PHILCO® 2000

PROGRAMMING MANUAL

PHILCO

PHILCO® 2000

Electronic Data Processing System

PROGRAMMING MANUAL

PHILCO CORPORATION

GOVERNMENT AND INDUSTRIAL GROUP — COMPUTER DIVISION

3900 Welsh Road
Willow Grove, Penna.

TABLE OF CONTENTS

TABLE OF CONTENTS (cont'd)

LIST OF FIGURES

CHAPTER I

FUNCTIONAL DESCRIPTION OF THE PHILCO 2000

The PHILCO 2000 is a large scale, general purpose, electronic data processing system. It is a single address system and operates in a parallel, asynchronous mode. High speed, reliability, and compactness are achieved by extensive use of transistors and printed circuits.

The PHILCO 2000 is the result of two major research operations - one concerned with parallel, asynchronous computers and the second with transistors. In 1952, the Institute for Advanced Study at Princeton, New Jersey, completed the IAS computer. IAS is a high-speed, binary, asynchronous, parallel computer. It proved very reliable and served as the basis for such computers as MANIAC and JOHNNIAC as well as for TRANSAC.

In 1953, Philco began the study of the desirability of using its Surface Barrier Transistors for high-speed switching circuits for digital computers. Results of extensive tests at the Lincoln Laboratories of the Massachusetts Institute of Technology and at Philco's transistor plant at Lansdale, Pa., proved that the long life and extreme reliability of these transistors made them well suited for digital computer use.

Philco's efforts in this area were recognized by the government with several orders for airborne, digital computers. Shortly thereafter, Philco produced the TRANSAC S-1000, the first large-scale, transistorized computer, which combined transistorized circuits with IAS design features. The next step was the production of the prototype of PHILCO 2000. In early 1957, the PHILCO 2000 was incorporated into Philco's product line and the first 2000 system was delivered in 1958.

THE CENTRAL COMPUTER

The heart of the PHILCO 2000 system is the Central Computer, the actual data processor of the system. The major components of the Central Computer are the internal storage units and the arithmetic and program sections. These components store the data and instructions, execute the instructions in sequence, and perform arithmetic and logical operations.

Magnetic Core Memory

The main internal storage is a high-speed, coincident-current, magnetic core memory with a basic capacity of 4096 words. A word is composed of 48 binary digits and may represent alphabetic, numeric, or alphanumeric information. When representing alphabetic information, the word is composed of eight binary-coded characters. When representing numeric information, the word is the equivalent of 14 decimal digits. Alphanumeric words contain less than eight binary-coded characters and a numeric equivalent of less than 14 decimal digits.

The basic core memory is available in one, two, four and eight units of 4096 words each. Thus, memory capacity ranges from 4096 to 32,768 words, or 32,768 to 262,144 characters, or 57,344 to 458,752 decimal digits.

Access to words in any unit of the core memory takes place in parallel in 10 microseconds (μs). Words are read from, or stored in, the memory in two cycles - a 4-microsecond read cycle and a 6-microsecond write cycle. When a word is read from memory, the read cycle reads and clears the memory location, and the write cycle restores the contents of the memory location accessed. When a word is stored in memory, the read cycle clears the memory location and the write cycle stores the word in the accessed memory location.

Substantial program running time is saved in the computer because a word read from memory may be operated on during the write cycle. Thus, an addition of a word from memory to the accumulator only requires the time to access memory - 10 microseconds. Furthermore, if the operation is one that takes more than 6 microseconds, such as multiplication, the write cycle is completely overlapped and the effective memory access time is only 4 microseconds. This is illustrated in the following diagram.

Read 4 μs	Write 6 μs	
Effective Access Time	←Multiply→	

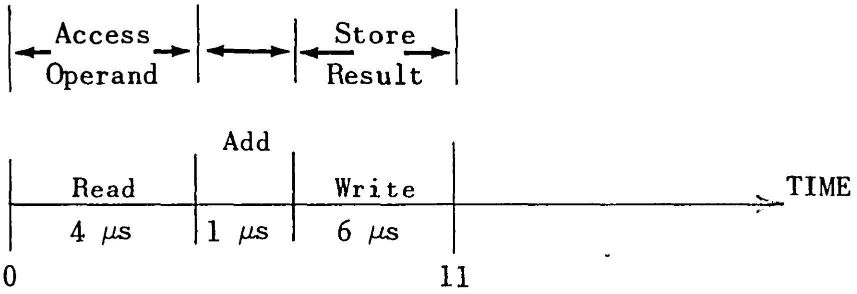
If the operation requires that the result be replaced in the original memory location, the write cycle is delayed until the operation is completed. Thus, the time to add a word from memory and replace the original word by the sum will be 4 microseconds, plus the addition, plus 6 microseconds. The effective access time in such a case is 5 microseconds.

In contrast to this split-cycle operation, unsplit memory cycle operations require two complete memory accesses plus the arithmetic operation. Therefore, a saving of 10 microseconds is realized during every split-cycle operation, as shown below.

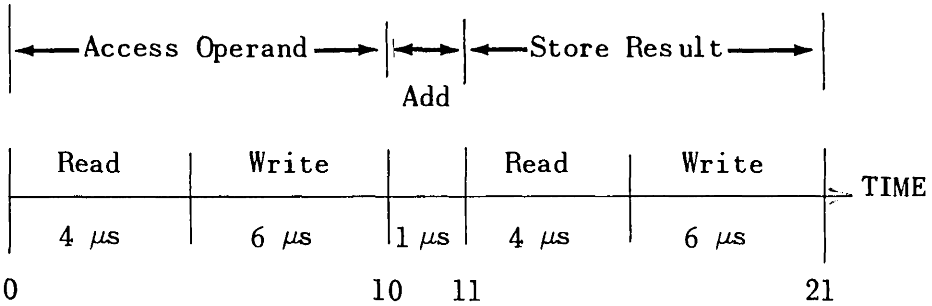
parallel transfer: the entire contents of A word/register are transfeered simultaneously in 1 access cycle.

serial transfer: each bit of A word/register is transfered in a seperate cycle.

Split-Cycle Replace Type Addition (AMS)



Unsplit Cycle Replace Type Addition



In addition to the split memory cycle, additional speed is achieved because the Central Computer operates in an asynchronous mode. That is, each operation is begun when a signal indicates that the last operation has been completed. Time is saved by not waiting for a signal from a "clock" as in a synchronous mode type of computer.

Magnetic Drum Storage

Intermediate speed storage in the computer is provided by magnetic drums. Each drum stores 32,768 words in eight bands of 4096 words each. This is the equivalent of 262,144 alphanumeric characters or up to 458,752 decimal digits. Up to 32 drums may be incorporated in one system. Since one drum controller handles up to four drums, eight drum controllers will control all 32 drums.

The drum, which is 24 inches long and 20 inches in diameter, revolves at a rate of 1750 rpm. One drum revolution, therefore, takes 34 milliseconds with an average access time of 17 milliseconds, plus 8 milliseconds for band selection to the first word of a transmission. Because each word is recorded in parallel, subsequent words come under the read-write heads every 8 microseconds - less time than it takes to store the word in the core memory. The words on the drum are interlaced to skip a location between successive words and to give a transfer time to the core memory of 16 microseconds per word.

The drums use the input-output register and memory access circuitry and are addressed individually by unit and drum addresses. The drum addresses range from zero to 32, 767. The information on a drum must first be transferred to core storage to be used. During the transfer, other input-output and processing are interrupted until the transmission is completed.

CENTRAL PROCESSOR

The Central Processor is the unit of the Central Computer which processes data and instructions. It consists of the arithmetic section, the program section, and the display and manual control section.

Arithmetic Section

Arithmetic in the computer may be performed in either the fixed point or floating point mode. Numbers are represented in pure binary form and are operated on in parallel. Negative numbers are represented in two's complement form. These design features increase the operating speed of the system.

Some of the flexible and timesaving additions to the basic types of arithmetic operations are multiplication yielding double length or rounded products, division with double length dividends (all divisions are self-correcting in the case of overflow), and combinations of multiplication and addition or multiplication and subtraction. In all arithmetic operations the operands may be in absolute value and the results may be stored in memory. These additional operations add to the flexibility of the 2000 and simplify the programming.

The basic transfer time and minimum addition time are one micro second. The average speeds of the arithmetic operations (including instruction and operand access) expressed in operations per second are shown below.

	Fixed Point	Floating Point
Addition and subtraction	66,700	66,700
Multiplication	20,300	24,900
Division	19,200	24,300

Program Section

The program section selects and executes the instructions stored in the core memory. Instructions are automatically selected in the sequence that they appear in memory. Since instructions are stored two to a word, two instructions are selected with each memory access, thereby reducing access time per instruction by approximately 50%.

The PHILCO 2000 has a repertoire of over 200 instructions, including 59 floating point instructions. This large number of instructions provides maximum program flexibility and minimizes the number of instructions per program. Ease in learning and using the instruction code is ensured by the simple mnemonic code and by the logical grouping of the instructions.

In addition to direct addressing, index registers may be utilized for address substitution and modification for most instructions. Registers may be selected in groups of 8, 16, and 32. The index register contents may be automatically increased and used for counting and addressing sequential locations. Also, the contents of the registers may be increased or decreased for convenient use in accumulating. Since the index registers are independent registers and not part of the core memory, program running time to use the contents of the registers is infinitesimal.

To further reduce program running time and to simplify the programmer's task, a number of unique instructions have been incorporated into the system. One is designed to facilitate the handling of records and fields of any size. Others simplify sorting, merging, and table lookup operations. Some reduce the effort necessary to write mathematical programs while another group simplifies the use of subroutines.

Display And Manual Control

An operator's console provides indicators, manual controls, and other facilities for monitoring the operations of the PHILCO 2000 system, for debugging programs, and for periodic maintenance. Adjacent to the console is the Console Typewriter which furnishes the operator with direct, immediate access to the core memory. By using the typewriter, the operator can insert control information into the program and check the intermediate results and control totals.

INPUT-OUTPUT SYSTEM

The PHILCO 2000 input-output system permits the programmer to make effective use of the high operating speeds of the Central Computer. All input-output data that can be scheduled is funneled through the Input-Output Processor unit. Random or high-priority input-output is connected with the Central Computer through the real-time channel. When no real-time connections are provided, this channel may be used for additional paper tape input-output. Because of the PHILCO 2000 design feature called Multiple Processing, all input-output operations may be programmed to proceed simultaneously.

Multiple Processing

The Multiple Processing technique of the PHILCO 2000 has greatly improved and enlarged upon the processing method often referred to as simultaneous read/write/compute. While the central processor is computing, as many as nine input-output devices may be processing data simultaneously. Four of the nine may be Magnetic Tape Units and four may be Punched-Card

Systems, High-Speed Printers, and Paper Tape Systems. The ninth may be either a Real-Time Scanner or a Paper Tape System. Each of the nine devices may be either reading or writing. Multiple Processing is possible because of a design feature that makes optimum use of memory (Memory-Sharing) and because of the advanced electronic design of both the Input-Output Processor and the Universal Buffer-Controller.

Input-Output Processor

The Input-Output Processor is the interconnecting and control link between the Central Computer and the 16 input-output channels. Each channel couples either a Magnetic Tape Unit or a Universal Buffer-Controller to the Central Computer. The standard data transfer rate over a channel is 90,000 alphanumeric characters per second. By means of a multiplexing technique, the Input-Output Processor can connect any four of the 16 channels to the Central Computer at one time. Up to 16 Input-Output Processors may be used in a PHILCO 2000 system to connect as many as 256 input-output channels with the Central Computer.

Each Input-Output Processor also controls four Universal Buffer-Controllers simultaneously. In this case, while transmission takes place between the computer memory and four tape units, any combination of four punched-card readers and punches, High-Speed Printers, and paper tape readers and punches may also be operating. Central Computer time is only used when data is transferred between the memory and a buffer-controller. The operations of the Punched-card, Paper Tape, and High-Speed Printing systems are essentially off-line when under the control of buffer-controllers.

Real-Time Channel

The real-time channel shares access to the magnetic core memory with the Central Processor, the Input-Output Processor, and the Magnetic Drum System. This channel may be used for paper tape input-output or with a real-time unit for real-time input-output.

Magnetic Tape Unit

The magnetic tape has a one mil mylar base, is one inch wide and comes in five real sizes: 600, 1200, 1800, 2400, and 3600 feet. Six-bit characters are recorded at a density of 750 to the longitudinal inch. The tape is pre-edited into areas called blocks, each of which contains 1024 six-bit characters. When tape is read by the computer, these characters are assembled into words. Each word contains eight characters, or 48 bits. Each block contains, therefore, 128 words. A full reel of tape contains 19,000 blocks or over 19,000,000 characters.

16 bits across the width of the tape are considered a frame. Each frame contains two characters, two parity bits (one per character), and two timing bits. One block contains 514 frames. Of these, 512 are data frames. A final frame provides a parity check on each channel, and an initial frame provides symmetry for backward reading.

A tape speed of 120 inches per second provides a maximum reading rate of 90,000 characters per second. Maximum efficiency, speed, and ease of use are achieved because one block size and one recording density are used throughout the system. Other magnetic tape specifications are listed in the summary of PHILCO 2000 characteristics.

Accuracy of reading and recording is ensured by parity checks and separate "read" and "write" heads. Immediately after information has been recorded, it is read back and checked for parity.

To insure against unintentional destruction of information, a physical snap ring is provided with each tape reel. Without the snap ring, recording cannot occur. Furthermore, a safety device can prevent the insertion of a snap ring. Whether or not the snap ring is used, data on the tape may be read.

Information written on magnetic tape may be binary-coded information, pure binary information, or a combination of both. Regardless of the data form, the Central Computer accepts six binary digits as a "character" and discards the parity bit. Because all numeric data may be recorded and read in pure binary form, tape and time savings of up to 50% will be realized. (A seven-place decimal number less than 2,097,152 is recorded in binary-coded decimal form in 42 bits and in pure binary form in 21 bits.)

Magnetic tape reading and writing operations are controlled by the Input-Output Processor. As many as four out of 16 tape units may be operated concurrently with Central Computer processing. Since the character transfer rate is 90,000 characters per second with one tape unit, with four tape units operating simultaneously the transfer rate is 360,000 alphanumeric characters or 628,000 decimal digits per second.

Universal Buffer-Controller

The Universal Buffer-Controller adds greatly to the flexibility and economy of the PHILCO 2000 system. This one unit controls off-line conversions between any two media or on-line communication between the Central Computer and any medium except magnetic tape. In general the buffer-controller acts as a buffering device between two input-output units or between one input-output unit and the Central Computer.

The input-output devices that may be operated with the buffer-controller include Punched-Card Systems, Magnetic Tape Units, Paper Tape Systems, and High-Speed Printing systems. Up to five Punched-Card, Paper Tape, or High-Speed Printer units may be connected to a buffer-controller in addition to two Magnetic Tape Units. If one or both Magnetic Tape Units are not used, their channels may be used by any other Input-Output Unit noted above. In the future any desirable device may be easily added to a buffer-controller. A simplified diagram of a buffer-controller is shown in the following figure.

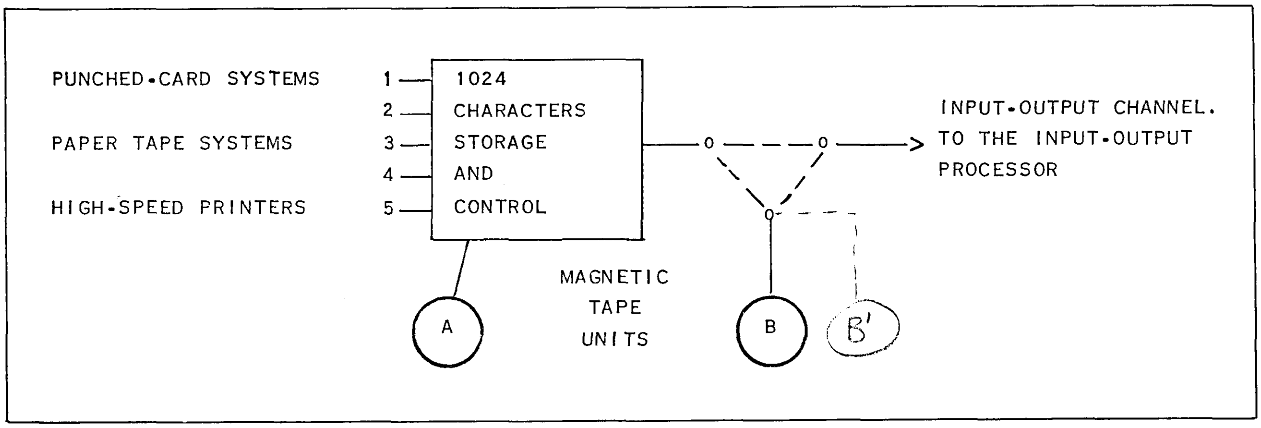


Figure 1. Universal Buffer-Controller

When used off-line, the buffer-controller controls the conversion of data from any medium to any other medium. For example, the buffer-controller is used to convert data from punched cards to magnetic tape, from tape to printer, tape to tape, etc.

Data Select is an additional off-line feature. When Data Select is used, only the data blocks containing selected control characters will be converted. Thus the data for up to 15 reports, for example, may be recorded on the same reel of magnetic tape or the same punched-card deck. Then the Universal Buffer-Controller may be used to segregate the data and prepare the reports in sequence from the single data source. Data Select simplifies and speeds up report preparation and allows the buffer-controller to perform an off-line function which in other systems requires Central Computer time.

The buffer-controller, which is switched on-line by a pushbutton, can be used as buffer storage for the intermediate speed input-output units, such as Paper Tape, Punched-Card, and High-Speed Printer systems. The transmission between the buffer-controller and an input-output unit is essentially off-line and does not require Central Computer time or control. The transmission between a buffer-controller and the Central Computer, however, is the same as it is for magnetic tape and is at the same rate: 90,000 characters per second. Also, as for magnetic tape, the transmission is time-shared with other input-output operations and Central Computer processing.

The two Magnetic Tape Units which may be connected to the buffer-controller are very flexible in their use. If tape unit A, as illustrated in the above figure, is provided with a buffer-controller, it is permanently connected to the buffer-controller. Tape unit A is used for off-line conversions to and from magnetic tape. If tape unit A is not provided with the buffer-controller, another input-output unit may be substituted.

Tape B channel now may have two tapes which may be used alternately on or off line.

Tape unit B is extremely flexible in its connection and use. For maximum flexibility of the buffer-controller, an optional electronic switch may be set by a pushbutton to connect tape unit B either on-line to the Input-Output Processor or off-line to the buffer-controller. Consequently, tape unit B may be used in one position as another on-line tape unit to retain the capacity of the system when the buffer-controller is engaged in an off-line operation. In the off-line position, tape unit B may be used as an alternate for tape unit A for conversion to or from magnetic tape. For a magnetic tape to magnetic tape conversion, tape unit B is used with tape unit A.

Another use of tape unit B is to record the output data of a program. It may then be switched off-line to convert the output data to another medium through the buffer-controller. This operation avoids the handling of tape reels and speeds preparation of reports. For economy, tape unit B may be permanently connected to the buffer-controller in the same fashion as is tape unit A. Finally, tape unit B can be omitted entirely and be replaced by another input-output system. In any case the Central Computer cannot communicate with a tape unit through the buffer-controller since this would result in poor utilization of the buffer-controller.

Punched-Card System

The Punched-Card System reads 2000 cards per minute by a new photoelectric reading technique and punches 100 or 250 cards per minute. The system may read or punch 51- or 80-column cards in either Hollerith or card image mode. The standard Hollerith code used in most punched-card installations has been expanded so that all 64 computer characters can be punched and read. The cards are translated from Hollerith code to computer code automatically. The card image mode facilitates the handling of binary information and packs twelve bits or two computer characters in each column. Card image mode also simplifies the use of punched cards prepared on different computers.

High-Speed Paper Tape System

Through the paper tape reader, data in the form of 5-, 6-, or 7-level punched paper tape may be read directly into the core memory at a rate of 1000 characters per second. With the paper tape punch, data may be punched onto paper tape at a rate of 60 characters per second. Both reading and punching are controlled by the paper tape controller.

For long tape life and higher operating speeds, reading is accomplished photoelectrically. The tape speed is 100 inches per second; up to 4096 words may be read with one instruction and provisions are made to pass blank tape. Operation of the reader is such that without reversing the tape or leaving a record or block gap, reading begins with the character immediately following the last character read.

The Paper Tape System may be connected directly to the Central Computer or to a Universal Buffer-Controller. The direct Central Computer connection is necessary when no Universal Buffer-Controllers are included in a PHILCO 2000 system. The direct connection may also be advantageous in a 2000 system since it allows the maximum number of input-output units to be operated simultaneously.

High-Speed Printing System

The high-speed printing system consists of the printer controller and the High-Speed Printer and operates in conjunction with a Universal Buffer-Controller.

Speeds of 600 or 900 lines a minute are obtained by the printer. (15,000 lines/min) By skip-feeding, non-printed areas are passed at a rate of 25 inches a second. Each line prints 120 characters spaced at ten characters per inch horizontally and six per inch vertically.

Information to be printed on-line is received from a buffer-controller in standard blocks of 1024 characters. The 64 computer characters fall into the following three major classes:

a. decimal digits, 0 through 9

b. alphabetic characters, A through Z

c. twenty-eight special symbols.

In normal operating mode, three of the special symbols are control characters and only the remaining 61 characters are printed. In memory dump mode, all 64 computer characters (including the three control characters) are printed.

(w, Δ, e)

Horizontal format is controlled by computer programming and plugboard editing. The plugboard is used to repeat characters on a line, and to suppress and rearrange fields. Vertical format is accomplished by a paper tape control-loop mounted on the print carriage mechanism.

The "print-on-the-fly" method of printing is used. One hundred and twenty hammers are arranged horizontally to be fired at a 2-3/4 inch diameter print roll which is constantly revolving about a horizontal axis. The 64 computer characters are spaced around the circumference of the print roll. The impulse hammers, when actuated, strike the paper and force it against a print ribbon which lies across the character face. The print ribbon is an inked, silk ribbon which is self reversing in operation and has a life of approximately 19 ~~100~~ printing hours. One major feature of this printer is the "controlled penetration" of the hammer. The hammer travel is controlled between physical stops and never actually strikes the print roll. This feature produces clearer printing and reduces wear on the hammer, print roll, and inked ribbon.

SUMMARY OF PHILCO 2000 CHARACTERISTICS

Central Computer

Memory capacity

a. basic core memory unit - 4096 words (32,768 alphanumeric characters)

b. memory unit expandable to 32,768 words (262,144 alphanumeric characters) in units of 4096 words

Internal characteristics

a. binary - parallel - asynchronous
b. fixed point arithmetic (floating point optional)
c. word length - 48 binary digits
d. two's complement notation

Instruction code

a. single address
b. two instructions per word
c. over 200 instructions (including 59 floating point)

Index registers - 8, 16, or 32

Magnetic drum system

a. drum capacity - 32,768 words (262,144 alphanumeric characters)
b. maximum of 32 drums in a system
c. average access time - 17 milliseconds
d. under control of magnetic drum controller

Input-Output

Input-Output Processor

a. simultaneous read/write/compute operation
b. sixteen input-output channels
c. four simultaneous transmissions with transfer rate of 360,000 characters per second
d. connection between Central Computer and Magnetic Tape Units and Universal Buffer-Controllers

Magnetic Tape Unit

a. reading/writing speed - 90,000 characters per second
b. tape dimension - up to 3,600' length, 1" width
c. reel capacity - over 19 million alphanumeric characters - 19,000 blocks
d. standard block size - 1024 characters (128 words)
e. density - 750 characters per linear inch
f. tape speed - 120 inches per second
g. immediate and automatic information verification of both reading and writing

Universal Buffer-Controller

a. capacity - 1024 characters
b. off-line data selection
c. off-line conversion of data between any two input-output systems
d. on-line buffer for all input-output systems except magnetic tape

Punched-Card System

a. photoelectric reader - 2000 cards per minute
b. punch - 100 or 250 cards per minute
c. plugboard format control

Paper Tape System

a. photoelectric reader - 1000 characters per second
b. punch - 60 characters per second
c. 5-, 6-, or 7-level tape

High-speed printing system

a. printing rate - 600 or 900 lines per minute
b. 120 characters per line
c. 64 printable characters
d. plugboard and/or computer format control
e. print-on-the-fly method of printing

CHAPTER II

DETAILED DESCRIPTION OF THE CENTRAL COMPUTER

In the previous chapter an overall view of the PHILCO 2000 was given. This chapter provides some of the details necessary to understand the basic programming requirements of the computer.

