencies which require this information to perform their part of the Air Defense mission. Some of the agencies which receive this information are:

1. Airborne interceptors through the Ground-to-Air Output System.

2. Other Direction Centers, Control Centers, etc., through the Ground-to-Ground Output System.

3. Manual Direction Centers, Airbases, etc., through the Teletype Output System.

CHAPTER 2 - BASIC PROGRAMS

INTRODUCTION

For proper operation, a digital computer must be able to solve a problem or series of problems in strict accordance with a known method of solution. Obviously, the computer cannot devise its own method of solution, therefore, all computer operations must be controlled. This is done with a series of instructions to the computer. This series of instructions is referred to as a program or computer program. This part of the text deals with some of the basic programs associated with the AN/FSQ-7 and AN/FSQ-8 computers.

DIRECTION CENTRAL ACTIVE (DCA)

The program which enables the AN/FSQ-7 to perform the function of Air Defense is the Direction Center Active (DCA) program. This program uses a total number of 90,000 memory locations. These are broken down into several dozen subprograms and approximately 200 tables. The exact number of instructions, programs, and tables varies as new versions, or "Models", of the DCA program appear.

How the computer is being used in relation to Air Defense can be determined by the use of four terms: Active, Standby, Simplex, and Duplex. At an operational site, the Active Computer will operate the Master Air Defense Program (DCA). When one computer is active, the other computer will be the Standby Computer. If the Standby Computer is cycling a program that is requesting transfer of certain information from the Active to the Standby Computer, the computers are in the Duplex Mode. The information that is transferred is to enable the Standby Computer to assume Air Defense if it becomes necessary with minimum lost time. If the computers cannot communicate with each other for any reason, they become Simplex.

The only function of the Active Computer is the operation of the DCA Air Defense Program. The Standby Computer has other duties in addition to "Standby", to take over the Air Defense function. These other duties involve preventative maintenance and data reduction programs which keep the computer reliable and determine the validity of the Air Defense previously performed.

Mention should be made here of an important little program, part of the DCA complex, called FIX. FIX senses parity errors, evaluates their impact on DCA performance, and if necessary or possible, "fixes" them. The variable impact of parity errors can be illustrated by considering on the one hand a parity error in an input radar data word, and on the other, a parity error in an instruction word. The radar data word can probably be discarded without any harm. If that word was a return on a track, the computer program will "dead-reckon" the track along its last known velocity and wait for the next return. If there is a parity error in an instruction word, the meaning of that instruction may be profoundly altered and the results would be catastrophic if the erroneous instruction were allowed to remain in the program. FIX would do something about this. First, it will set the alarm and print out the error. Then, it will attempt another read in. If the second read-in is correct, the computer will then continue with its program, but if it is incorrect, Parity Alarm will be generated and the computer will changeover to the Standby Computer.

FIX is one example of the "Fail Safe" principle, which is an important feature of any automated system. The premise is that no matter how reliable the physical components are, there is always the possibility of breakdown of one or more of these components; or that the system will be confronted with some situation that was not anticipated in the design. The more automatic a system is, the greater the risk that some malfunction will go undetected, doing serious and perhaps irreparable damage before a human monitor realizes it or has a chance to act. Built-in automatic error checks, such as parity circuitry or a program such as FIX, are of great value. The ability of a system to handle any situation must include the ability to recognize a situation which is beyond its own capability. When this situation occurs, the system should be able to alert maintenance and/or changeover to a standby system.

DIRECTION CENTRAL STANDBY (DCS)

A program which is to be run in the Standby Computer must be a program that will most efficiently and economically utilize its allotted time to diagnose and maintain a high reliability in the Standby Computer, and periodically analyze the Active Computer's performance. In addition, this Standby Program must monitor the Active Computer's alarms and be ready to assume operational Air Defense responsibilities should a malfunction occur in the Active Computer.

In order to coordinate this type of function, and implement the system by performing common operations, a control program is necessary. Thus, under normal operating conditions at a SAGE Sector, all standby maintenance programs will be governed by a Master Control Program. This program will be called DCS, and use as its medium of communications the Utility Control Console in the maintenance console area. All standby maintenance programs must be compatible with the Control Program.

DCS and all standby maintenance and utility programs are stored on tape and file protected. This tape will be located on tape drive Nr. 2.

To initiate standby operation, a single card is read into memory. This one-card program will check-load DCS into core memory and give it control. The Control Program (DCS) will then search the DCA Master Tape for the DCA Program, if in Duplex Mode.

The following refers to DUPLEX Mode.

DCS will review the DCA Master Tape (Standby Computer side, Tape Drive Nr. 1) and use a routine located on this tape to load the DCA Program on the appropriate drum fields. The Control Program also determines from the DCA Master Tape which fields are allocated for tables and used for temporary storage. It is the Standby Computers responsibility to maintain a valid DCA Program storage in the Standby Computer.

Completing the initial set up, the Control Program will then display appropriate information resulting from operator requests on the Utility Control Console (UCC), set up the safe data transfer and standby timing restrictions, and read the Standby Program from tape, if requested.

A.M.
DRUM

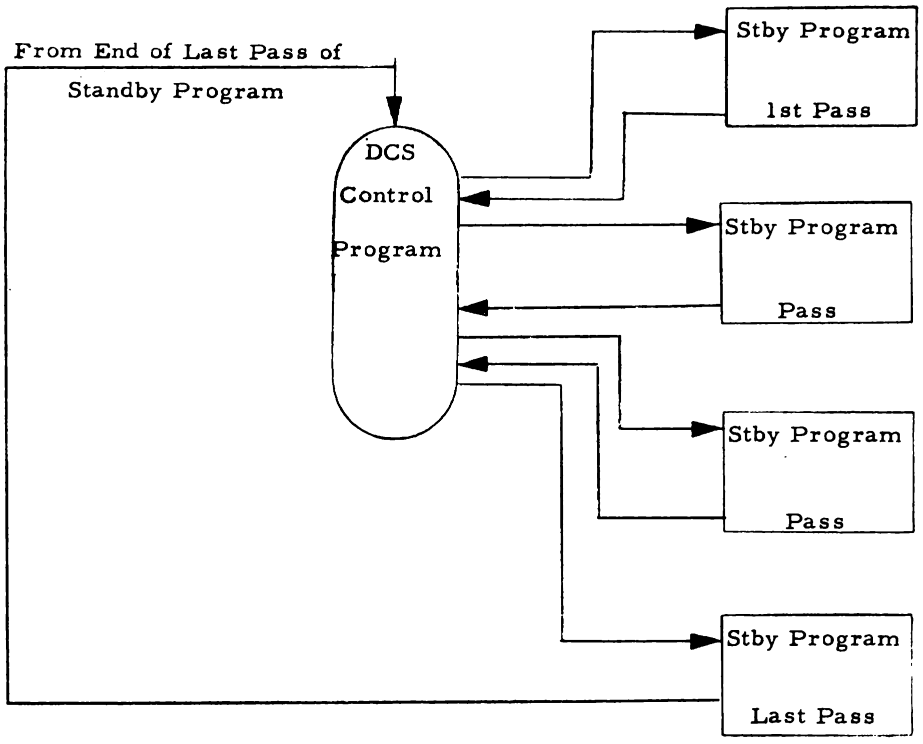


Figure 2-1. Maintenance Program Control

The Standby Program will run a complete "pass" of its program and branch back to DCS. A time check will be made to see if a safe data transfer is to be made. Whenever a safe data transfer is made, new time restrictions are set up. Refer to Figure 2-1.

Certain alarm conditions occurring in the Active Computer will cause an Alarm 1 condition in the Standby Computer and cause it to automatically branch into an emergency switchover condition, waiting for the duplex switch to be made active. When the Standby Computer is made active at the Duplex Switching console, the DCA start-over portion of the Air Defense Program (which is stored on drums) is read into core memory and given control.

The program communicates with the operator in three different ways:

1. Sense operator requests for switchover action, change in standby program selection, change in safe track data transfer frequency, etc., at the utility console.

2. Sense manual interventions for standby programs through the sense or A-B switches.

3. Prepare digital displays which will advise the status of operations and which signal required manual interventions by the operator and allow him to make logical decisions relating to the operation in progress.

SWITCHOVER

Switchover may be defined as the switching of the DCA Program operations from the Active Computer to the Standby, a function that makes the former Standby Computer, Active. Since the output of the DCA Program is a result of interaction between the program and simplexed computer I/O equipment, switchover involves a simultaneous switching of program and equipments.

There are only ~~three~~ TWO types of switchover which DCS prepares for:

1. Emergency Switchover is initiated in the Standby Computer by a malfunction in the Active Computer.

2. Scheduled Switchover is initiated in the Standby Computer by a pushbutton insertion on the Utility Control Console. Usually scheduled in advance, and accomplished when both computers are operating properly, scheduled switchover will transfer all tables and display drum fields from the Active Computer to the Standby Computer. The actual physical switching of the two computers is a manual operation performed at the Duplex Switching Console.

~~3~~. Alarm Nr. 1. An alarm generated by the Active Computer due to:

May Cause Emergency Switchover

a. Memory parity.

b. Inactivity.

c. Per 37

d. DCA being unable to startover, action will be the same as Emergency Switchover.

In order to maintain a close back-up of the Active Computer, DCS must accomplish the following:

1. Read in the DCA Program from Tape Drive Nr. 1 and store the programs on their assigned drum fields.

2. Periodically check the DCA Program drum fields for correctness.

3. At certain time intervals, as the Active Computer for a specified amount of information on the Air Defense situation and store this information.

NOTE: DCS will not try to perform an IC (intercommunications) transfer when an emergency switchover is requested. It is assumed that DCA is incapable of operating.

SAFE DATA TRANSFERS

There are four different types of IC transfers possible during the overall operation of DCS and switchover. The types of transfers are:

1. SAFØ transfer via IC Drum. This is a maximum of 30,000 words of data which is normally considered to be a minimum acceptable amount of active data that the Standby Computer needs to be prepared for switchover. Made automatically whenever DCS is operating in one of the Duplex Modes.

2. Table Transfer.

3. Display Fields Data (TD and DD).

4. Display Fields Data (RD).

Transfer of Table data, and Display Fields data occurs only when a scheduled switchover is requested by DCS. This is a maximum of $100{,}000_8$ words of data.

The frequency that safe data transfers are made also differ according to the different modes the computers are in. The three different modes and the frequency of transfers are as follows:

1. Alert Mode-every 30 seconds.

2. Normal Mode-every 2 1/2 minutes.

3. Simplex Mode-none.

CHAPTER 3 - MAINTENANCE PROGRAMMING

INTRODUCTION

A Maintenance Program is any program designed to indicate whether or not the computer is able to correctly perform its intended design function. If improper operation occurs the program should be able to specify the equipment responsible for the errors.

Many programs such as DCA or Utility programs may give some indication that an error is present, but provide little or no aid to the maintenance effort. The primary function of any maintenance program is to maintain computer reliability, i.e., to locate any existent or impending failures during scheduled maintenance periods, so that no machine failures occur during operating time.

To adequately perform this function the Maintenance Program must attempt to treat all circuits in a manner which approximates the ultimate applications of the computer, and to treat these circuits as strenuously as they will be treated by any other program. This criteria can only be met by treating the computer circuits as strenuously as possible.

A maintenance program should be used as a tool; much as an oscilloscope or screwdriver is used. Therefore it is necessary to know the capabilities and limitations of programs, not the step-by-step construction of a program.

BASIC REQUIREMENTS OF A MAINTENANCE PROGRAM

1. Validity -- A program is valid to the extent that it checks the equipment it is designed to check, and to the extent that it can accurately locate and specify to maintenance personnel a malfunction in the equipment area.

2. Reliability -- A program is reliable to the extent that it consistently locates and specifies malfunctions.

3. The third characteristic of a maintenance program is equally important but more difficult to define. This characteristic results from the computers being extremely fast, very reliable, but incredibly stupid. They perform exactly as ordered, which may not always be exactly as intended, i.e., they do what you tell them to do; not what you mean them to do.

The program used to maintain a computer will only be as logical, accurate, and comprehensive as is the programmer who creates the program. It may be valid and reliable, yet have little value as a maintenance tool because it is not sufficiently rigorous and thorough to completely test the equipment.

To be 100% comprehensive, the program must be capable of locating any and all failures regardless of the type, nature, frequency, location, etc., of the failure. Since many types of failures are first encountered after the computer has been in operation for some time, it is impossible to be 100% comprehensive at the time a program is designed. The program at the time it is designed will thus be as comprehensive as is permitted by the training and experience of the programmer.

TYPES OF FAILURES

Definition of failures: Any improper behaviour of a circuit not inherent in the equipment design.

1. Catastrophic -- This is a steady-state, continuously present, failure.

2. Intermittent -- This is a failure which is not continuously present, but occurs only once or at random intervals. Intermittent failures are extremely difficult to locate and frustrating, because they appear and disappear at random. They can manifest themselves as an inconsistent set of symptoms which make them nearly impossible to locate.

3. Machine state -- These are failures present only under certain conditions.

   a. After a certain sequence of instructions.

   b. After a particular instruction followed by a specific delay in time.

   c. At a particular machine duty cycle, or pulse repetition rate, etc.

The only sure way to detect such failures is to sequence all computer circuits through every possible combination of states. However, such a sequence would require several years of continuous machine time, assuming the programmers could conceive of all possible states and construct the required program.

FAILURE DETECTION

If a program runs once successfully, this does not necessarily lead to a firm conclusion since several alternate possibilities exist.

1. Possible conclusions if a program operates successfully:

   a. All the circuits checked are O.K.

   b. A circuit error exists, but did not occur during the instant of time in which this circuit was checked.

   c. An error occurred, but this particular type of error was not checked for by the program.

   d. A circuit error exists, but the program did not follow the peculiar sequence of events necessary to show up the error.

   e. The error indication utilized by the program failed, leading the program to believe that no error occurred.

f. An error exists but the section of the program which checks this type of failure did not operate for reasons unknown.

2. Because of the alternate possibilities that exist, it is necessary to:

a. Use multiple error indications.

b. Use controls and indicators to insure that all sections of the program operated.

c. Perform every possible check conceivable on this circuit.

d. Check for every possible type of circuit error in every test relating to the specified circuit.

e. Perform every test many times.

PROGRAMMING TECHNIQUES

There are many possible programming techniques which may be utilized to locate a malfunction. Five of the major techniques will be discussed here with their corresponding assumptions and limitations.

1. Start Small

This involves an initial check of as small an amount of circuitry as is possible. If this circuitry is operating properly, it is used in testing another circuit, etc. This process continues until a check has been performed on all circuitry for which a programmed check is possible. The assumptions and limitations are:

a. The minimum amount of circuitry required by a programmed check is not a small amount.

b. If a failure occurs during the first check, diagnosis is not easy, for other checks cannot be accomplished, since they depend upon the initial circuitry being in operation order.

c. An intermittent error may not occur during the time a circuit is being checked. If this error does appear when the circuit is being used to test other circuits, the test's validity is questionable.

d. The error may not be the result of a circuit component but be caused by interaction between circuits during a specific sequence of operations. This is beyond the scope of this type of program since it would require that every conceivable check be performed. This generally is not possible without using a great deal of circuitry.

2. Start Big

In this approach the entire system may be used to check a particular circuit or portion of the computer at a time.

The assumptions and limitations are:

a. If some basic error exists, it may be impossible for the program to maintain control and operate.

b. This type of program is very large and requires a great deal of time to construct and debug.

c. It is impossible to adequately check those portions of the computer required by the program to maintain control since so much circuitry is required for this purpose.

d. An intermittent error may disrupt program operation but yield no clues as to the identity of the error.

3. Multiple Clue Approach

Once an error is detected, the program attempts to obtain the same error using varying sequences of instructions. If the failure can be detected in a multiplicity of ways, it is only necessary to locate the common conditions, etc., to locate the failure. The assumptions and limitations are:

a. Hard to accomplish because it involves exhaustive research and programming sequences.

b. If an error can only be detected by one sequence of instructions, the program is at a loss for a common factor smaller than the computer itself.

c. An intermittent error may cause the program to lose control without giving any indications as to the nature of the error.

4. Process of Elimination

Certain types of failure cannot be directly analyzed. However, it is possible to employ routines to vindicate one area after another, and by a process of elimination, infer the error to be in the remaining area. The assumptions and limitations are:

a. All areas may check O.K. when individually checked because the error results from circuit interaction or timing conditions.

b. We assume the error to be in the remaining area. It may not be, particularly if the error is of an intermittent nature.

5. Program Assumptions

A maintenance program must assume the following:

a. That only a single failure, or unrelated multiple failures are present.

b. Control circuitry is operating correctly.

c. Errors can be isolated.

d. The failure is the result of a circuit error; not a design error.

TYPES OF MAINTENANCE PROGRAMS

1. The older maintenance programs were classed in two categories; Reliability and Diagnostic. The Reliability type checked only to see if an error existed. The Diagnostic type was used after the Reliability program to localize the trouble.

The disadvantages of these programs were:

a. The Reliability program indicated a failure existed but the Diagnostics could find no errors.

b. Reliability programs were not comprehensive enough and missed many errors.

c. Too many programs had to be run to isolate an error.

2. The more recent programs combine reliability and diagnostic functions in a single program, trying to eliminate these disadvantages. No consideration is extended toward making the program routines specifically reliability or diagnostic in nature. Rather, the program is concerned with attempting to locate every error which could occur, then tracing any error detected to smaller and smaller areas until the error locations can be specified. The four specific types of programs are as follows:

a. Overall Programs

Major application of this program type is the Drum, Inputs and Outputs areas. They start Big using the entire area under test, and then use the "Multiple Clue" and "Elimination Techniques" for error detection. An overall program consists of the reliability, diagnostic and marginal checking routines necessary to check an equipment area. The advantages are:

(1) Only one program necessary for a given equipment area.

(2) Only one card deck to load and write-up to refer to.

(3) More reliable error detection and error analysis is afforded.

(4) No indecision by the operator about which program to run.

b. Margin Line Routines (MLRs)

MLRs are a special type of program, each consisting of these sections.

(1) The Setup and Clear Routine -- Operates before margins are applied and insures that computer registers are properly set up for the test to be made.

(2) The Marginal Workout Routine -- operates with margins applied to one MC line and tests the circuitry on that MC line, then stores the results. This is a diagnostic type routine.

(3) The Check Routine -- after excursion is removed, the Check Routine interprets the results stored by the workout routine and prints the necessary data, indicating success or the nature of the failure.

The advantages of MLRs are:

(1) There is no question as to which program should check a particular circuit and how thorough a check should be made.

(2) Since margins are applied to a single line, the workout routine can usually be written such that failures in the equipment under test will not affect the validity of the results.

(3) MLRs have been written which run to completion and afford accurate error detection even when circuits fail under margins, and MLRs are giving accurate indications at excursions of 50 and 60 volts where earlier programs were often unable to isolate failure above 25 or 30 volts. This factor enables the MLRs to locate intermittent or unusual circuit malfunctions not detected at the lower excursion levels.

c. Automatic Diagnostic of System (ADIOS)

This system of diagnostic programs uses "Start Small" and "Multiple Clue" techniques. ADIOS will be a series of diagnostic programs which will test all of Central Computer, Card Machines and Drums. The technique here is to start very small, assuming that none of the circuits are working properly. This has been designed specifically for trouble shooting after an unscheduled switchover. Takes about 3 minutes to run.

d. Analysis Under Operational Conditions (AUOC)

This is a series of maintenance programs which really "Start Big", using the entire computer system for sequence checks of different equipment area. These programs operate under a master control and timing program in the same manner as the DCA system.

(1) The primary advantage of the AUOC system is that it permits all of the specified equipment areas to be checked or repaired simultaneously, thereby increasing the amount of maintenance which may be performed during a given period.

(2) A second advantage is that operation of AUOC after maintenance periods will enable a quick check of all equipment areas before releasing the computer to a customer.

(3) A third advantage is that AUOC, by utilizing the entire system, may be able to detect miscellaneous system (machine state) failures usually found by DCA but not located by maintenance programs.

MARGINAL CHECKING

1. Marginal checking is a testing technique in which components are artificially aged by voltage variations on the circuits.

2. This will assist in locating circuit changes before they progress to a point where normal machine operation is impaired.

3. The voltage variations themselves find no failures. It is the program routine that finds the failures, using voltage variations as a tool.

4. The types of margins are as follows:

a. Prescribed -- In this method, specified variations are made, with the expectation that the associated circuits should tolerate these excursions without failing during the program run.

NOTE: Running with prescribed margins requires only a single program pass at the specified variation; however it furnishes no information concerning the following:

(1) Is the circuit deteriorating?

(2) What is the present quality of this circuit?

(3) How good is it now compared to when it was new?

(4) How fast is the circuit deteriorating?

We actually know only one thing; i.e., the circuit did not fail in a manner the program could detect during the interval of time it was checked by the program.

b. Failure -- Voltage is varied to a point where the circuit fails.

NOTE: In this approach, each voltage is varied to the point which causes a failure GREAT ENOUGH TO BE DETECTED BY THE PROGRAM operating while the excursion is applied. Types of failures not checked for by the program may occur at variations lower than that at which an error is detected. This will be determined by the validity and comprehensiveness of the program.

This type of margin will require at least two program passes as an absolute minimum. To locate the specific failure point, the program must have one successful pass and one pass with a failure.

5. Specific Considerations

There are some circuits which may not be marginally checked, simply because no provisions have been made for voltage variation in these circuits. Margins may be of little or no assistance in locating failures in these circuits.

The voltage variation is applied to aggravate circuit changes which could cause a failure. However, an unusual effect of this variation is that certain kinds of errors, existent when voltages are normal, may be corrected for the duration of the excursion by changes in circuit parameters resulting from the voltage variations. Thus, the program should run a reliability pass before excursions are applied to determine if any failures exist at normal operating voltages. As added insurance, apply both positive and negative excursions to all lines; if a positive excursion conceals the defect, it should be aggravated by a negative excursion and vice versa.

DESIRED CHARACTERISTICS OF MAINTENANCE PROGRAMS

Those programs which are easily understood and easy to operate are more often used with confidence by maintenance personnel. In order to make better programs, these items are taken into account:

1. Speed and simplicity in operation.

2. Protection against operator errors.

3. Test or routine identification.

4. Error data -- give operator everything possible about the error.

5. Add options or special features.

6. Reloading Provisions - Drum storage of program.

7. Completeness - A thorough program checks the operator, the machine and itself, by covering for operator errors, providing manual routines, printouts, program flexibility, control options, etc.

CHAPTER 4 - THE COMPUTER WORD

COMPUTER WORD DESCRIPTION

A digital computer must be able to recognize the instructions it receives. Therefore, these instructions must be presented in a combination of digits, since this is the only "language" the computer can interpret. The AN/FSQ-7 and AN/FSQ-8 utilize a system of pure binary numbers for all operations; therefore, it is mandatory that computer instructions are coded in binary. Data must also be in binary form. To allow air defense data and instructions to be processed by the same circuits, a standard form of layout is necessary for both types of numerical information. This layout is referred to as a computer "word" and is composed of 32 bits in the AN/FSQ-7 and AN/FSQ-8. The computer word is divided into two half-words, simply referred to as the "left half-word" and the "right half-word". The reason for this division is that the Central Computer System has dual arithmetic elements, allowing two operations to take place simultaneously. The arithmetic elements are also designated as left and right, with the left arithmetic element processing the left half word of a computer word and the right arithmetic element processing the right half. Figure 4-1 shows the word layout for the AN/FSQ-7 and AN/FSQ-8. Each half-word consists of 15 magnitude bits, plus one bit for sign. In addition, an extra bit is included at the extreme left of the word. The bit is generated by the Central Computer and is used to check the information transfer to and from the Central Computer. This bit, called the parity bit, is not used in arithmetic computations and does not have to be assigned a position for instructions or data. It is included to show that a word stored in core memory actually consists of 33 bits - the two half-words plus the parity bit. Often, it is necessary to refer to single bits within a computer word; for this reason, the letter L or R is placed before the bit position to designate the proper half-word. For instance, a reference to L9 means that we are concerned with the ninth magnitude bit of the left-half word.

	Left Half-Word																Right Half-Word															
P	LS	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	RS	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Computer Word Layout

Table 4-1

1. INSTRUCTION WORDS

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

LEFT HALF WORD

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

1. Class Code

   a. Bits L/4-6

   b. Designates one of eight Instruction Classes.

2. Variation Code

   a. Bits L/7-12

   b. Designates specific instruction of the class, Programming Card shows Instruction class and variation octal code.

3. Index Selection Code

   a. Bits L/1-3

   b. Designates Index Register to be:

      (1) Loaded in (XIN) instruction

      (2) Sensed and reduced in (1 BPX)

      (3) Used for address control in (1 ADD) (1 MUL) etc.

      (4) Note: Programmers Card indicates when indexing can be used.

4. Special Option Bit Codes

   a. L/15=1

      Allows pulse to check <u>right</u> overflow.

   b. L/14=1

      Allows pulse to check <u>left</u> overflow.

   c. L/13=1

      Suppress overflow alarms (D.C. Site). Needed to stop false overflow alarms in <u>address modification</u>.

   d. L/12=1

      (1) D.C. Site only.

      (2) Provides 17 bit operation in some instructions. Note: This bit will be included in the operation code.

5. Index Interval Bits

a. Bits L/10-15: these bits can be used since they are necessary in the BPX, BSN, SDR, SEL, and PER instruction codes. Note: The dash (-) in their instruction code on the Programming Card.

b. With instructions TOB and TTB L10 is not part of the index interval code.

RIGHT HALF WORD

The right half word is referred to as the Director or Address Portion of the word. It specifies the Memory Unit, and Memory Address Selection for the different sites. In a DC the Selection would be done thus:

1. Bits L/S & R/S-15 specified memory unit.

a. $(0.00000)_8$ through $(1.77777)_8$ ---- Memory Unit Nr. 1 $(256)^2$. Memory Nr. 1 is selected if LS=0.

b. $(2.00000)_8$ through $(2.07777)_8$ ----- Memory Unit Nr. 2 $(64)^2$. Memory Nr. 2 is selected if LS=1 and RS-R11 contains at least one zero.

c. $(3.77760)_8$ through $(3.77777)_8$ ----- Test Memory Unit. Test Memory is selected if LS=1 and RS-R11 contain all "1's".

2. Memory Address

a. Bits R/S-15 specifies particular address of Memory Nr. 1.

b. Bits R4-15 specifies particular address of Memory Nr. 2.

c. Bits R/12-15 specifies particular address of Test Memory.

In a CC site however this same address selection does not ~~hold true.~~

Correct

The memory selection there would be:

~~1. Memory Nr. 1~~	~~0.00000-0.07777~~
~~2. Memory Nr. 2~~	~~0.10000-0.17777~~
~~3. Test Memory~~	~~0.20000-0.20017~~ 0.20000-0.20017

The right half of the instruction word also has other functions. Some of these are:

a. Used to specify the number of words to be transferred during an I/O operation (RDS, WRT).