PROGRAM CONTROL

The PHILCO 2000 Word

As previously defined, a PHILCO 2000 word is composed of 48 bits numbered left to right from 0 through 47. The word may be eight binary-coded characters, a 47-bit number with a sign bit, a data word containing pure binary as well as binary-coded information, a constant, or an instruction word containing two instructions. Regardless of the nature of a word, it is individually addressable by the PHILCO 2000 instructions; that is, each memory location has an address by which its contents may be located. A word may be addressed directly by specifying its address in an instruction or indirectly by specifying an index register which contains the address of the word.

Program Section

The program section selects and executes instructions in an ordered sequence. Instructions indicate how data is to be manipulated. Two instructions, the left and right half instructions, comprise an instruction word. The normal sequence of executing instructions is first the left half and then the right half instruction of one word, followed by the left half and then the right half instruction of the next succeeding instruction word.

Each computer instruction contains 24 bits divided into a 16-bit address part and an eight-bit command part.

Instruction Format

16 Bits	8 Bits
Address	Command

Command Part. The command part is further subdivided into the seven-bit command, C, and the function bit, F.

C = COMMAND bits
F = function bit
(floating pt; eight address;etc)

Command Part

1	7
F	C

Some instructions may require all eight bits to define a command; some require only seven, and others need seven bits modified by the function bit. The F Bit specifies whether the arithmetic is to be performed in the fixed or floating point mode.

Address Part. The address part is subdivided into an index register selector bit (S), a 3- to 5-bit index register number (N) to specify a particular index register, and a 10- to 12-bit variable field (V). The size N and V are determined by the number of index registers in a system. The address part may be a memory address or some number specified by the instruction. The address part of an instruction in a system with eight index registers is shown below.

V = VARIABLE field bits
(Address, increments, etc.)
N = INDEX REGISTER Number bits
S = Selector bit

Address Part

1	3	12
S	N	V

Instruction Format. Each PHILCO 2000 instruction can be specified by varying the configuration of the eight-bit command. Since the command code is pure binary, it is decoded in a compact, parallel, and very efficient manner. This may be appreciated by examining the command code in more detail.

	F	C						
Binary Digits	7	6	5	4	3	2	1	0
Non-Arithmetic		0						
Arithmetic		1						
Add		1	0	0				
Subtract		1	0	1				
Multiply		1	1	0				
Divide		1	1	1				
Do Not Store Result								0
Store Result								1

Assignment of zero or one to each of the eight positions results in a unique code combination which, when decoded, produces a specific command. Further designation of bits 3,2, and 1 might call for the absolute value of an operand, a "Clear" instruction, and other modifications of the four basic arithmetic operations. Thus the particular code combination

01000001

would add a word from memory to the word in the accumulator and store the result in memory.

To remember over 200 eight-bit instruction codes would be very difficult; therefore, to simplify programming, an English-decimal mnemonic code is used, and the preceding instruction is expressed simply as AMS, Add Memory and Store. This mnemonic code, which will be discussed in the next chapter, is translated into the machine or computer code.

The computer instruction is sequenced through various control registers. (See the block diagram at the end of this chapter.) The Program Register, PR, stores the selected pair of instructions to be executed. The Program Address Register, PA, contains the address of the next instruction word. The Memory Address Register, MA, holds the address of the memory location to be accessed. The Jump Address Register, JA, stores the address of the instruction following the last jump instruction. Of these registers, only the Jump Address Register is program-addressable. It is used to fabricate subroutine exit jump addresses. (Subroutines are discussed in Chapter VII.)

The sequence of operations in the program section begins with the transfer of the address in the Program Address Register, PA, to the Memory Address Register, MA, after this address has been established manually in the PA Register. Then the address is decoded and the corresponding instruction word is selected from memory and transferred to the Program Register. The address in the PA Register is then increased by one to become the address of the next sequential instruction word to be selected and executed.

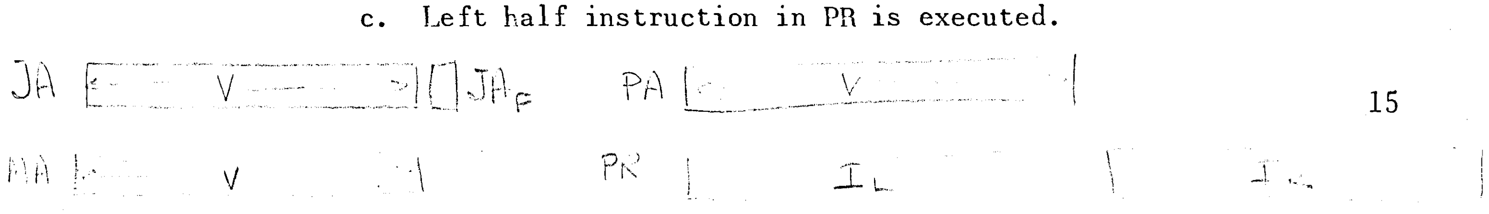
The left half instruction in the PR is executed first. The execution of the instruction may affect a word in memory, in the arithmetic section, or an address in the PA, JA, or an index register but it <u>cannot</u> directly affect the word in the PR. (A left half instruction may change the sequence of executing instructions so that the right half instruction is not immediately executed - but it cannot alter the right half instruction in the PR.)

The right half instruction is executed following the left half instruction, and the next pair of instructions is selected. This procedure may be described symbolically as follows: (Parentheses mean "contents of," and an arrow indicates a transfer of information.)

a. (PA) $\longrightarrow$ MA, (MA) specifies M; (M) $\longrightarrow$ PR

b. (PA) + 1 $\longrightarrow$ PA

c. Left half instruction in PR is executed.



d. Right half instruction in PR is executed.

e. Steps a through d are repeated.

This sequence is continued until a Halt, Repeat, or Skip instruction is executed, an error is detected, or a jump is effected.

DATA CONTROL

Arithmetic Section

The purpose of the arithmetic section is to perform arithmetic, comparisons, transfers of data, and other data manipulating operations.

The arithmetic section consists of an adder network and three arithmetic registers, which have one word (48-bit) capacities. The registers are the accumulator (A Register), the data register (D Register), and the multiplier-quotient register (Q Register). For floating point operations, an optional unit is added to the basic section.

The registers have the following functions:

a. D Register:

1. receives all data transferred between the memory and the arithmetic unit
2. receives all data transferred between arithmetic registers
3. contains the addend in addition, the subtrahend in subtraction, the multiplicand in multiplication, and the divisor in division
4. contains one of two factors in a comparison
5. permits access to index registers

b. A Register:

1. contains the augend in addition, the minuend in subtraction, and the dividend or the more significant half of the dividend in division
2. contains the sum in addition, the difference in subtraction, the product or more significant half of the product in multiplication, and the remainder in division
3. contains one of two factors in a comparison

augend + addend → sum
minuend − subtrahend → difference
dividend ÷ divisor → quotient
multiplier × multiplicand → product

c. Q Register:

1. contains the multiplier in multiplication; the less significant half of a double length product of a multiplication; the less significant half of a double length dividend in division; and the quotient of a division
2. may contain a factor in a comparison
3. contains a masking pattern during an extracting operation.

An arithmetic ~~instruction~~ command is interpreted as a fixed point instruction if the F bit is a zero. In the fixed point mode, the binary point of a data word lies immediately to the right of the sign or zero position as shown in the following diagram: (A description of floating point numbers will be found in Appendix E.)

0	1	2	3	4	43	44	45	46	47

Binary Point

The maximum positive number is

0	1	1	1	1	1	1	1	1	1

See pg. 90

which is less than one as far as the computer is concerned. Negative numbers are represented in two's complement form, the smallest computer negative number being minus one which is represented, however, in one's complement form. Any arithmetic result which would be outside the above limits produces a condition called overflow.

Index Registers

Index registers in the 2000 operate in several ways, depending on the type of instruction using them. Their uses may be categorized as follows:

a. Instruction Address Modification: After an index register is filled by the desired address modifier, the effective address of the instruction using this register is the sum of the contents of the index register and the V field of the instruction. Most PHILCO 2000 instructions can be address-modified in this manner, without altering the instruction in memory or the contents of the index register.

b. Counting: An index register may be set to automatically increase itself by one each time it is used. Used with an instruction which doesn't utilize the V portion (such as an arithmetic register-to-register transfer), the index register will contain the total number of executions of the instruction. Using the counting function for address modification permits consecutive memory locations to be addressed automatically.

c. Instruction Address Substitution: An address in an index register may replace an address part of an instruction in memory.

Input-Output Control

All input-output communication takes place between the magnetic core memory and the input-output units. Simultaneous computation and input-output operations are achieved by use of the Input-Output Processor, an Input-Output Register, and the special mode of operation described below.

Input data is collected in independent one-word registers in the Input-Output Processor or Paper Tape System while computation is in progress. After a word has been collected, it is transferred to a one-word Input-Output Register in the Central Computer. The program is then interrupted so that the word may be transferred, in parallel, to the core memory. During the transfer, the input unit is still in operation. After the transfer, the Central Computer resumes processing.

Output operations occur in a similar manner. The program is interrupted only when another word is required by the output unit. After the word has been transferred to the Input-Output Register, the program continues and the word is written or transferred to a buffer-controller simultaneously with the computer operation.

The Console Typewriter operates independently of the normal input-output circuitry.

TERMINOLOGY

The abbreviations of terms used in this manual excluding the mnemonic codes are defined as follows:

TAC:	Translator-Assembler-Compiler
PR:	The Program Register
MA:	The Memory Address Register
PA:	The Program Address Register
JA:	The Jump Address Register
IO:	The Input-Output Register
X:	An index register
c:	The counter indicator bit of an index register
M:	A memory location

A: The A Register
Q: The Q Register } one-word registers
D: The D Register

I: An instruction.

The following letters may be used as subscripts with M, A, Q, D, and I:

L: The left half of a word

R: The right half of a word

V: The variable field of an instruction

F: The function bit of an instruction

C: The seven-bit command part of an instruction

S: The index register selector bit of an instruction

N: The number of the index register selected.

Associated with X, the subscript, c, may be used; with JA, the subscript, F, may be used.

The following additional abbreviations are also used:

( ): The contents of

| |: The absolute value of the contents of

→: Is placed in

≡: Is equivalent to. (Is interchangeable with.)

For example, the abbreviations listed have the following meanings:

$D_{LV} \equiv$ the V part of the left half of (D)

$JA_F \equiv$ the F bit associated with JA

$(X_{31}) = D_V \equiv$ The contents of Index Register number 31 are equal to a V part of (D).

$x \leq y$: x is less than or equal to y.

$x \geq y$: x is greater than or equal to y.

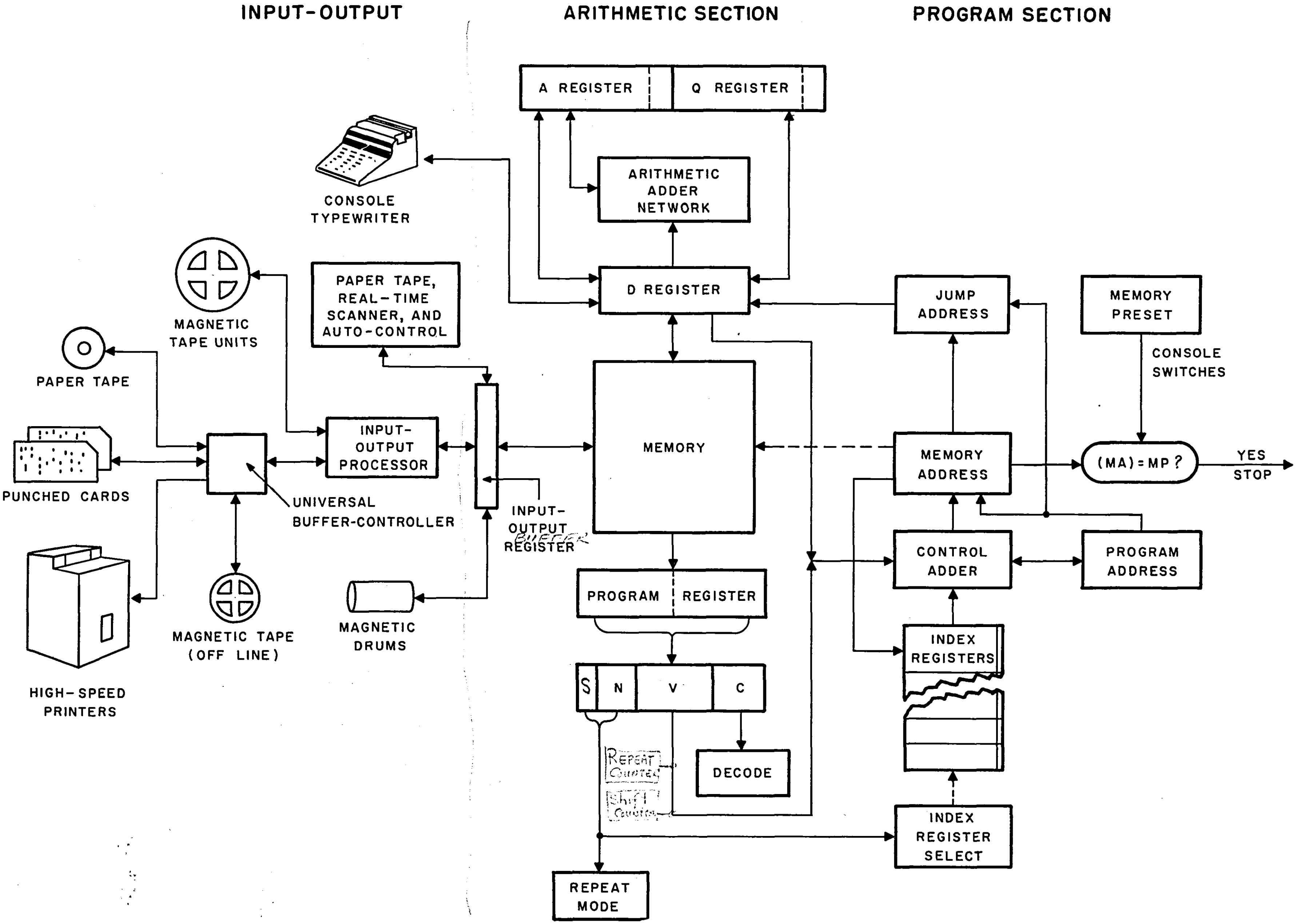


Figure 2. Simplified Diagram of the PHILCO 2000 System

CHAPTER III

INTRODUCTION TO CODING

TRANSFERS AND ARITHMETIC

In the previous chapter, the PHILCO 2000 instruction was described in terms of its size and structure. It was seen that a binary instruction code is very efficient for a machine. However, this code which is suitable to the computer is not convenient for programmers. Therefore, a completely functional English-decimal mnemonic code has been prepared for the PHILCO 2000. (The reader should be aware that from this point on the instructions described exist in the machine but not in the form shown here. The term "mnemonic instruction" is used to distinguish the code that the programmer writes from the "computer instruction" or code that the computer accepts.) The mnemonic code, by itself, is unacceptable to the computer and cannot cause it to function. To enable the computer to operate, it is necessary to use a translation program. This program, the Translator-Assembler-Compiler (TAC), converts the mnemonic code written by the programmer into the binary computer code.

The PHILCO 2000 Mnemonic Code

The PHILCO 2000 mnemonic code has been designed to facilitate the learning and use of the computer instructions, which number over 200. This has been done by grouping the instructions into classes of computer operations and then naming them mnemonically. Essential to data processing are the following instruction classes:

a. addition
b. subtraction
c. multiplication
d. division
e. transfer of data
f. jump
g. shifting
h. extracting
i. index register housekeeping
j. input-output
k. special.

Mnemonic Code Instructions

Just as a computer instruction has a command part and an address part, so does the mnemonic instruction. The command part of the instruction is composed of two to six letters and is divided into three sections. The command completely defines an operation in one of preceding instruction classes. It also specifies the origins of operands, the disposition of results, etc. The composition of the three parts is shown below:

	OPERATION	LOCATION OF OPERAND	OPTIONS
Size (in number of letters)	1-5	0-4	0-4

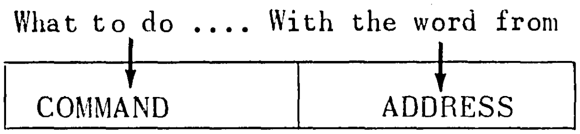
For example, Addition instructions (fixed point mode) are composed of the following letters:

OPERATION	LOCATION OF OPERAND	OPTIONS
Add or Clear Add	The operand from M or Q	In Absolute value and/or Store the result
A or CA	M or Q	A and/or S or blank

Thus, the possible operations are Add or Clear Add; the operand may be in memory or in the Q Register, and the possible options are that the operand may be in Absolute value and/or that the sum may be Stored. The following commands are possible in addition:

AM	:	Add Memory
CAQ	:	Clear Add Q
AQS	:	Add Q and Store
CAMA	:	Clear Add Memory in Absolute value
AQAS	:	Add Q in Absolute value and Store.

The command part is followed by the address part of the instruction. The address part usually specifies an address of a memory location or the number of places to shift a word. This may be shown as



Mnemonic instructions on the coding paper are shown below.

COMMAND									ADDRESS AND REMARKS						
A	M								1	2	3	4	$		Add bonus
A	Q	S							0	0	4	6	$		Add overtime and store

Writing programs for the PHILCO 2000 on coding paper is like writing a list of instructions for someone to perform a task. In each case the writer indicates specific operations to be performed. The written mnemonic code for the PHILCO 2000 doesn't look like computer code, but it does meet the requirements of being legible and easily understood.

Each line on the coding paper represents one instruction; each column or group of columns indicate a specific portion of the instruction. The L, or Label, column may be used, if necessary, to specify whether the instruction is a left half or a right half instruction.

The location field on the coding paper is used to specify locations for the instructions, if necessary. The only need to do this arises when one instruction must refer to another. In most cases the location field is left blank.

The command field generally indicates the command portion of the instruction; the address field generally indicates the address portion of the instruction.

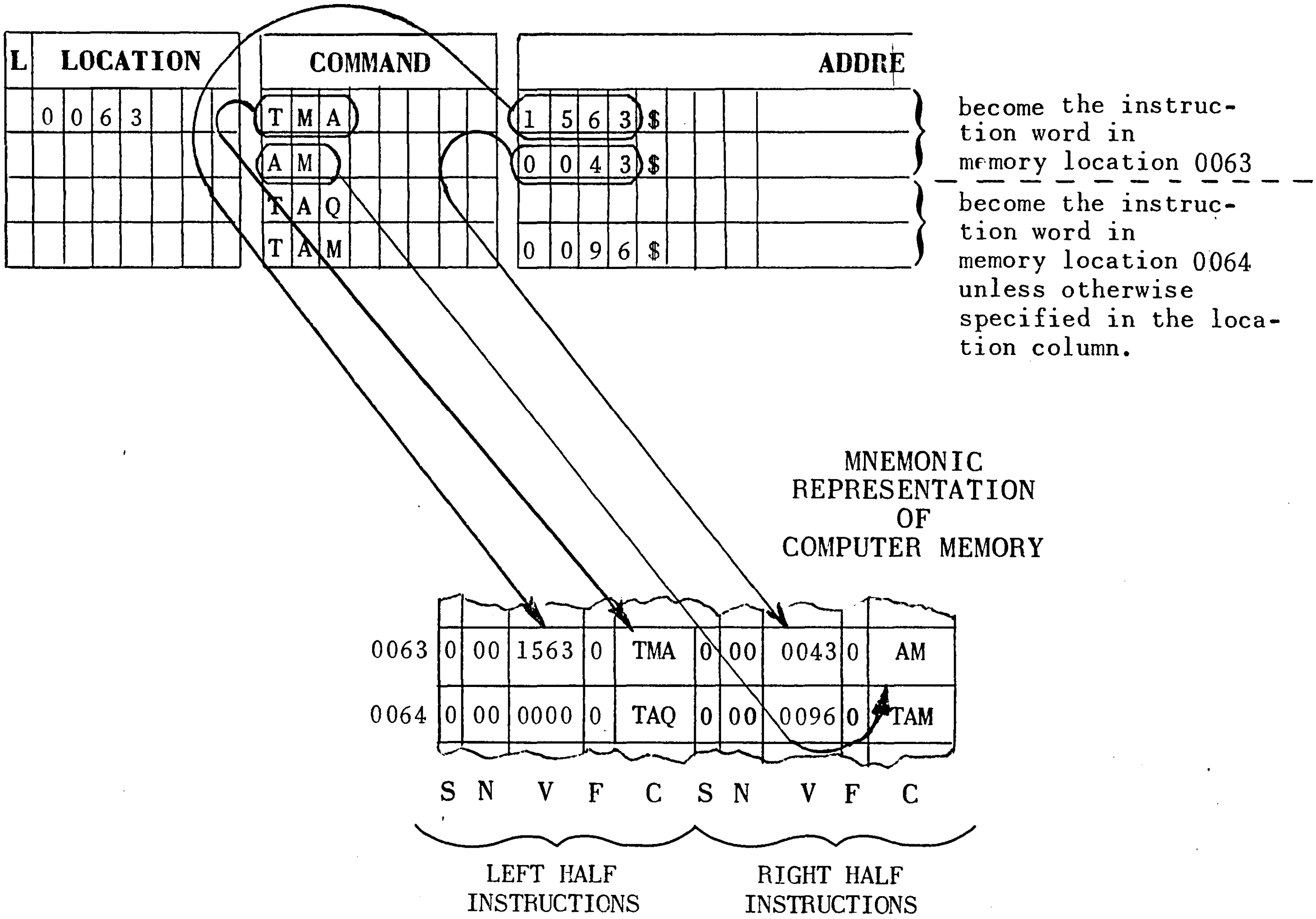
One PHILCO 2000 word is normally produced from two consecutive instruction lines. This results in the first instruction becoming the left half instruction and the second instruction becoming the right half instruction.

In all of the illustrative examples and exercises it is assumed that the computer instructions for each program have been stored in the memory by a manual loading procedure. It is also assumed that the data for each problem has been read into the memory by instructions which precede the example or exercise. The choice of memory locations, as far as operation mode is concerned, is arbitrary — every location is the same as every other.

In this chapter no concern will be given to computer representation of numbers or other data. When numbers and data are shown in memory locations or registers they are shown in English-decimal form. When the contents of a memory location or register are not significant or zero they are shown as 0——0, i.e., all zeros.

As indicated previously, two lines of mnemonic coding form one PHILCO 2000 instruction word. The following diagrams illustrate the computer representation of instructions:

WRITTEN MNEMONIC CODE



L	LOCATION	COMMAND	ADDRE
	0063	TMA	1563$
		AM	0043$
		TAQ	
		TAM	0096$

become the instruction word in memory location 0063

become the instruction word in memory location 0064 unless otherwise specified in the location column.