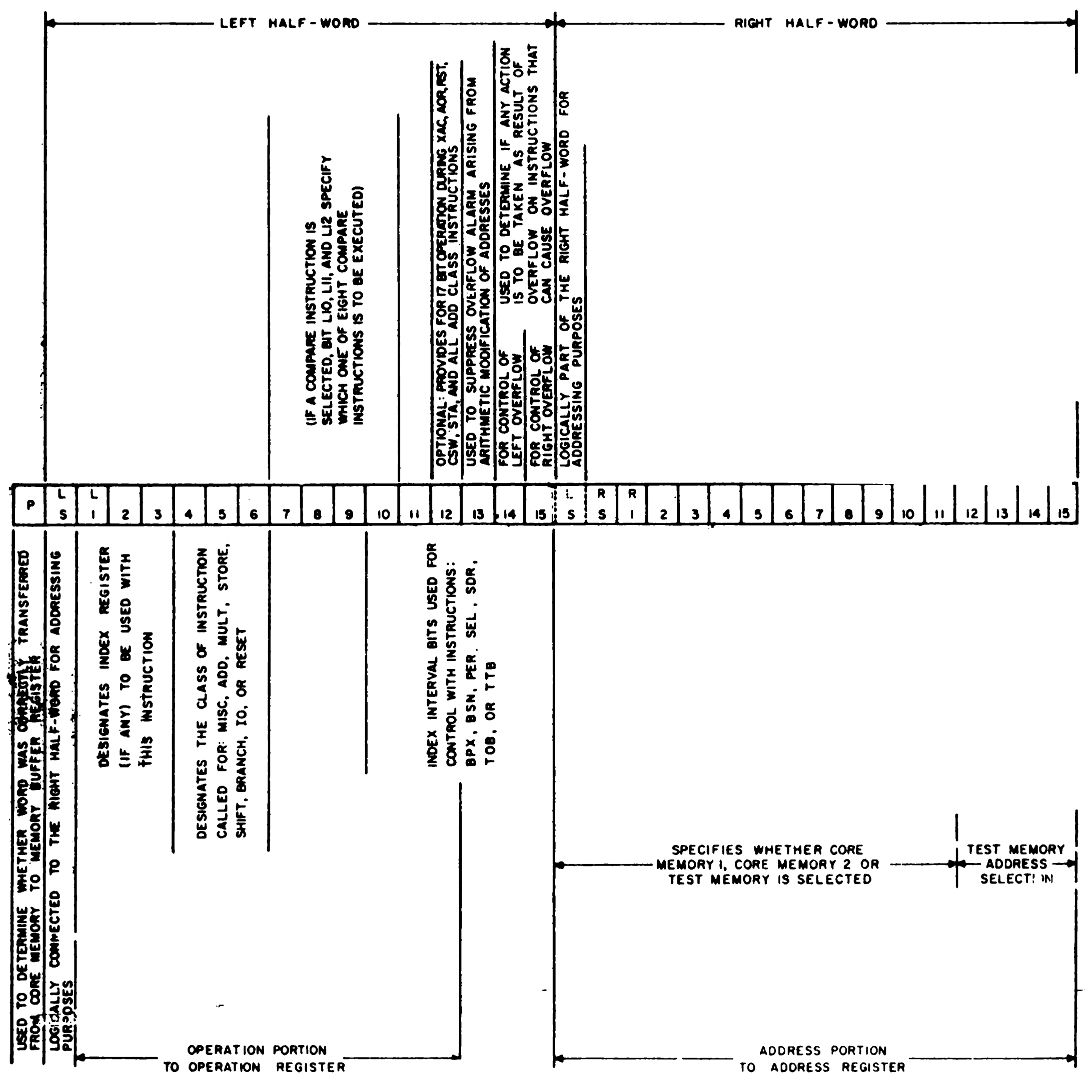

LEFT HALF-WORD
RIGHT HALF-WORD
P
L S
L 1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
L S
R S
R 1
USED TO DETERMINE WHETHER WORD WAS CORRECTLY TRANSFERRED FROM CORE MEMORY TO MEMORY BUFFER REGISTER
LOGICALLY CONNECTED TO THE RIGHT HALF-WORD FOR ADDRESSING PURPOSES
DESIGNATES INDEX REGISTER (IF ANY) TO BE USED WITH THIS INSTRUCTION
DESIGNATES THE CLASS OF INSTRUCTION CALLED FOR: MISC, ADD, MULT, STORE, SHIFT, BRANCH, IO, OR RESET
(IF A COMPARE INSTRUCTION IS SELECTED, BIT L10, L11, AND L12 SPECIFY WHICH ONE OF EIGHT COMPARE INSTRUCTIONS IS TO BE EXECUTED)
INDEX INTERVAL BITS USED FOR CONTROL WITH INSTRUCTIONS: BPX, BSN, PER, SEL, SDR, TOB, OR TTB
OPTIONAL: PROVIDES FOR 17 BIT OPERATION DURING XAC, AOR, RST, CSW, STA, AND ALL ADD CLASS INSTRUCTIONS
USED TO SUPPRESS OVERFLOW ALARM ARISING FROM ARITHMETIC MODIFICATION OF ADDRESSES
FOR CONTROL OF LEFT OVERFLOW
FOR CONTROL OF RIGHT OVERFLOW
USED TO DETERMINE IF ANY ACTION IS TO BE TAKEN AS RESULT OF OVERFLOW ON INSTRUCTIONS THAT CAN CAUSE OVERFLOW
LOGICALLY PART OF THE RIGHT HALF-WORD FOR ADDRESSING PURPOSES
SPECIFIES WHETHER CORE MEMORY 1, CORE MEMORY 2 OR TEST MEMORY IS SELECTED
TEST MEMORY ADDRESS SELECTION
OPERATION PORTION TO OPERATION REGISTER
ADDRESS PORTION TO ADDRESS REGISTER

Figure 4-1. Instruction Word Analysis

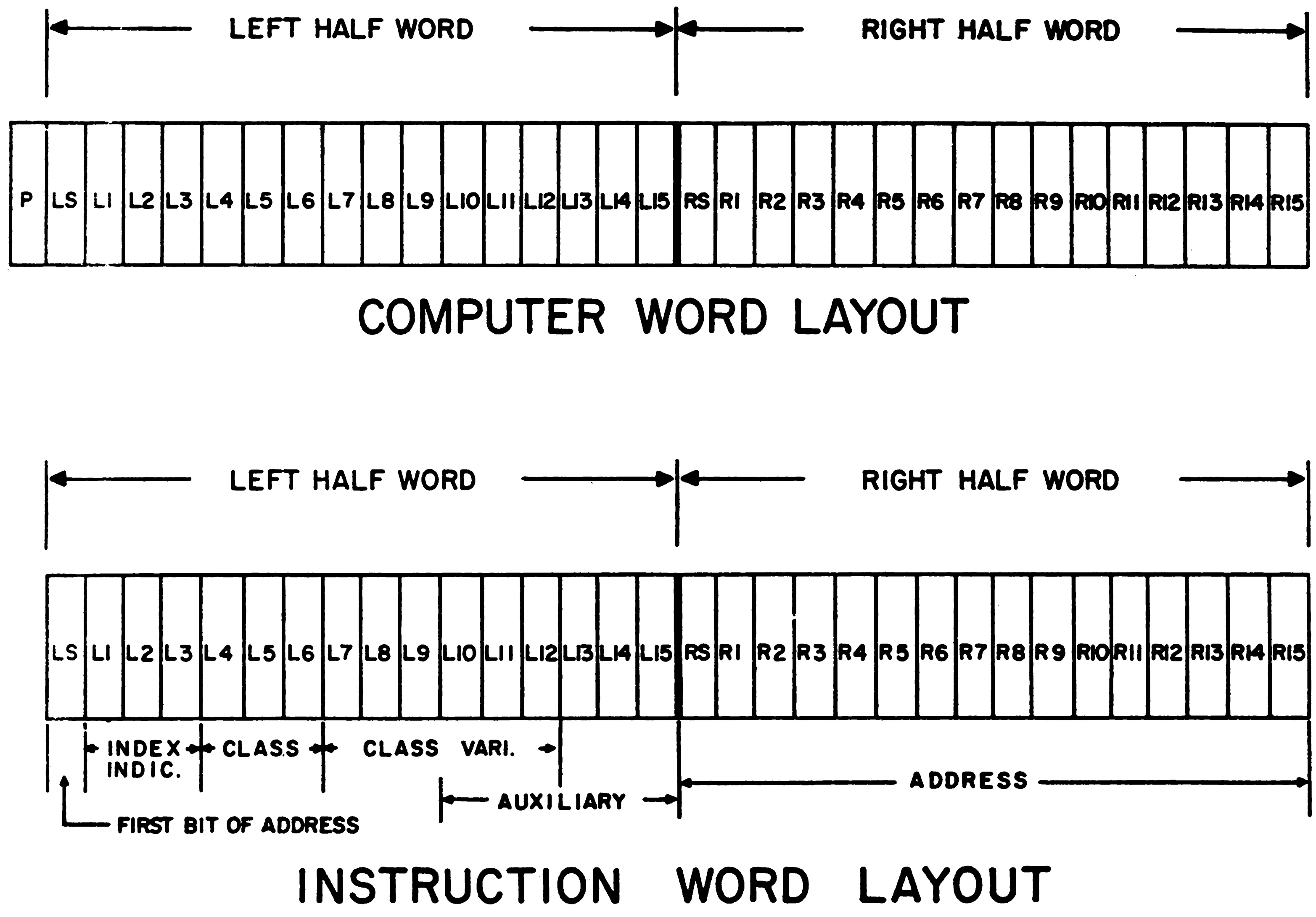

LEFT HALF WORD
RIGHT HALF WORD
P LS L1 L2 L3 L4 L5 L6 L7 L8 L9 L10 L11 L12 L13 L14 L15 RS R1 R2 R3 R4 R5 R6 R7 R8 R9 R10 R11 R12 R13 R14 R15
COMPUTER WORD LAYOUT
LEFT HALF WORD
RIGHT HALF WORD
LS L1 L2 L3 L4 L5 L6 L7 L8 L9 L10 L11 L12 L13 L14 L15 RS R1 R2 R3 R4 R5 R6 R7 R8 R9 R10 R11 R12 R13 R14 R15
INDEX INDIC.
CLASS
CLASS VARI.
AUXILIARY
ADDRESS
FIRST BIT OF ADDRESS
INSTRUCTION WORD LAYOUT

Figure 4-2

b. Loaded into specified index register during an XIN instruction.

c. Added to the specified index register during an ADX instruction.

d. R/5-15 loaded into the Drum Control Register during an SDR or SEL instruction.

e. R/10-15 Loaded into the Step Counter every PT-7

(1) Only significant if a shift class or SLR instruction is programmed.

(2) Designates the number of shifts to be performed.

2. DATA WORDS

Just as with instruction words, a data word consists of two half-words which are processed simultaneously by the dual arithmetic elements of the Central Computer System. Data received from various sources will have various forms, so the most general layout of a data word is what we shall consider. This layout conforms with the word layout down in Figure 4-2. The sign bit of a data word actually indicates, if the quantity is positive or negative, whereas it serves only as a code in an instruction word. Therefore, we have two half-words of 15 magnitude bit and a sign bit. All numbers in the AN/FSQ-7 and AN/FSQ-8 are treated as fractions. This restriction is placed on data primarily so that the multiplication of two numbers will always result in a product smaller than either of the numbers, thus positively avoiding overflow. Another consideration is that the results of a multiplication may be stored from one register without a loss of bit significance. As a result of this restriction, all numerical data within the AN/FSQ-7 and AN/FSQ-8 lies somewhere between the limits of +1 and -1. Data which has an actual magnitude of more than unity must therefore be scaled (factored) so that it appears in fractional form.

CHAPTER 5 - CENTRAL COMPUTER SYSTEM

The function of the Central Computer System is to process algebraically and logically the military tactical data supplied to it by the Input System via the Drum System; transferring the results back to the Drum System for subsequent distribution to the Display and Output Systems.

In addition to processing data, the Central Computer System operates as the main control for the Central. As data is being processed the Central Computer generates signals, as instructed, and sends them to the Drum System for utilization by the Input, Display, and Output Systems. These signals control the flow of data between systems, initiate operational cycles, set up control circuits for the coming operations, and in general, synchronize the actions of each system with those of the Central Computer System.

Functionally, the Central Computer System is divided into seven groups; Figure 5-1:

1. Instruction Control element

2. Selection control element

3. Program control element

4. Arithmetic Element

5. I/O Element

6. Memory Element

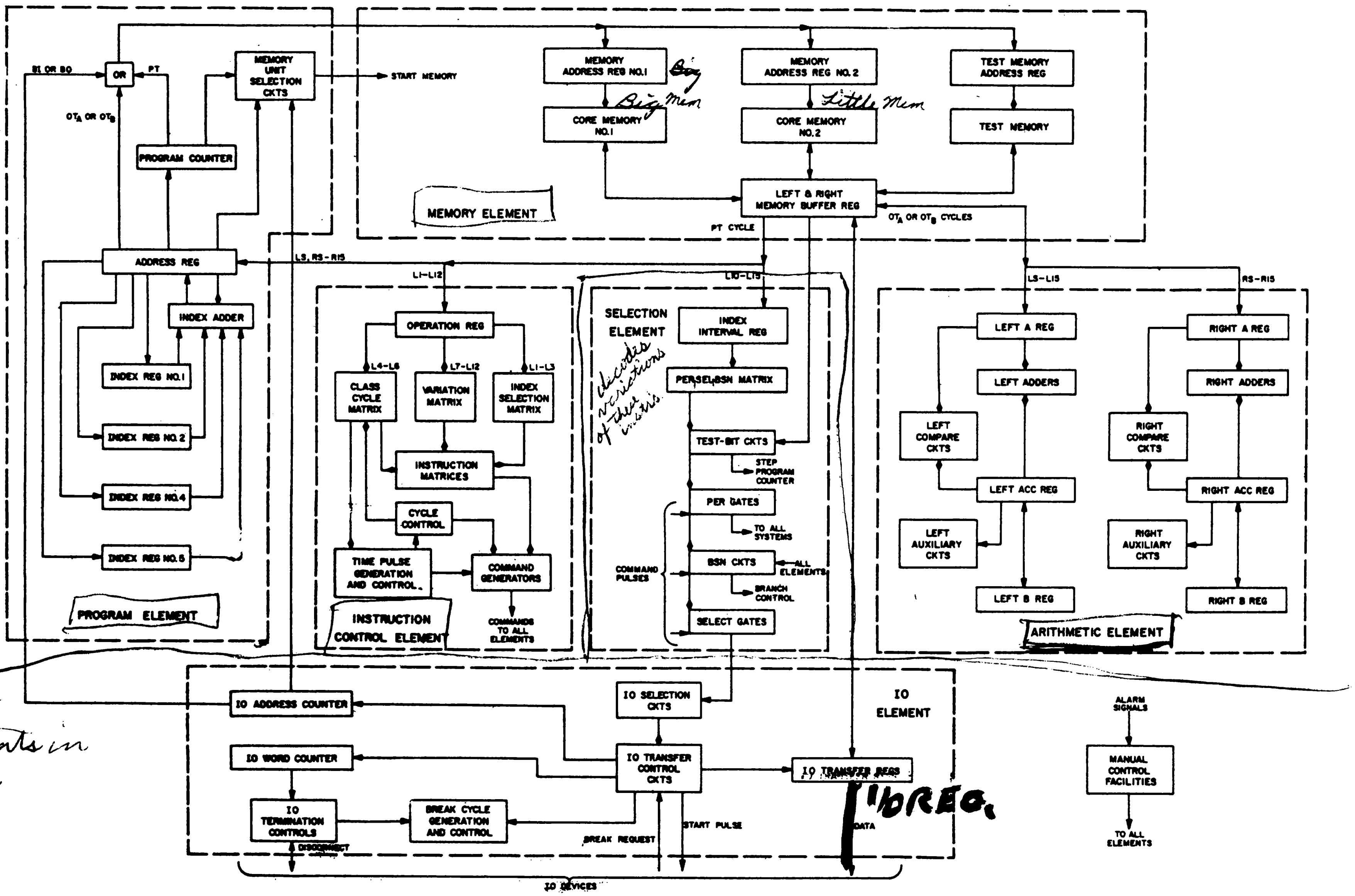
7. Manual controls and computer indicators

The instruction, selection, and program control elements sequence, coordinate, and control all processes in or allied with the operation of the computer. The entire instruction control element and a part of the program control element govern internal computer operations, while the selection control element and the remaining part of the program control element govern external computer operations primarily connected with IO devices.

The arithmetic element performs arithmetic processes using numerical data as instructed by the program. The associated IO units are commercial IBM machines modified for use with the computer and are not to be confused with the Input and Output Systems of the Central. The manual control facilities enable personnel to start, operate, shut down, and service the computer.

1. INSTRUCTION CONTROL ELEMENT, Figure 5-2.

The instruction control element accepts a portion of the instruction word from the memory element and decodes it. The decoder is capable of recognizing each of the various binary codes that indicate what operation is to be performed. More specifically,



Know 4 elements in squares

Figure 5-1

bits L1-L12 of the instruction word are transferred to a register, known as the operation register, which determines the class of instruction and the variation to be executed. The output of the operations register and its associated circuitry constitutes control signals which are sent to all the other elements of the Central Computer at specific times (depending on the instruction) and cause the instruction to be executed. The instruction control element is divided into three sections: The instruction decoder, the pulse generator and control, and the command generators.

The pulses generated by the instruction control element originate in a 2-mc crystal-controlled oscillator. The sine-wave output of the oscillator is clipped and shaped into positive pulses ranging from 20V to 40V above ground level in amplitude, 0.1 usec wide at the base and having a 0.5 usec repetition time. The pulses are distributed among a number of circuits, in which they perform many functions.

The following units are used for instruction decoding:

1. Operations register (0.3.1) bits L1-L12.

2. Cycle Control (0.3.1).

3. Class-Cycle matrix (0.3.1).

4. Variation Matrix (0.3.1).

5. Index selector matrix (0.3.1).

6. Instruction Matrices (0.3.2).

The six units in the above list receive their information from the left memory buffer register during the PT cycle of the instruction and select the class and specific variation.

1. Operations register

Bits L1 through L12 of an instruction word are transferred from the left memory buffer to the operations register during the PT cycle.

2. Cycle Control

The Cycle control employs six flip-flops to carry out four functions. The six flip-flops are:

a. PT-OT flip-flop

b. A-B flip-flop

c. Branch flip-flop

d. IO Interlock

e. CSW Control flip-flop

f. CSW Gate flip-flop

3. Class Cycle Matrix

The Class Cycle Matrix combines the information contained in the class-selection portion (bits L4, L5, and L6) of the operations register with the outputs of the PT-OT flip-flop and the A-B flip-flop to provide 17 combinations of class and cycle output levels.

4. Variation Matrix

The variation matrix decodes bits L7 through L12 of the instruction word to condition one output line specifying which variation of the selected class of instructions is desired. Output of the variation matrix and the index selector matrix are combined in the AND circuits of the appropriate instruction matrices; the output are d-c levels that condition command generators.

5. Index Selector Matrix

This central computer system contains four index registers (no.'s 1, 2, 4, 5), which may be used in the process of instruction indexing. In addition, the right accumulator register of the arithmetic element is also used as an index register, under certain circumstances, and functions as index register 3.

Instruction indexing is a process which automatically modifies the address part of an instruction without changing the basic instruction as it is stored in the core memory. Not all instructions are indexable. If the indexing process is to be used, the address portion of the selected instruction is modified immediately after the instruction is decoded.

The index selector matrix may select an index register for any one of the three purposes: the contents of the index register may be added to the address register during the execution of an indexable instruction. The index register may be reset to a new number during reset-class instructions; or the contents of the index register may be inspected and possibly modified during the execution of a BPX instruction.

Selection of an index register is effected by the output of the index selector matrix. There are six possible outputs: IX0 through IX5. When no index register is specified for the instruction in progress, IX0 is selected.

6. Instruction Matrices

The output of the class-cycle, variation, and index selector matrices are fed to the instruction matrix where they are combined to produce the levels needed to execute the selected instruction. These levels are applied to the necessary command generators. When the conditioned command generators are sensed by pulses from the time pulse distributor, they allow the pulses to emerge as commands. The commands are issued in proper sequence and at the proper time to accomplish the objective of the selected instruction. The same process is followed for all 63 instructions which the Central Computer is capable of performing,

In many cases, several different instructions require the same command at exactly the same pulse time. In this case, one command generator fed by an OR circuit is provided. The several d-c instruction lines are connected to the OR circuit so that any one of the lines can condition the selected command generator.

7. Command Generators

The functional arrangement of the instruction control element circuit is shown in Figure 5-3. The instruction decoder receives coded instruction pulses from the memory element by way of the memory buffer register. As a result of decoding by the various control and instruction matrices, a +10V level is applied to the suppressor grids of the command generators required for the execution of the instruction, and a -30V level to the suppressor grid of the remaining command generators. A command generator to which the +10V level has been applied is said to be conditioned; one to which the -30V level has been applied is called de-conditioned. Simultaneously with the generation and routing of these levels the pulse generation and control section produces three types of repetitive timing pulses of 0.1 usec duration.

Specific timing pulses are applied to the control grids of individual command generators. If a command generator is conditioned when it is strobed by a timing pulse, a command pulse is generated.

The commands issued by the command generator tubes either activate other parts of the instruction control element or are distributed among the other elements of the Central Computer System. The majority of the commands are transmitted to the Arithmetic element, where they initiate and control arithmetic operations.

Other commands are:

a. Applied to the program element to coordinate the overall operation of the Central Computer System.

b. Routed to the IO element to set up control which govern the transfer of information into or out of the IO devices.

c. Applied to memory element to initiate the transfer of information between Memory and the Central Computer.

Commands which are generated during every machine cycle are called common Commands. A complete list of all commands is shown in the Logic Index Appendix A Section 2.

2. SELECTION CONTROL ELEMENT

The selection elements are composed of various groups of logical circuits which enable us to perform a number of operations on all the systems in the AN/FSQ-7 and AN/FSQ-8. For instance, we can start an input test pattern generator or cause the

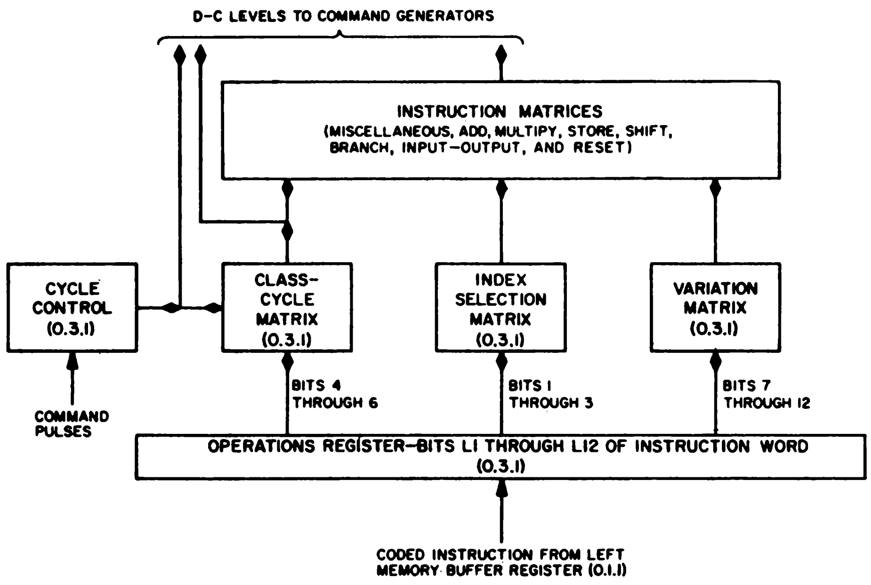

D-C LEVELS TO COMMAND GENERATORS
INSTRUCTION MATRICES (MISCELLANEOUS, ADD, MULTIPY, STORE, SHIFT, BRANCH, INPUT-OUTPUT, AND RESET)
CYCLE CONTROL (0.3.1)
CLASS-CYCLE MATRIX (0.3.1)
INDEX SELECTION MATRIX (0.3.1)
VARIATION MATRIX (0.3.1)
COMMAND PULSES
BITS 4 THROUGH 6
BITS 1 THROUGH 3
BITS 7 THROUGH 12
OPERATIONS REGISTER—BITS L1 THROUGH L12 OF INSTRUCTION WORD (0.3.1)
CODED INSTRUCTION FROM LEFT MEMORY BUFFER REGISTER (0.1.1)

Figure 5-2. Instruction Decoder, Simplified Block Diagram

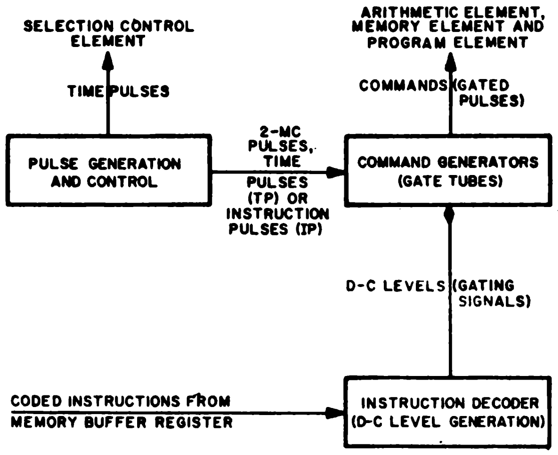

SELECTION CONTROL ELEMENT
TIME PULSES
ARITHMETIC ELEMENT, MEMORY ELEMENT AND PROGRAM ELEMENT
COMMANDS (GATED PULSES)
PULSE GENERATION AND CONTROL
2-MC PULSES, TIME PULSES (TP) OR INSTRUCTION PULSES (IP)
COMMAND GENERATORS (GATE TUBES)
D-C LEVELS (GATING SIGNALS)
CODED INSTRUCTIONS FROM MEMORY BUFFER REGISTER
INSTRUCTION DECODER (D-C LEVEL GENERATION)

Figure 5-3. Instruction Control Element, Simplified Block Diagram

tape units to rewind. These are primarily unconditioned commands; in each case, we are directing (or "operating") the computer to take some definite action. However, there are occasions when we wish to examine the status of various parts of the computer and take some action, depending on the status. For example, if we want to print something with the printer, we must first check to see if the printer is ready for use. This is known as "sensing" the status of the printer. If the printer is not ready, we can take corrective action depending on what options we have made available to the computer through the program. The third operation that the selection element performs is to determine whether an IO unit has been selected for operation and, if such is the case, to condition the IO element.

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 IO 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 IO 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.

3. PROGRAM CONTROL ELEMENTS

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

b. Program Counter

The Program Counter is a flip-flop register which is responsible for keeping track of the location of instructions within a program. During normal operation of a program, the program counter will contain the address of the next instruction to be executed.

c. Address Register

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

d. Index Registers

The Central Computer System can perform many routine functions such as sorting, tabulating, table makeup, etc., by the use of programs which will repeat a certain number of instructions as often as necessary. When a program such as this, called an interative program requires data stored in sequential locations, the data can be obtained through the use of a single instruction and an associated Index Register. Each time the instruction is executed, the amount remaining in the Index Register is added to the contents of the Address Register, causing an address modification. The AN/FSQ-7 and AN/FSQ-8 each contain four index registers, plus the right accumulator of the arithmetic element which is something used as a type of index register.

4. THE ARITHMETIC ELEMENT

a. General

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

b. A Registers

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

c. Adders

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

d. Accumulators

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

e. B Registers

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

5. IO ELEMENT

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

a. Controlling the amount of information that is transferred.

b. Determining where in memory we read or write the information.

c. Determining what IO unit is to be involved.

d. Acting as a buffer storage device between the IO unit and internal memory.

6. MEMORY ELEMENT

In the AN/FSQ-7, the physical core memory units are referred to as memory one and memory two. Memory one, which is called BIG MEMORY, contains $65{,}536_{10}$ or $200{,}000_8$ storage registers; memory two, which is called LITTLE MEMORY, contains $4{,}096_{10}$ or $10{,}000_8$ storage registers. In the AN/FSQ-8 both memory one and memory two contain $4{,}096_{10}$ registers. The core memories are non-volatile, meaning that they retain the information which is stored in them even when the power is not applied to the units. We also consider the memories to have random access, meaning that any memory location may be selected and read out in the same amount of time. This time is referred to as access time (or memory cycle) and is six microseconds in the AN/FSQ-7 and AN/FSQ-8. Thus, a minimum of six microseconds must elapse between successive word transfers. One more important point to consider is that readout from core memory is destructive; 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 a new word.

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

7. 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 Computer. 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.

OVERALL SYSTEM INFORMATION FLOW

The main transfer paths for both data and instructions within the Central Computer System are shown in Figure 5-1. Not all the paths have been shown; however, those which connect the registers and circuitry primarily responsible for decoding and the execution of instructions are indicated. The IO element has not been broken down to show any of its operational registers because it is not involved in the operation of the Central Computer to be discussed at this time. The IO element will be discussed more fully when IO programming and IO instructions are discussed.

CHAPTER 6

CODING OF LOGIC BOOK - reference logic index and CC Logic Vol. II index page.

1. First Digit designates "System"

    0. Central Computer
    1. Drums
    2. Inputs
    3. Outputs
    4. Display
    5. Power
    6. Warning Lights
    7. Maintenance Console
    8. None
    9. Test Equipment

2. Second Digit Designates "Logical Function".

3. Third Digit Designates "Section of Function".

4. Fourth Digit Designates "further breakdown if necessary".

5. Breakdown of "Logic Function" used in the Central Computer are as follows:

    a. (0.1) Core Memory

    b. (0.2) Timing and Distribution Controls

    c. (0.3) Operations

    d. (0.4) Addressing

    e. (0.5) Arithmetic

    f. (0.6) Index Interval Decoding

    g. (0.7) I/O, Sense, Operate, Misc.

    h. (0.8) Magnetic Tapes

CHAPTER 7

MACHINE TIMING all cycles have a 6μsec cycle or multiples of 6μsec,

In order to generate the individual commands that execute a computer instruction, a system has been devised using 2 mc (.5 usec apart) pulses.

They are distributed by the TPD (Time Pulse Distributer) on 12 lines as TP 0 through TP 11. This is basic machine timing. Every 6 usec a TP-0 is produced, and so on through TP-11. These are the same pulses that are used to control nearly all operations in the central computer. There are 5 (five) different types of Machine cycles. They are:

machine cycles

1. Program Time

During Program Time the TPD ring develops TP pulses which handle the transfer from memory to the instruction control element of instruction words. PT pulses are used to decode instruction words also.

2. Operate Time A

OTA pulses handle the transfer from memory to the arithmetic element of operands and data words. They also control most of the arithmetic processes.

3. Operate Time B

OT-B pulses are used to control the transfer of new data from the arithmetic element to core memory.

4. Break In

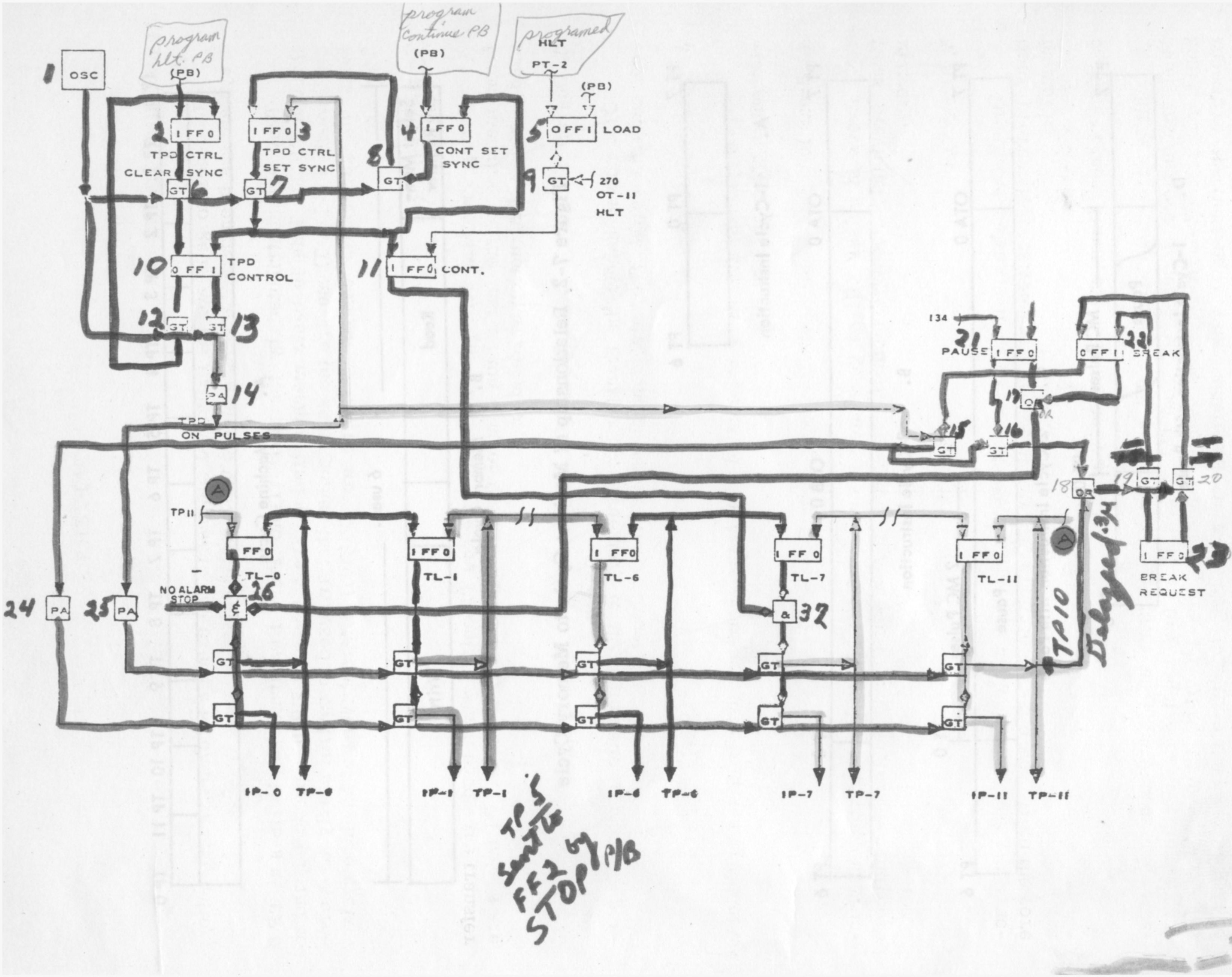
Break In pulses will be generated when an external device, acting under instructions previously received from the computer, has a word ready for transfer into core memory.