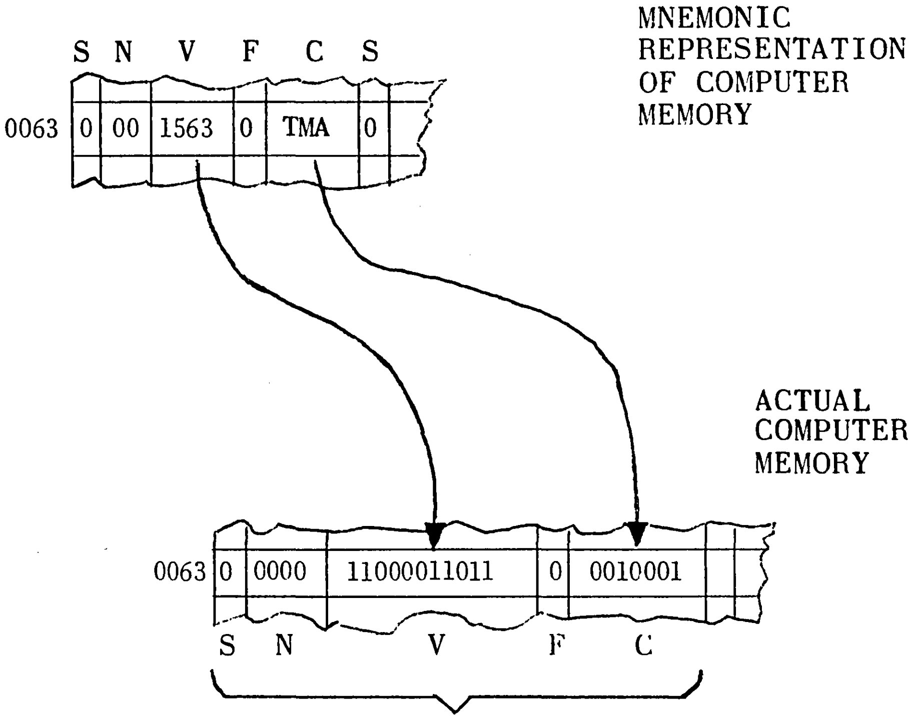
MNEMONIC REPRESENTATION OF COMPUTER MEMORY

	S	N	V	F	C	S	N	V	F	C
0063	0	00	1563	0	TMA	0	00	0043	0	AM
0064	0	00	0000	0	TAQ	0	00	0096	0	TAM

LEFT HALF INSTRUCTIONS

RIGHT HALF INSTRUCTIONS

The following would be the actual computer code for the left half instruction in memory location 0063:



TRANSFERS OF INFORMATION

Transfer Instructions

One of the most common operations in data processing is the transfer of information from one place to another. Illustrative data transfers are transfers of information from magnetic tape to memory and vice versa and of punched-card data to and from memory, transfers from memory to the arithmetic and control sections, and transfers from one arithmetic register to another.

Some of the functions of transfers are to provide the computer with data to be processed, to remove data which has been processed, to provide information for computer control functions, and to provide operands for arithmetic operations.

Transfer instructions can cause

a. transfers of one word
b. transfers of address parts of words
c. transfers of single binary digits.

All other types of data transfer can be accomplished by other classes of instructions, such as Input-Output instructions.

A transfer operation within the computer is analogous to transferring a number, written in one ledger column, to a second ledger column without affecting the original number. If there were a number in the second column it must have been erased before the transfer occurred. Thus, a transfer of information is a duplication of information. Also, the information is permanently stored unless replaced by (or for) a transfer. Transfer instructions in the computer duplicate a word or part of a word in another section of the computer. The original information is unaffected by the transfer and the contents of the receiving element are replaced.

Arithmetic transfers may be from core memory to the arithmetic unit, arithmetic unit to core memory, or arithmetic register to arithmetic register. This section will describe only these one word transfers. However, <u>all</u> transfer instructions have the following three letter format:

Letter 1	Letter 2	Letter 3
Transfer	From this Location	To this Location

All transfers involving the arithmetic section require the following combinations of letters:

Transfer	From	To
T	M A Q D	M A Q D

where M, A, Q, D are abbreviations for Memory location, A Register, Q Register, and D Register, respectively. Transfers from one location to the same location, i.e., from M to M or from A to A, or from Q to Q, or from D to D, are not possible.

The arithmetic transfer commands and their abbreviated definitions are as follows:

COMMAND			EXPLANATION
TMA	$(M) \longrightarrow A$	(m) → D → A	Transfer (M) to A
TMD	$(M) \longrightarrow D$		Transfer (M) to D
TMQ	$(M) \longrightarrow Q$	(m) → D → Q	Transfer (M) to Q
TAM	$(A) \longrightarrow M$	(A) → D → M	Transfer (A) to Memory
TAD	$(A) \longrightarrow D$		Transfer (A) to D
TAQ	$(A) \longrightarrow Q$	(A) → D → Q	Transfer (A) to Q
TQM	$(Q) \longrightarrow M$	(Q) → D → M	Transfer (Q) to Memory
TQA	$(Q) \longrightarrow A$	(Q) → D → A	Transfer (Q) to A
TQD	$(Q) \longrightarrow D$		Transfer (Q) to D
TDM	$(D) \longrightarrow M$		Transfer (D) to Memory
TDA	$(D) \longrightarrow A$		Transfer (D) to A
TDQ	$(D) \longrightarrow Q$		Transfer (D) to Q.

The parentheses are read as "the contents of", and the arrow indicates the transfer of information and the direction.

The D Register is the data register and is used in all arithmetic transfer instructions even though it is not always shown. When a word is transferred from memory to the arithmetic section, it is first transferred to the D Register. Conversely, when a word is transferred from the arithmetic section to memory, it is first transferred to the D Register. The same holds for transfers between registers. Therefore, several of the above transfers might be shown as follows:

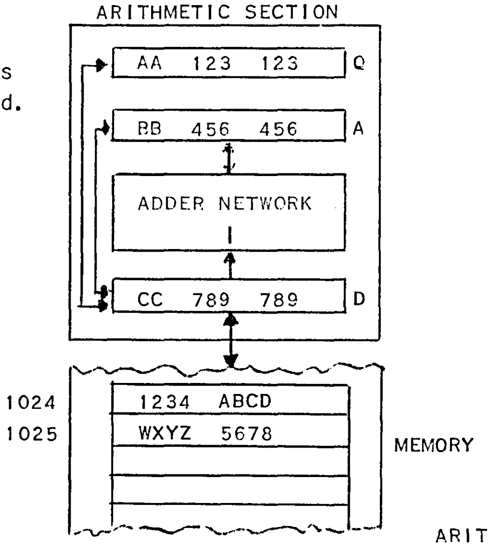
COMMAND	EXPLANATION		
TMA	(M) ⟶ D	then	(D) ⟶ A
TQA	(Q) ⟶ D	then	(D) ⟶ A
TQM	(Q) ⟶ D	then	(D) ⟶ M.

Transfer instructions involving memory require an address written in the address column to complete the instruction. If the transfer does not involve memory, the address column may be left blank. For example, two Transfer instructions may be written as follows:

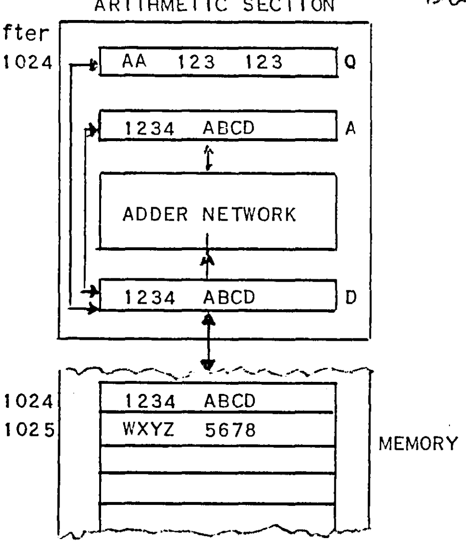
COMMAND								ADDRESS AND REMARKS						
T	M	A						1	0	2	4	$		
T	Q	M						1	0	2	5	$		

The first of the following diagrams represents the arithmetic section and the memory before the execution of the preceding instructions; the second and third diagrams represent the arithmetic section and the memory after execution of the instructions.

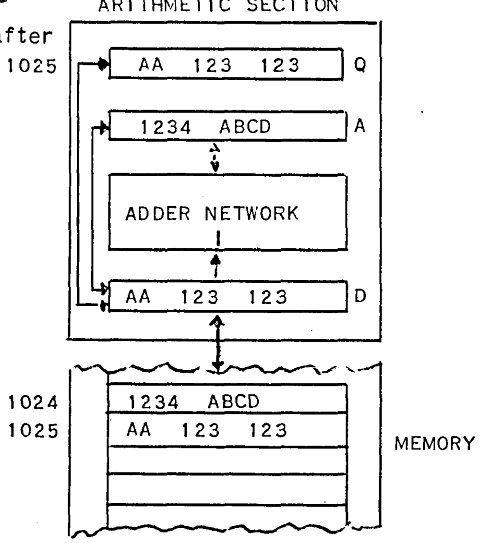
1. before the instructions are executed.



2. after TMA 1024



3. after TQM 1025



Frequently it is necessary to store an arithmetic result or other quantity in memory. This may either be a final or intermediate result. In either case it must be decided which particular memory location to use. The following factors are guides in making this decision:

a. Final results are usually stored in a special output area in memory.

b. Unless the input data is to be reused, input data areas may be used as output storage areas.

c. Any unused areas of memory may be used for intermediate storage.

In the examples on page 28 particular locations were chosen to illustrate a variety of storage uses. In preparing solutions to exercises, any locations may be used unless a specific location is designated.

Clear Instructions

Clear instructions are similar to Transfer instructions. These instructions - CM, CA, CD, and CQ - clear a location to zero. Unlike the Transfer instructions, however, the D Register is not affected, except in the CD instruction.

ARITHMETIC

Addition and Subtraction

In the PHILCO 2000, addition and subtraction take place with operands in the A and D Registers, such that

$$(A) + (D) \rightarrow A$$

and

$$(A) - (D) \rightarrow A.$$

This is read as the contents of the D Register are added to or subtracted from the contents of the A Register, and the result is placed in the A Register.

Prior to the addition or subtraction instruction, one operand, the augend or minuend, must have been placed in the A Register by a Transfer instruction or another arithmetic operation. The particular Addition or Subtraction instruction transfers the other operand from memory or from the Q Register to the D Register. However, the instruction may utilize the existing word in D. The addition or subtraction then takes place between the contents of A and D with the result going to A, as illustrated above.

Arithmetic operations in the 2000 are performed similarly to those done on a scratch pad by a person. The person writes down the two numbers - just as the 2000 places the two numbers in the A and D Registers. The arithmetic is then performed. The person writes the result on the pad for further use - just as the 2000 places the result in the A Register. This may be illustrated as follows:

	0———01247	A		0——— 0507634	A
+	0———0436	D	-	0————04271	D
	0——— 01683	A		0——— 0503363	A

When an operand comes from memory or the Q Register, either, both, or neither of the two options, A and S, are possible. The absolute value of the operand may be added or subtracted and the result may be stored in a specified memory location. (Vertical lines on either side of a location are read as "the absolute value of.")

$$(A) \pm |M| \longrightarrow M$$

$$(A) \pm |Q| \longrightarrow M$$

When an operand is in memory and the result is to be stored in memory, only one memory location is affected, and the operand is replaced by the arithmetic result.

Addition and subtraction commands may be graphically illustrated as follows:

Operation	Operand	Options
Add or Subtract	Operand to or from (A)	In Absolute value — — — — — Store result

The following letters are used to make up the commands:

	Operation	Operand	Options
Add	A	M Q	A S

	Operation	Operand	Options
Subtract	S	M Q	A S

As in the Transfer instructions, a result is first transferred to the D Register before it is stored.

The preceding arithmetic operations, therefore, would be shown as follows:

a. (M) or $(Q) \longrightarrow D$

b. $(A) \pm [(D)$ or $|D|] \longrightarrow A$

c. $(A) \longrightarrow D$, $(D) \longrightarrow M$, when the result is stored.

The basic addition and subtraction operations are

$$(A) \pm [(M) \text{ or } (Q) \text{ or } (D)] \longrightarrow A.$$

Clear Add, Clear Subtract

Supplementing the above instructions are the Clear Add and Clear Subtract instructions which clear the A Register to zero before adding or subtracting is accomplished. The Clear Add and Clear Subtract instructions may be written by prefacing the instructions on page 30 with a C.

The Clear Add instructions, in effect, result in one-word transfers. If the absolute value of the operand is used, the magnitude of a number, regardless of its sign, is transferred. Clear Subtract instructions transfer numbers with signs opposite to the original signs. In the floating point mode, these instructions can be used to normalize numbers.

Other Addition and Subtraction Instructions

The Addition and Subtraction instructions, AD and SD, use the existing word in the D Register with no options possible. The contents of the D Register are added to or subtracted from the contents of the A Register and the result is placed in the A Register. This is illustrated as follows:

$$(A) \pm (D) \longrightarrow A$$

However, the Addition and Subtraction instructions may be executed in the floating point mode by preceding the command with an "F".

All fixed point addition and subtraction commands, including abbreviated definitions, are listed on the following page.

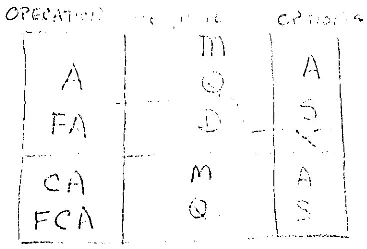
COMMAND	EXPLANATION	MEANING
AD	$(A) + (D) \rightarrow A$	Add D
AM	$(A) + (M) \rightarrow A$	Add Memory
AMA	$(A) + \lvert M \rvert \rightarrow A$	Add Memory, Absolute
AMS	$(A) + (M) \rightarrow M$	Add Memory, Store
AMAS	$(A) + \lvert M \rvert \rightarrow M$	Add Memory, Absolute and Store
AQ	$(A) + (Q) \rightarrow A$	Add Q
AQA	$(A) + \lvert Q \rvert \rightarrow A$	Add Q, Absolute
AQS	$(A) + (Q) \rightarrow M$	Add Q, Store
AQAS	$(A) + \lvert Q \rvert \rightarrow M$	Add Q, Absolute and Store
CAM	$0 + (M) \rightarrow A$	Clear Add Memory
CAMA	$0 + \lvert M \rvert \rightarrow A$	Clear Add Memory, Absolute
CAMS	$0 + (M) \rightarrow M$	Clear Add Memory, Store
CAMAS	$0 + \lvert M \rvert \rightarrow M$	Clear Add Memory, Absolute and Store
CAQ	$0 + (Q) \rightarrow A$	Clear Add Q
CAQA	$0 + \lvert Q \rvert \rightarrow A$	Clear Add Q, Absolute
CAQS	$0 + (Q) \rightarrow M$	Clear Add Q, Store
CAQAS	$0 + \lvert Q \rvert \rightarrow M$	Clear Add Q, Absolute and store
SD	$(A) - (D) \rightarrow A$	Subtract D
SM	$(A) - (M) \rightarrow A$	Subtract Memory
SMA	$(A) - \lvert M \rvert \rightarrow A$	Subtract Memory, Absolute
SMS	$(A) - (M) \rightarrow M$	Subtract Memory, Store
SMAS	$(A) - \lvert M \rvert \rightarrow M$	Subtract Memory, Absolute and Store
SQ	$(A) - (Q) \rightarrow A$	Subtract Q
SQA	$(A) - \lvert Q \rvert \rightarrow A$	Subtract Q, Absolute

COMMAND	EXPLANATION	MEANING
SQS	(A) - (Q) → M	Subtract Q, Store
SQAS	(A) - |Q| → M	Subtract Q, Absolute and Store
CSM	0 - (M) → A	Clear Subtract Memory
CSMA	0 - |M| → A	Clear Subtract Memory, Absolute
CSMS	0 - (M) → M	Clear Subtract Memory, Store
CSMAS	0 - |M| → M	Clear Subtract Memory, Absolute and Store
CSQ	0 - (Q) → A	Clear Subtract Q
CSQA	0 - |Q| → A	Clear Subtract Q, Absolute
CSQS	0 - (Q) → M	Clear Subtract Q, Store
CSQAS	0 - |Q| → M	Clear Subtract Q, Absolute and Store.

Micro-Flowcharts

The following two figures, called micro-flowcharts, graphically illustrate all possible additions and subtractions. The ovals containing questions represent computer functions which are necessary whenever an arithmetic order is interpreted by the computer. The operations of one complete and particular operation may be seen by tracing through the flowchart from "Select Instruction" to "Select Next Instruction." Certain details, which will be explained in a later chapter but which do not affect the basic operation, have been omitted from the flowcharts.

Note that in these flowcharts, as in others to follow, each branch on the chart produces another subclass of instructions.



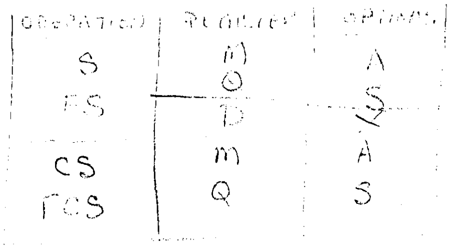
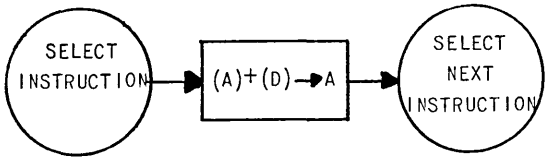


Figure 3. MICRO-FLOWCHARTS OF ALL ADDITION INSTRUCTIONS

1. IF THE ADDEND IS IN THE D REGISTER:



2. IF THE ADDEND IS IN MEMORY OR THE Q REGISTER:

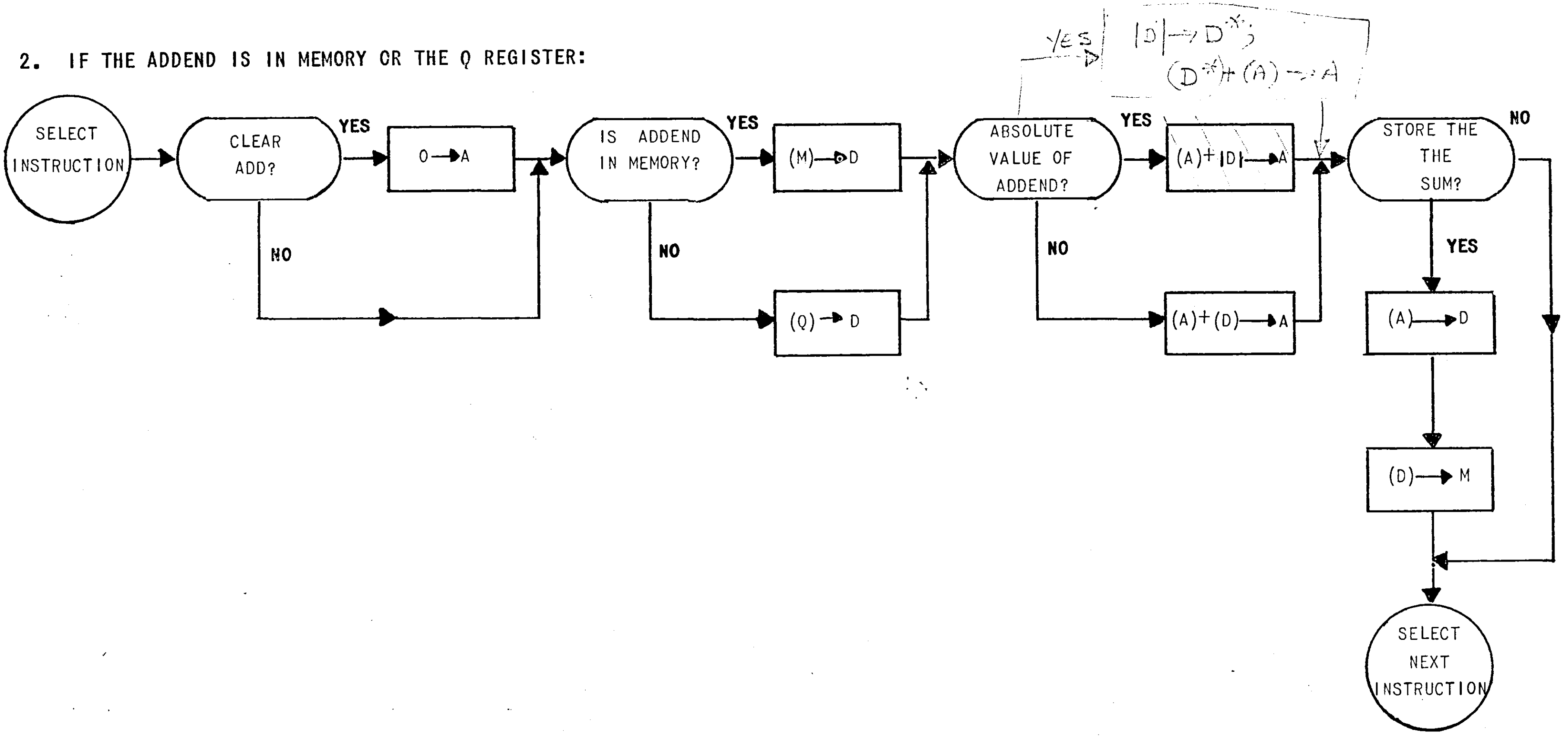
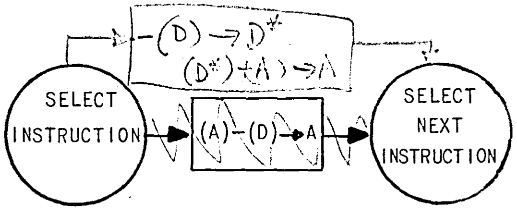
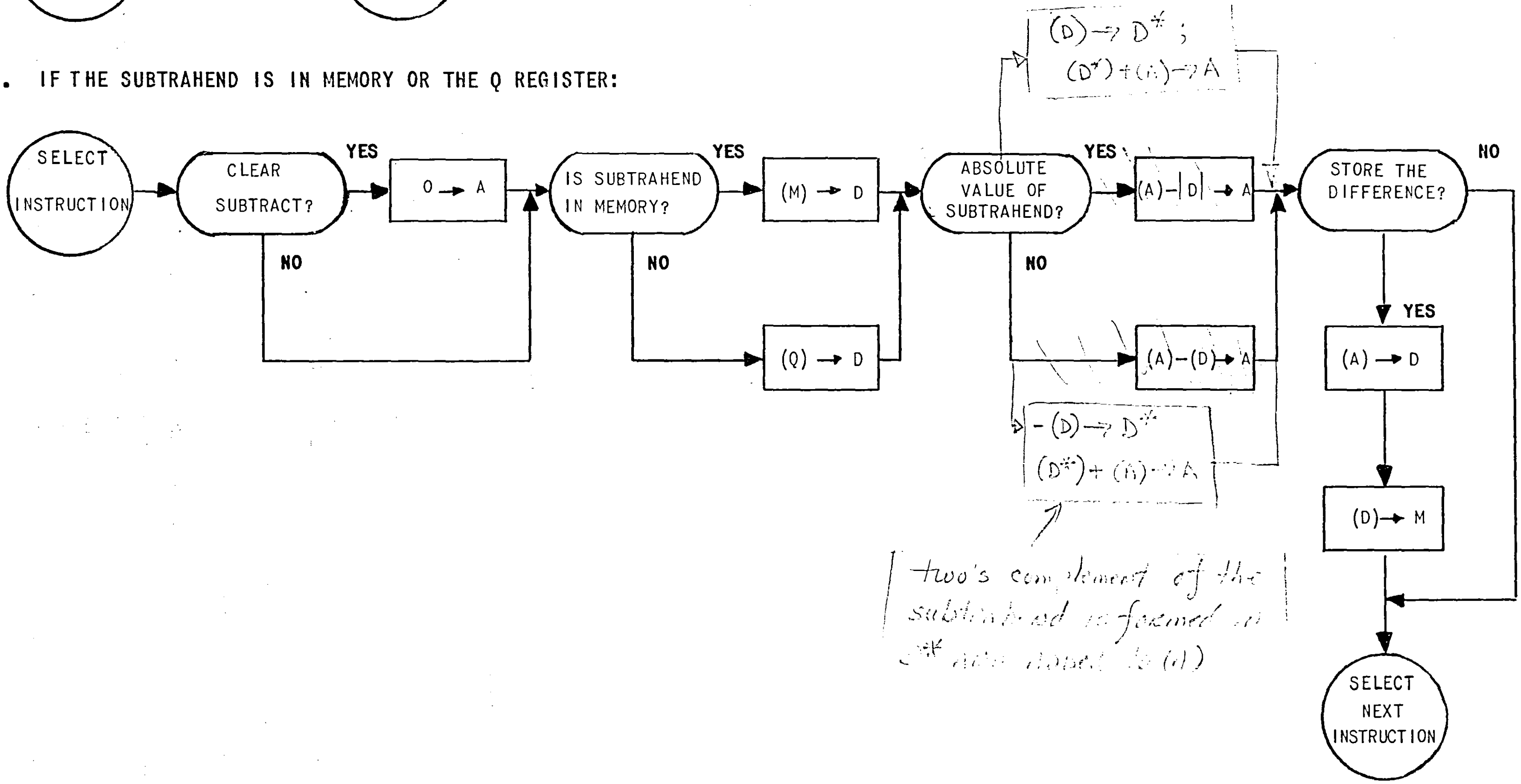


Figure 4. MICRO-FLOWCHARTS OF ALL SUBTRACTION INSTRUCTIONS

1. IF THE SUBTRAHEND IS IN THE D REGISTER:



2. IF THE SUBTRAHEND IS IN MEMORY OR THE Q REGISTER:



two's complement of the subtrahend is formed as D* and added to (A))

The following example illustrates the Addition, Subtraction, and Transfer instructions:

Example 1

A basic inventory operation is to add the quantity of an item of stock on order to the amount of stock on hand and then to subtract from this sum the quantity of the stock sold. This is a part of the operation known as updating an inventory.