5. Break Out

Break Out pulses will be generated when an external device, acting under instructions previously received from the computer, is ready to accept a word from the core memory.

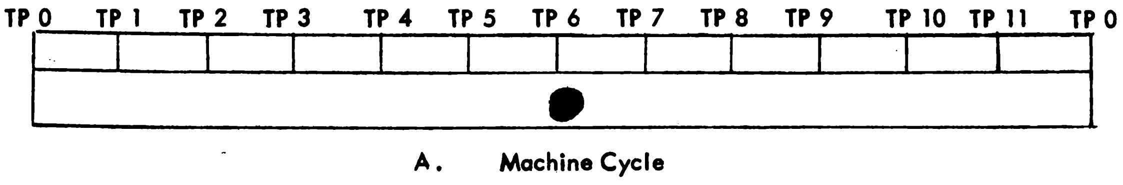
6. None of these times can exist at the same time.

All instructions require PT and therefore require at least 6 usec to operate. Most instructions require PT and OT-A and therefore require 12 usec. to operate. Some require PT-OTA-OTB, ((18 usec.). Some require PT-OTB, 12 usec., and some repetitive instructions require additional time. During this time the pulses used are generally straight 2 mc pulses. This time is referred to as an arithmetic pause. All of these combinations referred to above are known as instruction cycles.



OSC
program hlt. PB
(PB)
program Continue PB
programed HLT
PT-2
(PB)
1 FF 0
TPD CTRL CLEAR SYNC
TPD CTRL SET SYNC
CONT SET SYNC
0 FF 1 LOAD
GT
270 OT-11 HLT
0 FF 1 TPD CONTROL
1 FF 0 CONT.
PA
TPD ON PULSES
134
PAUSE
BREAK
OR
BREAK REQUEST
TP11
NO ALARM STOP
TL-0
TL-1
TL-6
TL-7
TL-11
&
TP10
Delayed 3M
IP-0
TP-0
IP-1
TP-1
IP-6
TP-6
IP-7
TP-7
IP-11
TP-11
TP 5 Sample FF2 by STOP P/B

Figure 7-1



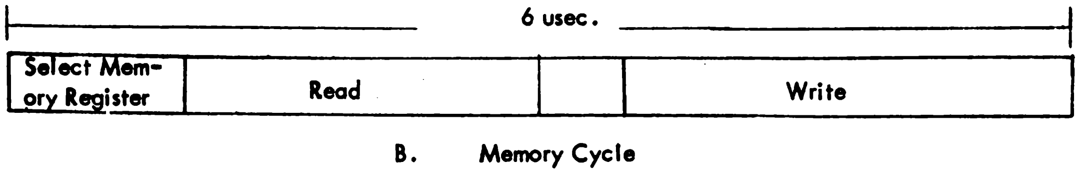


Figure 7-2. Relationship of Machine Cycle to Memory Cycle

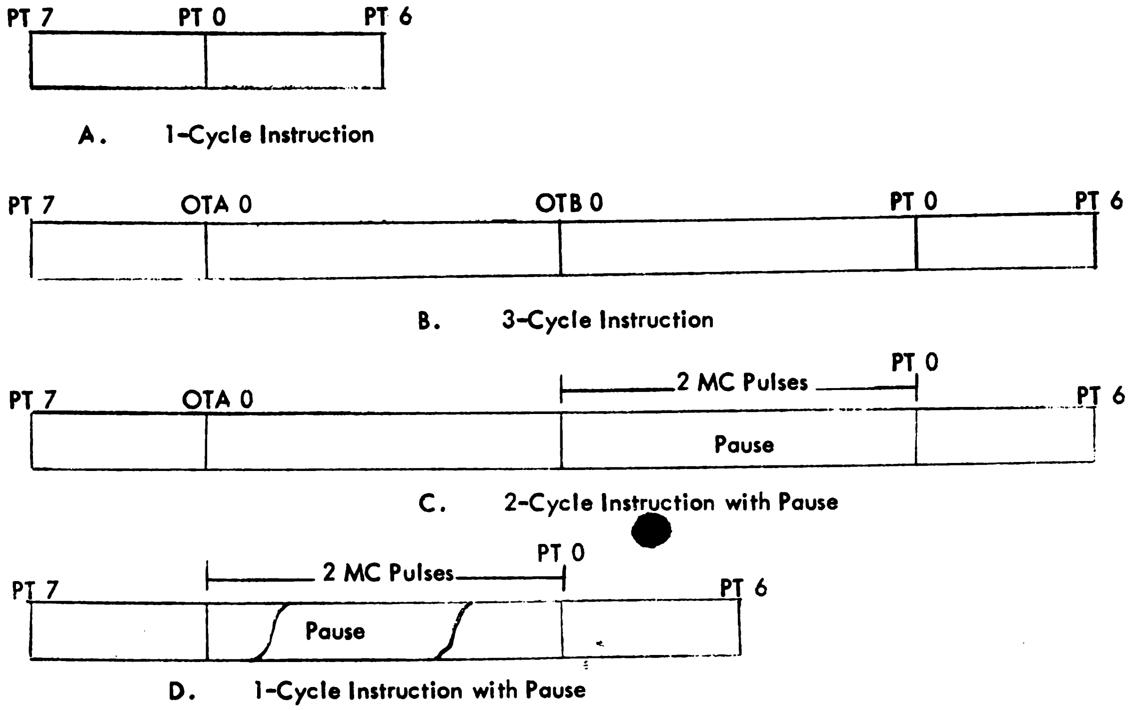
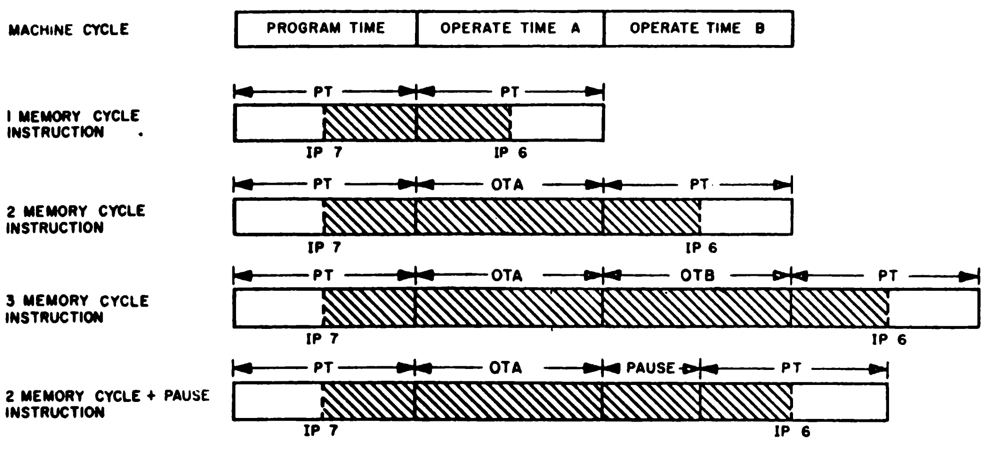
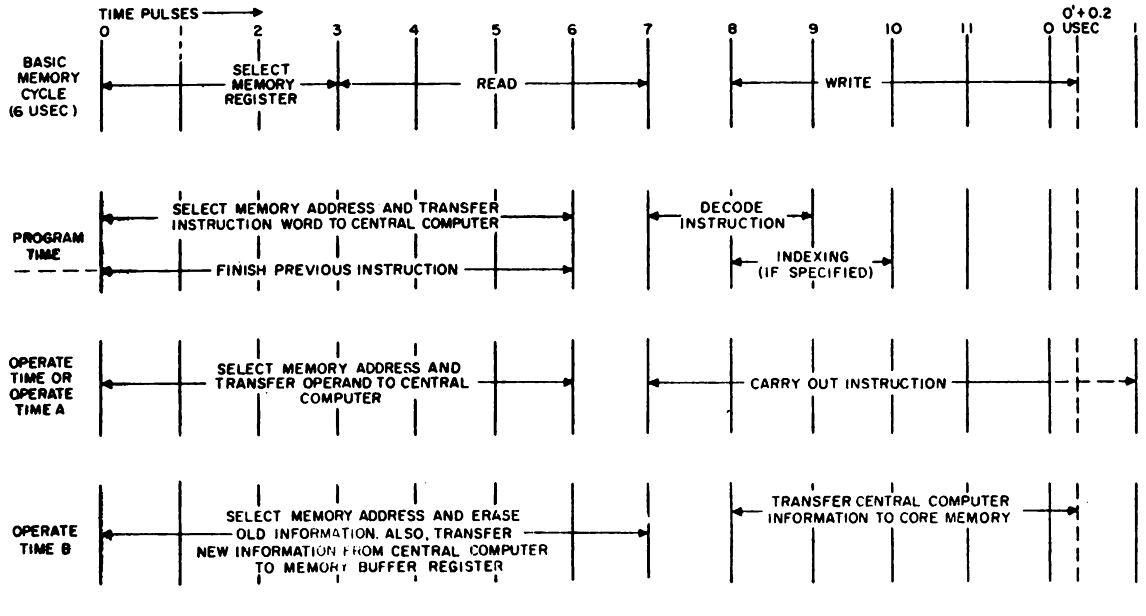


Figure 7-3. Various Cycle Configurations for AN/FSQ-7 and AN/FSQ-8



MACHINE CYCLE
PROGRAM TIME
OPERATE TIME A
OPERATE TIME B
1 MEMORY CYCLE INSTRUCTION
2 MEMORY CYCLE INSTRUCTION
3 MEMORY CYCLE INSTRUCTION
2 MEMORY CYCLE + PAUSE INSTRUCTION
PT
OTA
OTB
PAUSE
IP 7
IP 6

Figure 7-4. Machine and Instruction Cycles



TIME PULSES
0 1 2 3 4 5 6 7 8 9 10 11 0 0'+0.2 USEC
BASIC MEMORY CYCLE (6 USEC)
SELECT MEMORY REGISTER
READ
WRITE
PROGRAM TIME
SELECT MEMORY ADDRESS AND TRANSFER INSTRUCTION WORD TO CENTRAL COMPUTER
FINISH PREVIOUS INSTRUCTION
DECODE INSTRUCTION
INDEXING (IF SPECIFIED)
OPERATE TIME OR OPERATE TIME A
SELECT MEMORY ADDRESS AND TRANSFER OPERAND TO CENTRAL COMPUTER
CARRY OUT INSTRUCTION
OPERATE TIME B
SELECT MEMORY ADDRESS AND ERASE OLD INFORMATION. ALSO, TRANSFER NEW INFORMATION FROM CENTRAL COMPUTER TO MEMORY BUFFER REGISTER
TRANSFER CENTRAL COMPUTER INFORMATION TO CORE MEMORY

Figure 7-5. Comparison of Core Memory Cycle and Internal Machine Cycles

INSTRUCTION CYCLE

An instruction cycle always begins at PT-7. Therefore a one memory cycle instruction occurs from PT-7 to PT-6. A two memory cycle instruction occurs from PT-7 to PT-11, OT-0 to OT-11, and PT-0 to PT-6.

Why would an instruction cycle begin at PT-7? Because the instruction word is not available from memory before PT-7 and therefore the PT-0 to PT-6 time can be used by the prior instruction and is a part of its instruction cycle.

PAUSE

A pause is a time delay for an Operation to Complete before continuing. There are two types of Pauses (I/O Pause and Arithmetic Pause): I/O Pause occurs every time the computer tries to start a 2nd I/O operation while one is still in progress or if you try to halt the computer while in an I/O Operation. The I/O interlock is used in conjunction with this type pause. No instructions are operated during an I/O pause.

Arithmetic Pause is used for instructions with repetitive operations such as SHIFT, MULT, and SLR. 2 m.c. operation pulses are used at this time. Breaks can occur during Arithmetic Pauses. While in a Pause, the TPD ring is stopped after TP-11 at TL-0. When a break occurs during a pause, the TPD ring is started, and stops when the break cycle is complete.

CHAPTER 8 - BASIC COMPUTER INSTRUCTIONS

GENERAL

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

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

From the above discussion, it is apparent that to write a satisfactory program it is necessary to have a thorough knowledge of the instructions we can use. This includes execution time, the overall purpose of the instruction, when the instruction may be used, and the state of the computer after the instruction has been carried out. In addition, we should know whether the instruction can be indexed and what internal conditions must be satisfied before it can be executed. The following text describes 17 basic instructions that are used in the AN/FSQ-7. Each description contains the information listed above, and program examples of the instructions are given. Since most problems have several possible solutions, the program given for a particular problem may not represent the most efficient way of arriving at an answer. Rather, the programs are designed to show the application of individual instructions within a program.

HALT INSTRUCTION

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

CLEAR AND ADD INSTRUCTION

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

ADD INSTRUCTION

The operation of this instruction is similar to that of 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 any value that may be in the accumulators. The sum is placed in the accumulators, and the A registers are cleared to +0. The ADD instruction requires 12 usec for execution and may be indexed. The octal code for this instruction is 104. It should be noted that the ADD instruction can cause an overflow if the numbers added together are sufficiently large. If this happens, the result in the accumulator is meaningless. Because the arithmetic elements are dual, an overflow may occur in one accumulator and not the other; however, overflow in both accumulators may occur as a result of the same ADD instruction. Later on, we shall see how this condition (overflow) can be dealt with by a computer program.

LEFT ADD INSTRUCTION

The LEFT ADD (LAD) instruction is used to add the left half-word of the memory location specified to the contents of the left accumulator. This instruction is similar to the ADD instruction except that nothing is done to the right accumulator. The A registers are cleared to +0, then the left half word of the specified memory location is transferred to the left accumulator. The sum of the LHW of the memory location and the left accumulator appear in the left accumulator, the right accumulator remains unchanged. The octal operation code for the LAD instruction is 120 and may be indexed. Since addition only takes place in the left arithmetic element, overflow may occur in the left accumulator only. The LAD instruction requires 12 usec to operate.

FULL STORE INSTRUCTION

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

SAMPLE PROGRAMS INVOLVING ADDITION

Now that we have learned enough instructions (HLT, CAD, ADD, and FST) to solve a basic and simple problem involving addition only. However, the programming presented in this problem are the same as those outlined previously; i.e., to choose the proper instruction and place it in the proper place in our program. Assume that the problem is to add several sets of quantities together and store the results for future use. Before we write a program, we must know where in memory those quantities are stored initially; we must also know where to place their sum. In addition, we must have memory locations available for storage of the computer program itself. For ease of explanation, assume that memory locations 0_8-10_8 are available for the program; locations 100_8-200_8 are available for data, including the result. Further assume that the quantities we wish to add are contained in memory locations 100_8, 125_8, and 135_8. Let the quantities be designated A, B, C, D, E, and F, where A=1, B=2, C=3, D=4, E=5, and F=6, and have them located as follows:

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

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

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

Now we are ready to solve the problem. The program used is given below.

Table 8-1

BASIC ADDITION PROGRAM

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

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

The instruction itself is located in memory location 0.00000, and it says to CAD location 0.00100. This will cause the accumulators to be cleared to positive zero and the contents of 0.00100 to be added in. Since we specified that 0.00100 contains A, B, the left accumulator will contain A (1) and the right accumulator will contain B (2) after execution of the CAD instruction. It should now be obvious that the CAD instruction is the best instruction with which to start our program, since our problem is to add numbers together, and the ADD instruction would not suffice since it is possible (and probable) that the accumulators would not be cleared at the start of the program. Thus, using an ADD instruction might lead to a result we do not want. The next instruction, which is located in 0.00001, tells us to ADD 0.00125. After execution of this instruction, the left accumulator contains A+C (4) and the right accumulator contains B+D (6). Remember that simultaneous addition is carried on in both arithmetic elements. The third instruction, ADD 0.00135, will add in E (5) and F (6) so that the left accumulator contains A+C+E (11) and the right accumulator contains B+D+F (14). Now the result we wish to obtain is in the accumulators. However, to be of any real value, the result must be stored in memory, so the fourth instruction tells the computer to FST these sums in 0.00150. Now location 0.00150 contains A+C+E (11) in the left half-word and B+D+F (14) in the right half-word, regardless of what quantity may have been there previously. Of course, these sums are still contained in the accumulators. The last instruction in our program is HLT, which causes the computer to stop operation.

This problem has been simple, and no doubt the solution was obvious from the beginning. However, most problems do not offer such straight forward methods of solution. In some cases, it is advantageous to add one quantity before another, although in the above example it made no difference which quantities were initially placed in the accumulators, since the end result would have been the same. In addition, the choice of memory location 0.00150 for the result was arbitrary, the only restriction being that it had to be somewhere between locations 0.00100-0.00200, as mentioned before. In many cases, however, the results of computation have to be in a particular location or locations because another program may refer to that location on the assumption that the proper quantities have been placed there. The following example shows how a program may refer to a location that has just been used for storage. Assume that we wish to obtain the sums 2A + 2C, 2B + 2D, using the same data locations as those given in the first program. The program to obtain these sums is given below.

Table 8-2. MEMORY REFERENCE PROGRAM

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

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

CLEAR AND SUBTRACT INSTRUCTION

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

SUBTRACT INSTRUCTION

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

TWIN AND ADD INSTRUCTION

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

TWIN AND SUBTRACT INSTRUCTION

The TWIN AND SUBTRACT (TSU) instruction is employed to subtract the left half word of the specified memory register from both the left and the right accumulators. This instruction is similar in execution to the SUB instruction. The left half word is transferred to both A registers where it is complemented and then added to the accumulators. The difference appears in the accumulators, and just as with the

TAD instruction, overflow may occur in either or both accumulators as a result of the TSU instruction. The octal operation code that designates a TSU instruction is 140, and requires 12 usec to execute. This instruction may also be indexed.

SAMPLE PROGRAMS INVOLVING SUBTRACTION AND TWINNING

Now that we have learned enough instructions to cover two of the four basic arithmetic processes, it is possible to write programs which will solve problems that are more complex than those given previously. In the following examples, the locations of data are listed at the end of the tables containing the program. Assume that our programs will always start at location 0.00000 and that we have up to 0.00050 available for the instructions. The first problem is to compute two sets of values; namely, A-C+2E, B-D+E+F and A-C+2E, 2A-B-2C+D+3E-F. The program is executed as shown in the table below.

Table 8-3. SAMPLE SUBTRACTION PROGRAM

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

LEFT STORE INSTRUCTION

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

RIGHT STORE INSTRUCTION

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

SAMPLE PROGRAMS INVOLVING HALF-WORD STORAGE

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

Table 8-4. SAMPLE PROGRAM USING HALF-WORD STORE INSTRUCTIONS

LOCATION	OPERATION	ADDRESS
0.00000	CAD	0.00075
0.00001	ADD	0.00077
0.00002	TSU	0.00076
0.00003	LST	0.00430
0.00004	TAD	0.00077

0.00005	ADD	0.00077	
0.00006	RST	0.00430	
0.00007	SUB	0.00075	
0.00010	FST	0.00440	
0.00011	RST	0.00450	
0.00012	ADD	0.00440	
0.00013	LST	0.00450	
0.00014	HLT	0.00000	
0.00075	A	B	
0.00076	C	D	Constants
0.00077	E	F	
0.00430	First Result		
0.00440	Second Result		
0.00450	Third Result		

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

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

Table 8-5. ADDITIONAL EXAMPLE OF HALF-WORD STORAGE

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

CHAPTER 9 - BASIC BRANCH INSTRUCTIONS

GENERAL

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

BRANCH ON LEFT MINUS INSTRUCTION

The BRANCH ON LEFT MINUS (BLM) instruction examines the sign bit of the left accumulator and branches to the location specified in the address portion of the instruction if the sign bit contains a 1 bit. When the branching condition is met, the program counter, which has already been stepped, is transferred to the right A register, and the address register is transferred to the program counter. Thus, at the completion of a BLM instruction (assuming the branch condition is met), the program counter will contain the location of the next instruction we desire to execute. If the sign bit of the left accumulator is positive, the branch condition has not been met, and the next instruction executed will be the one immediately following the BLM instruction.

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

BRANCH ON RIGHT MINUS INSTRUCTION

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

TABLE CONSTRUCTION PROGRAM

Following is a program which illustrates the use of the BLM and BRM instructions. Assume that the problem is to examine both half words of a data word stored in memory and to construct a table of all negative values, using right half words, beginning at location 0.00100. We do not know the values of the left and right portions of the data

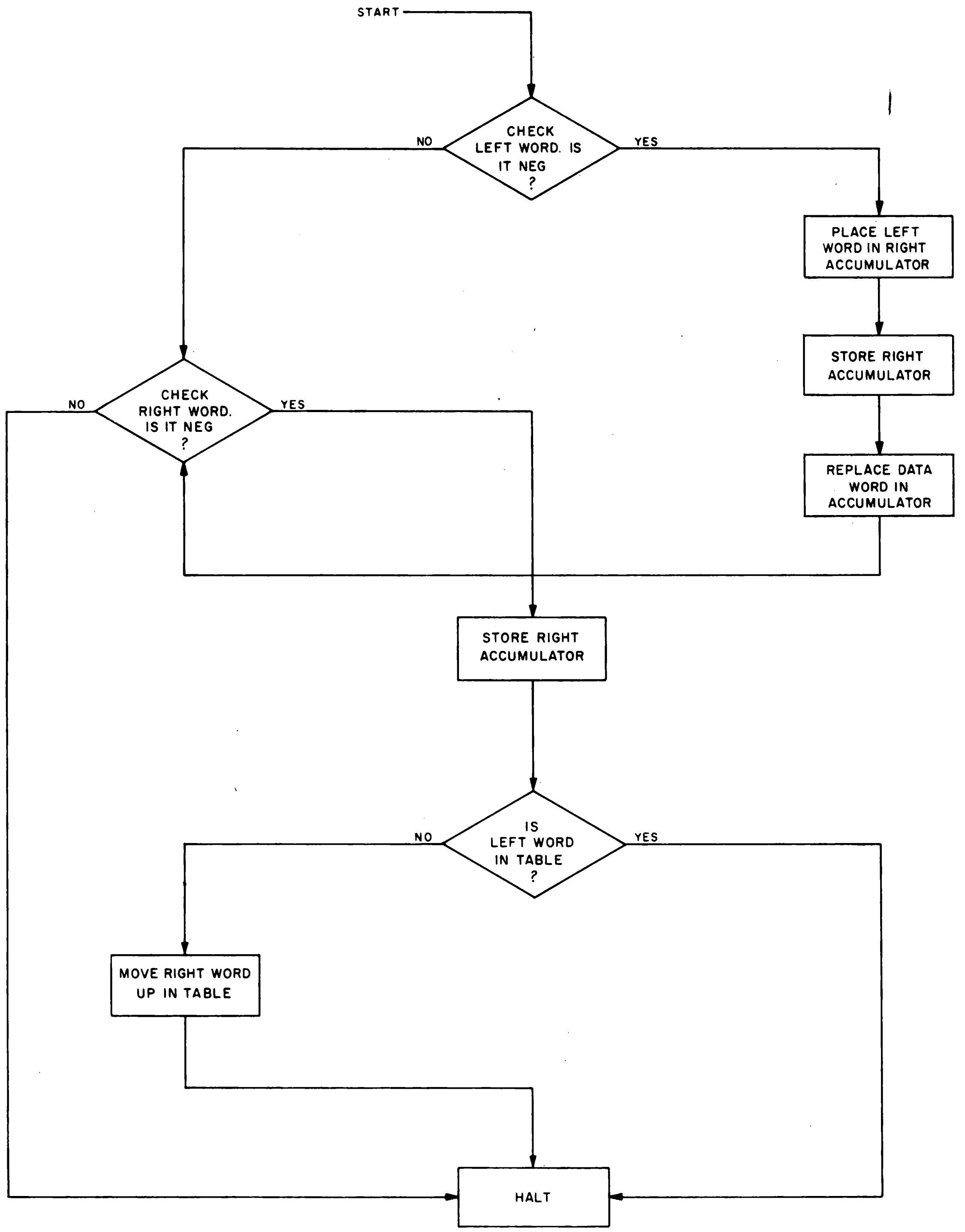

START
CHECK LEFT WORD. IS IT NEG ?
NO
YES
PLACE LEFT WORD IN RIGHT ACCUMULATOR
STORE RIGHT ACCUMULATOR
REPLACE DATA WORD IN ACCUMULATOR
CHECK RIGHT WORD. IS IT NEG ?
NO
YES
STORE RIGHT ACCUMULATOR
IS LEFT WORD IN TABLE ?
NO
YES
MOVE RIGHT WORD UP IN TABLE
HALT

Figure 9-1. Preliminary Flow Chart, Table Construction Program

word. The solution to this problem may be rather simple; however, when writing programs of the type that involves a number of decisions, it is useful to construct a program flow chart. This chart is actually a graphic representation of what is desired and depicts a method of arriving at a solution. The technique is to separate the problem into a number of blocks, each of which contributes to the solution. Then the actual programmed instructions that can be used to perform the function of each block are determined, and the end result is the finished program. The problem given above will be laid out using the flow chart method. It should be noted that flow charting is not always necessary in the solution of a problem, but will ease the task of the actual coding.

PRELIMINARY FLOW CHART

The problem stated that we are to examine a data word for negative quantities and construct a table from these quantities, if any are present. In addition, we know that this table is to start at location 0.00100 and that we must use the right half-word for storage. With this much information, we can lay out a preliminary flow chart. This chart should resemble the one shown in Figure 9-1.

FINAL FLOW CHART

From the examination of Figure 9-1, it should now be obvious that we can use the BLM and BRM instructions to arrive at a solution to this program. There are four possible arrangements of the signs within a data word, and this flow chart will handle all of them. Now we can proceed to fill in each of the blocks as done in Figure 9-2. For ease of explanation, assume that the data word we are interested in is located at address 0.00050 and that it contains the quantities X and Y in the left and right-half words, respectively. A listing of this program is also given below.

Table 9-1

LOCATION	OPERATION	ADDRESS
0.01000	CAD	0.00050
0.01001	BLM	0.01004
0.01002	BRM	0.01011
0.01003	HLT	0.00000
0.01004	CAD	0.01003
0.01005	TAD	0.00050
0.01006	RST	0.00100
0.01007	CAD	0.00050
0.01010	BLM	0.01002

0.01011	RST	0.00101
0.01012	CAD	0.00100
0.01013	BRM	0.01003
0.01014	CAD	0.00101
0.01015	RST	0.00100
0.01016	BRM	0.01003
0.00050	X	Y
0.00100		First value
0.00101		Second value

Note that Figure 9-2 specifies that we are checking X and Y to see if they are smaller than +0. This is necessary because -0 will contain a sign bit of 1 and will satisfy the branching conditions. Therefore, it is possible that one or both of the values could be -0 and not actually numbers with an absolute magnitude of less than zero. Later, we shall see how another branching instruction can be used to determine whether the quantities being tested are really negative numbers.

NUMBER SORTING PROGRAM

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

PRELIMINARY FLOW CHART

We know that our problem is to sort through three numbers and select the largest. Since we do not know the magnitude of the numbers, it is necessary to compare all three numbers against each other. This involves making two comparisons, determining which number is larger at each comparison, and finally storing the correct value. The flow which graphs this solution is shown in Figure 9-3.

FINAL FLOW CHART

The final flow chart for this program is shown in Figure 9-4. It can be seen that the use of the BLM instruction has enabled us to determine quite rapidly what is the largest number and to store it. Notice that there are two separate ways in which the third number compared can be found to be the largest number. Since this is the case, we can use one group of instructions to place this number in the desired memory location, and allow entry to this group from two separate points. For purposes of

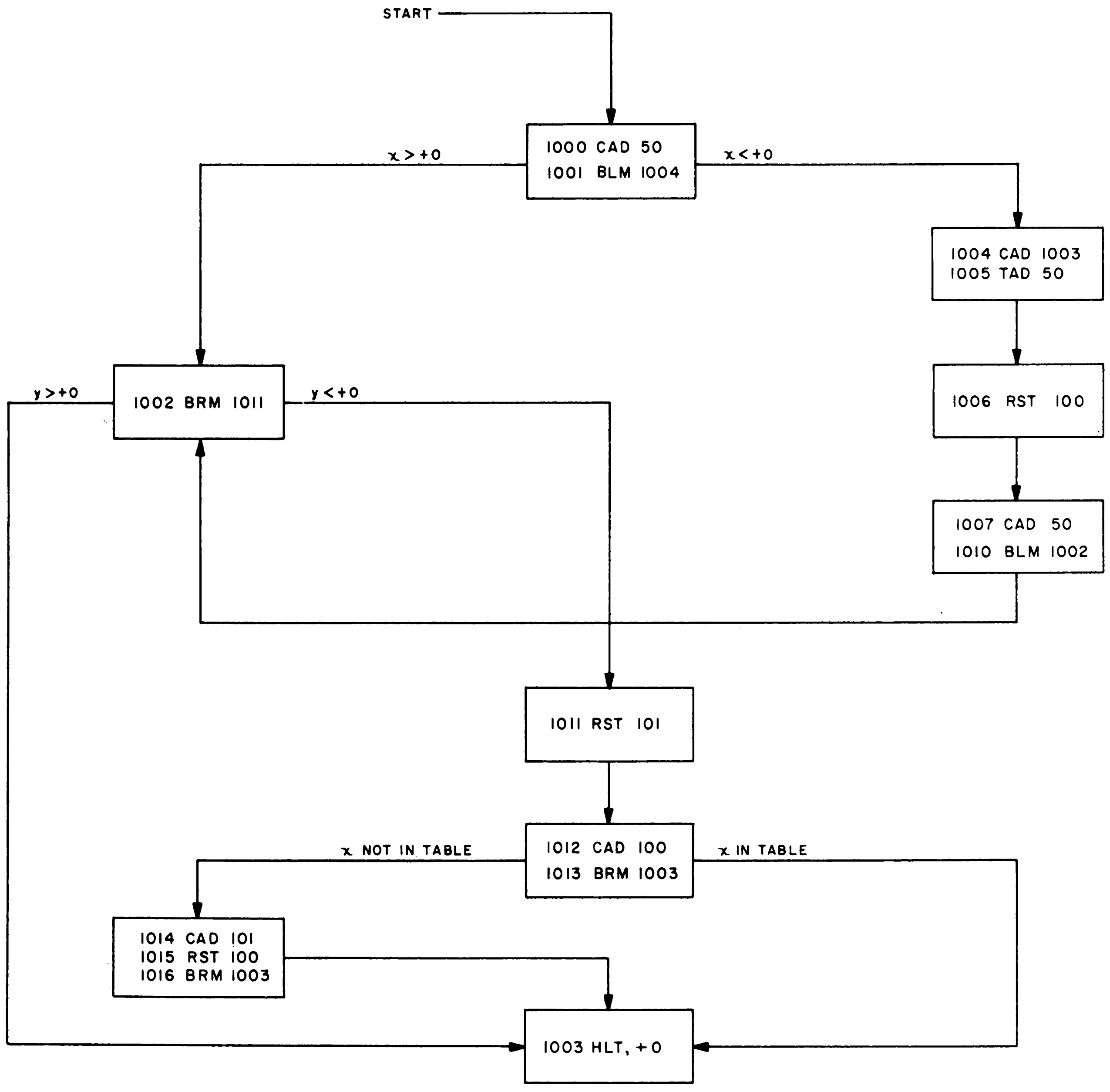


Figure 9-2. Final Flow Chart, Table Construction Program

explanation, we have assumed that the numbers to be compared are A, B, and C and that they are in memory locations 0.00100, 0.00101, and 0.00102, respectively. Location 0.00150 is used to store the result of the comparison. This program is listed in Table 9-2.

Table 9-2

LOCATION	OPERATION	ADDRESS	
0.00000	CAD	0.00100	
0.00001	SUB	0.00101	
0.00002	BLM	0.00011	
0.00003	CAD	0.00100	
0.00004	SUB	0.00102	
0.00005	BLM	0.00017	
0.00006	CAD	0.00100	
0.00007	FST	0.00150	
0.00010	HLT	0.00000	
0.00011	CAD	0.00101	
0.00012	SUB	0.00102	
0.00013	BLM	0.00017	
0.00014	CAD	0.00101	
0.00015	FST	0.00150	
0.00016	HLT	0.00000	
0.00017	CAD	0.00102	
0.00020	FST	0.00150	
0.00021	HLT	0.00000	
0.00100	A	-	Data
0.00101	B	-	
0.00102	C	-	
0.00150			Result

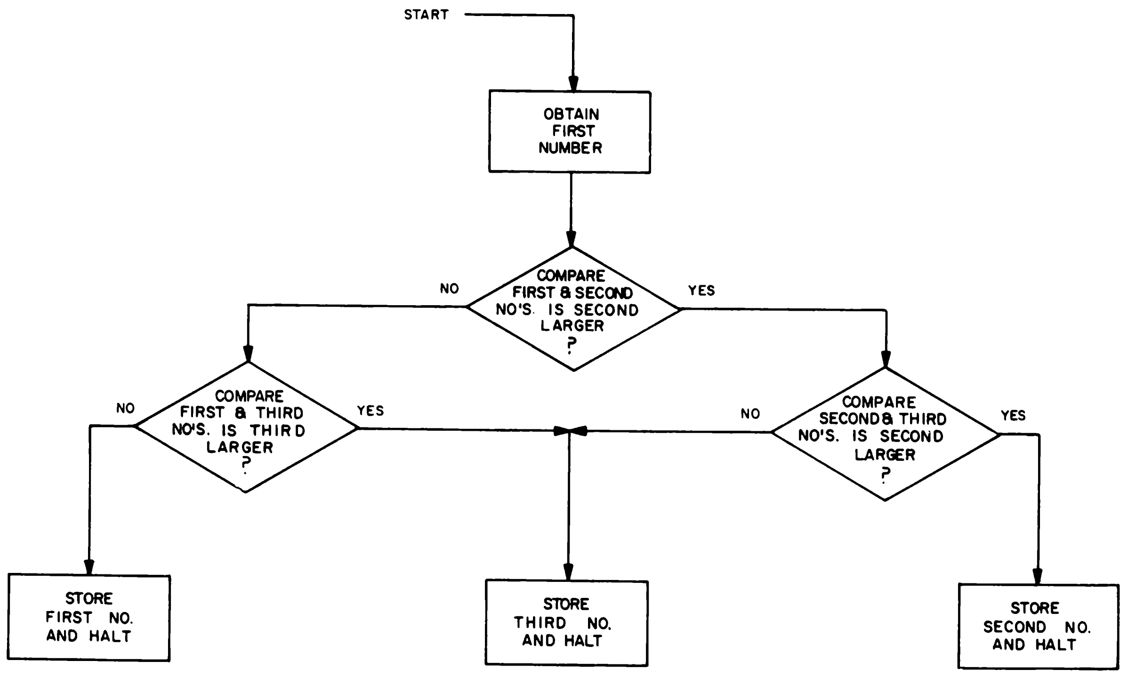

START
OBTAIN FIRST NUMBER
COMPARE FIRST & SECOND NO'S. IS SECOND LARGER ?
NO
YES
COMPARE FIRST & THIRD NO'S. IS THIRD LARGER ?
NO
YES
COMPARE SECOND & THIRD NO'S. IS SECOND LARGER ?
NO
YES
STORE FIRST NO. AND HALT
STORE THIRD NO. AND HALT
STORE SECOND NO. AND HALT

Figure 9-3. Preliminary Flow Chart, Number-Sorting Program

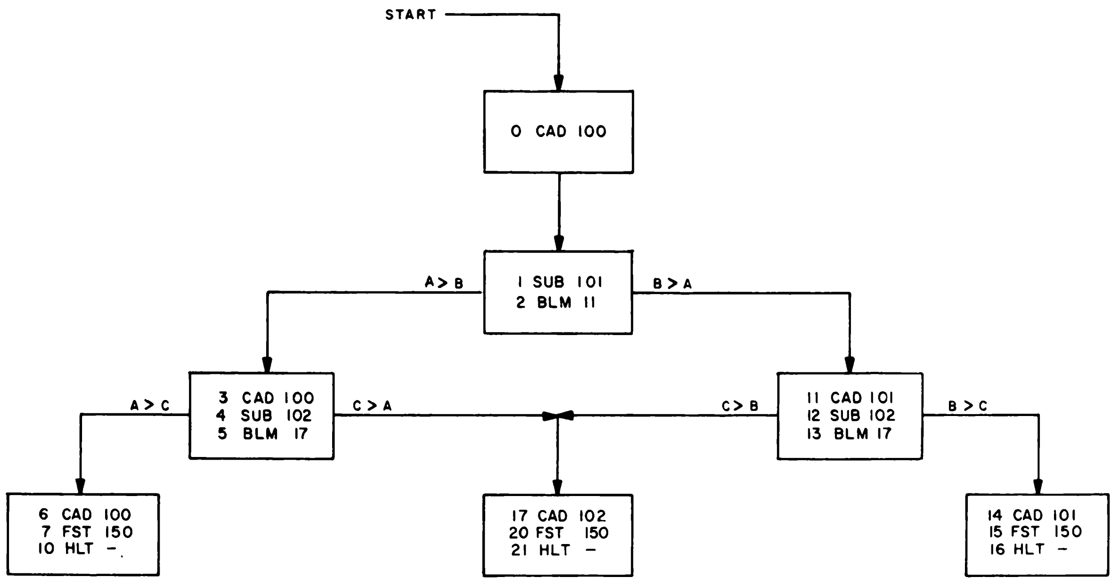


Figure 9-4. Final Flow Chart, Number-Sorting Program

BRANCH ON FULL MINUS INSTRUCTION

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

BRANCH ON FULL ZERO INSTRUCTION

The BRANCH ON FULL ZERO (BFZ) instruction checks the contents of both accumulators to see if they are 0. Since the AN/FSQ-7 and AN/FSQ-8 utilize both a positive and a negative zero, there are four combinations of positive and negative zero branching conditions. These combinations are given below.

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

The execution of this instruction is carried out in the following manner. First, the contents of both accumulators are made positive if they are not already so. Then the accumulators are complemented and made negative. A 1 is added to both accumulators by initiating a carry of 1 at bit 15, and the carry-out of the most significant bit is disregarded. If the accumulators contained either positive or negative zero, complementing and then adding 1 without the end carry would have rippled through the accumulator bits and cleared them to 0. At this point, the accumulators are complemented again, and the sign bits are checked. If the sign bits contain 1 bits, we know that the original values were either positive or negative zeros, and the branching condition has been satisfied. As with all branching instructions that find their conditions satisfied, the BFZ instruction will transfer the program counter to the right A register and the address portion of the instruction itself to the program counter if both sign bits are 1. Due to the number of operations involved, an OT cycle is required for the BFZ instruction steps described thus far. After the test of the sign bits has been made, the "carry 1" line is again pulsed, and the accumulators may be complemented, to restore them to their original values. Execution time for this instruction is 12 usec; the instruction is designated by a code of 540. The BFZ instruction cannot be indexed. The table below shows the various steps encountered during the execution of a BFZ instruction. In this case, we have assumed that both accumulators are zero but have different algebraic signs.

Table 9-3. EXECUTION OF BRANCH ON FULL ZERO (BFZ) INSTRUCTION

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

*These steps conditional, depending on sign of accumulator.

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

SAMPLE PROGRAM USING BFM AND BFZ INSTRUCTIONS

The following example illustrates how the BFM and BFZ instructions can be used to aid in the solution of a problem. Suppose we wish to sort a group of three data words and make up a table of those that have both half-words equal to or greater than zero. Because -0 and +0 both give the same results when used in arithmetic computation, it is necessary for us to make provisions in our program to check for -0. The preliminary flow chart for this problem is shown in Figure 9-5. Here the diamond-shaped "decision" blocks clearly indicate the use of the BFM instruction when checking for positive values. However, the possibility of a -0 satisfying the branching conditions of the BFM instruction has to be dealt with by a second check on the word, this time using the BFZ instruction. If the check discloses that the word is not -0, we know definitely that it is a true negative number and proceed to the next word. This program

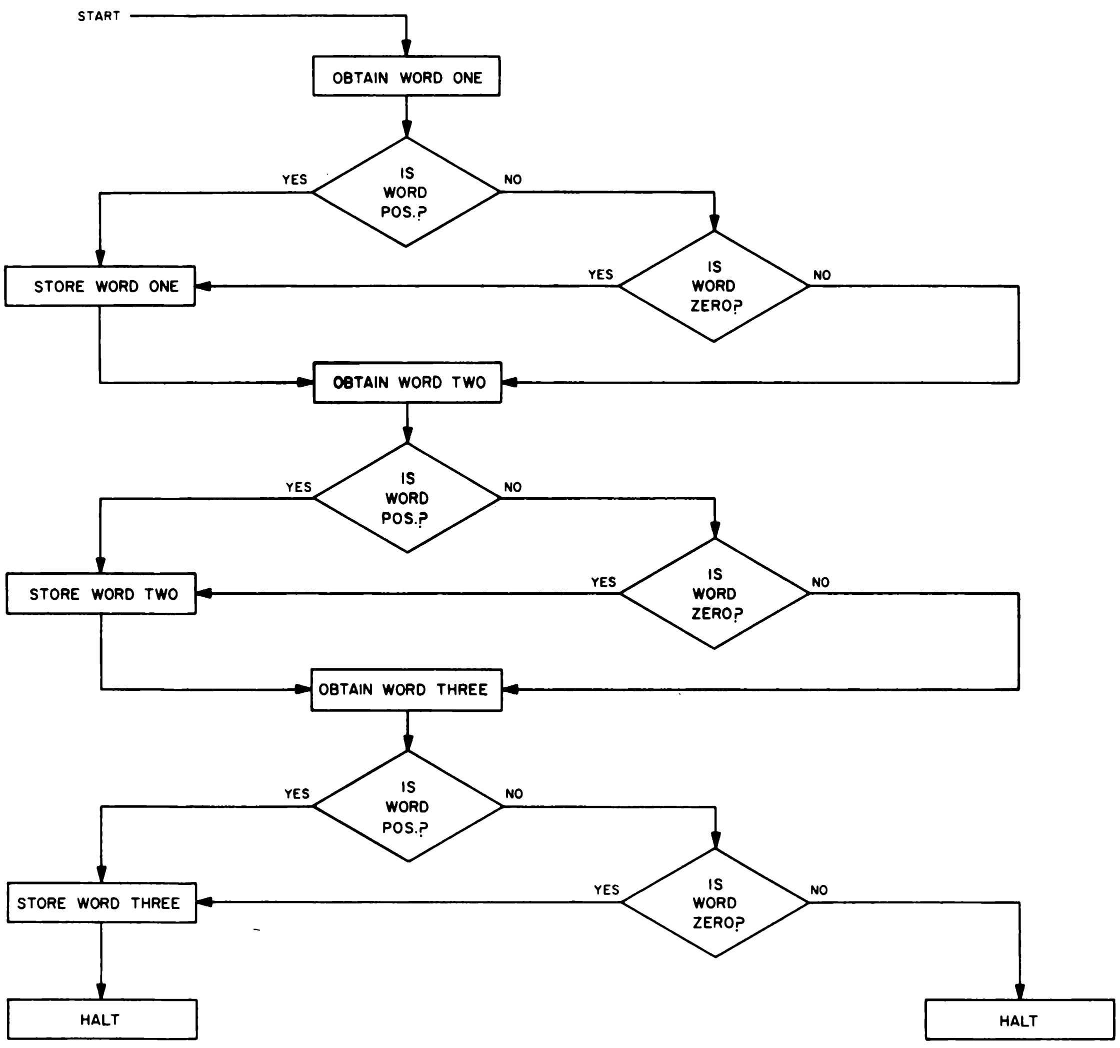


Figure 9-5. Preliminary Flow Chart Using Full Branch Instructions

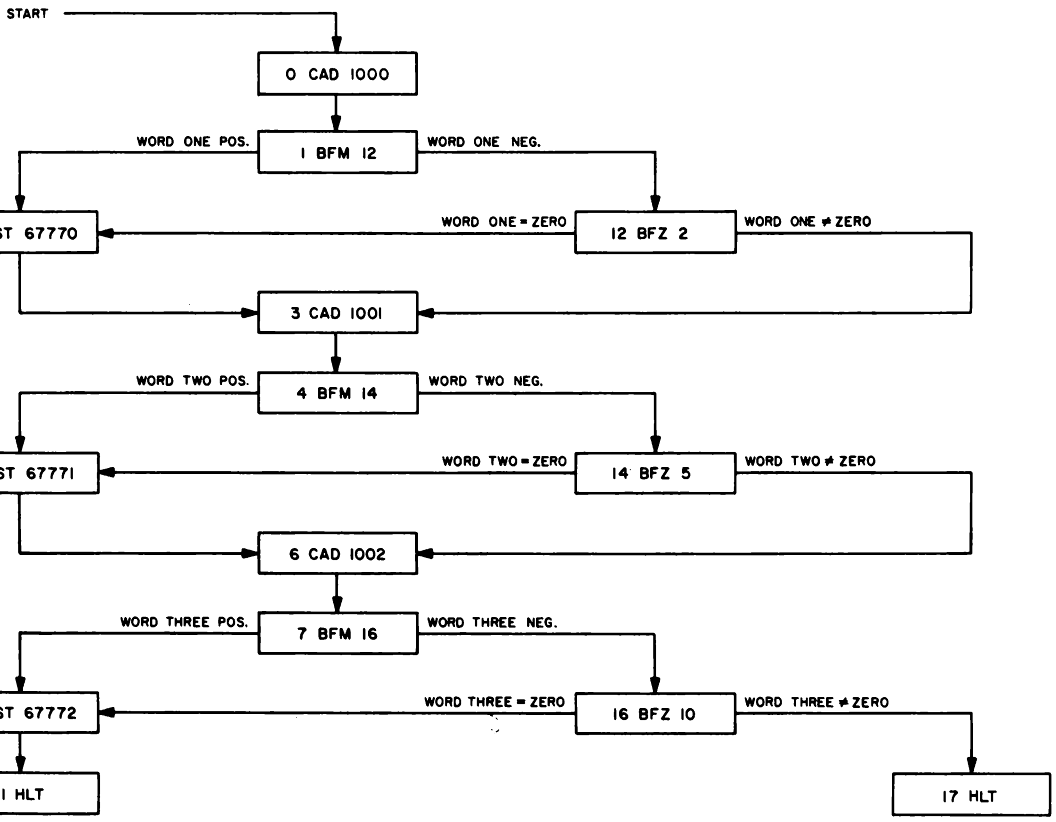


Figure 9-6. Final Flow Chart Using Full Branch Instructions

is also a logical program, since the data is left unchanged. We will assume that the program starts at location 0.00000 and the data is stored at 0.01000 through 0.01002. The table is to start at location 0.67770 and run in sequence. A finalized form of this program is shown in Figure 9-6 and listed in Table 9-4.

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

Table 9-4

CHAPTER 10 - ADD ONE RIGHT INSTRUCTION

The ADD ONE RIGHT (AOR) instruction is used to add one to bit R15 of the memory location specified by the address portion of the instruction. The execution of the AOR instruction takes place in the following manner. The right accumulator is cleared, and the contents of the selected memory register are transferred to the A registers. Then the carry 1 line into the bit R15 adder is pulsed, thus adding 1 to the address portion of the word. It should be remembered that the AN/FSQ-7 can have an address of 17 bits; therefore, if 17-bit operation is specified the LS bit is also used in the addition process. Also, an AOR instruction can cause computer overflow. After the address modification has taken place, the right accumulator is transferred to the right memory buffer (and bit LS of the left accumulator is transferred to the left memory buffer sign during 17 bit operation). The entire word (including the left half portion which has been stored in the left A register) is then replaced in its original memory location. The AOR instruction is similar to the FST instruction in that it takes an entire word from memory and replaces the same register with a different word. Because of this, an OTA and an OTB cycle are required in addition to the PT cycle, making execution time of the AOR instruction 18 usec. The octal operation code for this instruction, which is indexable, is 344.

USES OF THE AOR INSTRUCTION

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

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

ADDRESS MODIFICATION USING THE AOR INSTRUCTION

Assume that we have 11 numbers which are located in memory location 0.01100 through 0.01112. It is desired to find the sum of these numbers and store it in location 0.00150, using the least number of instructions possible. Of course, it is possible to use a series of ADD instructions, but this is quite lengthy. By use of the AOR instruction, we can use the same ADD instruction as many times as we need it. However, it

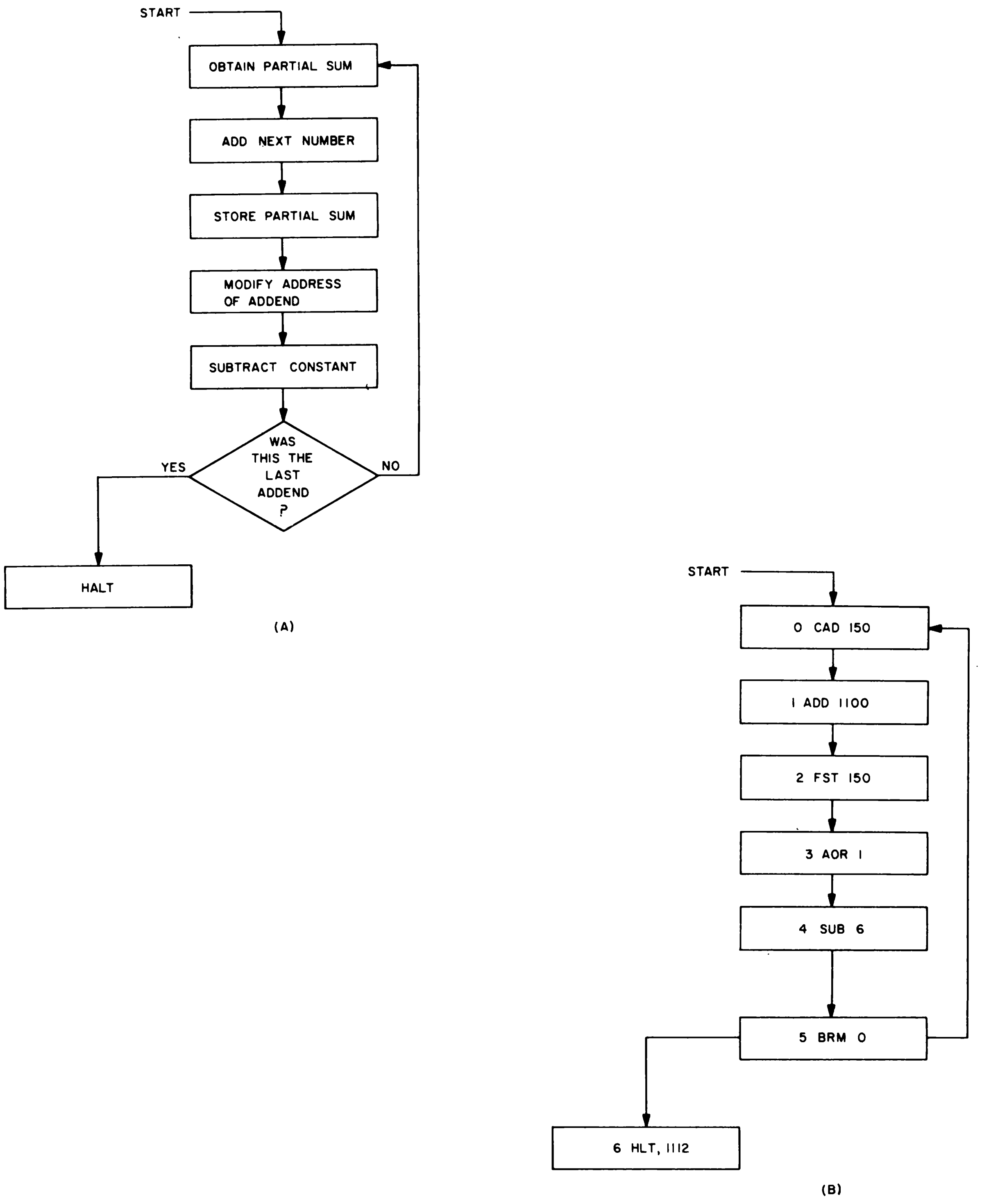


Figure 10-1. Address Modification Using AOR Instructions

is also necessary to have some sort of a control that tells us when we have added all 11 numbers. For this, we can use a constant which represents the last address we wish to use, and compare it against the address we are using. When the two match, we know we are using the last address we are interested in. We will assume that the sum of the 11 numbers will not cause an overflow and that the memory location 0.00150, which is to be used for final result storage, is cleared at the start of the program. The preliminary flow chart for this program is shown in Figure 10-1, part (A); the final flow chart is shown in part (B) of the same figure.

It can be seen from Figure 10-1 that we are modifying the address of the instruction in location 0.00001 by use of the AOR instruction at program step 3. Then the HLT instruction is subtracted from the new memory address which is left in the right accumulator after execution of the AOR instruction. The right half of the HLT instruction is being used as a constant; it contains the address of the last number to be added. Each time the address in the address in the accumulator is smaller than the final address, we will get a negative result. The BRM condition will be satisfied and will cause the program to select the partial sum and add another number to it. When the address being used reaches 0.00012, the AOR instruction will step it to 0.00013. The subtraction will leave 0.00001 in the right accumulator; the branch condition will not be met, and the program will fall through to the HLT instruction.

The program described above is listed below. Following that a straight-line program that achieves the same result without modification is listed for comparison.

Table 10-1

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

Table 10-2

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

Notice that while the program using the AOR instruction requires only seven memory locations as opposed to the 13 needed for the straight line program, the straight line program is considerably faster. The straight line program requires a total of 27 machine cycles, or 162 usec. On the other hand, the indexed program requires 13 machine cycles for each number added, or a total of 145 cycles (11x13 plus two cycles for the HLT instruction). This means 870 usec are needed to complete this program. Thus, we have two programs which arrive at the same result, one being shorter and the other being faster. As mentioned previously, one of these criteria will have to be sacrificed, depending on the situation.

COUNTING BY USE OF THE AOR INSTRUCTION

Now let us take an example of the other common use of the AOR instruction, that of stepping a pass counter. In a certain table, some words have both halves negative; several have only the right half negative; some have only the left half negative; and others have both halves positive. We wish to know the total number of each type of word. The table is located at memory location 0.01000 through 0.01500, and locations

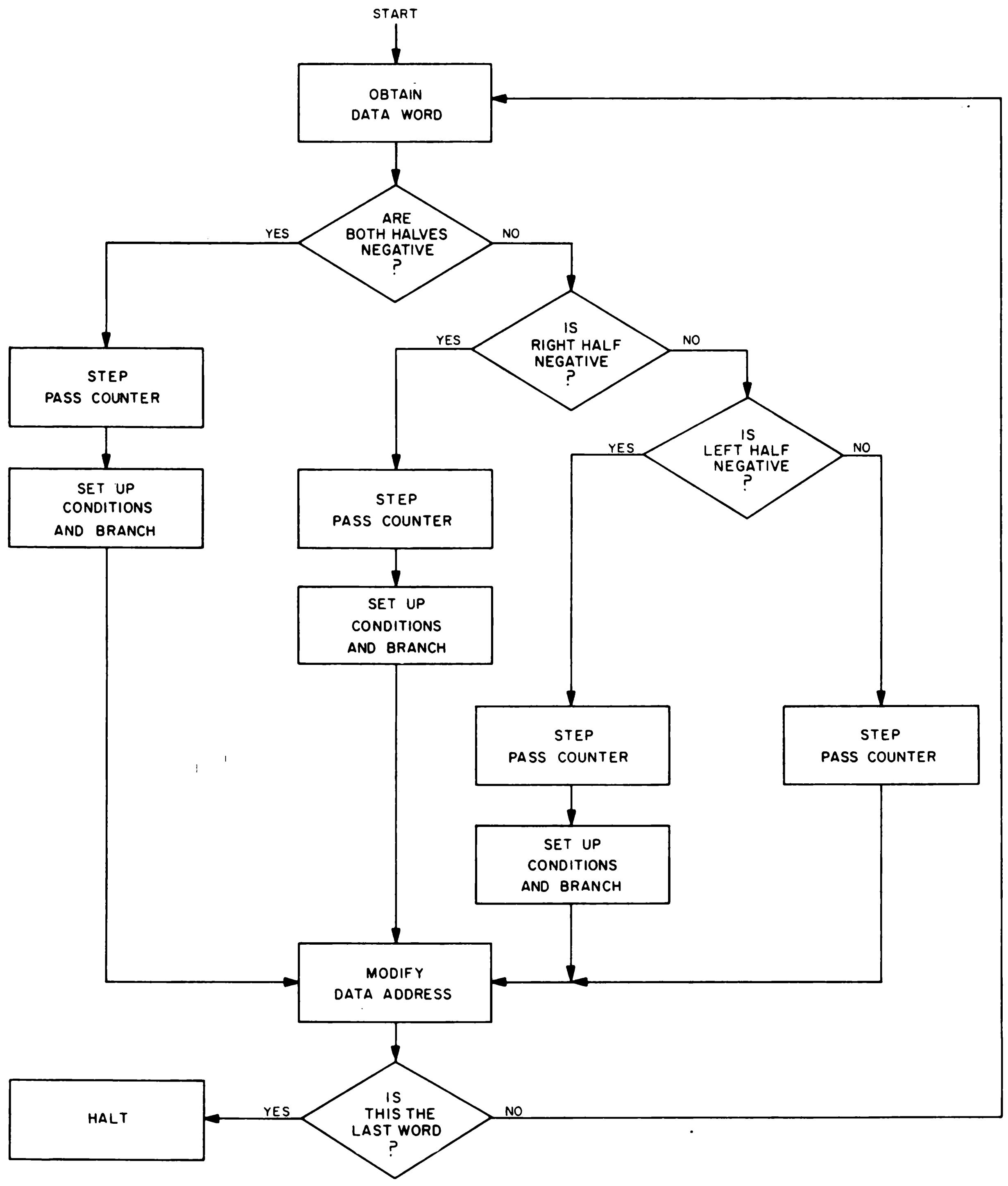


Figure 10-2. Counting by Use of the AOR Instruction, Preliminary Flow Chart

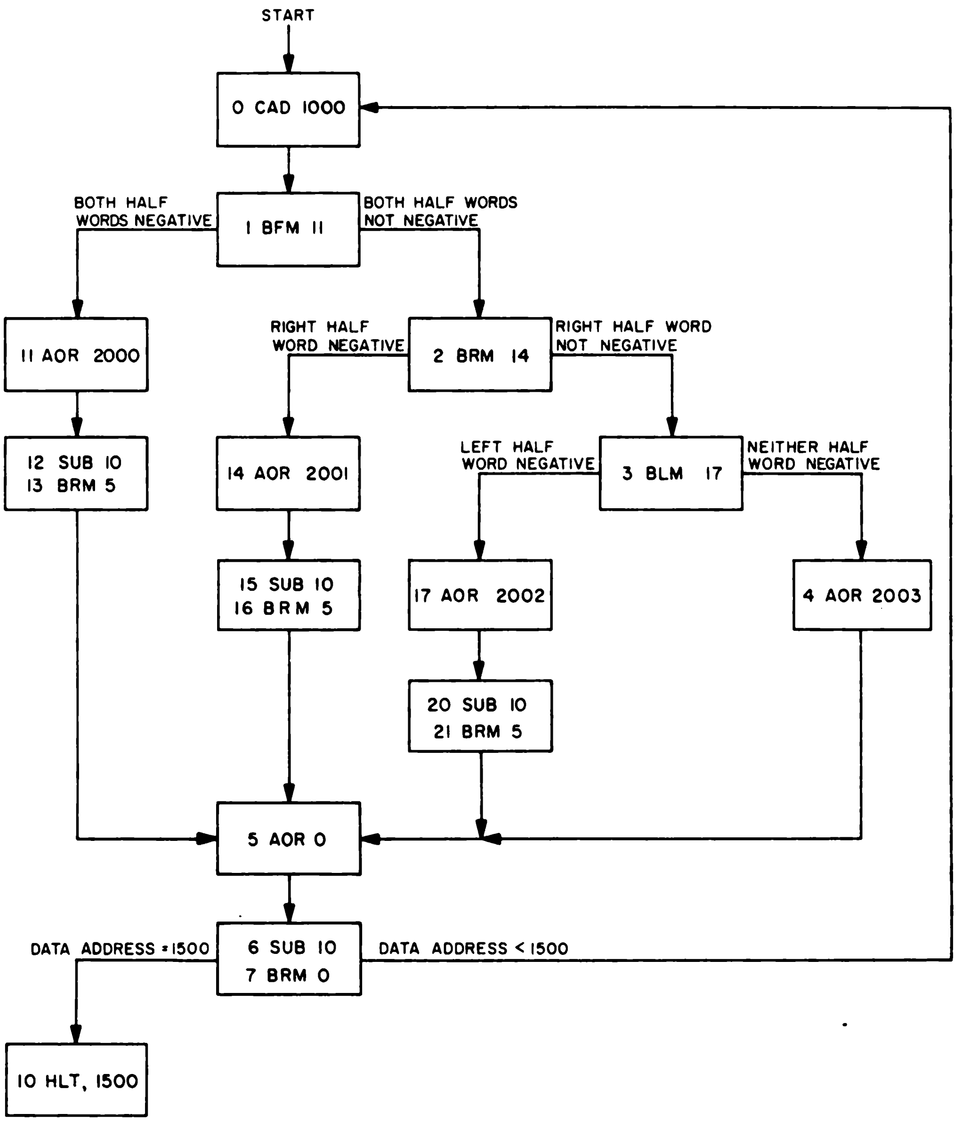


Figure 10-3. Counting by Use of the AOR Instruction, Final Flow Chart

0.02000 - 0.02003 will be used to store the number of full negative, right negative, left negative, and full positive words, respectively. The preliminary flow chart for this program is shown in Figure 10-2.