If the following numbers represent quantities of a particular kind of transistor:

a.	amount of transistors on hand:	11,463
b.	amount of transistors on order:	5,000
c.	amount of transistors sold:	7,500,

then the updated inventory (new amount on hand) would result from the following arithmetic:

$$11,463 + 5,000 - 7,500 = 8,963.$$

With data stored in the following memory locations, the problem is to perform the arithmetic described above and store the new on-hand amount in memory location 3971:

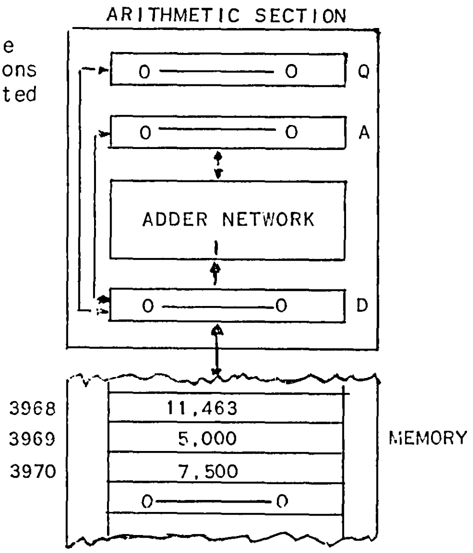
Memory Location	Contents
3968	On-Hand Amount
3969	On-Order Amount
3970	Sold Amount

The coding to do this follows:

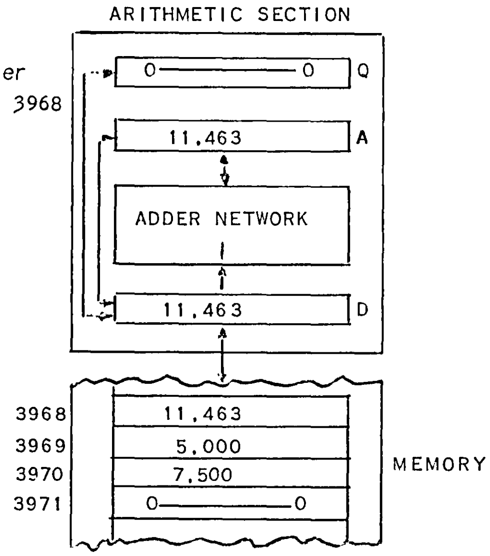
COMMAND								ADDRESS AND REMARKS						
T	M	A						3	9	6	8	$		Transfer Øn Hand to A
A	M							3	9	6	9	$		Add Øn Ørder to A
S	M							3	9	7	0	$		Subtract Sold from A
T	A	M						3	9	7	1	$		Transfer result to memory

Diagrams of the arithmetic section and the memory follow. Diagram 1 represents the two areas before the execution of the preceding instructions. Diagrams 2 through 5 represent the arithmetic section and the memory after execution of each instruction. The numbers shown in the registers and memory locations are assumed to be properly aligned. Treatment of non-aligned numbers will be given in Chapter VI.

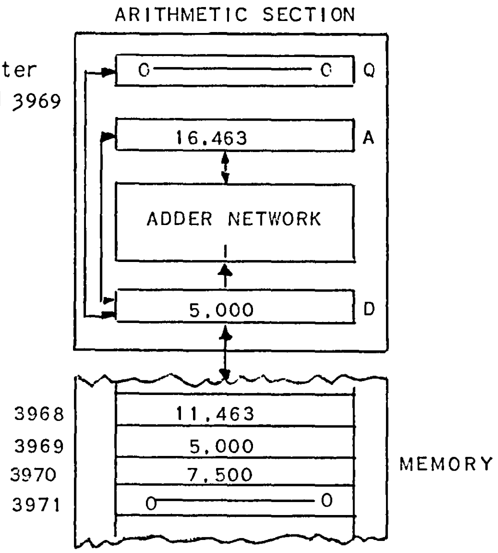
1. before the instructions are executed



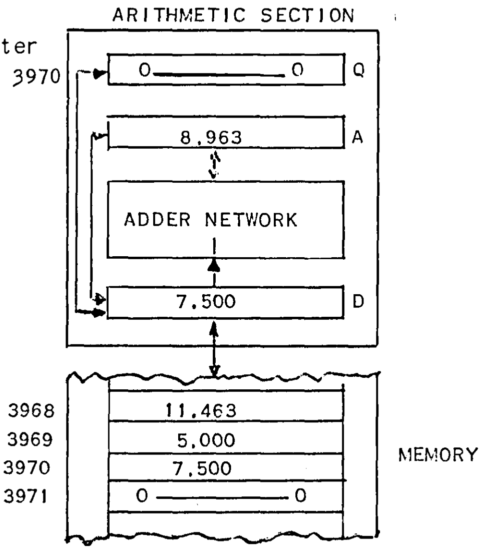
2. after TMA 3968



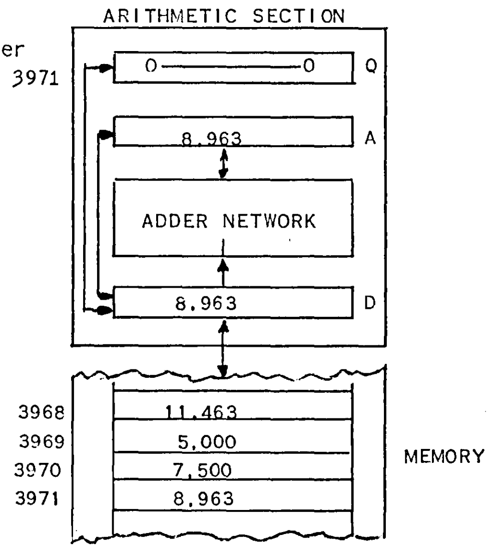
3. after AM 3969



4. after SM 3970



5. after TAM 3971



If it were desired to store the new on-hand amount in memory location 3970, only three instructions would be necessary. The third instruction would be SMS 3970: subtract the contents of memory location 3970 from the contents of the A Register and store the difference in memory location 3970. It should be realized that in doing this the amount sold is erased and replaced.

Similarly, if it were desired to replace the original on-hand amount with the new on-hand amount, the following instructions could be used:

COMMAND								ADDRESS AND REMARKS						
T	M	A						3	9	6	9	$		Transfer Øn-Ørder to A
S	M							3	9	7	0	$		Subtract Sold from A (on order)
A	M	S						3	9	6	8	$		Add Øn-Hand to A and store the sum in 3968

Example 2

The following figures for an employee's paycheck are stored in memory as shown:

Memory Location	Contents
3968	Gross Base Pay
3969	Overtime Pay
3970	Social Security Tax
3971	City Income Tax
3972	Federal Income Tax

Calculate the employee's net pay and store it in memory location 3973.

Solution:

COMMAND								ADDRESS AND REMARKS						
T	M	A						3	9	6	8	$		Transfer Gross Pay to A
A	M							3	9	6	9	$		Add Øvertime Pay to Gross Pay
S	M							3	9	7	0	$		Subtract Social Security Tax
S	M							3	9	7	1	$		Subtract City Income Tax
S	M							3	9	7	2	$		Subtract Federal Income Tax
T	A	M						3	9	7	3	$		Transfer net pay to memory

Exercises

1. Add the number from memory location 3840 to the numbers in memory locations 3968 through 3970. Each sum should replace its operand in 3968 through 3970.

2. Memory locations 3968 through 3972 contain numbers representing cash receipts. Compute their sum.

Multiplication

Several types of multiplication are possible in the PHILCO 2000. For each type, the multiplier must first be placed in the Q Register. This may be accomplished by a Transfer instruction, a Division instruction (quotients are developed in the Q Register), or a Shift instruction. The Multiplication instruction transfers the multiplicand to the D Register from a specified memory location or from the A Register. The actual multiplication then follows between the contents of the D and Q Registers.

One of two types of products may be specified in multiplication: unrounded double length products or rounded single length products. A double length product appears in the A and Q Registers, with the major half in A and the minor half in Q. In rounded multiplication the single length product appears in the A Register with the multiplier reappearing in the Q Register. Basic multiplication is shown in the formula,

$$\left[(M) \text{ or } (A)\right] \quad x \ (Q) \longrightarrow A,\ Q.$$

In each of the multiplications the options of storing the product (or major half of the product) and of using the absolute value of the multiplicand are possible. These operations may be shown symbolically as follows:

a. $(M) \text{ or } (A) \longrightarrow D$

b. $\left[(D) \quad \text{or} \ |D|\right] \quad x \ (Q) \longrightarrow A,\ Q$ unrounded
   or $\longrightarrow A$ rounded

c. $(A) \longrightarrow D,\ (D) \longrightarrow M$, when the result is stored.

A double length multiplication is illustrated as follows:

$$\boxed{+000000123450000} \quad D$$

$$\boxed{+000000678910000} \quad Q$$

X

$$\underset{A}{\boxed{+000000000000083}} \quad \underset{Q}{\boxed{+811439500000000}}$$

(Note that the + signs are illustrative only.)

Multiplication can be accomplished by shifting left which has the effect of multiplying by a power of ten.

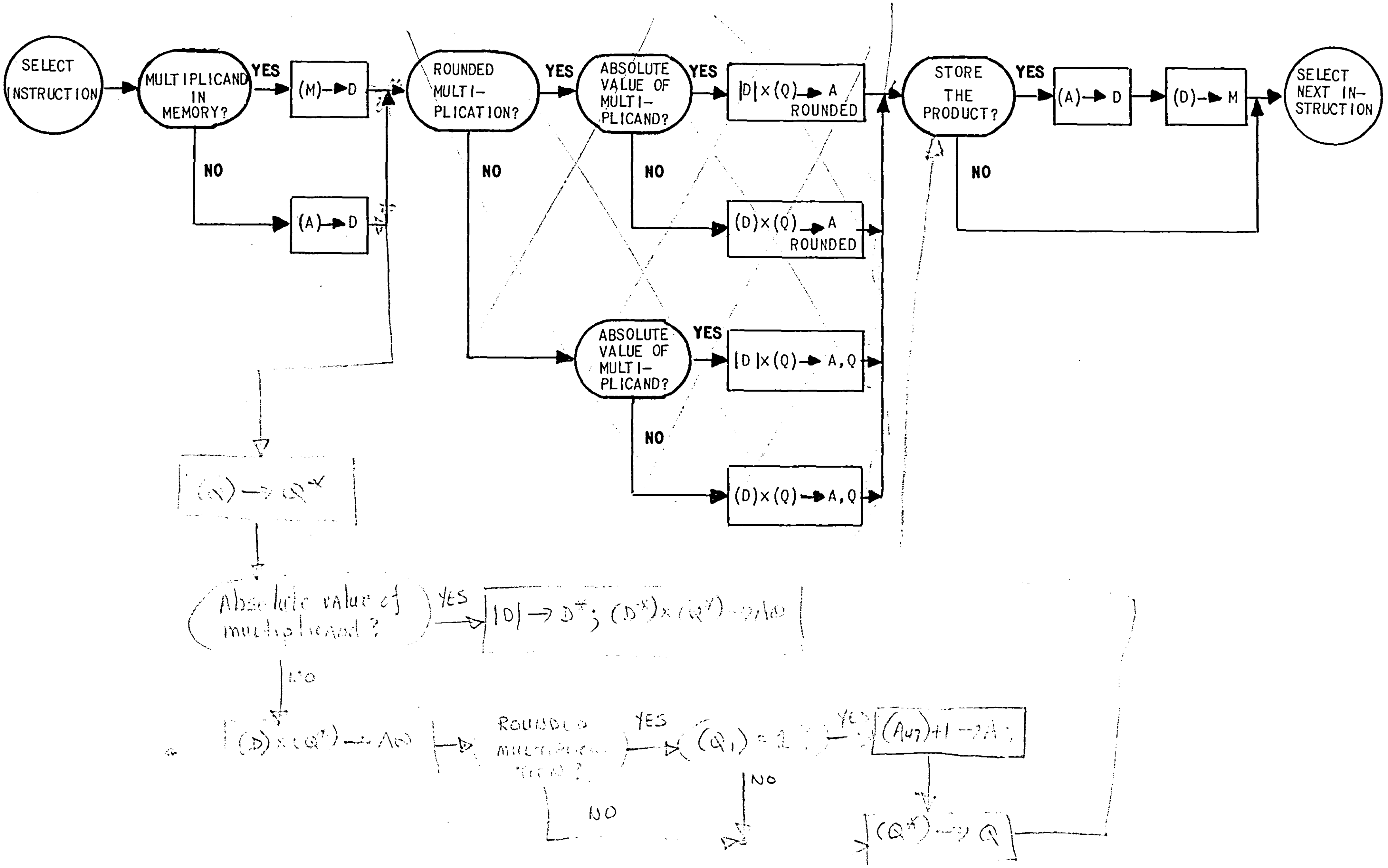
The multiplication command is constructed as follows:

Operation	Operand	Options
Multiply	(M) or (A) by (Q)	Absolute operand and/or Rounded result and/or Store result
M FM	A M	A R S

A list of multiplication commands and abbreviated definitions follows:

COMMAND	EXPLANATION	MEANING
MM	(M) x (Q) → A,Q	Multiply Memory
MMR	(M) x (Q) → A, rnd.	Multiply Memory and Round
MMS	(M) x (Q) → M,Q	Multiply Memory and Store (A)
MMRS	(M) x (Q) → M rnd.	Multiply Memory, Round and Store (A)
MMA	$\lvert M\rvert$ x (Q) → A,Q	Multiply Memory in Absolute value
MMAR	$\lvert M\rvert$ x (Q) → A,rnd.	Multiply Memory in Absolute value and Round
MMAS	$\lvert M\rvert$ x (Q) → M	Multiply Memory in Absolute value and Store (A)
MMARS	$\lvert M\rvert$ x (Q) → M rnd.	Multiply Memory in Absolute value, Round and Store (A)
MA	(A) x (Q) → A,Q	Multiply A
MAR	(A) x (Q) → A rnd.	Multiply A and Round
MAS	(A) x (Q) → M	Multiply A and Store (A)
MARS	(A) x (Q) → M rnd.	Multiply A, Round and Store (A)
MAA	$\lvert A\rvert$ x (Q) → A,Q	Multiply A in Absolute value
MAAR	$\lvert A\rvert$ x (Q) → A rnd.	Multiply A in Absolute value and Round
MAAS	$\lvert A\rvert$ x (Q) → M	Multiply A in Absolute value and Store (A)
MAARS	$\lvert A\rvert$ x (Q) → M rnd.	Multiply A in Absolute value, Round and Store (A)

Figure 5. MICRO-FLOWCHART OF ALL MULTIPLICATION INSTRUCTIONS



Special Multiplication Commands

Two special Multiplication instructions are possible in the PHILCO 2000. They are Multiply and Add (MAD) and Multiply and Subtract (MSU). The first step in each multiplication is (M) ⟶ D. Then this word is transferred to a duplicate of the D Register for the multiplication, and (A) ⟶ D. The multiplication yields a rounded product in A, and the contents of the Q Register are unaltered. The contents of the D Register are then added to or subtracted from the contents of the A Register, and the result replaces the contents of the A Register. At the conclusion of these instructions, the D Register contains the original contents of the A Register. Instruction MAD is especially useful for accumulating the sum of the products of two series of numbers.

$(m) \to D \to D^*;\ (A) \to D;\ (Q \to Q^*);\ (Q^*) \times (D) \to AQ;\ Q^* \to Q;$ ADD $(A) + (D) \to A$

SUB $-(D) \to D^*;\ (A) + (D^*) \to A$

Example 1

Compute the total cost of purchasing a quantity of items based on the following information and store the result in memory:

Memory Location	Contents
3968	Quantity Purchased
3969	Unit Cost
3970	Percentage Discount

As stated previously, no consideration will be given to the format of the numbers used in the example. Also it will be assumed that the significant digits of the products are only in the A Register. As will be shown in the next chapter, the programmer can ensure this result by a suitable arrangement of the data.

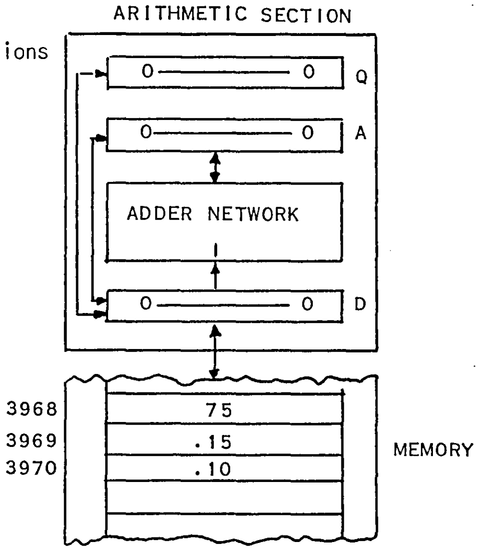
The necessary arithmetic in this example is (Quantity x Cost) x (1-% Discount). Numerous coding solutions are possible for this arithmetic operation. The following solution has been chosen because it is straightforward:

COMMAND								ADDRESS AND REMARKS					
T	M	Q						3	9	6	8	$	Transfer Quantity, (3968) to Q
M	M							3	9	6	9	$	Unit Cost x Quantity to A, Q
T	A	Q											Transfer(Unit Cost x Quantity) (A) to Q
M	M	R	S					3	9	7	0	$	Multiply % x (Q) to A and 3970
T	Q	A											Transfer (Unit Cost x Quantity) (Q) to A
S	M	S						3	9	7	0	$	Subtract Discount (3970) from (A) to 3970

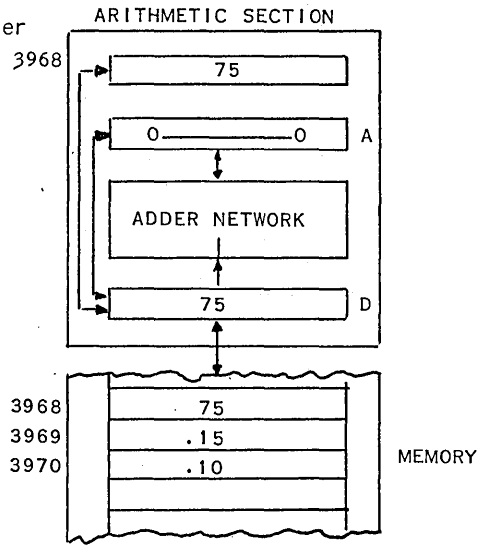
The following diagrams, which illustrate the effects of the preceding coding on the registers and memory, use the numbers below:

a.	quantity purchased:	75
b.	unit cost:	$.15
c.	% discount:	10% (.10)

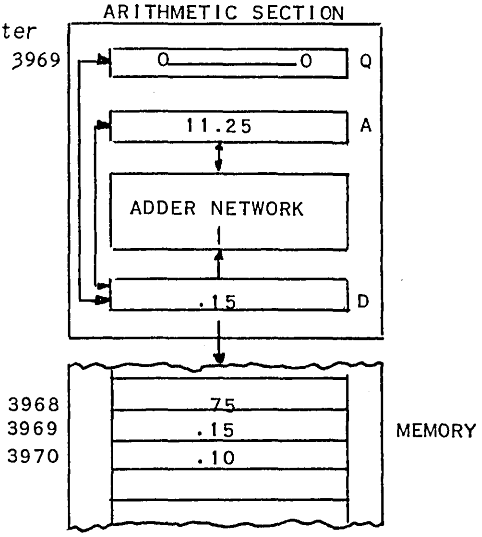
1. initial conditions



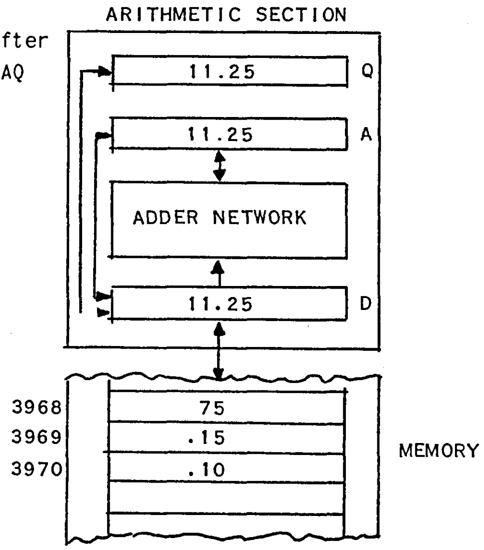
2. after TMQ 3968



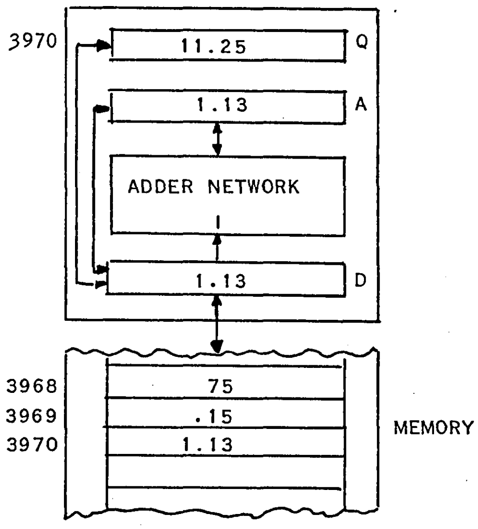
3. after MM 3969



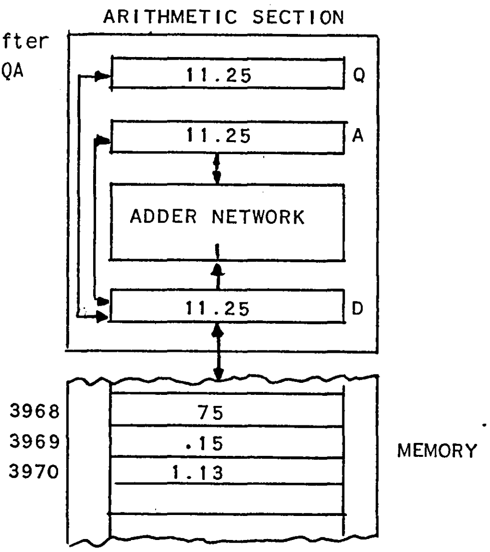
4. after
TAQ



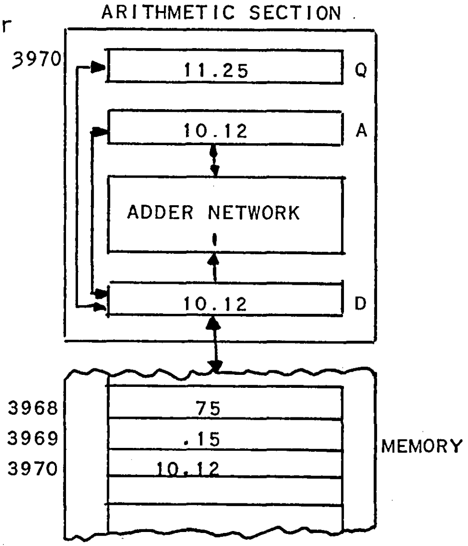
5. after
MMRS 3970



6. after
TQA



7. after
SMS 3970



Example 2

Although instruction MAD is normally used in matrix and statistical calculations, it will be helpful in the following type of operation:

Gross pay = hours x rate + overtime hours x overtime rate.

The factors are stored in the following memory locations:

Memory Location	Contents
3968	Hours Worked
3969	Base Rate of Pay
3970	Overtime Hours
3971	Overtime Rate of Pay

The coding to calculate gross pay is

COMMAND								ADDRESS AND REMARKS						
T	M	Q						3	9	6	8	$		Hours → Q
M	M							3	9	6	9	$		Hours x Rate → A, Q
T	M	Q						3	9	7	0	$		Øvertime Hours → Q
M	A	D						3	9	7	1	$		(Øvertime Rate x Øvertime Hours) +
														Hours x Rate → A

Exercise 1

Part of a production problem requires that the cost of manufacturing parts be calculated. The data includes the quantity of assemblies to be produced, the number of parts per assembly, and the unit cost per part. Compute the cost by multiplying Number of Assemblies x Number of Parts per Assembly x Unit Cost. Store the result in memory location 3971.

Memory Location	Contents
3968	Number of Assemblies
3969	Number of Parts per Assembly
3970	Unit Cost
3971	Result

Exercise 2

Memory locations 3968 through 3970 contain gross amounts due by customers. Each one is to be discounted by multiplying it by the factor in memory location 3967. (This number is actually a discount percentage subtracted from one.) Replace the gross amounts with the discounted amounts.

$$\frac{\text{DIVIDEND (A) or (AQ)}}{\text{DIVISOR (D)}} \longrightarrow \overset{(Q)}{\text{QUOTIENT}} + \overset{(A)}{\text{REMAINDER}}$$

Division

In the 2000, division involves single length or double length dividends. In double length division the dividend is in the A and Q Registers, with the major half in A and the minor half in Q. The sign of Q is ignored. In single length division, the dividend is only in A. In both cases the divisor is selected from memory and is placed in D.

The dividend must first be transferred to A, or A and Q, by transfer orders or by arithmetic operations. A double length dividend can be created by an unrounded multiplication. Then the Division instruction transfers the divisor from memory to the D Register, division proceeds, and the quotient is developed in the Q Register with the remainder appearing in A.

There are four divide orders. These operations may be shown symbolically as follows:

a. (M) → D

b. [(A) or (A,Q)] ÷ (D) → Q, remainder → A

c. (Q) → D, (D) → M, when the result is stored.

The following are the division commands and their abbreviated symbolic definitions:

COMMAND	EXPLANATION	MEANING
DA	(A) ÷ (M) → Q	Divide A
DAS	(A) ÷ (M) → M	Divide A and Store
DAQ	(A,Q) ÷ (M) → Q	Divide A and Q
DAQS	(A,Q) ÷ (M) → M	Divide A and Q and Store.

See pages 92 for overflow and 99 for division rules

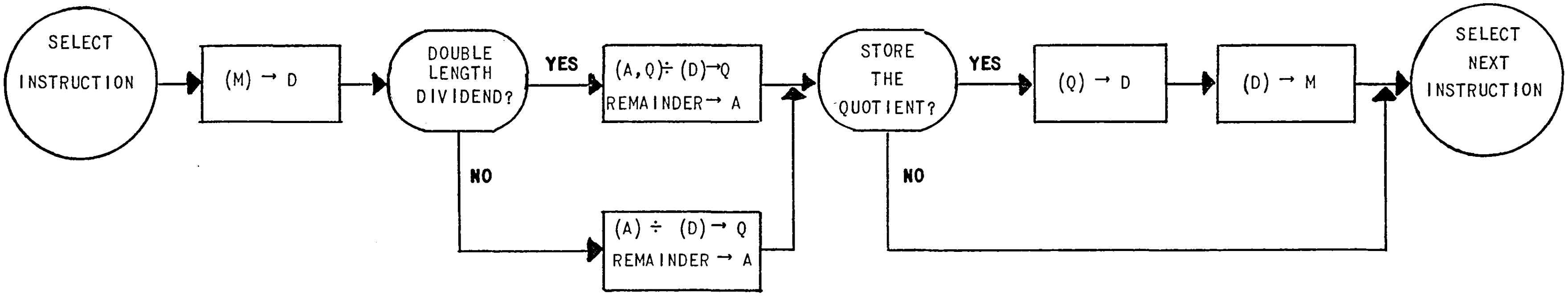
$|(D)| > |(A) \text{ or } (AQ)|$ for division to take place

ie:

```
A 0|0100
D 0|1000
  0|100
```

$\frac{4B4}{2B2} = 232$

Figure 6. MICRO-FLOWCHART OF ALL DIVISION INSTRUCTIONS



Division is more time consuming than any other arithmetic operation and, if possible, should be avoided. Two ways of doing this are multiplication by the reciprocal of the divisor, and shifting, which has the effect of dividing by a power of two. Shifting will be explained in more detail in Chapter VI.

Example

Memory locations 3968, 3969, and 3970 contain the cost of living indices for three years. Compute, and store in memory, the average index, i.e., (Index 1 + Index 2 + Index 3) ÷ 3 = Average Index.

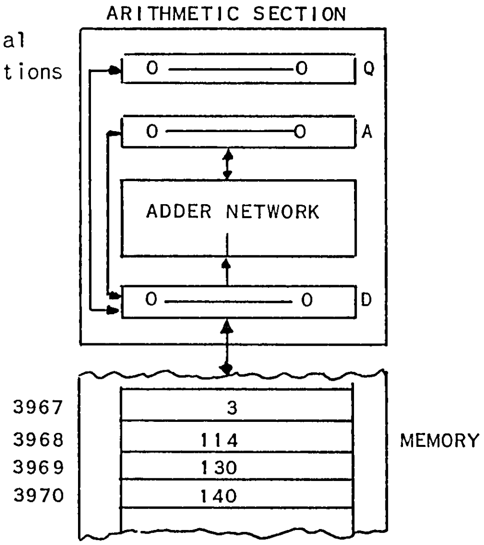
(Assume a constant of 3 in memory location 3967.)

COMMAND								ADDRESS AND REMARKS						
T	M	A						3	9	6	8	$		Transfer Index 1 to A
A	M							3	9	6	9	$		Add Index 2 + (A) → A
A	M							3	9	7	0	$		Add Index 3 + (A) → A
D	A	S						3	9	6	7	$		Divide (A) by 3 → 3967

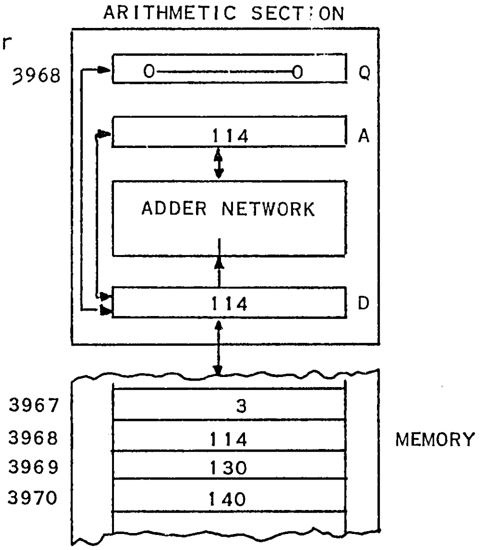
Memory Location	Contents
3967	3
3968	114
3969	130
3970	140

The following illustrations show the effects of the preceding instructions on the memory and arithmetic section.

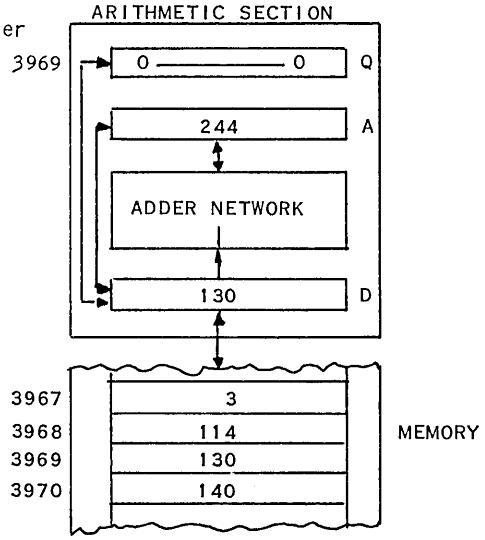
1. initial conditions



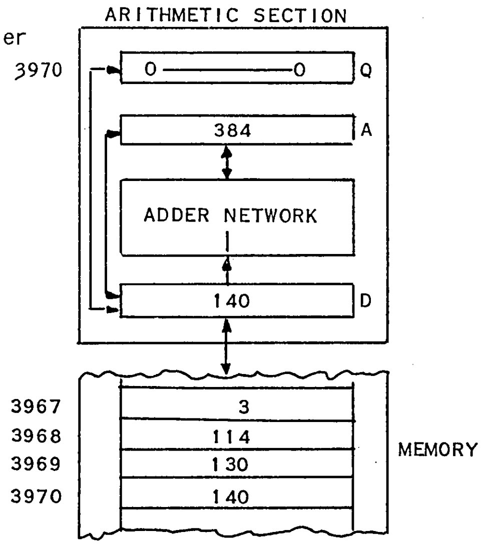
2. after TMA 3968



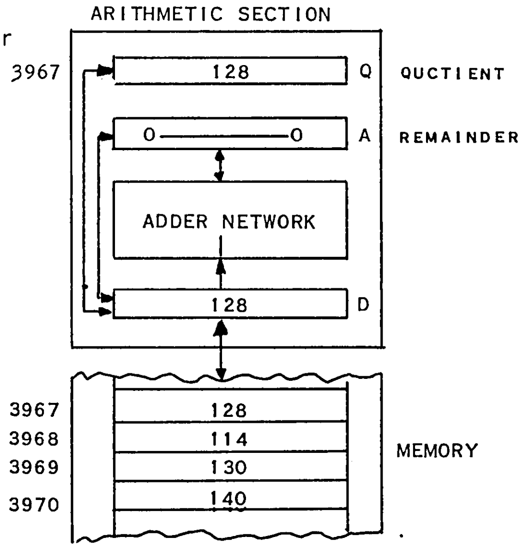
3. after AM 3969



4. after AM 3970



5. after DAS 3967



Exercise 1

The following information is given:

Memory Location	Contents
3968	The Number of Sales for Month 1
3969	The Number of Sales for Month 2
3970	The Number of Sales for Month 3
3971	The Total Dollar Receipts for Month 1 Sales
3972	The Total Dollar Receipts for Month 2 Sales
3973	The Total Dollar Receipts for Month 3 Sales

a. Compute the Average number of Sales for one month.

b. Compute the Average dollar Receipts for one month.

c. Compute the Average dollar Receipt for the Average monthly Sale.

1. $$\frac{\text{Sales 1} + \text{Sales 2} + \text{Sales 3}}{3} = \text{Average Sales}$$ B15

2. $$\frac{\text{Receipts 1} + \text{Receipts 2} + \text{Receipts 3}}{3} = \text{Average Receipts}$$ B2[illegible]

3. $$\frac{\text{Average Receipts}}{\text{Average Sales}} = \text{Average Receipt/Sale}$$

(Assume, as always, that the operands will be properly aligned throughout. Also assume that the number 3 is stored in memory location 3967.)

Exercise 2

An airplane travels a prescribed course of length (L) a number of times. It travels the course x times in a period of time (t_1), y times in a period of time (t_2), and z times in a period of time (t_3). Compute the airplane's average rate of speed.

$$R = \frac{L\,(x + y + z)}{t_1 + t_2 + t_3}$$

SUMMARY: TRANSFERS AND ARITHMETIC

a. Data transferred between memory and the arithmetic section and within the arithmetic section passes through the D Register.

b. The transfer instructions are

T (From Location 1) (To Location 2).

The locations are M, A, Q, D; location 1 can not be the same as location 2.

c. The basic arithmetic operations are

Addition:	(A) + [(M), (Q), or (D)] ⟶ A
Subtraction:	(A) - [(M), (Q), or (D)] ⟶ A
Multiplication:	[(M) or (A)] x (Q) → A, Q, or A rounded
Division:	[(A) or (A, Q)] ÷ (M) → Q, Remainder → A.

In every case except AD, SD, MAD, MSU, and division, the absolute value of one operand may be used.

d. Arithmetic rules of thumb:

1. Use replace type operations when the result is needed in memory and when one instruction less will be necessary.
2. When transferring a word from one memory location to another, use only the D Register. This saves time and doesn't disturb the other registers.
3. Whenever possible, keep an addition or subtraction operand which is to be used again in the Q Register to save memory access.
4. If possible, use an operand from the D Register. This provides the fastest operation speed.
5. Use addition instead of multiplication.
6. Use multiplication instead of division.
7. Use shifting instead of multiplication or division.
8. When computing the product of several numbers, each partial product may be transferred from the A Register to the Q Register for the next multiplication.
9. To accumulate the sum of products use MAD. The sum remains in the A Register while the products are formed between the contents of Q and M.
10. The contents of all registers and memory locations are unaltered by transfers from the registers and memory locations. Thus, in store type arithmetic operations the results are retained in the D Register and the original register (A or Q Register). The results may then be used without additional transfer operations.

FUNCTIONS OF ARITHMETIC REGISTERS

IN ARITHMETIC OPERATIONS

(Result Not Stored)

Operation	Time	A Register	D Register	Q Register
Addition	Before operation	Augend	Addend	-
	After operation	Sum	Addend	-
Subtraction	Before operation	Minuend	Subtrahend	-
	After operation	Difference	Subtrahend	-
Multiplication: Double Length Product	Before operation	-	Multiplicand	Multiplier
	After operation	Left half of product	Multiplicand	Right half of product
Multiplication: Rounded Product	Before operation	-	Multiplicand	Multiplier
	After operation	Rounded Product	Multiplicand	Multiplier
Division: Double Length Dividend	Before operation	Left half of dividend	Divisor	Right half of dividend
	After operation	Remainder	Divisor	Quotient
Division: Single Length Dividend	Before operation	Dividend	Divisor	-
	After operation	Remainder	Divisor	Quotient

Note that when the result is stored in memory, the D Register will contain the result rather than the operand shown in the table.

DECISION MAKING

Electronic data processing systems have proven very valuable in their "ability" to make simple, routine decisions. So significant is this feature that eventually all routine decisions might be made by machines, and management's time would be reserved for "the exception" decisions.

Routine decisions are those which result from answering such simple questions as

a. Does the employee have a bond deduction?

b. Is there a transaction for this record?

c. Has the customer remitted?

d. Have the deductions reduced a loan balance to zero?

On the basis of the answers to these questions appropriate action can be initiated.

Although these questions appear dissimilar they can be generalized as follows:

a. Is a number positive? negative? zero?

b. Is one identifying number or name equal to another?

c. Is one identifying number or name greater than another? Less than another?

The PHILCO 2000 can "answer" the first category of questions by "examining" the contents of an arithmetic register to "see" if its contents are positive, negative, or zero. It can answer the other questions by "comparing" the contents of two arithmetic registers to "see" if the contents of one are equal to, or greater than another.

Jump Instructions

To enable the 2000 to make decisions, Jump instructions have been provided. Depending upon the answer to a question, each Jump instruction is followed by one of two possible courses of action. Either the normal sequence of executing instructions is maintained, or it is interrupted, and a new sequence is begun.

The definitions that follow summarize the preceding statements and apply throughout this manual:

a. Jump: The operation that causes the computer's control section to select the next instruction to be executed from the memory location specified by the address part of a Jump instruction. This interrupts the normal sequence of executing instructions.

b. Conditional Jump Instruction: An instruction which causes a jump to be effected if certain conditions (usually within the arithmetic section) are satisfied. If the conditions are not satisfied, the next instruction to be selected will be the next sequential instruction.

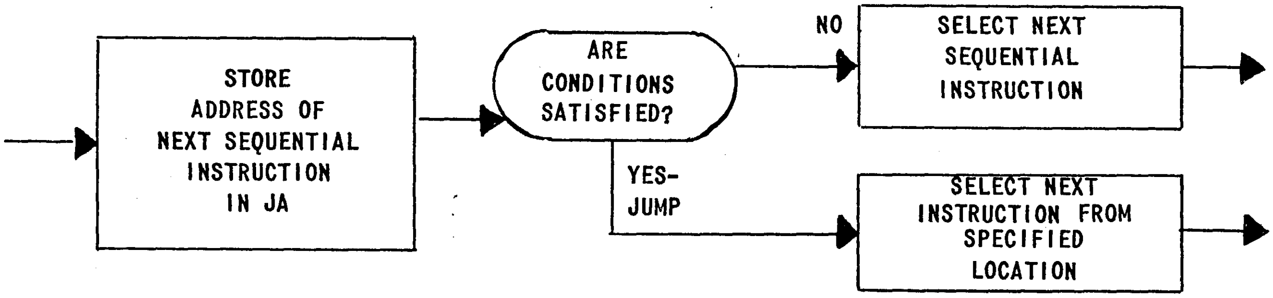
c. Unconditional Jump Instruction: An instruction which causes a jump independent of any conditions.

The PHILCO 2000 jump command codes begin with the letter J. The letters that follow the J specify the type of jump or the conditions. The following are jump commands:

JMP:	Unconditional Jump
JAN:	Jump if (A) are Negative
JAP:	Jump if (A) are Positive
JAZ:	Jump if (A) are Zero
JDP:	Jump if (D) are Positive

(Note that "positive" assumes only that the sign bit of a word is zero; "negative" assumes that the sign bit is one.)

The last four "if" jump commands are conditional jump commands. A graphic representation of conditional Jump instructions follows:


STORE ADDRESS OF NEXT SEQUENTIAL INSTRUCTION IN JA
ARE CONDITIONS SATISFIED?
NO
SELECT NEXT SEQUENTIAL INSTRUCTION
YES-JUMP
SELECT NEXT INSTRUCTION FROM SPECIFIED LOCATION

(The reason that the address of the next instruction is stored in JA will be given in Chapter VII.)

The complete PHILCO 2000 Jump instruction consists of a command part and an address part

COMMAND	ADDRESS
JMP	START

which reads: Jump to......... the instruction at this symbolically identified location.

The designation of the location jumped to may be any name, number, or combination of letters and numbers, necessary to identify the instruction to be executed following the jump. This designated location is called the "jump to address or location", and the instruction at this address is called the "jump-to instruction".

Example

The following illustrative example illustrates the addressing requirements for Jump instructions:

Memory Location	Contents
3968	Loan Balance
3969	Loan Payment
3970	Number of Active Loans
3971	Number of Cleared Balances

Compute the new loan balance. If it equals zero, add one to the number of cleared balances. If it doesn't equal zero, add one to the number of active loans. Assume that the constant of one is in memory location 3972.

For the sake of simplicity, only the instructions relating to the above operations are shown.

L	LOCATION	COMMAND	ADDRESS AND REMARKS	
		CSM	3969$	O - Loan Payment → A
		AMS	3968$	Loan Balance + (A) → 3968
		JAZ	ZRØBAL$	Jump if (A) are zero
		TMA	3972$	Add one to Number of
		AMS	3970$	Active Loans
.	....	....	....	....
	ZRØBAL	TMA	3972$	Add one to Number of
		AMS	3971$	Cleared Balances

This coding is read as follows:

a. Subtract the payment from the loan balance.

b. Test (A) for zero.

c. If (A) are zero, execute the instructions TMA 3972, AMS 3971 - at location ZROBAL.

d. If (A) are not zero, execute the next sequential instructions: TMA 3972, AMS 3970.

Symbolic Addressing

By now the reader is aware that symbolic addressing is a coding convenience. Before executing the program, the Translator-Assembler-Compiler will assign computer memory addresses to all symbolic addresses. If the programmer wants to have "jump to" coding begin in a particular memory location he writes the address of this memory location in the address part of the Jump instruction and the location column of the "jump to" instruction.

Unless the "jump to" address is known, it is convenient to temporarily omit the address parts of Jump instructions. Following a conditional Jump instruction, write the coding for the no jump case. After this, the next line can be used to begin the "jump to" coding, and the address part of the Jump instruction can then be completed. Caution must be exercised that the blank "jump to" addresses are filled in. Otherwise, they will cause jumps to memory location zero.

Equality Comparisons

Numeric or alphanumeric words may be compared for relative magnitude or equality. The two PHILCO 2000 jump commands performing equality comparisons are

JAED: Jump if (A) equals (D)

JAEQ: Jump if (A) equals (Q). (Q) → D

All comparisons in the computer take place between the A and D Registers. Therefore, in a comparison between the contents of the A and Q Registers, the contents of Q are automatically transferred to D, thereby erasing the contents of D.

Magnitude Comparisons

The two instructions JAGD and JAGQ cause magnitude comparisons. One number, or group of symbols, is compared to another number or group of symbols to determine their relative magnitudes. For comparison purposes, a word in the PHILCO 2000 should be considered either as a signed number or as an alphanumeric word.

If the word is alphanumeric it consists of eight binary-coded characters, whose relative values or weights can be determined from the subsequent chart. Then, depending on whether the words to be compared are alphanumeric or numeric, the appropriate Jump instruction is selected. The following Jump instructions cause magnitude comparisons:

JAGD: Jump if (A) are greater than or equal to (D). The contents of the registers are considered to be alphanumeric.

(A ≥ D)

JAGQ: Jump if the number in A is greater than or equal to the number in Q. (If the numbers are floating point numbers use JAGQF.) The contents of the registers are considered to be signed numbers.

A ≥ [illegible]
actually
(Q) → D

Representation of Characters

To make magnitude comparisons, every representable character must have a size or weight to distinguish it from every other character. This is analogous to the different values of numbers, i.e., 1 is smaller than 2; 3 is larger than 1, and so on.

The weights are determined by the six binary digits representing each character as shown in the table, PHILCO 2000 Binary Code for Alphanumeric Characters (page 58). The first two bits of each character are the Zone bits and the last four bits are the Numeric bits. Thus the binary configuration for the letter A appears as 010001 and contains an 01 zone and an 0001 numeric part.

The smallest configuration is 000000, and the largest is 111111. Within a column the lower characters are larger than the higher characters, i.e., H(011000) is larger than A(010001). From column to column, every character in a column is larger than any character in a column to its left, i.e., -(100000) is larger than O(000000), but is smaller than Δ.(110000).

Use of this table is illustrated by two alphanumeric comparisons using the JAGD instruction.

A Register	D Register	Jump ?
12345678	.2345678	No
ABC12345	ABC12344	Yes

Referring to the table, 1 is represented by 00 0001 while the period (.) is represented by 011011. Since 1 is not greater than . , the jump is not made.

PHILCO 2000

BINARY CODE FOR ALPHANUMERIC CHARACTERS

Zone / Numeric	00	01	10	11
0000 0	0 0	20 +	40 -	60 Δ (space)
0001 1	1	A	J	/
0010 2	2	B	K	S
0011 3	3	C	L	T
0100 4	4	D	M	U
0101 5	5	E	N	V
0110 6	6	F	Ø	W
0111 7	7	G	P	X
1000 0	10 8	30 H	50 Q	70 Y
1001 1	9	I	R	Z
1010 2	@	n filler	tab ┐	|
1011 3	=	● period	$	, comma
1100 4	;	)	*	(
1101 5	≡	%	<	>
1110 6	&	?	#	:
1111 7	' prime	"	⌴	e end of line Σ

XX XXXX
zone numeric
bits

Δ: Space symbol

Example

Memory locations 3968, 3969, and 3970 contain information for an inventory record. Memory locations 3840 and 3841 contain information for a transaction record.

Memory Location	Contents
3840	Transaction Stock Number
3841	Amount Ordered
3968	Inventory Stock Number
3969	On-Hand Amount
3970	Minimum Required Amount

a. If the stock numbers are the same (a match), perform the processing defined below.

b. If the numbers are not the same (no match), perform the coding at location ADVFILE.

c. If they match, determine whether or not the amount on hand is greater than or equal to the amount ordered, i.e., the order can be filled. If it is greater than or equal to, replace the on-hand amount by the difference between the on-hand and the ordered amounts and go on to the processing defined below. (If the on-hand amount is less than the ordered amount, go to the ADVFILE routine).