In this type of program, we can make good use of the various branching instructions to test for the conditions we are interested in. The final flow chart is shown in Figure 10-3. Notice that after the four possible combinations of half words have been checked, we step one of the four pass counters and branch (of fall through) to the data address modification which uses an AOR instruction in exactly the same manner as the program discussed earlier. The program using the pass counters is listed in Table 10-3.

Table 10-3

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

CHAPTER 11 - INDEXING

GENERAL

As previously explained, indexing is the process whereby the address portion of the instruction word is modified so that the same instruction may be used repeatedly but with a different operand each time it is executed. An example of this was given in the previous problems using the AOR instruction, where the AOR Instruction was used to modify the address portion of the instruction. While this method is certainly valid, it leaves several things to be desired. For instance, each time we wish to modify the address portion of the instruction, it is necessary to read the word out of memory, perform the addition, and then write the word back into memory. This is time-consuming and would make a program requiring several hundred modifications excessively long. In addition, notice that we can add only one to the address portion of the instruction word. If we wished to select every other register from a table of operands, two AOR instructions would be required to increase the address portion of the instruction by two, or else a constant of two would have to be used to perform the address modification. Finally, since the modification of the address takes place in the arithmetic element, overflow may be encountered when a true overflow condition does not exist.

From the above discussion, it can be seen that indexing by this method is unsatisfactory for our purposes. Therefore, the AN/FSQ-7 and AN/FSQ-8 makes use of four index registers to perform address modifications. These registers may be loaded with a value of up to 2^{16} (177777) and then added to the address portion of the instruction word. The index register actually contains 17 bits; however, the sign bit is used strictly as a control bit and does not affect the amount by which an address is changed. It is also possible to reduce the contents of the index register at any time by a separate instruction, causing the next address selected to be lower than the one previously selected by the instruction concerned. Index registers are especially valuable when we are writing computer programs because the address modification of an instruction does not require any additional time over and above the normal execution time of the instruction. Thus, when we wish to modify the address portion of an ADD instruction, we can do so and still execute the instruction in 12 usec, using the modified data address.

By referring to a block diagram of the Central Computer System, it can be seen that index registers designated 1, 2, 4, and 5 are transferred to the address register. The right accumulator is sometimes used as an index register and is designated as index register 3. It also may be transferred to the address register. Actually, only one of these five registers is selected at one time and its contents transferred to a portion of the address register, known as the index adder, during PT 7-11. Of course, the instruction from memory is also transferred out during this time, with the address portion of the word going to the address register. The original data address and the contents of the selected index register are added in the index adder, and the modified address is transferred to the memory address register. All of this occurs during PT 7-11, and therefore does not lengthen execution time of the instruction.

RESET INDEX REGISTER INSTRUCTION

The RESET INDEX REGISTER (XIN) instruction provides us with the capability of loading a value into index registers 1, 2, 4, or 5. No provision is made for loading the right accumulator with the instruction because it may be loaded in a variety of ways through the use of other instructions (such as CAD, etc). The address portion of this instruction does not refer to a memory address but contains the value which we wish to place in the specified index register. The index register which is to be loaded is designated by the first three bits in the left half-word of the XIN instruction. Only one index register may be loaded with one XIN instruction. Thus, if we wish to load index register 4 with a value of 100_{10}, we would write the instruction as 4 XIN 144 ($100_{10} = 144_8$). The octal operation code for the XIN instruction is 754, and it requires 6 usec to execute. Because it is used to load an index register, this instruction cannot be indexed by another index register.

BRANCH ON POSITIVE INDEX INSTRUCTION

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

The BPX instruction is executed in the following manner. The sign bit of the selected index register is sensed, and, if it is positive, the program counter, which contains an address one greater than the BPX instruction itself, is transferred to the right A register. Then the address register, which contains the memory location we wish to branch to, is transferred to the program counter and then to the memory address register. The address register is then cleared and loaded with the complement of bits L10-1L15. The specified index register and the complemented number are then added in the index adder, and the resultant difference, if positive, is placed back in the proper index register. Because no operand is required from memory, the BPX instruction is executed in 6 usec. In addition, the BPX instruction cannot be indexed by another index register.

It is important to remember that the sign of the index register is sensed before its contents are reduced, since this fact determines the amount with which the index register is loaded originally. For example, if we wish to select five sequential data addresses by use of an index register, we will load the index register with a value of 3. Some justification for this may appear necessary, but if we examine a sample program, the reason will become apparent. Assume that we wish to take the sum of the data located in addresses 0.01000 - 0.01004. The program is listed below:

Table 11-1

LOCATION	OPERATION	ADDRESS
0.00000	1 XIN	0.00003
0.00001	CAD	0.01000
0.00002	1 ADD	0.01001
0.00003	1 $\text{BPX}_{(01)}$	0.00002
0.00004	FST	0.10000
0.00005	HLT	0.00000

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

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

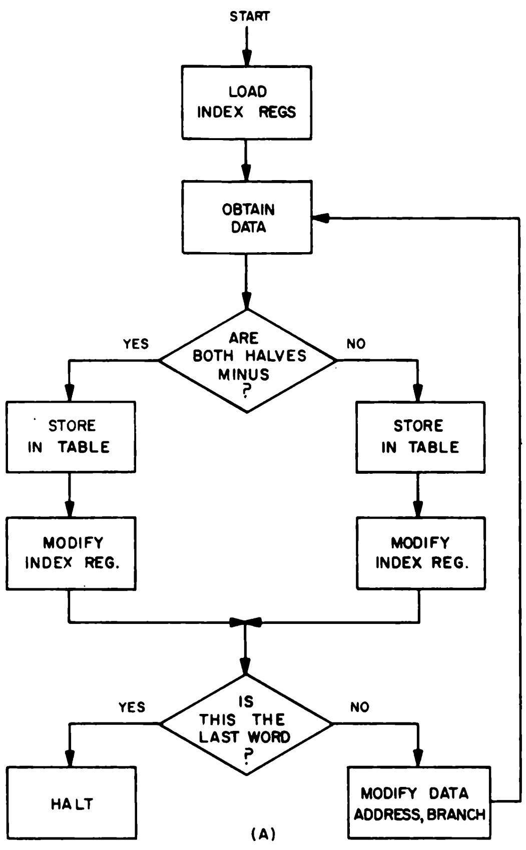
APPLICATIONS OF THE BPX INSTRUCTION

The following examples show how the BPX instruction may be used in various programs. Assume that we wish to sort through a table that is stored in data locations 0.02500 through 0.03000. We want to place all numbers with both half-words negative into a table starting at location 0.00500 and the rest of the numbers into a table starting at location 0.04000. The sample flow chart for this program is shown in Figure 11-1 (Part A). We have no way of knowing how many, if any, of these numbers are negative in both half-words; therefore, it is wise to provide a halt after each table address modification in the event that all numbers were placed in one table or the other. Part B of Figure 11-1 gives the final flow chart for this problem. The program is also listed below.

Table 11-2

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

As another example of the uses of the BPX instruction, assume that we wish to program a delay of 120 usec in to the Central Computer System. There are various



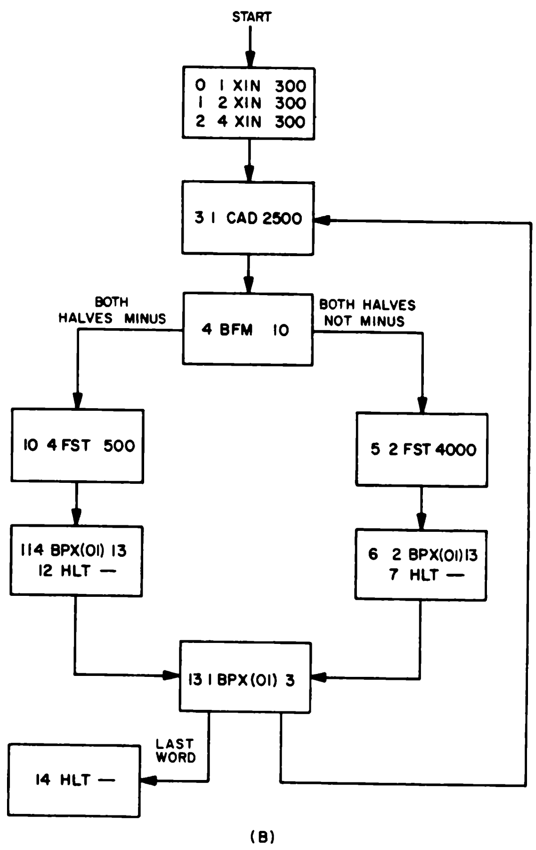


Figure 11-1. Table Sorting by Using the BPX Instruction

ways in which this could be done, but using the BPX instruction to branch back to itself is probably the one method which uses the least space. A sample program showing the delay is listed below.

Table 11-3

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

Step 0.00002 will load index register 5 with a value of 22_8 or 18_{10}. This step consumes 6 usec. The next step causes the index register to be stepped down by a decrement of one and to branch to the same instruction. What we are doing, in effect, is continually sensing the sign of index register 5, waiting for it to go negative. The sign bit will be positive 18_{10} times; however, the branch condition will have been met before the index register is stepped from 0.00001 to 2.00000, so the BPX instruction will be executed 19_{10} times or a total of 114 usec. Adding the 6 usec for the execution of the instruction will give the required 120 usec delay.

USING THE BPX INSTRUCTION AS AN UNCONDITIONAL BRANCH

There is one important feature of the BPX instruction which we have not discussed. Although the name of the instruction implies the use of an index register, it is also possible to use this instruction as an unconditional branch. This is accomplished by omitting a number from index indicator bits L1-L3. When this is done, the BPX instruction will automatically transfer program control to the address specified in the right half portion of the BPX instruction. The execution time remains at 6 usec, and the octal operation code is simply 510. When the BPX instruction is used in this manner, it can often save several steps in a routine.

For instance, let us consider the number-sorting routine which was presented in Figure 9-3. This program sorted three numbers, found the largest and stored it in a specified location. The BLM instruction was used in this program and 18_{10} steps were required. Now let us solve the same program using the BPX instruction as an unconditional branch. The three numbers were A, B, and C, and were stored in the left half portions of locations 0.00100, 0.00101, and 0.00102, respectively. The largest number was to be stored in location 0.00150. The preliminary flow chart for this program is shown in Figure 11-2. Notice that this flow chart arrives at the same place from two

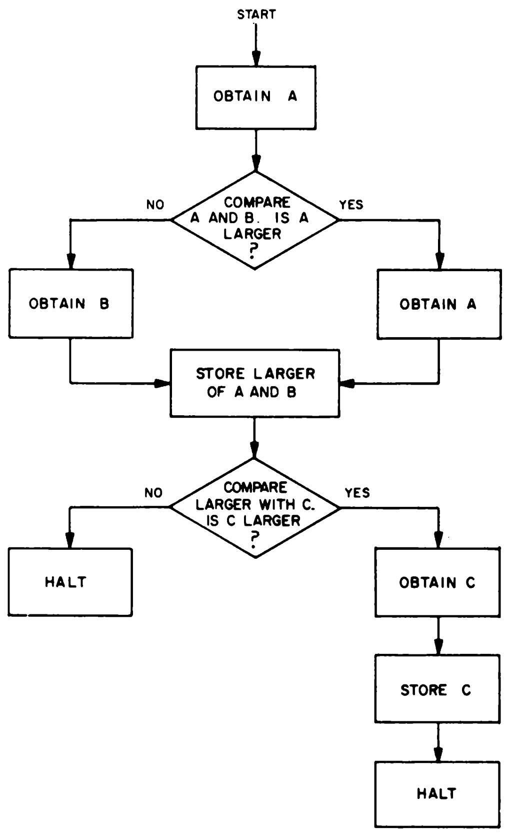


Figure 11-2. Number-Sorting Using Unconditional Branch, Preliminary Flow Chart

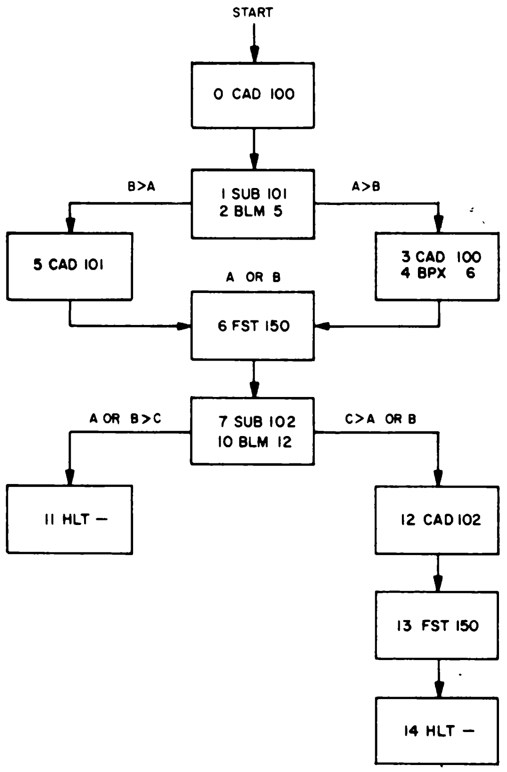


Figure 11-3. Number-Sorting Using Unconditional Branch, Final Flow Chart

different outcomes; since the computer proceeds sequentially, some provision must be made to allow us to do this. The BPX instruction is what is used, and the final flow chart is listed in Figure 11-3. The program is written in 13_{10} steps, a saving of five memory locations over the former program. This program is listed below. This is just one example of the use of an unconditional branch, but it serves to show the importance of this instruction. It is also used widely when entering and leaving program subroutines and with various other programming techniques. The BPX instruction as an unconditional branch is more fully explained on the following pages.

Table 11-4

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

INDEXING TECHNIQUES

GENERAL

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

ADDITIONAL USES OF THE BPX INSTRUCTION

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

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

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

Table 11-5. BPX Instruction Configurations

INSTRUCTION	ACTION
0 BPX (-)	Unconditional branch
3 BPX (-)	Unconditional branch

1 BPX(01)	Branch if index register 1 is positive and reduce its contents by one
2 BPX(01)	Branch if index register 2 is positive and reduce its contents by one
4 BPX(01)	Branch if index register 4 is positive and reduce its contents by one
5 BPX(01)	Branch if index register 5 is positive and reduce its contents by one
1 BPX(00)	Branch if index register 1 is positive
2 BPX(00)	Branch if index register 2 is positive
4 BPX(00)	Branch if index register 4 is positive
5 BPX(00)	Branch if index register 5 is positive
6 BPX(-)	No operation for 6 usec
7 BPX(-)	No operation for 6 usec

RESET INDEX REGISTER FROM RIGHT ACCUMULATOR INSTRUCTION

The RESET INDEX REGISTER FROM RIGHT ACCUMULATOR (XAC) instruction is used to load the specified index register with the contents of the right accumulator. The instruction is executed by first clearing the address register and then transferring the contents of the right accumulator into the address register. The address register then transfers its contents to the selected index register, which has also been cleared. The octal operation code for the XAC instruction is 764, and it requires 6 usec to execute. This instruction does not utilize the address portion, since the index indicator bits determine where the contents of the right accumulator are to be placed. For this reason, the XAC instruction is not indexable.

As an example of the use of the XAC instruction, let us assume that we are going to process some tactical data on the AN/FSQ-7 or AN/FSQ-8. We are interested only in data that has been received after midnight, but we do not know how much data has been received. We shall assume that the data we are concerned with is the status of various missiles at various sites and that all status reports transmitted to the AN/FSQ -7 or AN/FSQ-8 after midnight will have a 1 in the LS bit position. Memory locations 0.00300 through 0.00550 are reserved as a block for unsorted missile status reports, and locations 0.06000 through 0.06250 are reserved as a block for those reports received after midnight. Once the reports have been sorted, we wish to arrange them in a form in which they may be further processed. Our problem here is not quite as simple as some we have dealt with in earlier examples, because more than one operation is involved. First, we must determine what data we are going to process, then temporarily store it, and, at the same time, record the amount of data that we will be processing. It is obvious that index registers will be used in this operation to obtain the raw data.

Recording the amount of usable data can be done by stepping a cleared location. The flow chart for this problem is shown in Figure 11-4, and the program is listed below.

Wherever possible, preliminary and final flow charts will be consolidated in one figure (as has been done in Figure 11-4) throughout the remainder of this chapter.

Table 11-6

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

It can be seen that this performs several operations with relatively few instructions. Besides performing the preliminary sorting necessary, the program will set an index register so that further processing may be done on the reports stored in location 0.06000 - 0.06250. It is necessary to reduce the contents of the index register after

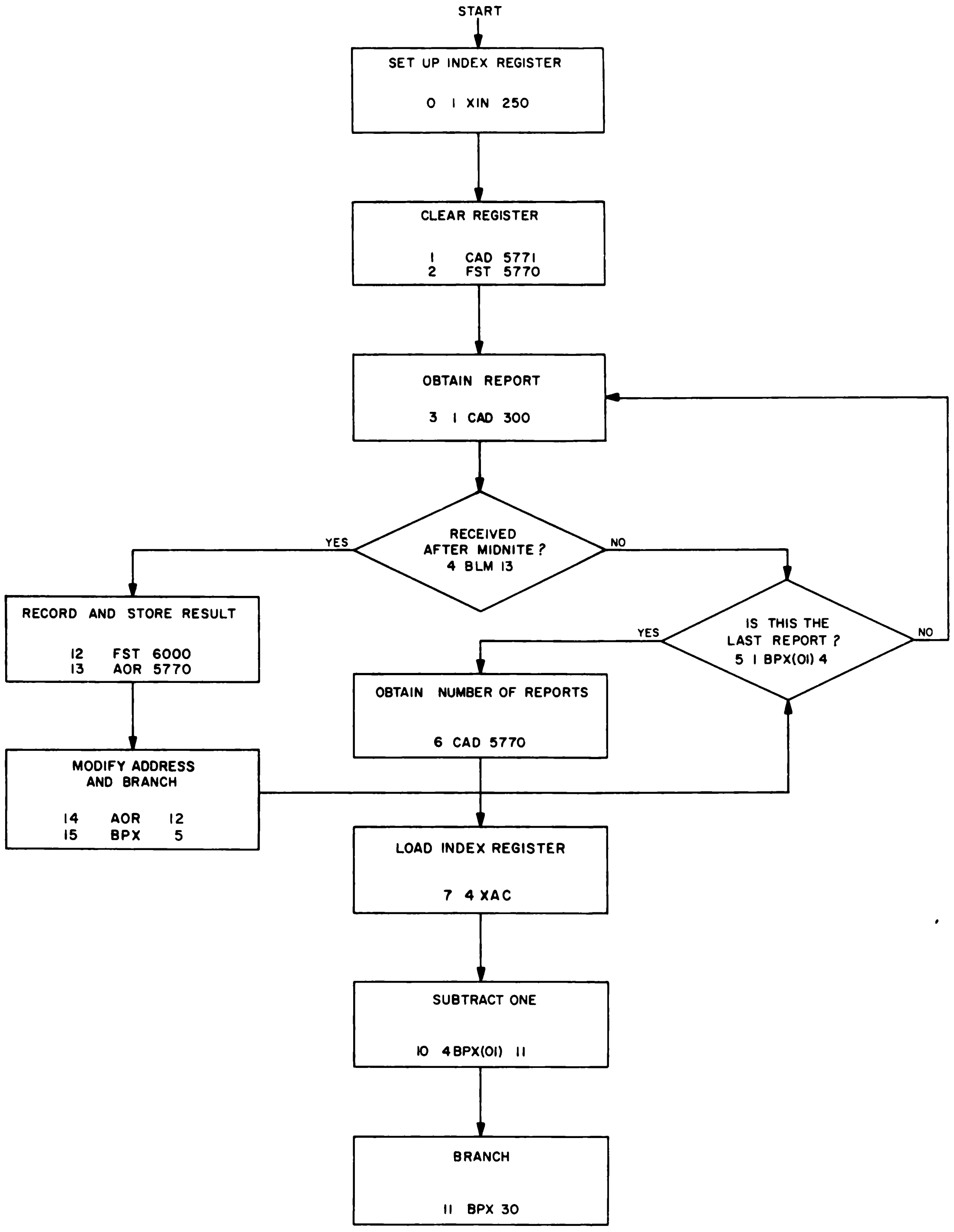


Figure 11-4. Data Sorting and Counting Program Flow Chart

it is loaded so that the proper number of loops will be executed. Another instruction then branches to a routine which will then do the final processing on the status reports.

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

Table 11-7

LOCATION	OPERATION	ADDRESS
0.00000	1XIN	0.00002
0.00001	CAD	0.01420
0.00002	1SUB	0.00060
0.00003	BFZ	0.00006
0.00004	1BPX(01)	0.00001
0.00005	HLT	0.00000
0.00006	1CAD	0.13350
0.00007	2XAC	0.00000
0.00010	BPX	(To desired routine)
0.01420		Result of previous program
0.00060	0.00000	0.00004
0.00061	0.00000	0.00010
0.00062	0.00000	0.00020
0.13350	0.00000	0.00002
0.13351	0.00000	0.00003
0.13552	0.00000	0.00004

Here, it can be seen that while we are dealing with the values 4_8, 10_8, or 20_8, we wish to load an index register with corresponding values of 3_8, 4_8, or 5_8. In addition, this program contains one more feature that we have not discussed yet, but which is very useful in programming. This is the HLT instruction in location 0.00005. We are assuming that the previous program has placed one of the three allowable results in the right accumulator. If this is the case, the program will always branch around the HLT instruction, even on the last pass through it. However, if, for some reason, an incorrect value was stored in location 0.01420, the program would make three unsuccessful attempts to compare and then fall through to the HLT instruction. This is known as an "error halt" because, under normal circumstances, we would not halt but branch to the desired routine after setting index register 2. Therefore, if the program does halt at this location, we know the result of the previous program was incorrect.

The XAC instruction also has a configuration that enables it to be used for more than its stated purpose. The various configurations are listed below.

Table 11-8

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

STORE "A" REGISTER INSTRUCTION

The STORE "A" REGISTER (STA) instruction is used to replce the right half portion of the specified memory location with the contents of the right A register. The right A register is used as the modifying register in this instruction because many other registers utilize it during their execution, and its contents are often valuable. One example is the entire class of branching instructions which automatically transfer the contents of the program counter to the right A register during their execution. An application of this is shown below. The STA instruction is executed in much the same manner as the RST instruction except that the right A register is used to supply the new address portion of the memory location rather than the right accumulator.

The STA instruction requires 18 usec to execute and may be indexed. It has an octal operation code of 340.

ADD INDEX REGISTER INSTRUCTION

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

The ADX instruction is used primarily when it is desired to preserve the contents of an index register for a particular reason. Up to this time, we have had no way of obtaining the contents of an index register once it has been loaded. However, with the ADX instruction, we can specify an address portion of 0.00000 and thus obtain the value of the specified index register which will be transferred to the right A register. Then a STA instruction may put the number in a desired location.

PROGRAMMED USE OF THE ADX AND STA INSTRUCTIONS

The main reason for preserving the contents of an index register is to set up control for some other program or because the index register is required midway through the execution of a program for loop control of another program. This occurs frequently in various programs run on the AN/FSQ-7 and AN/FSQ-8 because there may be several routines or subprograms that require indexing control and there are only four index registers. If we are executing an indexed program which is being controlled by index register 1, and a more important program requests the use of the same index register, it is necessary to preserve the original contents of the index register so that the program in progress at the time of interruption may continue from the point of interruption once the intervening program is completed. This can best be shown by an example of just such an occurrence. Assume that we have a program which is contained in locations 0.00150 through 0.00156. The function of the program is to add several numbers together with the use of an index register. If, at some point in the addition, a large negative number is encountered which causes the right accumulator sum to go negative, we wish to branch to a number-sorting program using the same index register, which is contained in locations 0.01625 through 0.01634. After this program is completed, we wish to return to our original program and complete it from its point of interruption. The student should not be concerned with the actual function of these two programs; the illustration is to show the transfer of control back and forth between these routines, one of which has priority over the other. The fact that priority exists has been established by the fact that a control word containing a large negative value may have been placed in the table of operands used by the first program. Thus, even a third program enters into the picture. However, this fact is beyond the scope of the example at this time. Assume that this condition may occur and that the program must be written to include this occurrence. The flow chart for this program (or programs) is shown in Figure 11-5.

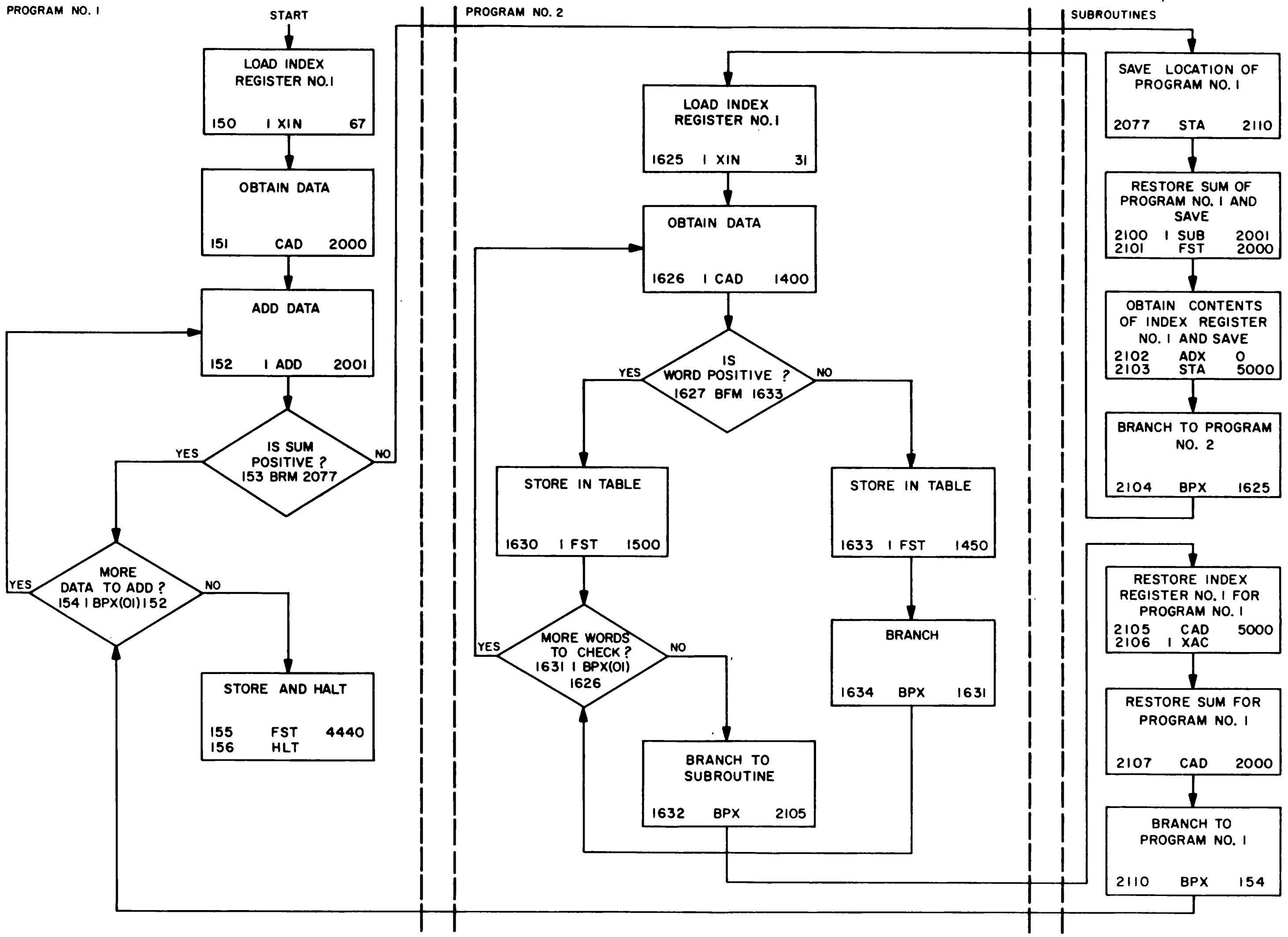


Figure 11-5. Flow Chart Showing One Index Register Used in Two Programs

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

CHAPTER 12 - ADVANCED COMPUTER INSTRUCTIONS

CLEAR AND ADD MAGNITUDE INSTRUCTION

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

DIFFERENCE MAGNITUDE INSTRUCTION

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

ADD B REGISTERS INSTRUCTION

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

MULTIPLY INSTRUCTION

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

TWIN AND MULTIPLY INSTRUCTION

The TWIN AND MULTIPLY (TMU) instruction multiplies the left half-word contained in the specified memory location by the contents of the left and right accumulator. Thus, the TMU instruction causes the left half-word to be used as the multiplier and in both the left and right arithmetic elements. Aside from this difference, the TMU instruction is executed in the same manner as the MUL instruction. The execution time of the TMU instruction is 16.5 usec, plus or minus 1/2 usec, and is an indexable instruction. It is designated by an octal operation code of 254.