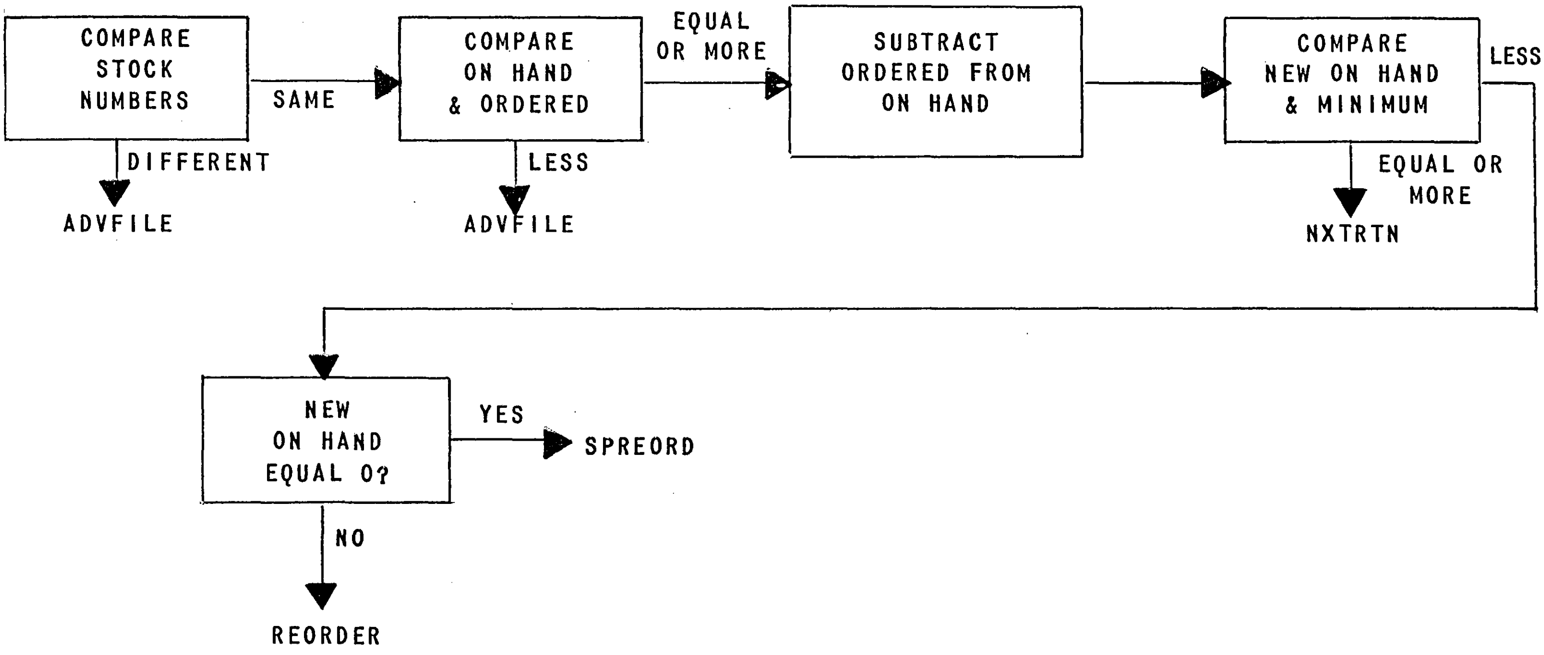
d. Compare the new on-hand amount, i.e., the above difference, to the minimum required amount.

e. If the new on-hand amount is greater than or equal to the minimum, go to NXTRTN.

f. If it is less, go to REORDER - unless it is zero, in which case go to SPREORD.

The routines ADVFILE, NXTRTN, REORDER, AND SPREORD will not be coded for this example.

This problem may be graphically explained as follows:



The coded solution to this problem becomes

L	LOCATION	COMMAND	ADDRESS AND REMARKS	
		TMA	3968$	Inventory Stock Number → A
		TMD	3840$	Transaction Stock Number → D
		JAED	START$	Jump to START if (A) = (D)
		JMP	ADVFILE$	If no match, jump to ADVFILE
	START	TMA	3969$	Øn-Hand Amount → A
		TMQ	3841$	Amount Ørdered → Q
		JAGQ	UPDATE$	Jump to UPDATE if (A) ≥ (Q)
		JMP	ADVFILE$	If (A) < (Q) jump to ADVFILE
	UPDATE	SQS	3969$	Øn Hand - Ørdered → A, 3969
		TMQ	3970$	Minimum Required Amount → Q
		JAGQ	NXTRTN$	Jump to NXTRTN if (A) ≥ (Q)
		JAZ	SPREØRD$	Jump to SPREØR if (A) = 0
		JMP	REØRDER$	Jump to REØRDER if (A) < (Q) and (A) ≠ 0

Analysis of the Coding

TMA 3968 and TMD 3840 place the Stock Numbers in the A and D Registers so that they may be compared for equality. JAED performs the comparison and jumps to START if (A) = (D). If inequality is the case, the next instruction, JMP ADVFILE, is executed. TMA 3969 and TMQ 3841 place the On-Hand and Ordered Amounts in the A and Q Registers to compare their relative magnitudes.

JAGQ performs the comparison and jumps to UPDATE if (A) ≥ (Q). (JAGQ rather than JAGD is used because the information being compared is numeric rather than alphanumeric, i.e., binary-coded.) If (A) < (Q), more material has been ordered than is on hand, and the next instruction, JMP ADVFILE, is executed.

SQS 3969 subtracts the Ordered Amount (in the Q Register) from the On-Hand Amount (in the A Register) and places the result in the A and D Registers and memory location 3969.

TMQ 3970 places the Minimum Required Amount in the Q Register to compare it to the new On-Hand Amount in the A Register. JAGQ performs the comparison and jumps to NXTRTN if the new On-Hand Amount is greater than or equal to the Minimum Required. If the new On-Hand Amount is less than the Minimum Required, a reorder is necessary.

JAZ SPREORD checks (A) for zero. If the new On-Hand Amount is zero, SPREORD is jumped to; if (A) are not zero, the next instruction, JMP REORDER, is executed.

[illegible]
[illegible] 126
Pg (14)

Exercise

The following data in memory pertains to an employee:

Memory Location	Contents
3968	Number of Hours Worked
3969	Hourly Pay Rate
3970	Overtime Pay Rate
3971	Number of Exemptions
3972	Union Dues
3973	Hospitalization Contribution
3974	Year to Date Gross Pay
3975	Year to Date Net Pay
3976	Year to Date Social Security Tax
3977	Year to Date Income Tax

Definitions:

Overtime Hours = Hours worked in excess of 40.

Gross Pay = Hours (not more than 40) x Hourly Rate + Overtime Hours x Overtime Rate.

Income Tax = [Gross Pay -(13 x Number of Exemptions)] x .18.

Social Security Tax = 3.00% x Gross Pay.

Net Pay = Gross Pay - Income Tax - Social Security Tax - Union Dues - Hospitalization Contribution.

This exercise has the following parts:

Part 1:

Determine if the employee worked overtime. If so, store the Overtime Hours in memory location 3978.

Part 2:

Compute Gross Pay, store in 3979, and add it to Year to Date Gross Pay.

Part 3:

Compute Income Tax, store in 3980, and add it to Year to Date Income Tax. If the computed Income Tax is negative, assume the tax is zero.

Part 4:

Compute Social Security Tax, store it in 3981, and add it to Year to Date Social Security Tax. The new Year to Date Social Security Tax must not exceed $144.00. Therefore, do not deduct the full 3.00% if it will cause the Year to Date total to exceed $144.00.

Part 5:

Using the above results, compute Net Pay, store it in 3982, and add it to Year to Date Net Pay. Make only the deductions that can be made in the above order of priority.

The necessary constants for this routine are stored as follows:

Memory Location	Contents
3000	40
3001	13
3002	.18
3003	.03
3004	144

SUMMARY: DECISION MAKING

a. Jump instructions are necessary to allow for alternate paths of processing. A jump may take place depending on the comparison of one word with another or one binary digit with another.

b. The jump is effected by transferring the address part of the Jump instruction to the Central Computer control section. When a Jump instruction is written, the address part of the instruction is also written in the location column of the instruction to be executed next IF the jump is effected.

c. Regardless of the type of jump, the address of the next sequential instruction is stored in the Jump Address Register, JA.

d. The Jump instructions listed below may be summarized as follows:

1. Jump if (D) are positive.
2. Jump if (A) are positive, negative, or zero.
3. Jump if (A) equal (D) or (Q).
4. Jump if (A) are equal to or greater than (D) or (Q).

COMMAND	EXPLANATION
JMP	Unconditional Jump
JDP	Jump if (D) are positive
JAP	Jump if (A) are positive
JAN	Jump if (A) are negative
JAZ	Jump if (A) are zero
JAED	Jump if (A) equals (D)
JAEQ	Jump if (A) equals (Q)
JAGD	Jump if (A) are greater than or equal to (D): alphanumeric
JAGQ	Jump if (A) are greater than or equal to (Q): numeric

(Note: Jump instructions involving the Q Register only will be described in Chapter VI.)

e. Because the computer may only compare (A) with (D), (Q) are transferred to D in the JAEQ and JAGQ commands.

f. Decision making rules of thumb:

1. When comparing (A) and (D), fill the A Register before the D Register. This is a must because words transferred to the A Register go through the D Register.

2. The signs of arithmetic results, except after division, may be determined by testing the sign of the A Register with JAN or JAP. Use JAZ to determine if a sum, difference, or rounded product is equal to zero. (This use of JAZ does not apply to floating point zero.)

3. When writing instructions, it is helpful to leave "jump to" addresses blank until the "jump to" coding can be written. This enables the programmer to keep track of the coding which remains to be written. Be careful to fill all addresses which have been left blank.

4. Equality comparisons (Jump instructions) apply to alphanumeric as well as to numeric words.

5. JAGD may be used for positive numbers because the actual magnitude of the number is not affected by the sign, which is zero.

6. JAGQ may be used for alphanumeric words if the sign positions are zero, i.e.,as if positive numbers were being compared.

CHAPTER IV

FLOWCHARTING

BASIC OPERATIONS

As programming operations become more detailed and complex, it grows increasingly difficult to remember and write down all of the possibilities in a problem which must be coded. To minimize this condition, a system to graphically represent the logical flow of processing has been devised. This system is called flowcharting.

A good flowchart is, in effect, a very detailed and accurate statement of the problem and at the same time is one type of solution. (Coding is another solution.) The basis of flowcharting is that a program can be represented as a series of several kinds of operations connected in a logical sequence. The following are typical operations in programming:

a. transfers of information

b. arithmetic operations

c. logical decisions

d. input-output.

In addition to these basic types of operations there are

e. start and stop situations

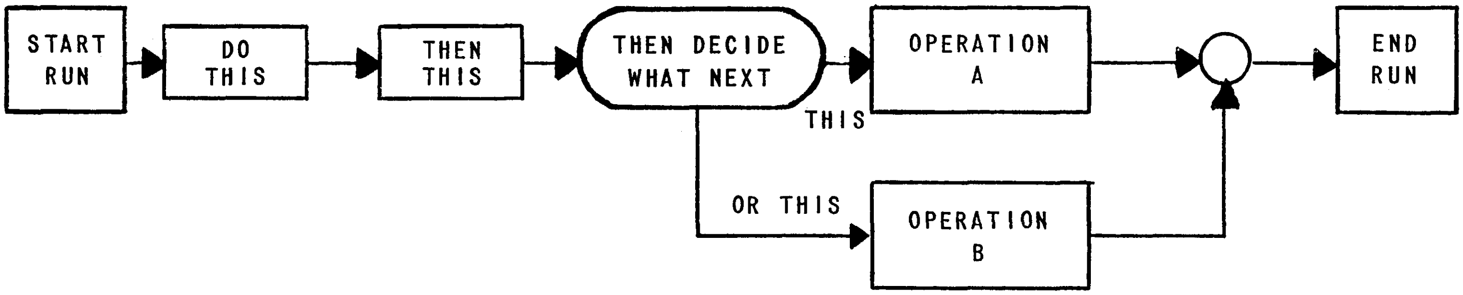
f. flowchart connections - jumps

g. subroutines

h. program switches.

The subject of subroutines and program switches will be covered in Chapter VII.

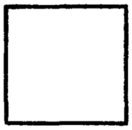
The use of flowcharts can be illustrated by the following flowchart:



FLOWCHART SYMBOLS

Because flowcharts contain many operations, it is convenient to have each type of operation appear in a box of unique shape. The following shapes are recommended for flowcharting and will be used throughout this manual.

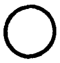
a. Starts and stops will appear in squares:



b. Transfers and arithmetic operations will be shown in rectangles of various sizes:

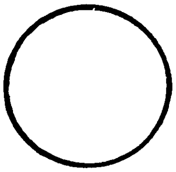
c. Decisions will be shown in flattened ovals of various sizes:

d. Flowchart connectors are small circles of various sizes:

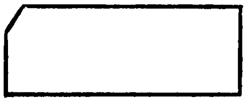


e. Input-output media utilize the following forms:

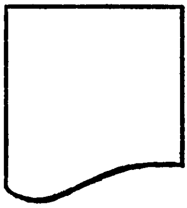
1. Paper tape or magnetic tape:



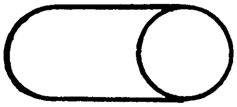
2. Punched cards:



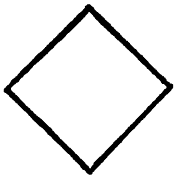
3. Console Typewriter:



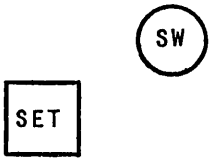
4. Magnetic drums:



f. Subroutines utilize diamonds:

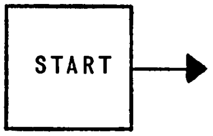


g. Program switches are small circles and are set by small squares:



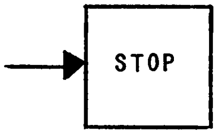
Illustrations of Symbol Usage

STARTING



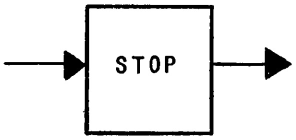
Start the program or computer run.

and



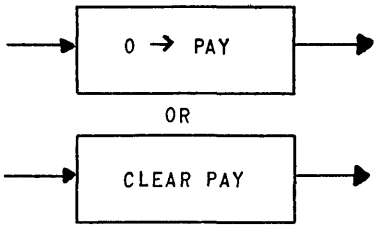
Stop computation: specify reason (as end of run, data error, etc.

STOPPING



Stop computation: continue when advance bar is depressed.

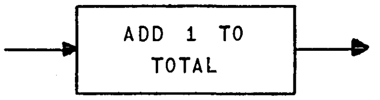
TRANSFERS



Transfer zero to the location containing the pay.

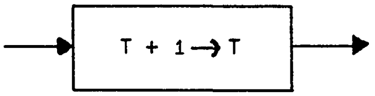
and

ARITHMETIC OPERATIONS



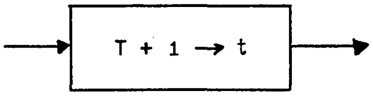
Add one to the location containing the total.

OR

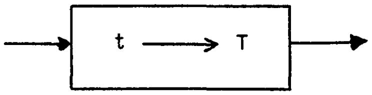


The old total is replaced by the new.

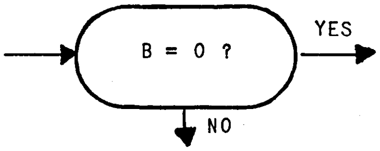
It is convenient in arithmetic operations to distinguish between the operation in which an original value is changed, as in the example above, and the operation which merely holds the result in a different memory location or register. If the old total were not to be replaced, the operation could be shown as



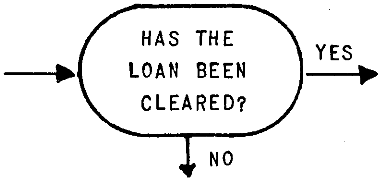
To replace the old total



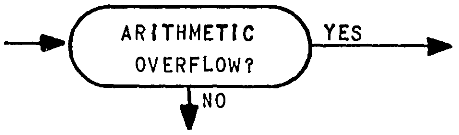
DECISIONS



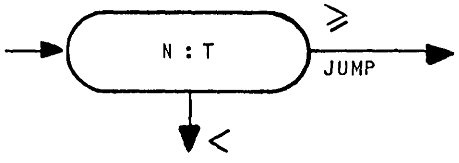
A decision based on a number B (Balance) being zero



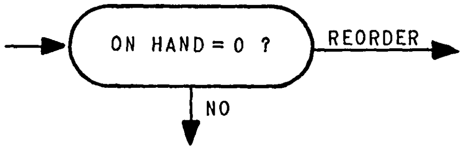
This is the same operation described in words.



Sampling of the overflow indicator



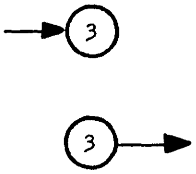
If desired, the two factors being compared can be shown inside the oval and the type of comparison can be shown with symbols outside the oval.



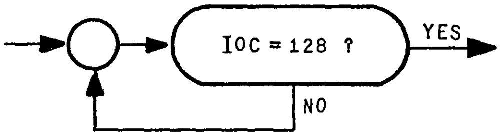
It will often be convenient to write the symbolic jump address on the appropriate arrow.

FLOWCHART CONNECTIONS

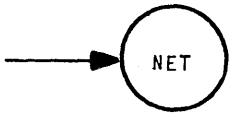
Whenever it is necessary to change the course of flow a connector is used.



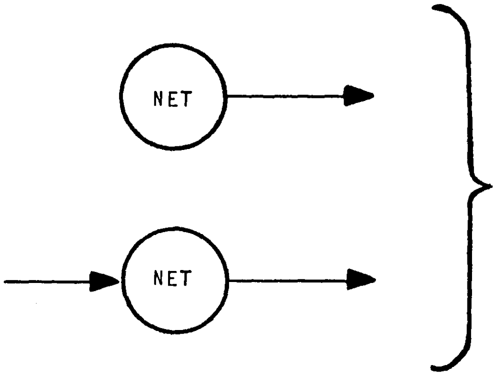
Because of limitations of paper, a numbered connector is used to indicate a change in course.



To connect merging lines, an unnumbered connector may be used. (IOC = Input-Output Counter)



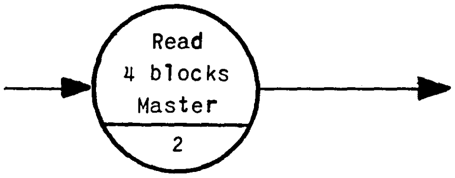
A jump to the part of the program called NET (JMP NET)



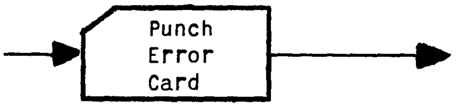
The part of the program called NET follows the arrow from a connector.

The arrow on the left may or may not indicate a jump.

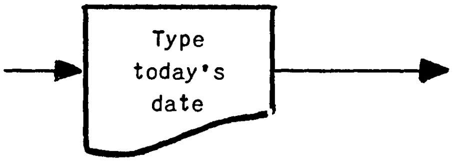
INPUT-OUTPUT



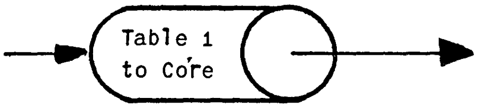
Paper tape or magnetic tape. The entry below the line is a unit number or file identification.



Punched cards



Console Typewriter

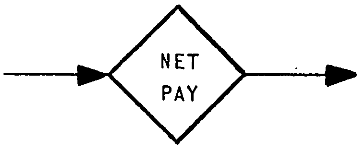


Magnetic drums

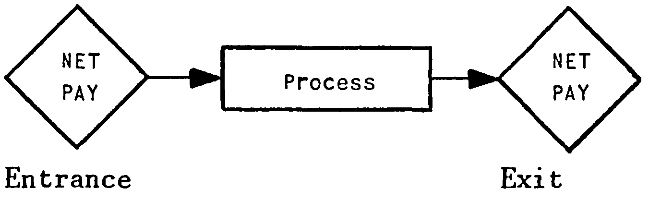
OTHER SYMBOLS

Use of the following symbols will be explained in Chapter VII:

SUBROUTINES

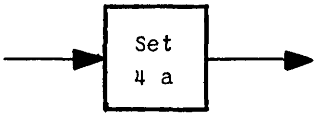


Execute the net pay subroutine and return. (JMP NETPAY)

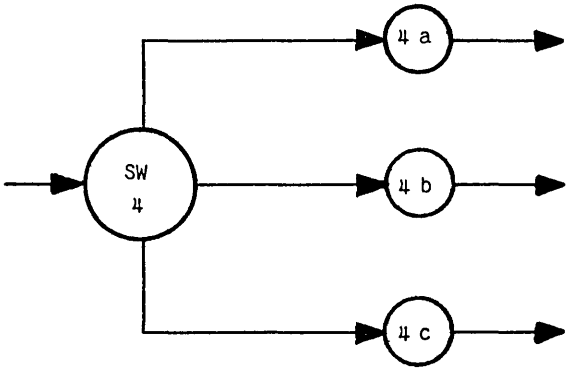


The net pay subroutine

PROGRAM SWITCHES



Set program switch 4 to the "a" path.



Path "a" of switch 4

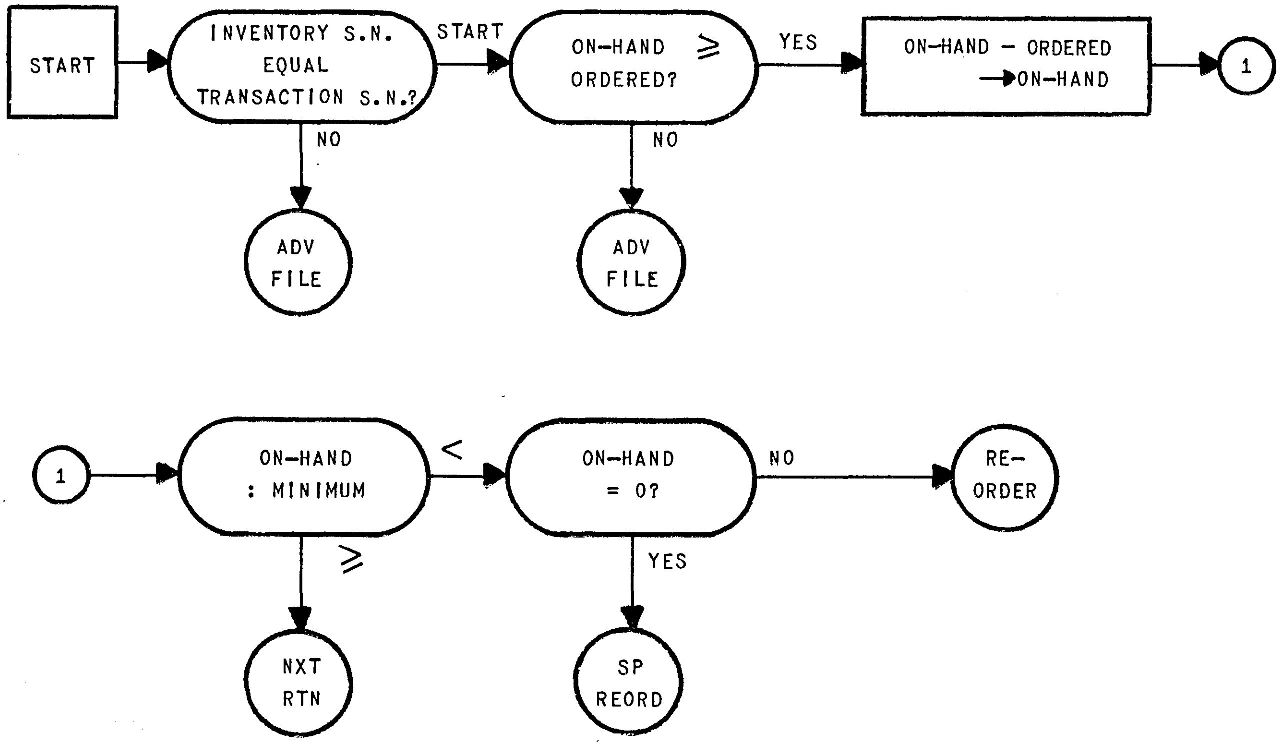
Path "b" of switch 4

Path "c" of switch 4

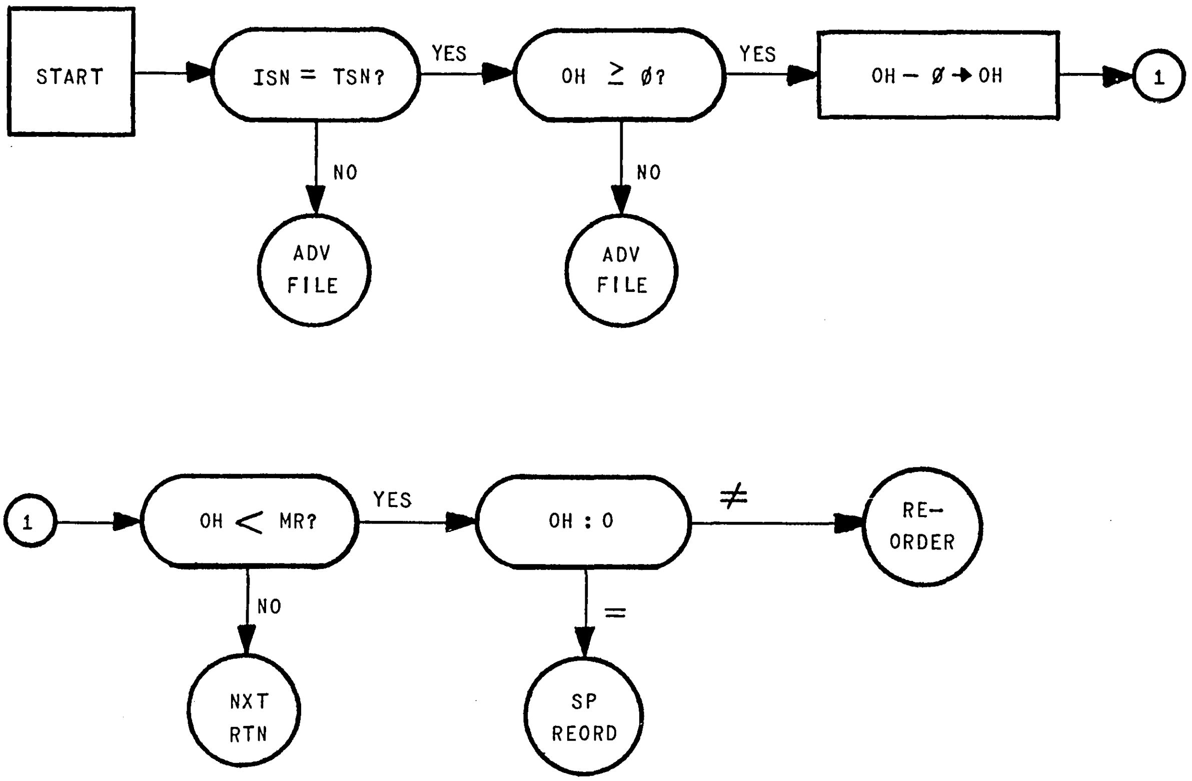
Example

The example in Chapter III, page 58, is used to illustrate two ways of flowcharting.