DIVIDE INSTRUCTION

Execution of the DIVIDE (DVD) instruction produces the quotient of the contents of the specified memory address (divisor) and the contents of the combined accumulators and B registers (dividend). The execution of this instruction leaves an unsigned quotient of 16 significant bits in the B register and the remainder and sign of the quotient in the accumulators. The Central Computer System carries out the process of trial subtractions and shifting with the use of 2-mc pulses from the time pulse generator, as with the MUL instruction. However, a trial subtraction cannot be performed with one pulse, so the pulses are separated into groups of five pulses each, called divide time pulses (DVTP). Each DVTP cycle, whose pulses are numbered DVTP O-DVTP 4, causes the execution of one trial subtraction. Sixteen trial subtractions are required, yielding an arithmetic pause time of 39.0 usec plus or minus 1/2 usec. The addition of the normal PT-OT cycles brings total execution time of the DVD instruction to 51.0 usec plus or minus 1/2 usec. The DVD instruction is indexable and is specified by an octal operation code of 260.

TWIN AND DIVIDE INSTRUCTION

The TWIN AND DIVIDE (TDV) instruction, which operates like all twin instructions, uses the left half-word contained in the specified memory location as the divisor for both the left and right arithmetic elements. The TDV instruction is executed in the same manner as the DVD instruction. Execution time of the TDV instruction is 51.0 usec plus or minus 1/2 usec; this instruction is indexable. The octal operation code for this instruction is 264.

SHIFT LEFT AND ROUND INSTRUCTION

The SHIFT LEFT AND ROUND (SLR) instruction provides the means for 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) in the combined accumulators and B registers, and the result of a division leaves a 16-bit unsigned quotient plus the remainder in the combined accumulators and B registers. Neither of these results is compatible with the word length of the AN/FSQ-7 or AN/FSQ-8, since we deal with half-words of 16 bits, and these numbers are almost twice as long as the maximum length we can handle. Therefore, some method must be found to reduce these numbers to 15 significant bits and yet preserve the most accuracy it is possible to give to the magnitude of the number. This is accomplished by the SLR instruction, whose execution is explained below.

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

SHIFT INSTRUCTIONS

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

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

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

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

The time required for the execution of the shift instructions depends on the number of shifts involved. If six or less shifts are specified the instruction may be executed in 6 usec. If more than six shifts are required, the Central Computer System uses an arithmetic pause to complete the operation. Execution time for shifts of six or more can be determined by the formula: $t = \frac{7+N}{2}$, where N equals the number of decimal shifts that are called for. For example, DSR 10 would require $\frac{7+8}{2}$ = 15/2 or 7.5 usec to execute. It should be noted that only bits R10 through R15 are decoded to determine the number of shifts; therefore, the maximum number of shifts that may take place during any one instruction is 63_{10} or 77_8. If more than 77_8 shifts are called for, only the value represented in bits R10 and R15 will actually be performed. For instance, if we try to shift 105_8 positions, only 5_8 shifts will be executed. A description of the various shifting instructions used in the AN/FSQ-7 and AN/FSQ-8 is presented below.

DUAL SHIFT LEFT INSTRUCTION

The DUAL SHIFT LEFT (DSL) instruction combines the accumulator and B register of both arithmetic elements into two 32 bit registers which may be shifted only to the left. The sign bits of both accumulators are duplicated in bit 15 of the B registers as the shifting operation begins, and bits shifted out of accumulator bit 1 positions are lost. The DSL instruction is designated by the octal operation code of 400.

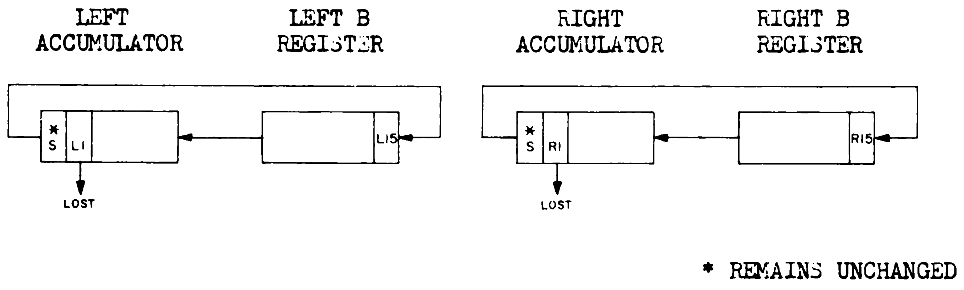


Figure 12-1. Dual Shift Left (DSL)

DUAL SHIFT RIGHT INSTRUCTION

The DUAL SHIFT RIGHT (DSR) instruction combines the accumulator and B register of both arithmetic elements into two 32 bit registers for the purpose of shifting numbers to the right. The sign bits of both accumulators are duplicated in accumulator bit 1 positions, and bits shifted out of B register bit 15 positions are lost. An octal operation code of 404 is used to specify a DSR instruction.

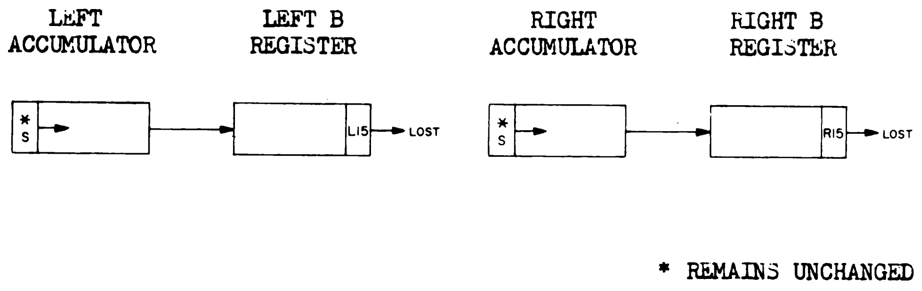


Figure 12-2. Dual Shift Right (DSR)

LEFT ELEMENT SHIFT RIGHT INSTRUCTION

The LEFT ELEMENT SHIFT RIGHT (LSR) instruction combines the left accumulator and left B register into a 32 bit shifting register. The right accumulator and right B register are not affected by the execution of this instruction. The sign of the left accumulator is duplicated in the L1 bit position, and all bits shifted out of bit 15 of the B register are lost. The LSR instruction is designated by an octal operation code of 440.

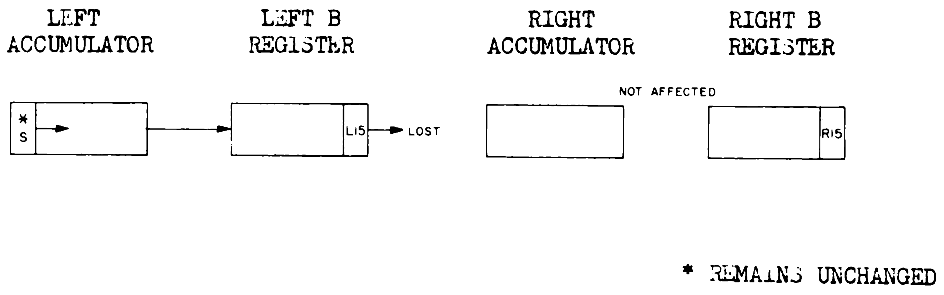


Figure 12-3. Left Element Shift Right (LSR)

RIGHT ELEMENT SHIFT RIGHT

This instruction is similar to the LSR instruction except that it deals with the right arithmetic element only. Thus, the right accumulator and right B register are combined into a 32 bit shifting register whereas the left arithmetic element is not affected. The RIGHT ELEMENT SHIFT RIGHT (RSR) instruction duplicates the sign bit of the right accumulator in the R1 bit position, and all bits shifted out of bit position 15 of the right B register are lost. The RSR instruction has an octal operation code of 444.

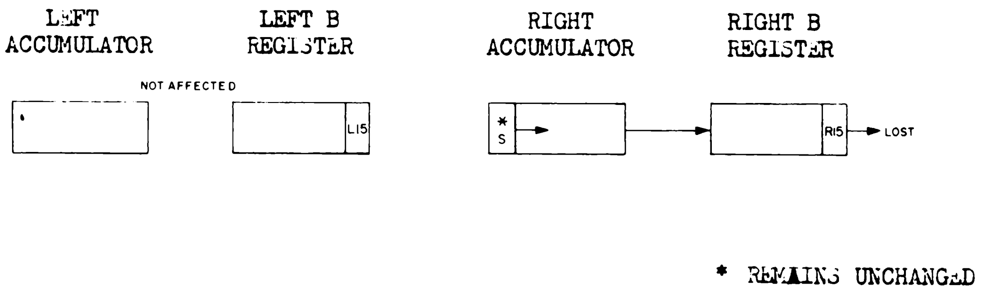


Figure 12-4. Right Element Shift Right (RSR)

ACCUMULATORS SHIFT LEFT INSTRUCTION

The ACCUMULATORS SHIFT LEFT (ASL) instruction shifts both accumulators to the left the number of places specified by the right half-word. The execution of this instruction does not affect the contents of the B registers. The sign bits are duplicated in bit position 15 of the accumulators, and all bits shifted out of positions L1 and R1 are lost. Because this instruction does not combine the accumulators and B registers, only the 15 magnitude bits of the accumulators may be shifted. Thus, an ASL 17_8 (or greater) will duplicate the contents of the sign bits in all positions of the accumulators. The ASL instruction is specified by an octal operation code of 420.

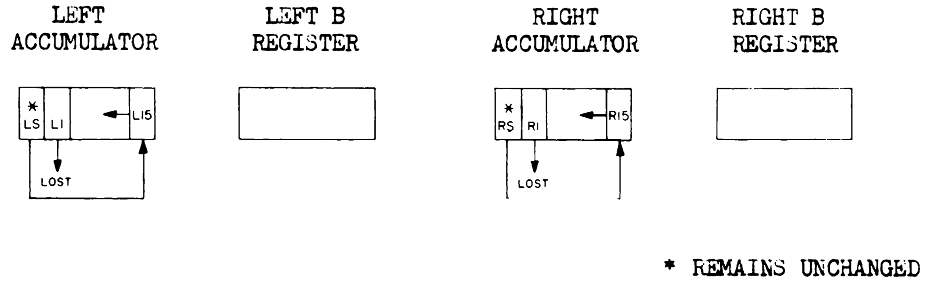


Figure 12-5. Accumulators Shift Left (ASL)

ACCUMULATORS SHIFT RIGHT INSTRUCTION

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

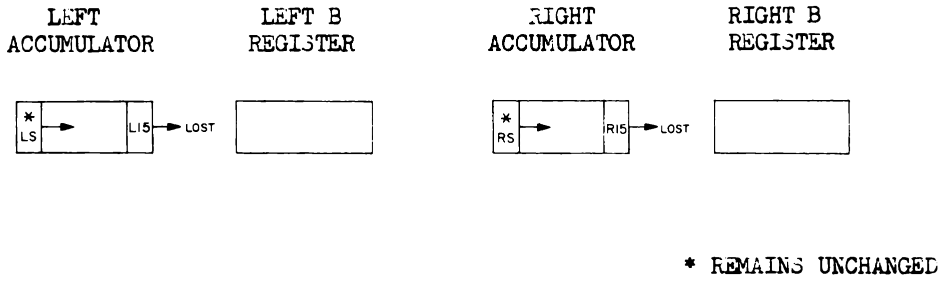


Figure 12-6. Accumulators Shift Right (ASR)

CYCLE INSTRUCTIONS

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

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

DUAL CYCLE LEFT INSTRUCTION

The DUAL CYCLE LEFT (DCL) instruction combines the accumulators and B register of each arithmetic element into a 32 bit cycling register. Bits shifted out of the accumulator sign bit positions are re-entered into the bit 15 position of the B registers. Bits shifted out of the sign bit position of the B registers are shifted into bit 15 of the accumulators. When more than 40_8 shifts are specified, the final result will be the number specified in the right half-word minus 40_8. For instance, when DCL 50_8 is specified, 50_8 shifts will take place, but will have the same effect as a DCL 10_8 instruction. The DCL instruction will not cause an overflow and may be executed in 6 usec if six shifts or less are called for. If more than six shifts are specified, execution time is variable. The DCL instruction is designated by an octal operation code of 460.

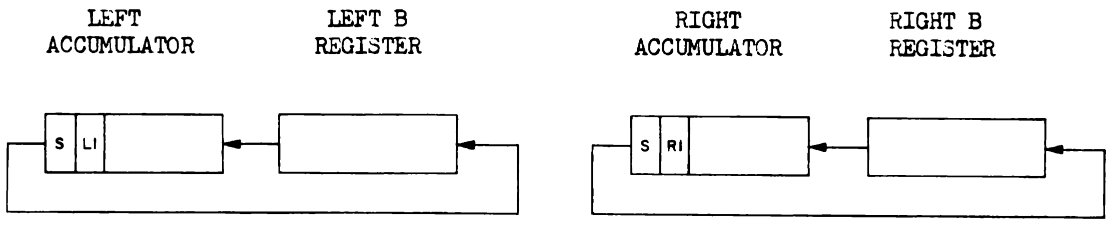


Figure 12-7. Dual Cycle Left (DCL)

FULL CYCLE LEFT INSTRUCTION

The FULL CYCLE LEFT (FCL) instruction is used to interchange bit positions of the left and right half-words in the arithmetic element. Thus, the two accumulators are combined as shown following. The contents of the B registers are not affected by the execution of the FCL instruction. Bits leaving the LS bit position are re-entered into the R15 bit position, and bits leaving the RS bit position are entered into the L15 bit position. If 40_8 shifts are called for, the FCL instruction, specifying 40_8 shifts, will bring the bit positions into their original places and will be equivalent to no shifting action. When more than 40_8 shifts are called for, the effect is the same as it is for the DCL instruction. Execution time of the FCL instruction can be 6 usec if six shifts or less are needed; otherwise it is variable. The FCL instruction, which will not cause an overflow, is designated by an octal operation code of 470.

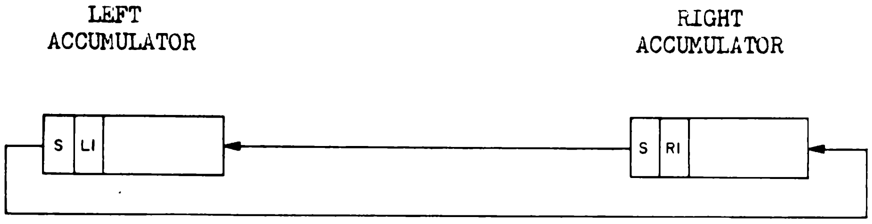


Figure 12-8. Full Cycle Left (FCL)

PROGRAM EXAMPLES OF SHIFT CLASS INSTRUCTIONS

Let us assume that we wish to find out how many numbers stored in memory locations 0.01000 - 0.01031 are larger in absolute magnitude than a particular constant. We can do this by placing the constant in the accumulators and then using a DIM instruction to compare magnitudes. However, the action of the DIM instruction is such that the original contents of the accumulators are first duplicated in the B registers. Therefore, after testing to see if the number meets our condition, we can DCL and restore the number to the accumulators without going into memory again. The flow chart to accomplish this is shown in Figure 12-9.

As another example of the use of shift instructions, let us examine a table that contains information in the right half-word only. The left half-word contains +0. The right half-words have marker bits (1's) placed at one particular position within a half-word, but the bit position used does not have to be the same in each word. What we want to do is make up another table telling us in what bit position the marker bit is located in each half-word. The program will search locations 0.34040 through 0.34047 and store the results in locations 0.00050 through 0.00057. A constant of 20_8 is stored in the right half-word of memory location 0.00300. The program flow chart for this example is shown in Figure 12-10.

After obtaining the location, we check to make sure that it contains an operand by use of the BFZ instruction at step 0.00002. If the branch condition is not satisfied, we know that the right half-word is not 0 and contains a marker bit. Then we shift the right accumulator one bit position to the right and step an indicator register. If the next check for 0 reveals that the right half-word has been shifted so that the marker bit is no longer in the word, the position of this marker can be determined by obtaining a constant of 20_8 and subtracting the contents of the indicator register from it. Thus, if the marker bit for a particular word was located in bit position R9, seven shifts would

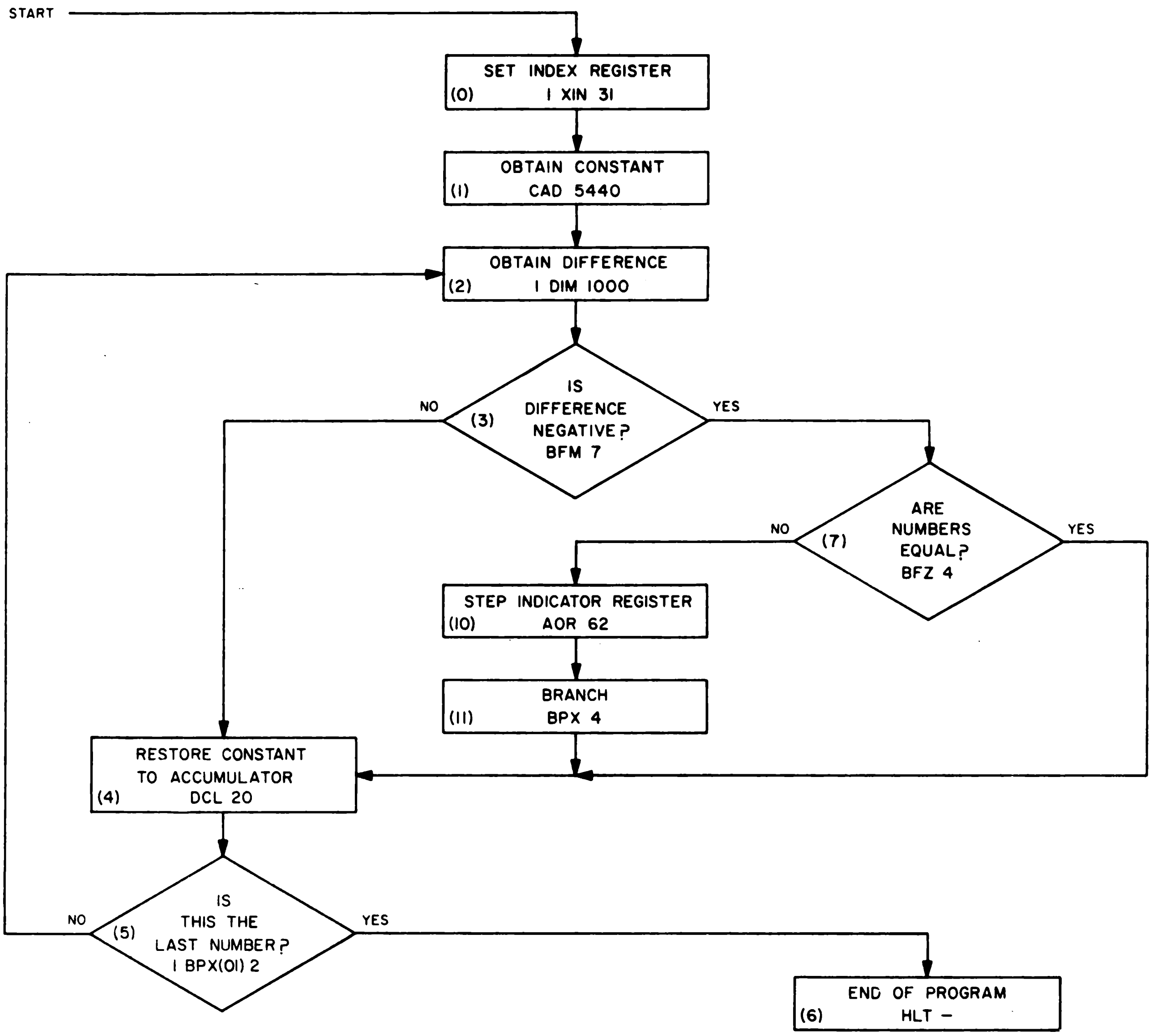


Figure 12-9. Magnitude Sorting Program

be required to remove this bit from the right half-word. Subtracting seven from our constant of 20_8 would leave 11_8 (or 9_{10}) in the accumulator. The marker location is then stored, the indicator register is cleared, and the next operand selected if the program is not complete.

As a final example of the use of shift class instructions in programming, assume that we have operands A, B located in memory location 0.00500, and operands C, D located in memory location 0.00501. We wish to obtain and store the quantity 4(A+C), (B+D)/2 in memory address 0.00700, which is cleared at the start of the program. The program that will perform this is listed in Table 12-1.

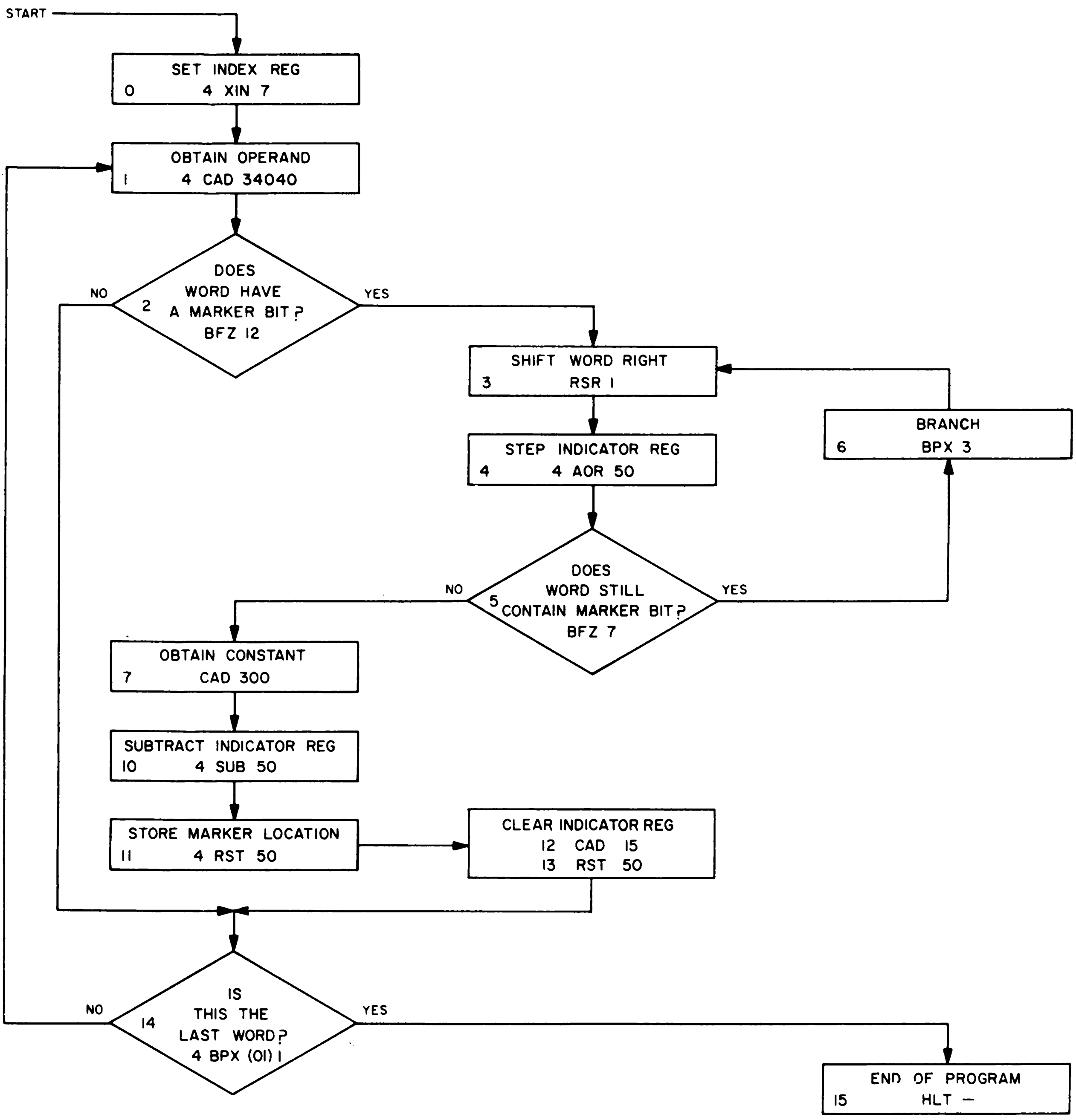


Figure 12-10. Marker Bit Identification Program

Table 12-1

LOCATION	OPERATION	ADDRESS
0.00120	CAD	0.00500
0.00121	ADD	0.00501
0.00122	RSR	0.00001
0.00123	RST	0.00700
0.00124	ASL	0.00003
0.00125	LST	0.00700
0.00126	HLT	0.00000
0.00500	A	B
0.00501	C	D
0.00700		Final result

LOGICAL INSTRUCTIONS

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

EXTRACT INSTRUCTION

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

PROGRAM EXAMPLE OF THE EXTRACT INSTRUCTION

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

Table 12-2

LOCATION	OPERATION	ADDRESS
0.00760	1XIN	0.00400
0.00761	2XIN	0.00400
0.00762	5XIN	0.00400
0.00763	1CAD	0.25000
0.00764	ETR	0.00500
0.00765	BFZ	0.00772
0.00766	1CAD	0.25000
0.00767	2RST	0.40000
0.00770	2BPX(01)	0.00775
0.00771	HLT	0.00000
0.00772	1CAD	0.25000

0.00773	5RST	0.70000
0.00774	5BPX(01)	0.00775
0.00775	1BPX(01)	0.00763
0.00776	HLT	0.00000
0.00500	0.00000	0.00001 (Control word)
0.25000 - 0.31000		Data Storage
0.40000 - 0.40400		Storage for odd numbers
0.70000 - 0.70400		Storage for even numbers

Notice that the control word contains all 0's except for the least significant bit position of the right half-word. This position, which contains a 1, will leave bit 15 of the half-word unchanged during execution of an ETR instruction. After the ETR instruction has been executed, the accumulators will be cleared by the control word with the exception of bit 15. If bit 15 originally contained a 1, it will still contain a 1, and the BFZ condition immediately following the ETR instruction will not be satisfied. If this is the case, the program will fall through to the routine for storing the number in the odd table. If the bit was 0 originally, it will still be a 0, and the program will branch to location 0.00772, where the number will be stored in the even table.

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

LOAD B REGISTERS INSTRUCTION

The LOAD B REGISTERS (LDB) instruction is used to load the B registers with the contents of the memory location specified by the right half portion of the instruction. The LDB instruction could be used when it is desired to place a word in the B registers prior to execution of a program. The main use of the LDB instruction is to place a control word or mask into the B registers in preparation for the execution of another logical instruction, called Deposit, which is explained below. The LDB instruction is indexable and requires 12 usec to execute. It is designated by an octal operation code of 030.

DEPOSIT INSTRUCTION

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

The DEP instruction may be indexed and requires a total of 18 usec to execute due to the fact that the word is obtained from memory, modified, and then returned to memory. An octal operation code of 360 is used to designate a DEP instruction. Because the execution of this instruction is rather difficult to follow, a typical example using a half-word is shown in the table below when the B register contains a mask of 0.111 000 000 111.

Table 12-3. Deposit Instruction Execution

ACTION	ACCUMULATOR	A REGISTER	MEMORY LOCATION
Start	0.010 101 111 000 011	0.000 000 000 000 000	0.000100 011111101
Complement	1.101 010 000 111 100	0.000 000 000 000 000	0.000100011111101
Transfer	1.101 010 000 111 100	0.111 000 000 000 111	0.000100011111101
Logical Multiply	0.101 000 000 000 100	0.111 000 000 000 111	0.000100011111101
Complement	1.010 111 111 111 011	0.111 000 000 000 111	0.000100011111101
Logical Add	1.010 111 111 111 011	0.111 100 011 111 111	0.000100011111101
Logical Multiply	0.010 100 011 111 011	0.111 100 011 111 111	0.000100011111101
Store	0.010 100 011 111 011	0.111 100 011 111 111	0.101100011111011

EXCHANGE INSTRUCTION

The EXCHANGE (ECH) instruction is used to interchange the contents of the accumulators with the contents of the memory location specified by the right half portion of the instruction. Execution of this instruction takes place in the following manner. The word from core memory is first transferred to the A registers, then the accumulator contents are transferred to the memory buffer registers, and on into the location just read. Finally, the A registers are added to the cleared accumulators, ending the execution. Because a memory cycle is required to read the word out and another one is needed to store the accumulator contents, the ECH instruction is a 3-cycle instruction and thus requires 18 usec execution time. This instruction, which may be indexed, is designated by an octal operation code of 350.

USE OF EXCHANGE INSTRUCTION

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

Table 12-4. Relocation Program

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

COMPARE INSTRUCTIONS

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

COMPARE LEFT HALF-WORDS INSTRUCTION

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

COMPARE RIGHT HALF-WORDS INSTRUCTION

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

COMPARE FULL WORDS INSTRUCTION

This instruction is used to compare a full word from memory with the contents of the accumulators. The COMPARE FULL WORDS (CMF) instruction is executed in much the same manner as the CML and CMR instructions. The accumulator contents are complemented and the A registers are cleared. Then the word from memory is transferred to the A registers and a comparison is made. A successful comparison will cause the next instruction in sequence to be executed; a no compare indication will cause the program to skip the instruction immediately following the CMF instruction. It should be remembered that the entire word must compare in order to cause a successful indication; if one half-word compares and the other does not, the result is still a no-compare indication. The CMF instruction is executed in 12 usec and may be indexed. It is designated by an octal operation code of 046.

COMPARE MASKED BITS INSTRUCTION

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

TABLE 12-5. Compare Masked Bits Instruction Execution

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

Compare Difference Instruction

This table illustrates the execution on a half-word only; however, the full word is processed in the same manner. We will assume that the B register is loaded with a mask of 0.000 111 111 000 111.

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

PROGRAM EXAMPLES USING COMPARE INSTRUCTIONS

Suppose we are stepping two indicator registers by two different programs. We want to go through a definite sequence of programs when the contents of the two registers reach the same value, otherwise we will continue to execute the programs which are stepping the indicator registers. We can cause our program control to branch to a short subroutine which will test the equality of the two indicator registers at specific intervals. Assuming that the indicator registers are contained in the right half-words of the memory locations, the subroutine which will test the contents of the registers is given below.

Table 12-6

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

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

Table 12-7

LOCATION	OPERATION	ADDRESS
0.00410	CAD	0.06250
0.00411	LDB	0.03341
0.00412	CMM	0.03325
0.00414	BPX	Execute this program
0.00414	BPX	Read in next program
0.06250	0.56000	0.00000 (Control word)
0.03341	0.77000	0.00000 (Mask of 0.111 111 000 000 000)
0.03325		Current program number

COMPARE DIFFERENCE LEFT HALF-WORDS INSTRUCTION

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

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

COMPARE DIFFERENCE RIGHT HALF-WORDS INSTRUCTIONS

The purpose of the COMPARE DIFFERENCE RIGHT HALF WORDS (CDR) instruction is to test the equality of the right accumulator contents and the operand contained in the right half of the memory location specified by the instructions. As with all compare instructions, the instruction immediately following the CDR instruction will either be executed or skipped, depending on the outcome of the comparison. Execution of the CDR instruction is started by complementing the accumulators and clearing the A registers. The specified memory location contents are then transferred to the A registers, and right A register and right accumulator are compared. Following this, both A registers are added to the complemented accumulators, thereby generating the difference between the numbers. If the final contents of the accumulators are still in complement form, we know that their original contents were equal to or larger than the operand; if the final accumulator contents are positive, it means that the operand was larger than the original value in the accumulators. Twelve usec are required to execute the CDR instruction, which is indexable and is designated by an octal operation code of 043.

COMPARE DIFFERENCE FULL WORDS INSTRUCTION

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

COMPARE DIFFERENCE MASKED BITS INSTRUCTIONS

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

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

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

Table 12-8

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

PROGRAM EXAMPLES USING COMPARE-DIFFERENCE INSTRUCTIONS

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

Table 12-9

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

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

As an example of the use of the CDM instruction, let us check the equality of bit positions R10 - R15 of two registers. We wish to execute a certain subroutine if the contents are equal and another one if the contents are unequal and the number placed in the accumulator is higher in magnitude than the number it is being compared with. This program is listed in Table 12-10.

Table 12-10

LOCATION	OPERATION	ADDRESS
0.00001	CAD	0.03314
0.00002	LDB	0.06000
0.00003	CDM	0.03315
0.00004	BPX	To subroutine A
0.00005	BRM	To subroutine B
0.00006	HLT	0.00000
0.03314	0.00650	0.02540
0.03315	0.43120	0.01630
0.06000	0.00000	0.00077

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

TEST BITS INSTRUCTION

TEST ONE BIT INSTRUCTION

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

TEST TWO BITS INSTRUCTION

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

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

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

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

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

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

BIT SELECTION FOR TOB AND TTB INSTRUCTIONS

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

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

Table 12-11. TOB and TTB Bit Selection

L11-L15	BIT(S) SELECTED	
	TOB	TTB
00	LS	LS*
01	L1	Ls,L1
02	L2	L1,L2
03	L3	L2,L3
04	L4	L3,L4
05	L5	L4,L5
06	L6	L5,L6
07	L7	L6,L7
10	L8	L7,L8
11	L9	L8,L9
12	L10	L9,L10
13	L11	L10,L11
14	L12	L11,L12
15	L13	L12,L13
16	L14	L13,L14
17	L15	L14,L15
20	~~RS~~	RS*
21	R1	RS,R1
22	R2	R1,R2
23	R3	R2,R3
24	R4	R3,R4
25	R5	R4,R5
26	R6	R5,R6

CONT. ON 112

PROGRAM EXAMPLES OF TOB AND TTB INSTRUCTIONS

Previously, we illustrated several programs which required that a test be made on one specific bit to determine a course of action. There are several ways we can get a bit into position for testing. One way is to cycle it into a sign bit position and then use a branch instruction. Another way is to use the ETR instruction and set all the inactive bits in the word to be tested to 0's except the particular bit we are interested in, and then execute a BFZ instruction. Now, with the addition of the TOB and TTB instructions, we can test any bit we desire without shifting or masking. For instance, assume that we wish to check a certain program which had been read into memory to make sure that 17-bit operation had been specified for all instructions within the program. We can do this very simply because we know that the presence of a 1 bit in L12 of an instruction word which does not ordinarily make use of the auxiliary bits indicates that 17-bit operation is desired. (We will assume that the program we are to test contains only 0's in all bit positions except where 17-bit operation is desired.) The program which will perform this check is given below.

Table 12-12

LOCATION	OPERATION	ADDRESS
0.00501	1XIN	0.00123
0.00502	1TOB(14)	0.01340
0.00503	BPX	0.00506
0.00504	1BPX(01)	0.00502
0.00505	HLT	0.00000
0.00506	1ADX	0.01340
0.00507	STA	0.03000
0.00510	HLT	0.00000
0.01340 - 0.01463		Location of program

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

As an example of the TTB instruction application, let us assume that after we have performed the program discussed above, we again want to run through it and store the address of every instruction which specifies the right accumulator (3) as its index register. Since the index indicator contains three bits (L1 through L3), we can use a combination of TOB and TTB instructions to accomplish this job. The program to perform this check is given below.

Table 12-13

LOCATION	OPERATION	ADDRESS
0.00510	1XIN	0.00123
0.00511	2XIN	0.00123
0.00512	1TOB(01)	0.01340
0.00513	1TTB(03)	0.01340
0.00514	BPX	0.00522
0.00515	BPX	0.00522
0.00516	BPX	0.00522
0.00517	1ADX	0.01340
0.00520	2STA	0.00500
0.00521	2BPX(01)	0.00522
0.00522	1BPX(01)	0.00512
0.00523	HLT	0.00000

This program will examine the contents of L1 with the TOB instruction. If L1 is a 1, indicating index registers 4, 5, 6, or 7, the program counter will be stepped an additional time, skipping to the BPX in location 0.00514. This will cause the computer to branch to the end of the program where the next memory location will be selected. If L1 is a 0, indicating index register 0, 1, 2 or 3, the program will execute the TTB instruction in location 0.00515. This TTB instruction will check bits L2 and L3 for a bit combination of 11. If any other bit combination (00, 01, or 10) is found, the program immediately branches to a step which reduces the index register by one and selects a new instruction in the program for testing. If a bit combination of 11 is found, the address of the instruction it is found in is stored, starting at location 0.00623, and the next instruction is selected for testing. Notice that we use two different index registers in this program, one to control the memory locations selected for testing and the other to control the storage of applicable addresses. We could have used the same index register to store the addresses; however, the chances are that the index register controlling the address selection will be stepped several times for each time that we store an address (which contains an instruction using index register 3). Therefore, our table of addresses would not be in sequential order. To achieve sequential order, a separate index register was employed.

Table 12-14

27	R7	R6, R7
30	R8	R7, R8
31	R9	R8, R9
32	R10	R9, R10
33	R11	R10, R11
34	R12	R11, R12
35	R13	R12, R13
36	R14	R13, R14
37	R15	R14, R15

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

CLEAR AND ADD CLOCK INSTRUCTION

EXECUTION

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

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

OPERATE INSTRUCTION

The OPERATE (PER) instruction is used to initiate a wide varity of actions within the AN/FSQ-7 and AN/FSQ-8, such as starting a test pattern generator or rewinding a magnetic tape reel. The PER instruction variation which is to be performed is designated by bits L10-L15 of the instruction word (auxiliary bits). The execution of the PER instruction depends chiefly on a decoding matrix, which allows for the transmission of a pulse to a unit specified by the auxiliary bits. As previously mentioned, this pulse can cause several actions to take place, depending on what unit is receiving the pulse. Execution time of the PER instruction requires 12 usec. Because the address portion of this instruction is meaningless, it may not be indexed. An octal operation code of Ø1- is used to specify a PER instruction. These bits are placed in L4-L9 of the instruction word, leaving all the auxiliary bits free to specify what action is to take place.

Table 12-15

OPERATE UNIT	ACTION	INDEX INTERVAL CODE
Condition lights 1-4	Turn on	01-04
Intercommunication indicators 1-4	Turn on in other Computer	10-13
Test Clock	Complements FF controlling the stepping of the clock. Clock may now be stepped by BPX instructions.	14
Area discriminator (spare)		17
Area discriminator	Turn on for cycle beginning with radar data.	20
Marginal Checking	Start excursion	21
	Stop duplex excursion	22
	Stop simplex excursion	23
I/O Interlock	Clear	27
SD Camera	Start mode 1	31
	Start mode 2	32
Reserved for SD Cameras 3 and 4		33, 34

OPERATE UNIT	ACTION	INDEX INTERVAL CODE
Digital Display	Start at slot 0, 1st section Start at slot 107, 2nd section. (Not used at present)	35
Print operate hubs 1-10	Energize	51, 62
Input System testing	Start LRI and XTL pattern generator and start GFI continuous pattern generator.	63
	Switch GFI pattern generator type I. Has to be in Mode II.	64
	Set GFI azimuth FF.	65
	Set GFI target FF.	66
Selected tape unit	Set prepared	67
	Backspace	70
	Rewind	71
	Write end-of-file	72
Card Punch	Punch what normally goes into columns 17-32 in columns 1-16 on card.	73
	Gang punch 1-16	74
I/O Address counter	Lock at current address until I/O interlock is cleared.	75
Scan counter	Set to 0	76
	Step by 1	77

BRANCH ON SENSE INSTRUCTION

The BRANCH ON SENSE (BSN) instruction provides for a branch of program control to the address specified in the right half portion of the instruction if a designated condition exists. This condition is determined by decoding the auxiliary bits (L10-L15) of the instruction word. We can sense for such things as overflow in either or both accumulators, checking a tape drive unit to see if it is ready to be used, and various other conditions. The execution of the BSN instruction takes place in the following manner. The auxiliary bits are decoded in the same matrix used by the PER instruction. These bits will partially condition one unit selected by the bits. If the unit being sensed is active, it completes the conditioning necessary for a positive indication. The BSN instruction then strobes all the sense units in parallel and causes a branch of program control if a unit is on. Only one unit may be on at the time the check for branch takes place. If a unit is found to be active, the contents of the program counter are transferred to the right A register and the right A register and the contents of the address register are transferred to the program counter. (This is the same action which results when a condition is satisfied for any of the other branch instructions such as BRM and BPX). If the unit being sensed was not active, the branch will not be executed, and the next sequential instruction will be selected. Execution of the BSN instruction requires 12 usec, and it is not indexable. The BSN instruction is designated by octal operation code of 52- in bit positions L4-L9. Bits L10-L15 are thus made available to specify what unit is to be sensed.

Table 12-16

Sense Unit	Condition For Branch	Index Interval Code	BSN Turns Unit Off
Condition lights 1-4	On	01-04	Yes
Inactivity On	On	05	Yes
Tapes not prepared	Not Prepared	10	No
I/O Unit Not Ready	Not Ready	11	No
Left Overflow On	On	12	Yes
Right Overflow On	On	13	Yes
I/O Interlock On	On	14	No
Memory Parity Error	On	15	Yes
Drum parity error (addressable)	On	16	Yes
Tape parity error	On	17	Yes
Duplex Marginal Chg. exc. on	On	20	No
Sense Switches 1-4 (Active)	On	21-24	No
Drum Parity (Status)	On	25	Yes
Simplex MC Exec. on	On	27	No
Duplex Switch	Active	30	No
Printer sense 1, 2	Energized	31, 32	No
Output alarm on	On	33	Yes

Sense Unit	Condition For Branch	Index Interval Code	BSN Turns Unit Off
GFI range signal LRI & XTL timing on	On	34	Yes
SD Camera on	Taking picture	35	No
Display System (On for data coming from TD Drums)	Displaying Track	37	No
Other Com. alarms 1,2	On	41, 42	Yes
Other computer intercom. 1-4	On	43-46	Yes
GFI North Azimuth on	On	47	Yes
Nonsearch Alarm on	On	50	Yes
OB Drum Parity	On	51	Yes
Illegal Address or Section on	On	52	Yes
Output Message Parity	On	53-60	Yes

CHAPTER 13 - MANUAL CONTROLS

DUPLEX MAINTENANCE CONSOLES

The two DUPLEX MAINTENANCE CONSOLES (DMC) of the AN/FSQ-7 equipment are identical. Each console contains the controls and indicators for starting, loading, monitoring, testing, and stopping the computer with which it is associated. The extent to which the operation of a computer can be controlled from its DUPLEX MAINTENANCE CONSOLE depends upon the basic operating status of the computer. Computer status is controlled by the TEST-OPERATE switch.

When the TEST-OPERATE switch is in the OPERATE position, the computer is said to be in OPERATE Status. The OPERATE Status is normally used when running the operational program. DCA (Direction Center Active) is another name for the operational program. While the computer is in operate it can:

1. give alarm indications
2. be started and stopped
3. provide indications of register contents
4. generally provide a means of controlling computer functions.

In the TEST Status (with TEST-OPERATE switch in the TEST position) the primary use of the computer is to perform maintenance. While the computer is in TEST, it can:

1. give alarm indications
2. be started and stopped
3. provide indications of register contents
4. provide means for stepping or cycling through program steps
5. allow marginal checking
6. allow the maintenance man, through use of the controls and indications, to isolate troubles.

Most of the controls on the DMC are Interlocked. In other words, certain conditions must be met before these controls can be used. There are four basic Interlocking Levels and several variations of these four. The four basic Interlocking Levels are:

1. COMPUTE, which is present when the computer is computing (Continue FF set)

2. NON-COMPUTE, which is present when the computer is not computing, (Continue FF clear)

3. TEST, which is present when the TEST-OPERATE switch is in the TEST position.

4. TEST NON-COMPUTE, which is present when the TEST-OPERATE switch is in the TEST position and the computer is not computing (Continue FF clear).

The variations of these Interlocking Levels are found on the following chart, together with their associated controls.

Table 13-1

Name of Control	Interlocking Condition	Logic Page
	These only effect comp. when these conditions are present	
1. TEST-OPERATE Sw	Computer in standby (can be overridden)	7.1.7.
2. Program Continue P.B.	Non-Compute Ground	0.2.1.
3. Program Stop P.B.	-48 Compute	0.2.1.
4. Instruction Step	-48 Test Non-Compute	0.2.1.
5. Memory cycle	-48 Test Non-Compute	0.2.1.
6. Rest FF	-48 TEST	0.2.1.
7. Clear Memory	-48 TEST	0.2.1.
8. Ready I/O Units	None	7.1.13.
9. Master Reset	-48 TEST	0.2.1.
10. Load from Card Reader	-48 Non-Compute	0.2.1.
11. Load from A.M. Drums	-48 Non-Compute	0.2.1.
12. Start from Test Memory	-48 Non-Compute Simulates BPX	0.2.1.
13. Single Pulse	TEST Non-Compute Ground	0.2.2.
14. Sense Sw (4)	None	0.7.4.
15. Select Test Memory	-48 TEST	0.2.1.
16. Memory Normal/Reverse	-48 TEST	0.4.1.
17. T.M. Assign-Unassign	None	7.1.5.
18. T.M. Plugboard	None	7.1.5.
19. Clear Alarms	None	0.2.1.
20. Reset Audible Alarms	None	7.1.14.
21. Complement	Ground TEST Non-Compute	0.2.4.
22. Three-Two-Three Auto	+10 Test Non-Compute	0.2.4.
23. C.P.C. Rotary	+10 Test Non-Compute	0.2.5.

Name of Control	Interlocking Condition	Logic Page
24. Delay-Off-No-Delay	+10 Test	0.2.5.
25. Memory, Drum, Tape Parity alarm	None	0.7.5.
26. Stop-Branch	None	0.2.7.
27. Overflow Alarm	None	0.7.5.

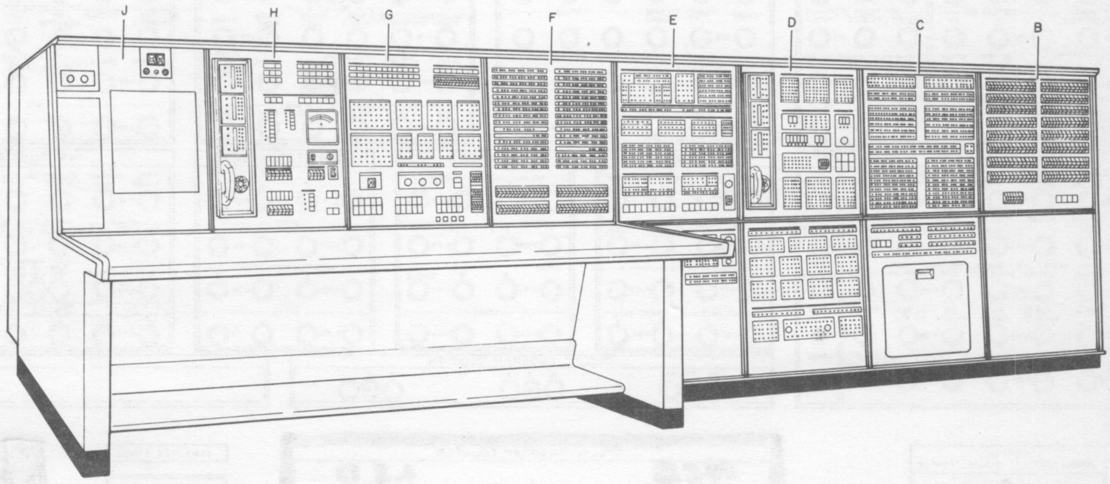


Figure 13-1. Duplex Maintenance Console (Unit 1)

MEMORY NORMAL-REVERSE SWITCH

The function of the NORMAL-REVERSE Switch (Logic 0.4.1.) is to swap the addresses of "Big" and "Little" memories. The reason this switch exists is because of the way programs are normally loaded into the computer. The programs are usually loaded sequentially into memory, starting at location 0.00000. This means that if MEM I is inoperative it is impossible to load a maintenance program to diagnose the problem. By changing the addresses around, the program may be loaded into MEM II to check MEM I.

Table 13-2

NORMAL ADDRESSES		REVERSE ADDRESSES	
MEM I	0.00000 to 1.77777	MEM I	2.00000 to 3.77757
MEM II	2.00000 to 3.77757	MEM II	0.00000 to 1.77777
TEST MEM	3.77760 to 3.77777	TEST MEM	3.77760 to 3.77777

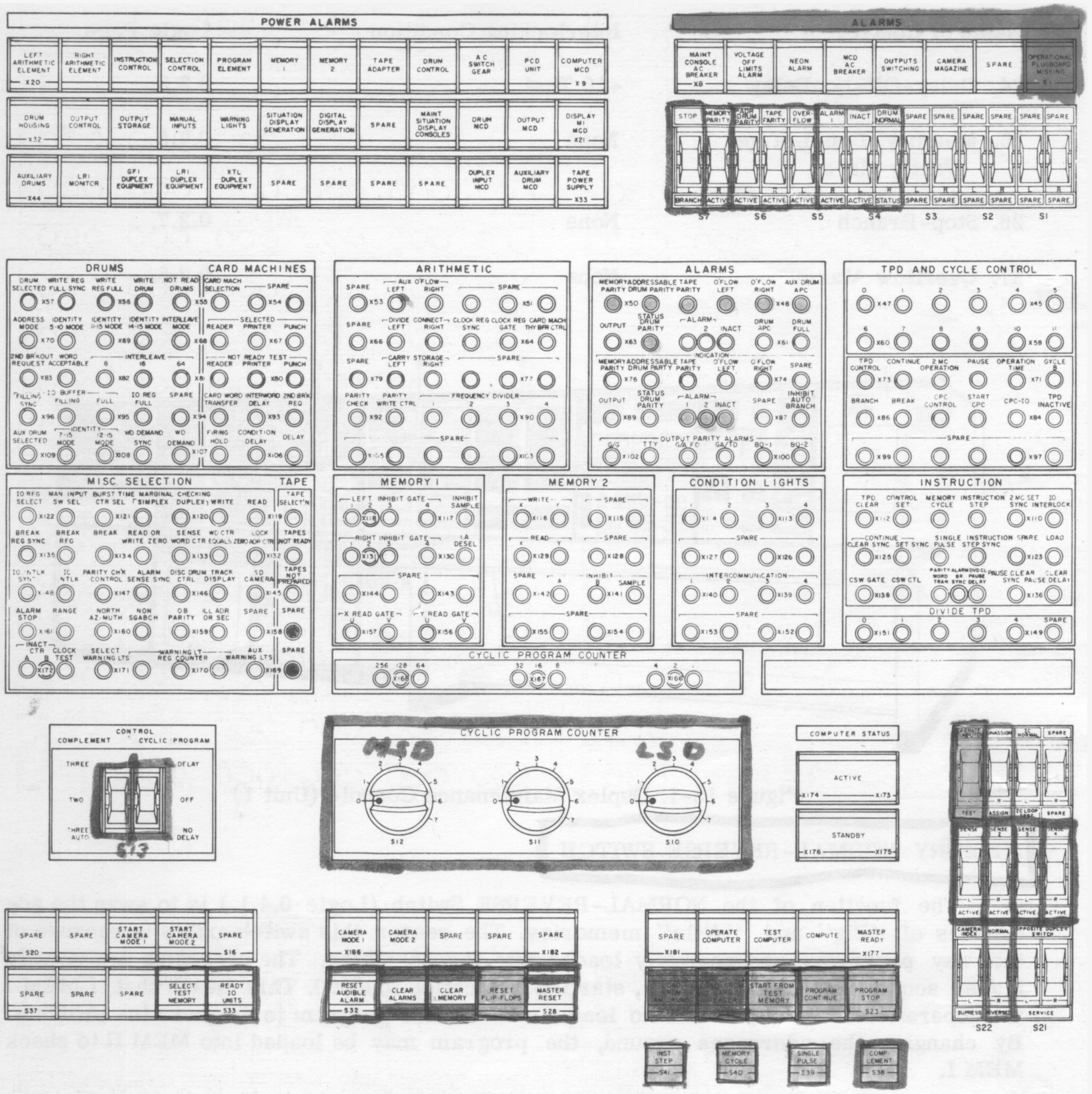


Figure 13-2. Duplex Maintenance Console, Module G

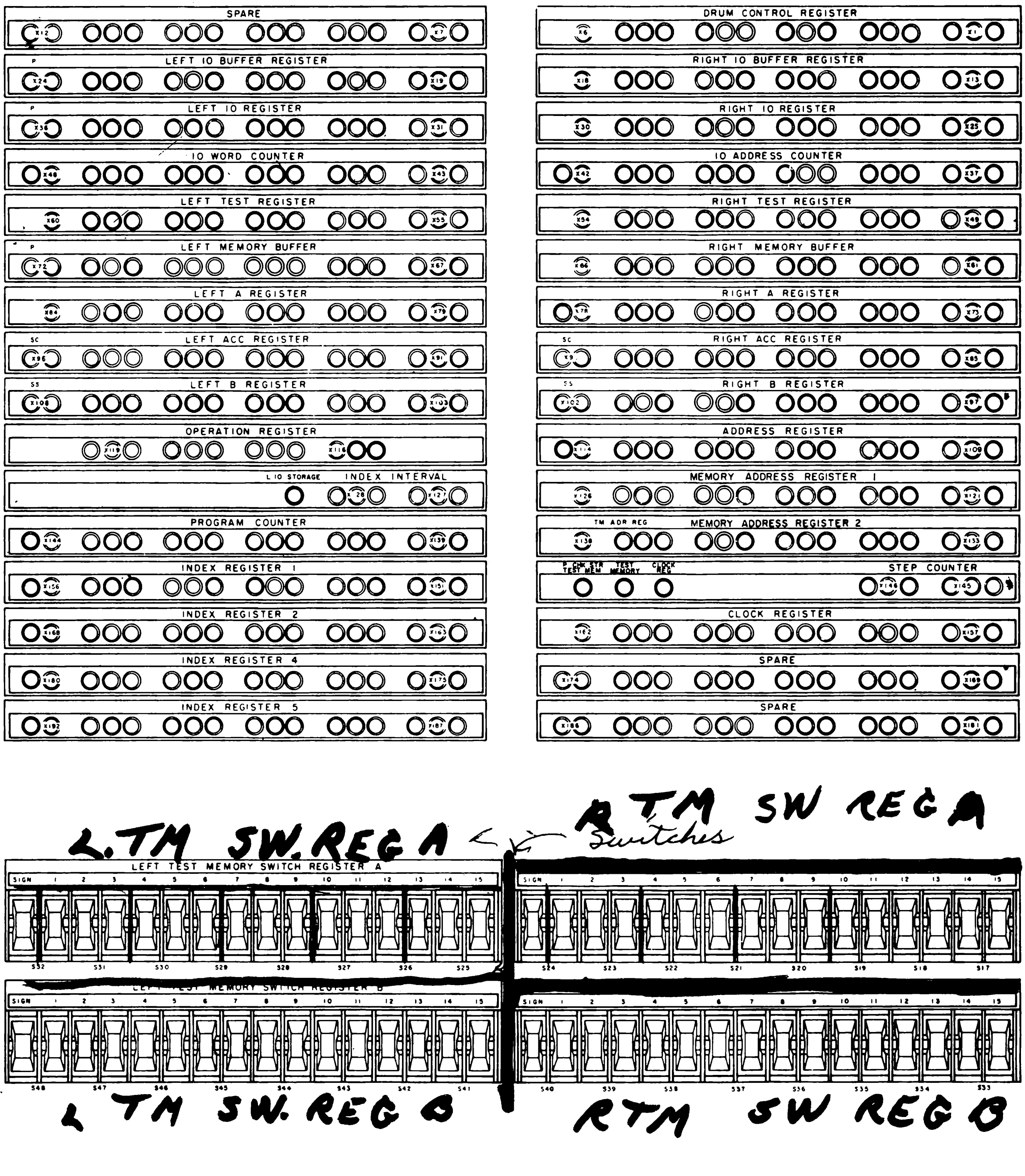


Figure 13-3. Duplex Maintenance Console, Module F

The way this change of address is brought about is to use the pulse which normally starts MEM I to start MEM II in the Reverse position, and vice versa.

PROGRAM CONTINUE PUSH BUTTON (PB)

This PB is used to continue the execution of a program that has been interrupted. It is operative whenever the computer is halted in the course of a program, regardless of computer status. If the Continue PB is depressed the TPD ring is started up and computer operation resumes from the point at which the program was interrupted. When tracing Logic 0.2.1., note that the Continue P.B. when depressed, gives you a Clear Alarms function as well as a Continue function. This function will be discussed later when the Clear Alarms PB is covered.

PROGRAM STOP PB

The PROGRAM STOP PB is used to interrupt the computer program without removing power or losing stored information. It is operative whenever the computer is running, regardless of computer status.

When this pushbutton is depressed, the computer completes the instruction it is currently executing and executes a simulated HLT instruction. If the IO Interlock is on, all IO transfers are completed before the HLT instruction is executed.

When the computer is stopped by the switch, the address of the succeeding instruction in the interrupted program is retained in the program counter and the coded instruction is contained in the memory buffer register.

The operation may be fully understood by tracing Logic 0.2.1. twice; once with the IO Interlock on, and then with it off.

SINGLE-PULSE PB

The SINGLE PULSE pushbutton (Logic 0.2.2.) is used to advance the program one timing pulse at a time. It is normally used only in diagnostic testing after an error stop in a reliability program. It can be used to advance the entire program up to the point of error, Timing pulse by Timing pulse; as a rule, however, it is used only after the source of trouble has been isolated to a single instruction step or memory cycle.

MEMORY CYCLE PB

[illegible]

The MEMORY CYCLE PB (Logic 0.2.1.) is used to advance the computer program one memory cycle at a time. Since each memory cycle is synchronized with a machine cycle (i.e., a PT or an OT cycle), this switch can be used to break an instruction into its component parts. An instruction of the store class, for example, requires a PT cycle and two OT cycles (OTA and OTB) for its execution. If the computer is halted at such an instruction, only the PT cycle is performed the first time the MEMORY CYCLE PB is depressed; the instruction is transferred from memory to the operation and address registers, and the computer halts. When the switch is depressed again, the OTA cycle is performed (an operand is brought from memory) and again the computer halts. The third time the switch is depressed, the OTB cycle is performed (the data is stored in core memory) and the computer halts for the third time. At each halt, the operator can check the neon indicators on the Duplex Maintenance Console and perform any tests and troubleshooting operations required. If an even finer analysis is required, the program may be advanced a time pulse at a time by means of the SINGLE PULSE PB.