1.



2.



Key:

ISN	=	Inventory stock number
TSN	=	Transaction stock number
OH	=	On-hand
Ø	=	Ordered
MR	=	Minimum required

From this it should be evident that the coding operation can be greatly simplified if the problem is analysed and defined in a flowchart.

It should also be evident that the first flowchart is easier to read but takes more space. The programmer may draw a flowchart as detailed as the coding. Although this produces the largest flowchart, the detail simplifies the coding. When flowcharting then, the programmer should strive for a compromise that suits his own interests.

Exercise 1

Flowchart the exercise in Chapter III, page 59.

Exercise 2

Flowchart and code a payroll operation which employs the following data:

Memory Location	Contents
3840	Employee Number
3841	Type of Record Code
3842	Dollar Amount
3968	Employee Number
3969	Bond Deduction Code
3970	Bond Accumulation
3971	Company Store Balance

Memory locations 3840, 3841, and 3842 contain data which refers to an employee's weekly paycheck. The remaining data refers to an employee's permanent record.

a. Determine if the paycheck data applies to the permanent record. If it doesn't, go to NXTMAN.

b. If both records refer to the same employee, determine if the type of record code equals the bond deduction code. If the codes are not equal, the dollar amount is a company store payment.

c. On the basis of the above determination, adjust the appropriate accumulation in the permanent record; add to the bond total or subtract from the store balance.

d. Determine if the employee can purchase an $18.75 bond or has eliminated his store balance, whichever is appropriate.

If the bond total equals or exceeds $18.75, deduct this amount and go to BOND. Go to NXTMAN after the last step.

(Assume that the amount, $18.75, is stored in memory location 3839.)

CHAPTER V

PHILCO 2000 ARITHMETIC AND CONSTANTS

REPRESENTING NUMBERS AND DATA

The basic operations of data processing and the associated computer instructions were discussed in Chapter III. This chapter attempts to give the reader a more complete understanding of PHILCO 2000 arithmetic. It also describes the Translator-Assembler-Compiler (TAC) method of representing constants.

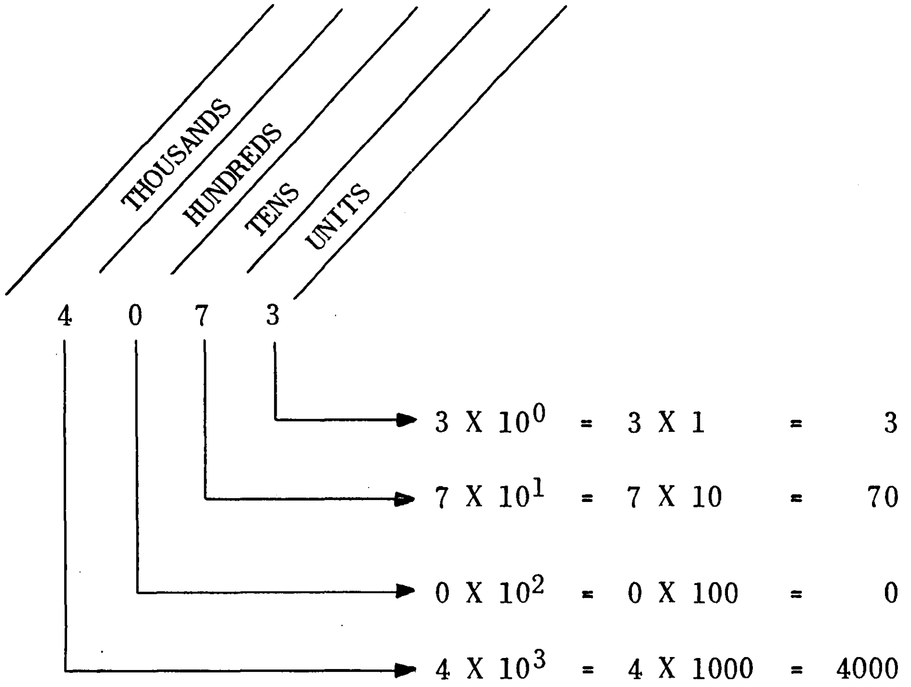
The PHILCO 2000, like most present day computers, uses the binary digit (bit) as the basic unit of information. The reason for this is that the two digits of the binary number system are easier to represent and use electronically than are the ten digits of the decimal system. However, since people normally use the decimal number system, some method or device must be provided to enable the programmer and the computer to communicate in a common language. The computer's method of accomplishing this is the Translator-Assembler-Compiler.

In most cases it will be sufficient for the programmer to think and write in English-decimal terms. In some cases, however, such as scaling numbers, shifting, extracting, and in certain programming techniques, a knowledge of binary representation and arithmetic is necessary.

Decimal Number System

Before discussing binary numbers and arithmetic, the decimal number system will be briefly reviewed. This is a positional number system of base ten in which a digit can have one of ten possible values from zero to nine: 0, 1, 2, ..., 9, and in which a digit position determines an associated power of ten.

For example the decimal number, 4073, is described as follows:



$$3 \text{ X } 10^0 = 3 \text{ X } 1 = 3$$

$$7 \text{ X } 10^1 = 7 \text{ X } 10 = 70$$

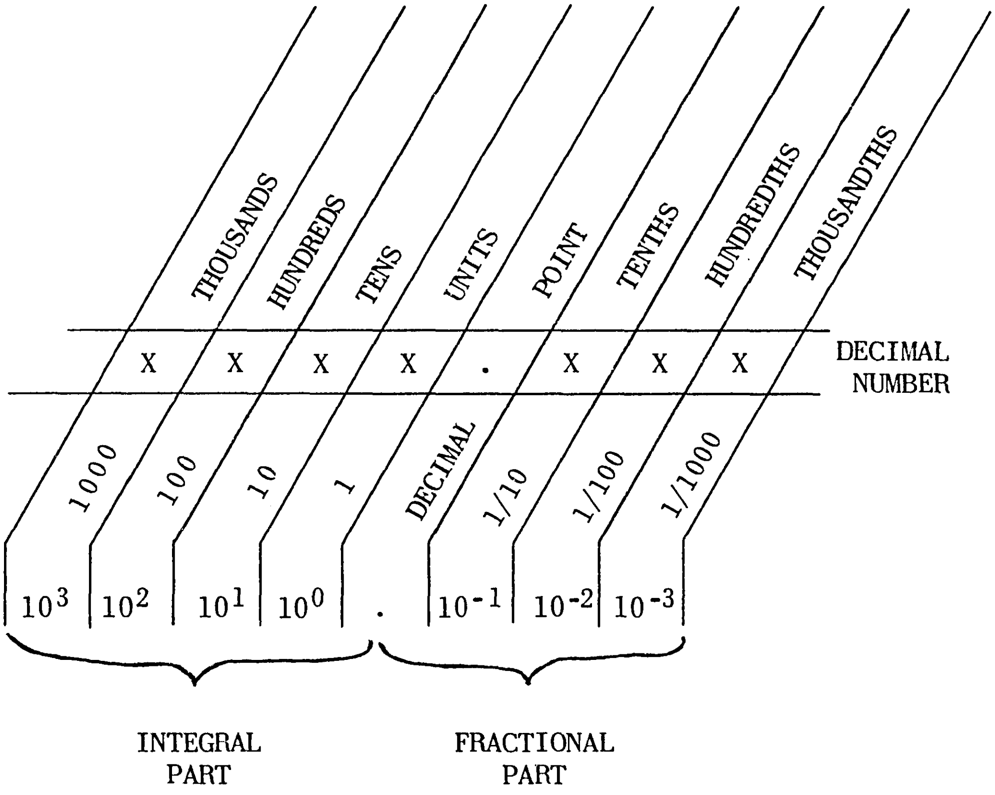
$$0 \text{ X } 10^2 = 0 \text{ X } 100 = 0$$

$$4 \text{ X } 10^3 = 4 \text{ X } 1000 = 4000$$

and the value of the decimal number = the sum of the products = 4073.

Each digit position has a value equal to the product of the digit appearing in the position and a corresponding power of ten. To the left of the decimal point powers of ten are of ascending order; to the right of the decimal point powers of ten are of descending order.

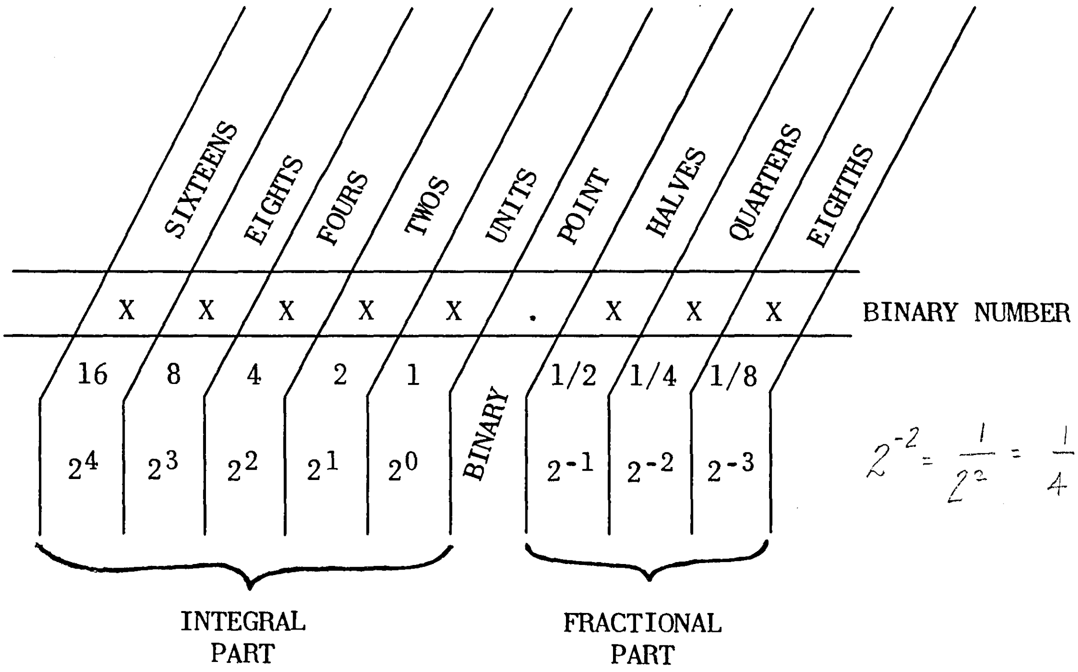
Thus the positional weights of a decimal number are represented as follows:



Regardless of the base in a positional number system, each number can be considered as the sum of the products obtained by multiplying the digits by the corresponding powers of the base.

Binary Number System

In the binary number system the base is two, and digits can have one of two values: zero or one. The positional weights (powers of two) of a binary number can be represented as follows:



$2^{-2} = \frac{1}{2^2} = \frac{1}{4}$

From this it can be seen that to the left of the binary point powers of two are of ascending order; to the right of the binary point powers of two are of descending order.

Decimal-Binary Equivalents

The decimal equivalent of the binary number, 101101, for example, is determined as follows:

1 0 1 1 0 1

→1 X 2^0 = 1 X 1 = 1

→0 X 2^1 = 0 X 2 = 0

→1 X 2^2 = 1 X 4 = 4

→1 X 2^3 = 1 X 8 = 8

→0 X 2^4 = 0 X 16 = 0

→1 X 2^5 = 1 X 32 = 32

value of the binary number = sum of the products = 45.

Correspondingly, the binary fractional number, .1011, has the equivalent decimal value, .6875, and is determined as follows:

```
                                                  .1  0  1  1
.5     =  1 X .5     =  1 X 1/2  = 1 X 2^-1  <----'  |  |  |
.0     =  0 X .25    =  0 X 1/4  = 0 X 2^-2  <-------'  |  |
.125   =  1 X .125   =  1 X 1/8  = 1 X 2^-3  <----------'  |
.0625  =  1 X .0625  =  1 X 1/16 = 1 X 2^-4  <-------------'
.6875  =  sum of the
          products   = value of the binary number.
```

When these results are combined, the decimal equivalent of the binary number, 101101.1011, is 45.6875.

The most frequent necessity for reading binary information is from the operator's console, since the contents of the computer's registers are displayed in binary form. This reading can be simplified by grouping every three bits and assigning to them their equivalent ~~decimal~~ octal value. It should be noted that every three bits can have a value from 0 to 7.

Weights		Weights		Weights	
4 2 1		4 2 1		4 2 1	
0 0 0 =	0	0 1 1 =	3	1 1 0 =	6
0 0 1 =	1	1 0 0 =	4	1 1 1 =	7
0 1 0 =	2	1 0 1 =	5		

When this grouping is completed, the resulting number is then an octal number. For example, the number, 110111101011010, separated into groups of three bits, could be written in octal notation as follows:

```
        1 1 0 1 1 1 1 0 1 0 1 1 0 1 0

   1 1 0   1 1 1   1 0 1   0 1 1   0 1 0
     6       7       5       3       2
```

When octal notation is used, therefore, the reading and writing of binary numbers is simplified.

Since $2^3 = 8$, which is the base of the octal system, any grouping of 3 binary digits can be directly translated into its octal equivalent.

Binary Representation of Data

Source data usually appears in English-decimal form. To be intelligible to most computers, such data must be in some binary form. Many computers, including the PHILCO 2000, represent each character by six bits called a binary-coded character, BCC. (Refer to the chart in Chapter III, page 58, for the representation of every binary-coded character.)

The following diagram illustrates binary-coded characters, and then a binary-coded number is compared to a pure binary number:

0	1	2	3	4	5	7	8	9
000000	000001	000010	000011	000100	000101	000111	001000	001001

The pure binary representation of the binary coded number 12345789 is

101111000110000110111101.

Some computers perform arithmetic with binary-coded numbers; however, the 2000 and others perform arithmetic only on pure binary numbers. Each type of representation has its own merits, with the advantages of speed of operation and compactness of numbers going to the binary computer. For example, a 48-bit PHILCO 2000 number is the equivalent of decimal digits which would require 84 or 90 bits to represent in binary-coded form.

binary arithmetic

Whenever it is necessary to determine the number of bits needed to represent a decimal number, Appendix B (Binary and Decimal Equivalents) can be used. For example, the number of bits necessary to represent any three decimal digits is 10, although the same number of bits will represent a decimal number up to 1023.

not justi

100_{10} = 144_8 = 001 100 100 (9 bits)

511_{10} = 777_8 = $\underbrace{111\ 111\ 111}_{9\text{ bits}}$ 1023_{10} = 1777_8 = $\underbrace{1\ 111\ 111\ 111}_{10\text{ bits}}$

Number System Conversion

Occasionally it is desirable to manually convert a number from one system to another, e.g., to find the binary representation of a decimal number or the decimal representation of a binary number. These conversions may be performed as follows:

Decimal to Binary - Integral Numbers

a. Divide the decimal number by two; the remainder will either be one or zero.

b. If the remainder is one, the least significant digit of the binary number is one.
If the remainder is zero, the least significant digit is zero.

c. Divide the quotient by two.

BCC is a 6-digit code

BCD is a 4-digit code

d. If the remainder is one the next to the least significant bit is one.
If the remainder is zero, the bit is zero.
(Note: Bits should be written from right to left.)

e. Repeat the above steps until a quotient becomes zero.

This process of successive division and recording of remainders may be used to convert any integral number from a higher to a lower base.

The decimal number, 76, is converted to binary as follows:

decimal to binary

Successive Stages
of Equivalent
Binary Number

$$\begin{array}{r} 38 \\ 2\overline{)76} \end{array} \quad \text{Remainder} \longrightarrow \underline{0}$$

$$\begin{array}{r} 19 \\ 2\overline{)38} \end{array} \quad \text{Remainder} \longrightarrow \underline{0}\ 0$$

$$\begin{array}{r} 9 \\ 2\overline{)19} \end{array} \quad \text{Remainder} \longrightarrow \underline{1}\ 0\ 0$$

$$\begin{array}{r} 4 \\ 2\overline{)9} \end{array} \quad \text{Remainder} \longrightarrow \underline{1}\ 1\ 0\ 0$$

$$\begin{array}{r} 2 \\ 2\overline{)4} \end{array} \quad \text{Remainder} \longrightarrow \underline{0}\ 1\ 1\ 0\ 0$$

$$\begin{array}{r} 1 \\ 2\overline{)2} \end{array} \quad \text{Remainder} \longrightarrow \underline{0}\ 0\ 1\ 1\ 0\ 0$$

$$\begin{array}{r} 0 \\ 2\overline{)1} \end{array} \quad \text{Remainder} \longrightarrow \underline{1}\ 0\ 0\ 1\ 1\ 0\ 0 \quad = \quad 76$$

By dividing by eight to convert to octal first, fewer divisions are performed. The above conversion may be performed as follows:

decimal to octal

Successive Stages
of Equivalent
Octal Number

$$\begin{array}{r} 9 \\ 8\overline{)76} \end{array} \quad \text{Remainder} \longrightarrow 4$$

$$\begin{array}{r} 1 \\ 8\overline{)9} \end{array} \quad \text{Remainder} \longrightarrow 1\ 4$$

$$\begin{array}{r} 0 \\ 8\overline{)1} \end{array} \quad \text{Remainder} \longrightarrow 1\ 1\ 4$$

octal to decimal (integers only)

$765_8 = 501_{10}$

```
   7 | 8
  62 | 6
 501 | 5
```

```
   7
 × 8
  56
 + 6
  62
 × 8
 496
 + 5
 501
```

The binary equivalent of the octal number is

octal to binary

Octal	1	1	4
Binary	0 0 1	0 0 1	1 0 0

<u>Decimal to Binary - Fractional Numbers</u>

```
  0.25
     2
  0.50
     2
  1.00
     2
  0.00
.010 = .25₁₀
```

a. Double the decimal number. (multiply by 2)

b. If the product is greater than one, record a one and delete the integral part of the product. If the product is less than one, record a zero. (The bits are written from left to right, i.e., away from the binary point.)

c. Repeat a and b until a resulting product equals one exactly or until the desired length of the binary number is attained.

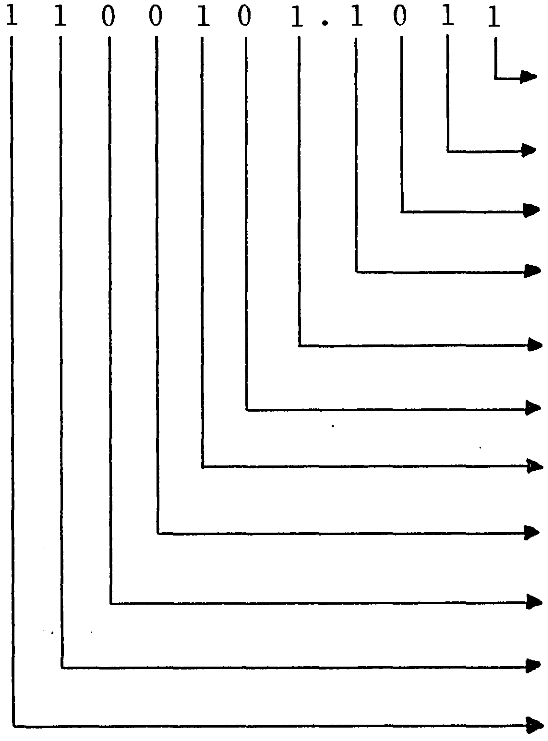
The fractional decimal number, .6875, is converted to binary as follows:

	Decimal Number	Successive Stages of Equivalent Binary Number	
	0.6875 × 2		
Double	1.3750	.1	
Delete 1	0.3750 × 2		
Double	0.7500 × 2	.10	
Double	1.5000	.101	
Delete 1	0.5000 × 2		
Double	1.0000	.1011	Equivalent binary number
Delete 1	0.0000	and all further bits are zero.	

This method can be generalized to convert any fractional number from a higher to a lower base, i.e., decimal (base 10) to octal (base 8).

Binary to Decimal. To convert from a number of a base less than ten to its decimal equivalent, simply find the sum of the products of each digit by the corresponding power of the base.

The decimal equivalent of the binary number, 1100101.1011, is determined as follows:



1 X .0625 = 0.0625
1 X .125 = 0.125
0 X .25 = 0.0
1 X .5 = 0.5
1 X 1 = 1.0
0 X 2 = 0.0
1 X 4 = 4.0
0 X 8 = 0.0
0 X 16 = 0.0
1 X 32 = 32.0
1 X 64 = 64.0

Decimal equivalent = sum of the products = 101.6875

(The reader should verify for himself that this is the correct number.)

BINARY ARITHMETIC

Rules of Binary Arithmetic

Since the binary number system uses only two digits the rules of binary arithmetic are quite simple. These rules are shown below.

Addition:

$$\begin{array}{r} 0\\ +1\\ \hline 1 \end{array}\qquad \begin{array}{r} 1\\ +0\\ \hline 1 \end{array}\qquad \begin{array}{r} 1\\ +1\\ \hline 1 \curvearrowleft 0 \end{array}\qquad \begin{array}{r} 0\\ +0\\ \hline 0 \end{array}$$

(Carry 1)

(Borrow 1)

Subtraction:

$$\begin{array}{r} 1 \curvearrowright 0\\ -1\\ \hline 1 \end{array}\qquad \begin{array}{r} 1\\ -0\\ \hline 1 \end{array}\qquad \begin{array}{r} 1\\ -1\\ \hline 0 \end{array}\qquad \begin{array}{r} 0\\ -0\\ \hline 0 \end{array}$$

Multiplication:

$$\begin{array}{r} 0\\ \text{x}1\\ \hline 0 \end{array}\qquad \begin{array}{r} 1\\ \text{x}0\\ \hline 0 \end{array}\qquad \begin{array}{r} 1\\ \text{x}1\\ \hline 1 \end{array}\qquad \begin{array}{r} 0\\ \text{x}0\\ \hline 0 \end{array}$$

Division:

$$\frac{0}{0} = \text{Not defined}\qquad \frac{1}{0} = \text{Not defined}\qquad \frac{1}{1} = 1\qquad \frac{0}{1} = 0$$

Note: "Carry 1" means that a one is carried to the next (left) bit position. "Borrow 1" means that a one is borrowed from the next (left) bit position.