The MEMORY CYCLE PB is electrically interlocked so that it is operative only when the computer is halted and in test status. It is normally used only in diagnostic testing after an error stop in a reliability program. It can be used to advance the program, memory cycle by memory cycle, to the point of error; as a rule, however, it is used only after trouble has been localized to a single instruction requiring several machine cycles for its execution.

INSTRUCTION STEP PB

The INSTRUCTION STEP PB (Logic 0.2.1.) is usedto advance the computer through one complete instruction, including all PT and OT cycles requires for its execution. The computer halts automatically when the instruction has been executed so that the operator can observe the neon indicators on the Duplex Maintenance Console and perform any tests and troubleshooting operations. A single instruction may involve several machine cycles. If finer analysis is required, an instruction may be advanced through its component machine cycles by means of the MEMORY CYCLE PB or through its individual time pulses by means of the SINGLE PULSE PB.

This switch is electrically interlocked like the MEMORY CLCLE PB so that it is operative only when the computer is halted and in test status. It is normally used only in diagnostic testing after an error stop in a reliability program. It can be used to advance the program, instruction step by instruction step, to the point of error. As a rule, however, it is used only after trouble has been localized in a small portion of the program.

RESET FLIP-FLOPS PB

The RESET FLIP-FLOPS pushbutton (Logic 0.2.1.) is used to clear the control flip-flops and registers, which are not ordinarily cleared in normal operational or test programs and which must be cleared to prevent the introduction of unwanted information into a new program or into a complementing operation. The switch is electrically interlocked so that it is operative only when the computer is in test status. This switch is usually used before loading a new program and before testing the computer flip-flops by means of the COMPLEMENT switches.

The RESET FLIP-FLOPS PB, when depressed, activates a relay cycle that initiates the following control cycle:

1. Stop the generation of time pulses and instruction pulses by clearing the TPD-control flip-flop.

2. Clear the IO word counter, the IO register, the IO address counter, the IO buffer register, the program counter, and most of the flip-flops in the selection control and instruction control units.

3. Clear most flip-flops in the right and left arithmetic units. Refer to Logic 0.2.4. to find exactly what FFs and registers are cleared.

CLEAR MEMORY PB

The CLEAR MEMORY pushbutton (Logic 0.2.1.) is used to clear the live test register and the 3.77760 locations in core memory when the computer is halted in test status. The effect produced manually by this switch is identical with that produced by the simulated execution of the following instructions:

Clears Mem. 17[illegible] times because of illegal addresses

$SEL_{(04)}$		Selects IO Register
LDC	0.00000	Clears IO Address Counter
RDS	3.77761	Actually sets the IO Word Counter to the complement of 3.77760 but will read 3.77761 times from the cleared IO Register because the simulated RDS instruction makes no provision for Command 290.
HLT		

READY IO UNITS PB

This control (Logic 7.1.8.) merely puts the IO units under the control of the computer, enabling the computer to read from the Card Reader, write on the Card Recorder (Punch) or the Line Printer, or read or write Tapes. In other words, make the IO units ready for use by the computer.

MASTER RESET PB

The MASTER RESET pushbutton (Logic 0.2.1.) is used to prepare the IO units, the memory element and the Central Computer System for the loading of a new program. It combines the operations of the following pushbuttons, which are discussed separately:

1. RESET FLIP-FLOPS

2. CLEAR MEMORY

3. READY IO UNITS put I/O units under computer control

The MASTER RESET pushbutton is operative when the computer is in test status whether it is running or halted.

SELECT TEST MEMORY PB

The SELECT TEST MEMORY pushbutton (Logic 0.2.1.) makes it unnecessary for the operator to perform more than one manual operation to clear all the control flip-flops and registers affected by the RESET FLIP-FLOPS PB and to reset the program counter to 3.77760, which is the first address in test memory. In other words, this control clears all previous program data, (except memory) and selects the first address in test memory as the point from which computer operation may resume. Since an operator can write directly into the first address of test memory by setting the switches in toggle switch register A, this pushbutton permits him to manually clear the computer and to set up whatever control he requires. This switch is operative when the computer is in the test status.

START FROM TEST MEMORY PB

The START FROM TEST MEMORY (Logic 0.2.1.) pushbutton sets up the computer controls to initiate the execution of the program contained in test memory (toggle switch registers and/or test memory plugboard). This program may consist of a single instruction used to branch program control to a test program already stored in memory, or it may include all the instructions required to perform a specific function(e.g., to set up controls to transfer additional information into core memory from an IO device). This PB is operative only when the computer is halted. Simply stated, this PB simulates a BPX to 3.77760.

LOAD FROM CARD READER PB

+ 0.2.3 0.2.6

The LOAD FROM CARD READER PB pushbutton (Logic 0.2.1.) is used to introduce a loading or control program into the first 24 core memory registers by setting up the computer controls to transfer the information contained on a single IBM binary-punched card. To do this the following program is simulated:

```
SEL(01)

LDC      0.00000      by controls clear - clears addr cnt.

RDS           30

SEL(01)

BPX      0.00000
```

KNOW THIS

The second $SEL_{(01)}$ and the BPX are done by skipping from TL-6 to TL-8 which simulates a HLT instruction. This simulated HLT checks the IO Interlock taking care of the $SEL_{(01)}$. The BPX is taken care of by inhibiting Command 270 of the HLT instruction. This allows the computer to advance past PT6 of the HLT and since the Program Counter hasn't been stepped by this simulated program, Location 0.00000 is brought out and operated. Since this loading program can direct the computer to read additional information from any input source, this switch provides a convenient means of introducing the desired operating program into the computer. I/O wd.cntr. end carry clears I/O intlk

LOAD FROM AM DRUMS

The operation of the LOAD FROM AM DRUMS PB (Logic 0.2.1.) is almost the same as the LOAD FROM CARD READER PB. The difference is that the first 24 registers of the first field of the AM-A drum are loaded sequentially into core memory starting at location 0.00000. The simulated program is as follows:

KNOW

```
SDR(02)
LDC       0.00000
RDS            30
SDR(02)
BPX       0.00000
```

SENSE SWITCHES

The four SENSE SWITCHES (Nr. 1,2,3,4)(Logic 0.7.4.) provide the operator with a flexible means of controlling the execution of a program, if this control feature is written into the program. During the execution of such a program, the sense switches are sensed by appropriate BSN instructions to determine what action is desired by the maintenance operator; i.e., whether the program should continue sequentially or branch to the address contained in the right half-word of the BSN instruction. Each switch is associated with the correspondingly coded BSN instruction. If, for example, SENSE SWITCH Nr. 1 is in the inactive (up) position, the computer continues with the next step of the program after a BSN$_{(21)}$ instruction has been executed. If the switch is in the ACTIVE position, however, the program branches directly to the address contained in the right half-word of the BSN$_{(21)}$ instruction, instead of executing the next instruction at the address specified by the program counter. The operation of the other three switches is identical, except that they are associated with three other BSN instructions.

[illegible]

use 3/3 auto sw/

COMPLEMENT CONTROLS

The first control to be discussed will be the COMPLEMENT SWITCH (Logic 0.2.4.) which controls the operation of the COMPLEMENT PB. It is a three position switch that determines how the Central Computer FFs are to be complemented. In the up (three) position it allows the FFs to be complemented 3 times at a .5 msec rate, every time the Complement PB is depressed. Its other (two) position allows the FFs to be complemented twice. When in the down (three auto) position, complementing is done by 1 sec clock pulses, bypassing the Complement PB altogether. Every 1 sec clock pulse is allowed to complement the FFs 3 times at a 5 msec rate.

The COMPLEMENT PB is for complementing of the Central Computer FFs if the COMPLEMENT SWITCH is in the Three or Two position.

CYCLIC PROGRAM COUNTER (CPC)

The computer can be set up to run automatically through a program and stop at a predetermined point in the program. This capability is very helpful in finding the cause of machine state failures.

The CPC (Logic 0.2.5.) itself is a 9 stage counter whose setting is controlled by the three CPC switches. These are 8 position rotary switches which are controlled by the CPC DELAY-OFF-NO DELAY SWITCH. To operate the CPC:

a. Calculate number of pulses desired and set CPC counter switches.

b. HLT machine.

c. Place in the A switches a BPX to portion of program that is to be checked.

d. Set CPC lever switch to Delay or No Delay.

e. To stop operation, place CPC lever switch in the OFF position.

When calculating the number of pulses you must take into consideration that 3 extra pulses will be allowed out of the TPD ring. In other words, if you set the counter switches to 4, 7 pulses will be generated. You must also remember not to count break pulses because the CPC is not stepped during breaks.

The CPC Control switch controls whether the CPC is on or off and also the amount of delay between successive passes through the portion of program being checked.

In the No Delay position recycling is started by 2 MC pulses and there is only about 3 msec between successive passes. This position would be used when scoping for a trouble.

The DELAY position is used when you want to troubleshoot by observing the neons and in this position, CPC recycling is controlled by 1/32 sec clock pulses. Note that these clock pulses only control the starting of the CPC. Once started, it is stepped by 2 MC TPD on pulses as in the NO-DELAY position. The time between passes is about 1/16 sec in the DELAY position. Consider the following example:

```
3.77760    BPX        400
    400    1XIN         0     to PT3 time
    401    SEL(04)
    402    LDC       1000
    403    RDS         10
    404    1BPX(01)   404
    405    HLT
```

12 pulses
3 "
15 total
- 3 extra pulses
12 ~~[illegible]~~ needed pulses
or
$14_{(8)}$ pulses-setting
+ 7 cycle up comp.
$23_{(8)}$ CPC setting

CPC would be set to what number to determine if:

	ANSWERS
a. Index Register Nr. 1 was being cleared properly.	$32_{(8)}$
b. PT/OT FF was being set properly.	$41_{(8)}$

c. PT/OT FF was being cleared properly. $55_{(8)}$

d. IO address counter was being cleared properly $74_{(8)}$

e. Branch FF was being set properly. $7_{(8)}$

f. Branch FF was being cleared properly. $20_{(8)}$

ASSIGN-UNASSIGN SWITCH

TM Plugboard

Test Memory consists of a plugboard, 2 rows of 32 toggle switches called "A" and "B" switches and the Live Test Register (LTR) which is a flip-flop register. Test Memory addresses go from 3.77760 - 3.77777.

The plugboard has room for $20_{(8)}$ computer words on it. Since Test Memory has only $20_{(8)}$ addresses, it is obvious that all locations on the board, the A and B switches and the LTR cannot be used at the same time. To control which will be used we have the ASSIGN-UNASSIGN SWITCH (Logic 7.1.5.), and control plugs (hubs) on the Test Memory plugboard. There are four of these hubs for each address on the board; A., B., L., and P. "A" stands for the A Switches, "B" for the B Switches, "L" for the Live Test Register and "P" for the Plugboard.

In the UNASSIGNED position all TM addresses are controlled by the plugs; A., B., L., and P. If a T.M. address is selected and the switch is in UNASSIGN, the control hubs are checked and the appropriate location brought out to the Memory Buffers. For example, let's assume address 3.77773 is selected and the A hub is plugged. The contents of the A Switch register will be brought out. If the plug was taken out of A and put in B, the B Switches would be brought out. Care must be taken to insure that only one control hub is plugged for each address. If more than one is plugged a logical add will take place in the Memory Buffers.

When the ASSIGN-UNASSIGN SWITCH is in the ASSIGNED position, address 3.77760 is the A switches, 3.77761 is the B switches, 3.77762-3.77776 are controlled by the plugs and 3.77777 is the LTR.

CLEAR ALARMS PB

Break hold path on Relay

This pushbutton (Logic 0.2.1.) clears all Central Computer Alarm Indicator FFs (Logic 0.2.7.) and clears all Central Computer Alarm Sensable FFs except Alarm I and Alarm II (Logic 0.7.4.)

The indicator FFs do nothing more than light a neon on the DMC to show the operator what alarm has occurred and sound off the Audible Alarm, which is a buzzer.

CLEAR AUDIBLE ALARM PB

The AUDIBLE ALARM (Logic 7.1.14.) is turned off by this PB.

SUMMARY QUESTIONS

1. What are the two basic statuses of the computer?

2. The Compute level is +10v when:

3. The Non-Compute level is +10v when:

4. When in reverse the Normal/Reverse Memory Switch allows usual memory Nr. 1 addresses to select:

5. Will the Normal/Reverse Memory switch change Test Memory Addresses?

6. The Continue PB sets the ________________ FF first.

7. When the IO Interlock is on and the Program Stop PB is depressed, what happens?

8. The Instruction Step FF allows one entire instruction to be operated and then:

9. The memory cycle PB lets the TPD Ring step once from:

10. The Single Pulse PB allows one TP pulse at a time to step the:

11. The Reset FFs PB is only operative when in what status?

12. Clear Memory PB loads memory from the:

13. Master Reset PB Resets FFs, Clears Memory, and:

14. Start from TM sets the Program Counter to:

15. List the program that is simulated by the Load from Card Reader PB.

16. In "Three Auto" position of the complement switch, the computer FFs will be complemented at what rate?

CHAPTER 14 - ALARMS

GENERAL

This section describes the alarm indicators and alarm control circuits that are actuated by alarm conditions during the operation of the Central Computer System.

The Central Computer operating alarm conditions provide the operator with an indication (by means of neons and the alarm buzzer) that the sequential execution of the program is not progressing normally and/or that faulty operation has been detected. In addition to sounding the audible alarm buzzer and lighting the appropriate indicator lamp, the error-detecting signal will also interrogate the setting of appropriate duplex maintenance console switches to determine what program action, if any, should be executed. These switch settings determine whether the error alarm is to be ignored or whether the computer is to stop or branch control to test memory.

The alarms that we will cover are Memory Parity, Tape Parity, Drum Parity, Alarm I, Alarm II, and Inactivities. With the exception of Alarm II all of these alarms have associated switches. If an alarm occurs and its switch is inactive (off) an indicator will be set and the audible alarm sounded. If the appropriate alarm switch is active (on) the indicator is set, alarm is sounded and the alarm goes on to check another switch called the Branch/Stop Switch. If this switch is in the Stop position the alarm will cause the computer to halt. If in the Branch position the program control will be branched to the middle of Test Memory, address 3.77770.

MEMORY PARITY ALARM OPERATION

Assume that we have a memory parity with these conditions:

1. Memory Parity Switch active.
2. Branch/Stop Switch in Branch position.
3. Inhibit Auto-Branch FF set (this is a FF to give some program control to the Branch/Stop operation).

Tracing these conditions through logic (0.2.7.) and paying close attention to timing, we will find that the computer finishes any OTA or OTB cycles and doesn't go into the Alarm Branch operation until it comes to a PT-5 dy. Tracing logic with the Branch/Stop Switch in the Stop position should now show that the computer will stop at a TL-0.
at store erroneous parity word in Live test reg.

IO PARITY ALARM OPERATION

An IO Parity Alarm can occur from a bad word on tapes or drums. When tracing in logic (0.2.7.) notice that the bad word is not sent to the L.T.R. and that the timing is different. Trace for the following sets of conditions:

at BI-1 alarm gen.
and Tape 1. Drum Parity Switch Active and Branch/Stop Switch to Branch.
and Tape 2. Drum Parity Switch Active and Branch/Stop Switch to Stop.
3. [redacted] Parity Switch Active and Branch/Stop Switch to Branch.
at PT-5 dy. - branch to fix

INACTIVITY ALARMS

Inactivity Alarms, TPD or Loop, are generated if the DCA program is not progressing properly. In normal operation (Logic 0.2.7.) throughout DCA we have $PER_{(05)}$ and $PER_{(06)}$ instructions which start and stop, respectively, the operation of a circuit called the Inactivity Counter. If DCA isn't progressing properly either too many or too few $PER_{(05)}$ instructions will occur causing Loop inactivities. TPD inactivities occur when no TP11's happen for a certain period of time. Following a $PER_{(05)}$ to start it, the inactivity counter will generate a Loop Inactivity Alarm for the following sets of conditions:

1. Two consecutive 8 second clock pulses with no intervening $PER_{(05)}$ with the Inactivity Switch active and the Branch/Stop Switch to Branch.

2. Two consecutive 8 second clock pulses with no intervening $PER_{(05)}$ with the Inactivity Switch Active and the Branch/Stop Switch to Stop. (Computer will not stop.)

3. Three consecutive $PER_{(05)}$'s with no intervening 8 second clock pulse, with the Inactivity Switch Active and the Branch/Stop Switch to Branch.

4. Three consecutive $PER_{(05)}$'s with no intervening 8 second clock pulse; with the Inactivity Switch active and the Branch/Stop Switch to Stop. (Computer will not stop.)

A TPD inactivity will be generated by two consecutive 1/32 second clock pulses with no intervening TP-11 if the Inactivity Switch is Active and the Branch/Stop Switch is in the Branch position.

ALARM I

The Alarm I (Logic 0.2.7.) alarm is used by the Active Computer to tell the Standby Computer to prepare for a switchover. The Alarm I may be generated by:

1. Two Memory Parities without an intervening $PER_{(40)}$, if the Inhibit Auto Branch FF is set, Memory Parity Switch Active and the Branch/Stop Switch in Branch.

2. Two inactivities with no intervening $PER_{(40)}$ if the Inactivity Switch is Active and the Branch/Stop Switch is in Branch.

3. Alarm Branch restart failure if the Alarm I Control FF is set, Branch/Stop Switch to Branch and no TP-11's for 4 to 8 seconds after attempting a Memory Parity or Loop Inactivity Alarm Branch or no TP-11's for from 1/32 of a second to 4 seconds after attempting a TPD Inactivity Alarm Branch.

4. A $PER_{(37)}$ in the Active Computer will generate an Alarm I in the other computer. The Alarm I Switch must be active to generate an Alarm I.

ALARM II

Alarm II informs you that the SAFØ (Safe Data Zero) transfer from DCA to DCS is not correct and it is not safe to switch computers. This alarm may be generated (Logic 0.2.7.) by:

1. An SDR$_{(26)}$ (IC own) APC (Angular Position Counter) error from the active to the standby computer.

2. An SDR$_{(16)}$ (IC other) APC error generates an Alarm II in own computer.

OVERFLOW ALARM OPERATION

Overflow is a condition that exists when two numbers result in a quantity which exceeds the capacity of the machine (accumulators).

Assume that the special option bits are set to allow unsuppressed overflow, the Overflow Alarm Switch Active, Inhibit Auto Branch FF set and the Branch/Stop Switch to branch or stop (Logic 0.2.7.) The operation is the same as a normal alarm branch or alarm stop operation except that one more instruction will be operated before the alarm branch or alarm stop action is initiated.

SUMMARY QUESTIONS

1. What is the purpose of the alarm indicators and alarm control circuits?

2. At what PT time does the computer go into the alarm branch for a Memory Parity error?

3. What can cause an IO Parity Alarm?

4. An Inactivity Alarm because of the absence of what instruction?

5. What conditions will generate a Loop inactivity alarm?

6. What alarm sets up a switchover of computers?

7. When will an Overflow Alarm occur?

APC counts angle of address from 0° located on drum

CHAPTER 15 - PARITY

GENERAL

The basic definition of parity is equality with respect to an established quantity or standard. More applicably, parity may be defined as uniformity in either the oddness or evenness of number. The latter concept, in a slightly modified version, is used in the AN/FSQ-7 equipment for detecting errors incurred during the transfer or storage of binary information. Although parity is sometimes employed to verify the results of computations, it does not serve this particular function in the AN/FSQ-7 Central Computer System.

In an error detection scheme based on the more appropriate definition of parity, all words entering, leaving, or circulating within a system must have uniform parity; that is, every binary word either naturally exhibits or must be made to exhibit a parity in common with that of all other words in the system. Before elaborating on this principle, a further clarification of parity is necessary from the standpoint of its application in the Central Computer System. First, the established parity for all words is odd. Second the parity does not hinge on the absolute numerical value of a word of information; rather, it is based on the number of binary 1's contained in a word, be it an instruction word or a data word. Since the established parity is odd, each word in the Central Computer (which has parity) must have an odd number of 1's. Obviously, the parity of many words will naturally be odd. Those words having an even parity are given odd parity. This is accomplished, without changing the information contained in the normal 32-bit word, by prefixing to it a parity bit. When the natural parity of a word is even, the parity bit is made a 1; it is left at 0 when the original parity is odd. Having assigned an odd parity to all words, every word can then be checked for conformance with this characteristic. If, upon subsequent examination, a word found to have even parity, it can rightfully be assumed that an error was generated either in storing or in transferring the word from a particular source to a specific destination. For purposes other than checking the accuracy with which information is transferred, the parity bit is meaningless and is discarded before a data word is entered into calculation or before an instruction word is decoded.

As a result, a practical application of parity requires circuits that are capable of (1) determining whether the number of binary 1's in a word of information is odd or even, (2) assigning a uniform parity to all words, (3) checking the parity of words to which parity has been assigned, and (4) generating a signal when a word with incorrect parity is detected. The first requirement is integral to both parity assigning and checking.

MEMORY PARITY CIRCUIT

The memory parity circuit performs a dual function: it assigns a uniform parity to all words originating within and without the Central Computer to which no parity has been previously assigned, and it checks the parity of words to which parity has been assigned. The latter function constitutes the test on the accuracy with which data is transmitted from the core memory array to the memory buffer register (MBR), and from several of the input sources to the MBR. Since the MBR is situated at the crossroads of all information paths in the Central Computer System, both parity assigning and checking take place at the MBR.

Of the several sources of words which are transferred through the MBR, only tapes and specific drum fields have a parity bit assigned. Therefore, words from these sources need only be checked for correct parity; all other input sources require that the parity circuits assign a parity bit. All words read out from memory, of course, have parity bits and require only parity checking.

It was mentioned earlier that the parity operations performed at the MBR are assign and check. An assign operation may be considered to consist of a parity count followed by the writing of a 1 or 0 in the parity bit flip-flop. The checking operation also begins with a count but is followed by a parity check. It will be noted that the counting procedure is common to both parity assign and check. For convenience, the counting operation will be treated first and will be followed by the overall operation of the parity control circuits.

PARITY COUNTING

The counting operation (Logic 0.1.2.) neither counts nor totals the number of 1's in a binary word but only ascertains whether the total is odd or even. A count is begun with the application of a pulse to the two gates associated with the R15 flip-flop in the MBR. (A register flip-flop and its associated gates and OR circuits will be termed a stage). One of the gates is conditioned by the 0 side of the flip-flop, the other by the 1 side. If R15 happens to be a 1, the counting pulse will pass through the gate conditioned by the 1 side and will emerge on the odd line. Conversely, if R15 is 0, the count pulse will pass through the gate conditioned by the 0 side and will exit on the even line. Although the remaining stages in the register are each complicated by the inclusion of two additional gates, the operation at each stage is identical with that described for R15 and its associated gates. The additional gates are necessary because the count pulse may arrive at any subsequent stage on either an odd or even line, whereas at the first stage only a single input line was encountered.

Upon leaving the first stage, the count pulse successively samples the gates at each of the remaining stages. The line on which it enters a particular stage is dependent upon the parity count at the preceding stage. The line on which it leaves that stage depends on the content of the associated flip-flop. Since the flip-flop may be set at either 1 or 0, and since the count pulse may arrive on either of two lines, the combination of these considerations gives rise to four possible conditions. The four conditions and the resultant output line (parity count) produced by each are listed below and are applicable to all but the first stage.

Table 15-1

INPUT LINE (PARITY)	CONTENT OF MBR FF	OUTPUT LINE (PARITY)
ODD	0	ODD
ODD	1	EVEN
EVEN	0	EVEN
EVEN	1	ODD

It will be noted that a pulse sampling the gates of a flip-flop containing a 0 leaves the stage on the same line it entered on. A count pulse arriving on either an odd line or an even line, upon sampling a 1, exits on a line opposite to that on which it entered a stage. In all cases, the action is equivalent to a progressive determination of the parity at each and every stage.

In summary, a parity count pulse determines the overall parity of a word by traversing the entire MBR and sampling the bit at every stage to sense the parity of the word up to and including that stage. The final parity is obtained when the pulse samples and leaves the last stage in the MBR.

PARITY ASSIGNING AND CHECKING

The parity assigning and checking operations differ (Logic 0.1.1, 0.1.2 and Timing Chart) depending on the type of word transfer:

1. All words being read out of core memory during PT, OTA, or BO cycle.

2. Words from the arithmetic element being read into core memory during OTB cycle. Computer words are transferred from either the accumulators or the A registers into memory through the MBR. Since no parity bit exists in these words, it is necessary to assign parity. Such a transfer always takes place during an OTB cycle.

3. Words from IO devices being transferred into core memory with (a) parity or (b) no parity. The devices that have parity assigned to their words before they are transferred to Central Computer are tapes, aux drums, and main drums except fields 22, 23, 32, 33, 47, and 60 thru 70. Words transferred from any other IO device to Central Computer must have parity assigned in the MBR.

Table 15-2. BASIC TIMING CHARTS

PT CYCLES	
PT-0 delayed	SM Clear Parity Check FF and Parity Write FF with Command 41.
PT-7	Set Parity Check FF and start P.C. (Parity Count)
PT-10	Check Parity with Command 55 . 2μ
OTA CYCLES	
OTA-0 Delayed	Same as PT-0 delayed
OTA-7	Same as PT-7
OTA-10 delayed	Same as PT-10 delayed

BO CYCLES

BO-0 delayed	Same as PT-0 delayed
BO-7	Same as PT-7
BO-10	Same as PT-10 delayed

OTB CYCLES

OTB-0 delayed	SM, Clear Parity Check FF and Parity Write FF with Command 41.
OTB-2	Set the Parity Write FF
OTB-3	Parity Count (to assign parity). The Parity Check FF is set but will not be checked yet.
OTB-7	Parity Count, Set Parity Check FF and Clear Parity Write FF.
OTB-10 delayed	Check parity. This is a check on the Parity assignment circuitry.

BI CYCLES (IO devices with Parity, Parity Check Control FF set)

BI-0 delayed	SM, Clear Parity Write FF and Parity Check FF with Command 41.
BI-2	Set Parity Write FF
BI-3	Clear Parity Write FF
BI-3 delayed	Parity Count, Set Parity Check FF
BI-7	Check Parity for IO Parity error

BI CYCLES (IO device without Parity, Parity Check Control FF clear)

BI-0 delayed	SM, Clear Parity Write FF and Parity Check FF with Command 41.
BI-2	Set Parity Write FF
BI-3 delayed	Parity Count, Set Parity Write FF and write good parity in new word.
BI-7	Parity Count, Set Parity Parity Check FF again
BI-10 delayed	Check Parity, this is a check on the parity assignment circuitry.

SECET

SUMMARY QUESTIONS

1. Give the parity bit for each of the following words:

   a. 1.000 101 111 101 011, 1.001 010 011 100 111

   b. 0.111 011 010 001 100, 1.111 110 101 011 000

   c. 1.43571, 0.45632

   d. 0.21307, 1.53214

2. In Central Computer, parity is assigned in what register?

3. A parity count must be performed when it is __________ or when it is ________ .

4. During what 2 types of machine cycles is parity assigned?

5. Why is parity checked immediately after it is assigned?

6. What drums don't have parity?

7. Will a parity error be detected if no word is transferred to the memory buffers during PT?

8. Will a parity error be detected if two 1 bits are lost from a memory word?

9. Will a parity error be detected if a zero is lost?

10. When reading a word from an IO device with parity, the parity count for checking occurs at what time?

11. When reading a word from an IO device without parity, the parity count for checking occurs at what time?

NOTES

NOTES

NOTES

NOTES

SAVE A LIFE

If you observe an accident involving electrical shock,
DON'T JUST STAND THERE - DO SOMETHING!

RESCUE OF SHOCK VICTIM

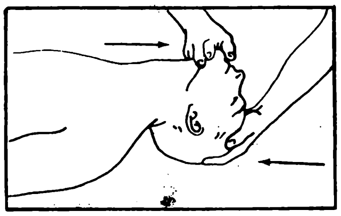
The victim of electrical shock is dependent upon you to give him prompt first aid. Observe these precautions:

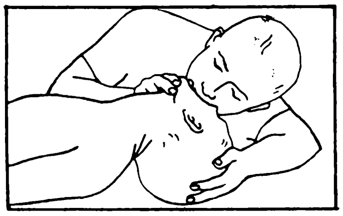
1. Shut off the high voltage.
2. If the high voltage cannot be turned off without delay, free the victim from the live conductor. REMEMBER:
   a. Protect yourself with dry insulating material.
   b. Use a dry board, your belt, dry clothing, or other non-conducting material to free the victim. When possible PUSH - DO NOT PULL the victim free of the high voltage source.
   c. DO NOT touch the victim with your bare hands until the high voltage circuit is broken.

FIRST AID

The two most likely results of electrical shock are: bodily injury from falling, and cessation of breathing. While doctors and pulmotors are being sent for, DO THESE THINGS:

1. Control bleeding by use of pressure or a tourniquet.
2. Begin IMMEDIATELY to use artificial respiration if the victim is not breathing or is breathing poorly:
   a. Turn the victim on his back.
   b. Clean the mouth, nose, and throat. (If they appear clean, start artificial respiration immediately. If foreign matter is present, wipe it away quickly with a cloth or your fingers).





   c. Place the victim's head in the "sword-swallowing" position. (Place the head as far back as possible so that the front of the neck is stretched).
   d. Hold the lower jaw up. (Insert your thumb between the victim's teeth at the midline - pull the lower jaw forcefully outward so that the lower teeth are further forward than the upper teeth. Hold the jaw in this position as long as the victim is unconscious).
   e. Close the victim's nose. (Compress the nose between your thumb and forefinger).
   f. Blow air into the victim's lungs. (Take a deep breath and cover the victim's open mouth with your open mouth, making the contact air-tight. Blow until the chest rises.) If the chest does not rise when you blow, improve the position of the victim's air passageway, and blow more forcefully. Blow <u>forcefully</u> into <u>adults</u>, and <u>gently</u> into <u>children</u>.
   g. Let air out of the victim's lungs. (After the chest rises, quickly separate lip contact with the victim allowing him to exhale).
   h. Repeat steps f. and g. at the rate of 12 to 20 times per minute. Continue rhythmically without interruption until the victim starts breathing or is pronounced dead. (A smooth rhythm is desirable, but split-second timing is not essential).

DON'T JUST STAND THERE - DO SOMETHING!