Examples of Binary Arithmetic - Positive Numbers

Addition:

$$\begin{array}{r} 5\\ +3\\ \hline 8 \end{array}\qquad \begin{array}{rl} \underline{111} & \text{Carries}\\ 101 & \\ +011 & \\ \cline{1-1} 1000 & \end{array}$$

$$\begin{array}{r} 29\\ +22\\ \hline 51 \end{array}\qquad \begin{array}{rl} \underline{111} & \text{Carries}\\ 11101 & \\ +10110 & \\ \cline{1-1} 110011 & \end{array}$$

$$\begin{array}{r} 4.75000\\ +3.59375\\ \hline 8.34375 \end{array}\qquad \begin{array}{rl} \underline{1111} & \text{Carries}\\ 100.11000 & \\ +\ 11.10011 & \\ \cline{1-1} 1000.01011 & \end{array}$$

Subtraction:

```
         15                1111
         -8               -1000
         --               -----
          7                0111

         54.5390625        110110.1000101
        -19.0234375        -10011.0000011
        -----------        --------------
         35.5156250        100011.1000010
```

Multiplication:

```
          5                 101
         x3                 x11
         --                 ---
         15                 101
                           101
                           ----
                           1111

           4.750            100.110
          x3.625          x  11.101
          ------          ---------
           23750            100110
           9500            1001100
          28500           100110
         14250           100110
        ---------       -----------
        17.218750        10001.001110
```

Division:

```
            2                      10
       4 ) 8              100 ) 1000
                                100
                                ----
                                0000

            2.125                  10.001
       5 ) 10.625         101 ) 1010.101
                                101
                                ------
                                  00 101
                                     101
                                     ---
                                     000
```

Complements in Arithmetic

An understanding of complements in arithmetic is necessary to the understanding of PHILCO 2000 arithmetic. The complement of a given number is defined as the difference between the given number and the base of the number system raised to an appropriate power. The appropriate power (exponent) is equal to the number of digits necessary to represent the given number. It may also be defined as that number which produces a one (followed by zero, or zeros) when added to the given number.

For example:

Given the decimal number 4

Its ten's complement is $\begin{array}{r}\underline{+6}\\10\end{array}$ because $10^1 - 4 = 6$

Given the decimal number 136

Its ten's complement is $\begin{array}{r}\underline{+864}\\1000\end{array}$ because 1000, i.e., $10^3 - 136 = 864$

Given the binary number 101

Its two's complement is $\begin{array}{r}\underline{+011}\\1000\end{array}$ because $1000 - 101 = 011$.

These complements are named from the bases of their respective number systems. By using complements, <u>all</u> arithmetic in a computer may be performed by addition. Subtraction, for example, would be performed as follows:

$$\begin{array}{r}7\\\underline{-4}\\3\end{array} \equiv \begin{array}{rrl} & 7 & \\ & \underline{+6} & \text{(Ten's complement of 4)}\\ 1 \hookleftarrow & \boxed{3} & \end{array}$$

The last carry is ignored, because $7 - 4 = 7 + (10-4)\underline{-10}$ to yield the answer of three.

(Note: The symbol, $\equiv$, means "is equivalent to.")

Examples

$$\begin{array}{r}469\\\underline{-237}\\232\end{array} \equiv \begin{array}{rrl} & 469 & \\ & \underline{+763} & \text{(ten's complement of 237)}\\ 1 \hookleftarrow & \boxed{232} & \end{array}$$

"overflow" is ignored

and

$$\begin{array}{rcr}26 & = & 11010\\\underline{-14} & = & \underline{-01110}\\12 & = & 01100\end{array} \equiv \begin{array}{rrl} & 11010 & \\ & \underline{+10010} & \text{(two's complement of 01110)}\\ 1 & \boxed{01100} & \end{array}$$

A simple method of obtaining the two's complement of a binary number is to change all zeros to ones and all ones to zeros and then add one to the rightmost bit. The number obtained before the addition of one is called the one's complement.

Example

```
Given the number                 1011010 (=90)      132₈ [illegible]

Its one's complement is          0100101
                                     + 1
                                 -------
and its two's complement is      0100110 (=38).     46₈ = 38

Proof:  The given number         1011010 = 90
       +Its two's complement,   +0100110 =+38

Produces a one followed by
                     zeros:     10000000 =128 = 2⁷,    200₈ = 128₁₀

i.e., the next higher power of
the base.
```

Two's Complement Arithmetic

Because of the nature of complements, as explained above, negative binary numbers can be represented in two's complement form. The corresponding arithmetic is called two's complement arithmetic.

To illustrate two's complement arithmetic, six-bit numbers (rather than 48-bit numbers) are used. Minus eight in two's complement form appears as

```
        S|
  + 8 = 0|001000
        1|110111    One's complement
        +|     1
         -------
  - 8 = 1|111000    Two's complement.
```

Similarly for minus three,

```
        S|
  + 3 = 0|000011
        1|111100    One's complement
        +|     1
         -------
  - 3 = 1|111101    Two's complement.
```

In the computer, as in these examples, the first bit is a sign indicator. Positive numbers begin with zero whereas negative numbers begin with one.

Examples of Two's Complement Arithmetic - Negative Numbers

Addition:

```
               S
   15          0|001111
 +(-3)         1|+111101
 -----          --------
   12       1  1|001100
        no carry is required

   14          0|001110
 +(-19)        1|+101101
 -----           -------
   -5          1| 111011
```

```
0 0 11
1 1 00
    11
  1 1 01

0 010 011
1 101 100

1 101 101
```

Proof of the last sum is obtained by complementing the result

```
 111011    Sum
 000100    One's complement of sum
+     1
 ------
 000101    Two's complement
```

and the magnitude of the result is seen to be 5.

Another example is

```
  -3        1|111101
+(-5)       1|+111011
 ----          ------
  -8       1 1|111000
```

One's complementing is the same as subtracting a number from the largest digit of its order - ie 9 −3 = 6 = one's comp of 3

Proof: complementing the sum

```
 000111
+     1
 ------
 001000 = 8
```

Two's complementing is the same as subtracting a number from the smallest digit of the next higher order - ie - 10 −3 = 7 = two's comp of 3

Subtraction is performed by obtaining the two's complement of the number to be subtracted and then performing addition.

Subtraction:

```
   15        001111
   -8       +111000    Two's complement of 8
   --        ------
    7     1 |000111

   15        001111
 -(-3)      +000011    Two's complement of -3
 -----       ------
   18        010010

   -3        111101
 -(-5)      +000101    Two's complement of -5
 -----       ------
    2     1 |000010
```

PHILCO 2000 ARITHMETIC

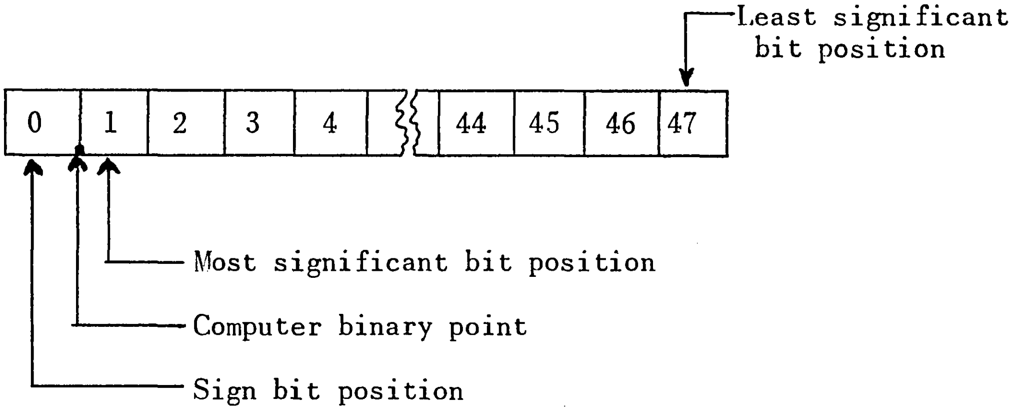
The PHILCO 2000 incorporates the features of two's complement arithmetic to perform all arithmetic by the single process of addition. Subtraction, then, is performed by the addition of the two's complement of the number to be subtracted. Multiplication is a process of addition and shifting; division is a process of addition of two's complements and shifting.

Computer Representation of Numbers

Positive and negative numbers in the computer are represented by 48 bits in the same way as the previous six-bit numbers were. That is, if the sign bit is zero, the 48 bits represent a positive number. If the sign bit is one, the 48 bits represent a negative number in two's complement form.

The arithmetic section assumes that all numbers are fractional with the binary point positioned between the sign and the most significant bit positions. However, as will be shown later, the programmer may assume the binary point is positioned anywhere in relation to the 48 bits.

The following configuration of any number is assumed in the computer:



Thus, the computer's interpretation of the following numbers would be:

010000...0 = 0.10000...0 = 1/2 = .5

001000...0 = 0.01000...0 = 1/4 = .25

011000...0 = 0.11000...0 = 3/4 = .75

010010...0 = 0.10010...0 = 9/16 = .5625

101000...0 = 1.0100 ...0 = -3/4 = -.75

Fractional Arithmetic

Because the 2000 is a fractional computer the differences between fractional number and whole number arithmetic are important. These differences arise from the fact that fractional numbers are aligned at the left rather than the right. In multiplication, the product is generated to the right. For example, .1 x .1 (in either decimal or binary arithmetic) yields .01. As a result, a product is never larger in absolute value than the multiplier or the multiplicand. In division, the divisor must be larger than the dividend so that a fractional quotient can be obtained.

The following examples illustrate fractional arithmetic in the 2000:

Multiplication:

$$\begin{array}{rcr} .5 & & 0.1000...0 \\ \times .5 & & 0.1000...0 \\ \hline .25 & = & 0.0100...0 \end{array}$$

Division:

$$\overset{(A)}{.125} \div \overset{(D)}{.5} = .25 \qquad \frac{0.0010...0}{0.1000...0} = 0.0100...0$$

Since the binary point precedes the number, the largest possible computer number is less than plus one and is represented by a zero in the sign bit position and ones in the remaining 47 positions.

0	1	2	3		44	45	46	47
0	1	1	1		1	1	1	1

This number is equivalent to .99999999999999.... which is very close to but not equal to one.

The smallest computer number is minus one and is represented by a one in the sign position followed by all zeros.

0	1	2	3		44	45	46	47
1	0	0	0		0	0	0	0

$-1 \leq n < +1$
where n is any
computer number

(This is the one's complement of the largest positive number.)

Therefore, within the PHILCO 2000, all numbers resulting from arithmetic operations must be within the following range:

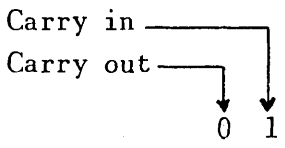
a. They must be less than plus one.

b. They must be greater than or equal to minus one.

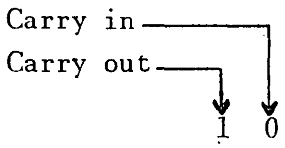
Any results, which would be outside this range, produce overflow.

Overflow

When overflow occurs, it usually signifies that an invalid result has been formed. However, it may be used as a control for such operations as double precision arithmetic and counting. Some examples of addition causing overflow follow:



```
  0.10000...0  =  1/2
+ 0.10000...0  =  1/2
-------------     ---
  1.00000...0  ≠  1
```



```
  1.00000...0  =  -1
+ 1.10000...0  =  -1/2
-------------     ------
  0.10000...0  ≠  -1 1/2
```

Note that overflow changes the sign but that all other positions in the sum are correct. Another definition of overflow is that the carry into the sign position is not the same as the carry out of the sign position (no carry is a carry of zero). The occurrence of overflow in the computer's arithmetic operations is summarized below.

Addition and Subtraction:	Overflow occurs if a result equals or exceeds a computer value of plus one or is less than a minus one.
Multiplication: -1.0×-1.0 should $= +1.0$	Overflow occurs if a computer value of minus one is multiplied by minus one. This should yield plus one. After such a multiplication, the product reappears as minus one. Addition or subtraction in a MAD or MSU instruction may bring the plus one back into representable range. Overflow, however, will still be indicated. The addition or subtraction in a MAD or MSU instruction may also cause overflow, even if the product is a representable value.
Division: $|A, AQ| \geq |D|$ ie $\frac{0100}{0010}$	If the dividend is larger than the divisor (in absolute value), potential overflow is detected and division is not performed. Instead, the A and Q Registers are altered as follows and the next instruction is selected.

if $|(A)|$ or $|(AQ)| > |(D)|$

Register \ Type of Division	Single Length	Double Length
A Register	Shifted numerically one place to the right.	Shifted numerically one place to the right (into Q).
Q Register	Cleared to zero.	Shifted numerically one place to the right.

If the store option were used with the instruction, the contents of Q would be stored in memory, replacing the divisor.

if $|(A)$ or $AQ)| = |(D)|$

The following table indicates the results if the dividend is equal to the divisor (in absolute value):

A

D

Sign of Dividend	+	+	-	-
Sign of Divisor	+	-	+	-
Potential Overflow	Yes	Yes	No	No
Quotient	No division. Dividend (A or AQ) is shifted one place to the right.		-1	One's complement of -1 (011....111)
Remainder			Two's complement of the divisor	Equal to divisor

Detection of Overflow

For the detection of overflow, the programmer must consider the following:

a. the overflow indicator
b. the overflow instructions
c. the overflow switch on the control panel of the console

Overflow Indicator

The overflow indicator has two states or conditions, one and zero. "One" is indicated by a neon light on the panel being on; for "zero," the light is off. Normally, the indicator is automatically cleared to zero before each Arithmetic instruction, Shift instruction, or four of the Index Register instructions (AIXOL, AIXOR, SIXOL, SIXOR). It is automatically set to one each time overflow occurs.

The programmer may test the indicator to see if overflow has occurred with the JOF or JNO instructions, as long as no instruction which would clear the indicator occurs between the time overflow occurred and the JOF or JNO instruction.

JOF: Jump if Overflow is indicated.

Jump to the location specified by the address portion of the instruction if the overflow indicator is set to one. If the overflow indicator is zero, proceed to the next instruction. THE overflow, if set, is always cleared.

JNO: Jump if No Overflow is indicated.

Jump to the location specified by the address portion of the instruction if the overflow indicator is set to zero. Proceed to the next instruction if the overflow indicator is one. The overflow, if set, is always cleared.

Overflow Instructions

The clearing of the indicator by the Arithmetic, Shift, and Index Register instructions may be inhibited by the ICOS instruction until a convenient time for testing the indicator occurs.

ICOS Inhibit Clearing the Overflow indicator

This instruction clears the overflow indicator and then inhibits its future clearing by Arithmetic, Shift, or Index Register instructions.

This inhibition on the clearing of the overflow indicator may be removed only by the ICOZ instruction.

ICOZ Remove Inhibition on Clearing the Overflow indicator

This instruction removes any inhibition on clearing the overflow indicator set by the ICOS instruction.

Overflow Switch

The overflow switch on the console is an on-off switch which can be used to stop the computer when overflow occurs. It has effect only if there is no inhibition on the clearing of the overflow indicator. When it is set to ON, the computer will stop if overflow has occurred, and no inhibition was set by an ICOS instruction and the next instruction is not a jump overflow instruction. ?

The operations which are automatically performed by the computer when overflow occurs are as follows:

a. The overflow indicator is set.

b. If there is no inhibition on clearing the overflow indicator, the overflow switch is examined.

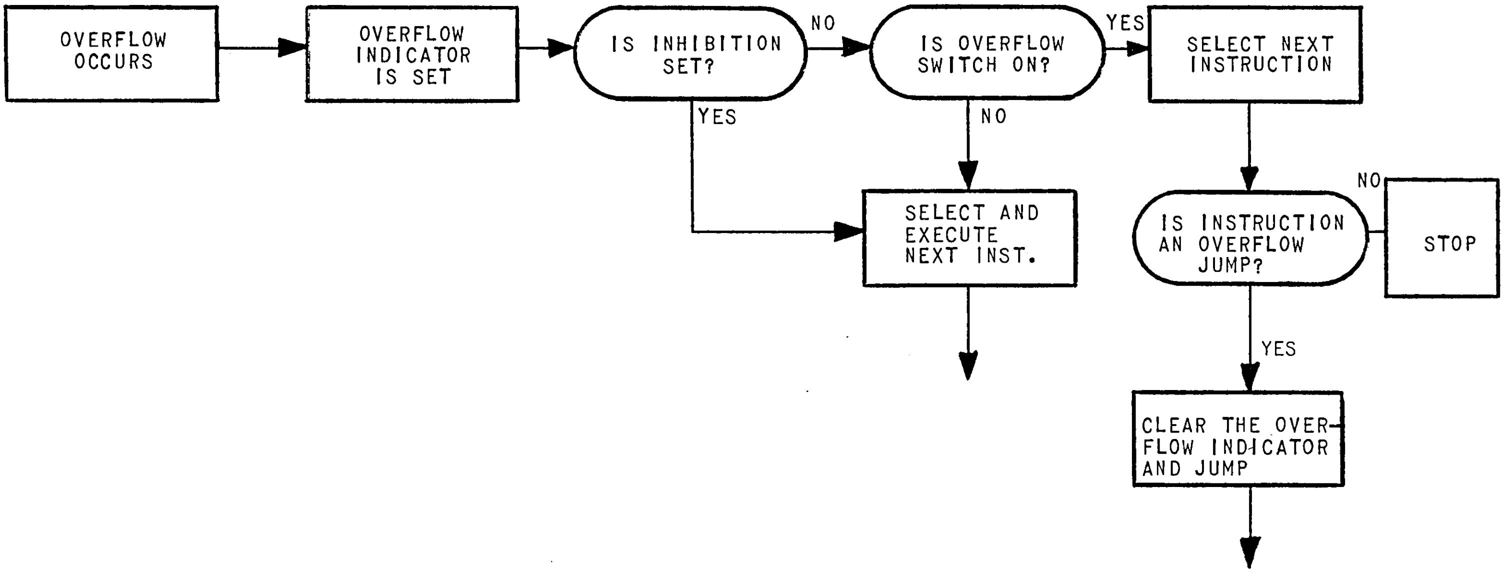
1. If the switch is off, control proceeds to the next instruction.

2. If the switch is on, the next instruction is selected and examined.

   a) If the next instruction is an overflow Jump instruction, it is executed.

   b) If the next instruction is not an overflow Jump instruction, the computer stops with the overflow neon lighted.

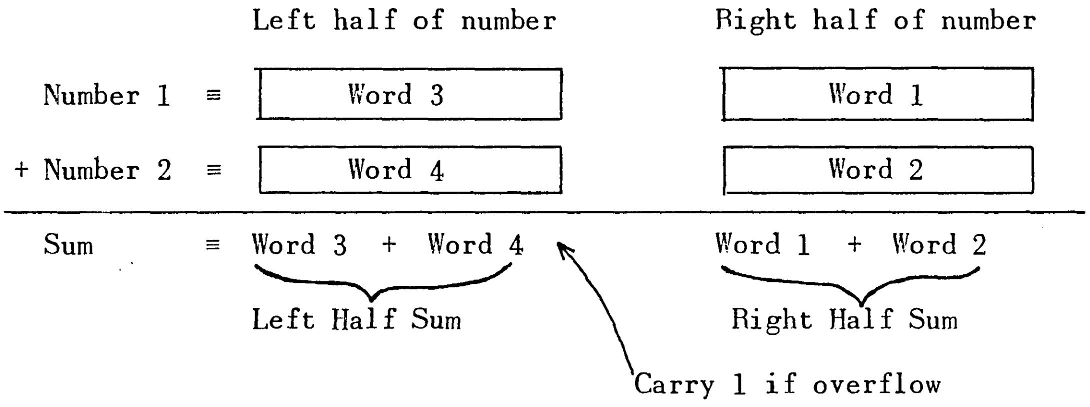
A flowchart of these steps is shown below.


OVERFLOW OCCURS
OVERFLOW INDICATOR IS SET
IS INHIBITION SET?
NO
YES
IS OVERFLOW SWITCH ON?
YES
NO
SELECT NEXT INSTRUCTION
SELECT AND EXECUTE NEXT INST.
IS INSTRUCTION AN OVERFLOW JUMP?
NO
STOP
YES
CLEAR THE OVERFLOW INDICATOR AND JUMP

Use of Overflow

Overflow may be expected and used as a control, or it may be unexpected, i.e., caused by an error in the data or in scaling the numbers. (Scaling is described in the next section of this chapter.)

When overflow is used as a control, the overflow switch should be in the off position. Overflow is used as a control in double precision addition and subtraction in which two words are used to represent a number. To add (or subtract) two numbers, corresponding halves are added (or subtracted). When overflow occurs in the addition of the right half words, a carry must be added to the addition of the left half words.



Example

The coding to perform double precision addition with positive numbers is shown below; it is assumed that the overflow switch is off. The numbers and their corresponding memory locations are as follows:

Memory Location	Contents
3968	Right Half Number 1
3969	Left Half Number 1
3970	Right Half Number 2
3971	Left Half Number 2
3972	Right Half Sum
3973	Left Half Sum
3974	A One in the Sign Position
3975	A One as the Least Significant Digit

The number in 3974 is needed to correct the sign position if overflow has occurred, and the number in 3975 is used for the carry of one.

L	LOCATION	COMMAND	ADDRESS AND REMARKS	
		TMA	3968$	Øbtain the Right Half Sum
		AM	3970$	
		JNØ	NØFLØW$	Jump if no carry to Left Half Sum
		AM	3974$	Correct the sign of Right Half Sum
		TAM	3972$	Store Right Half Sum
		TMA	3975$	Transfer "Carry" ⟶ A
		AM	3969$	LH
	ADD	AM	3971$	Øbtain and store Left Half Sum
		TAM	3973$	2 LH
		JMP	NXTRTN$	
	NØFLØW	TAM	3972$	Store Right Half Sum
		TMA	3969$	Prepare to obtain Left Half Sum
		JMP	ADD$	

Overflow is also used as a control in counting. If an operation is to be performed a number of times, a sum can be kept which will overflow when the desired number of operations has been performed. This is an infrequent use for overflow.

Overflow is more often used to detect errors in data, or in scaling. (For present purposes an error in scaling can be taken to mean that a result became larger or smaller than was assumed possible or that numbers being added or subtracted were placed too close to the sign position.) When overflow occurs in either of these cases, two possible courses of action exist: The computer can be made to stop, or the programmer can "program around" the overflow.

The computer will stop whenever the overflow switch is on and an overflow Jump instruction doesn't immediately follow the overflow. When the computer stops the operator can examine the instructions being executed and the operands which caused the overflow.

The ICOS instruction can be used when the overflow indicator is to be examined after groups of instructions. The purpose of doing this is to guarantee the detection of overflow without using an overflow Jump instruction after each arithmetic operation.

Example

Assume that a series of arithmetic operations are to be performed. Overflow may occur anywhere in the computation and it is undesirable to place an overflow Jump instruction after each Arithmetic instruction. One way of avoiding this is to execute an ICOS instruction just before the computation begins and an overflow Jump instruction after the computation ends. Prior to running the program the overflow switch should be off.

If overflow occurs, the overflow Jump instruction should direct the processing to a part of the program which tells the operator, by printing on the Console Typewriter, to set the overflow switch and which then stops the computer. After setting the switch, the operator depresses the advance bar. Then an ICOZ instruction and a jump to the start of the computation is executed. When the computation is repeated, overflow recurs and the computer stops at the point of overflow. The operator thus can determine where in the program the overflow occurred and why it occurred.

A skeleton form of the coding for this procedure is shown below. Note that in this case the Jump instruction signifies overflow by not jumping. An instruction which cannot cause overflow precedes JNO because the computer will not stop if overflow occurs immediately preceding an overflow Jump instruction.

L	LOCATION	COMMAND	ADDRESS AND REMARKS	
		ICØS		Inhibit clearing of overflow
	START	.	.	
		.	.	The series of computations in
		.	.	which overflow can occur
		.	.	
		.	.	
				An instruction which cannot cause overflow
		JNØ	NXTRTN$	If no overflow, jump to continue the program
		.	.	
		.	.	"Set overflow switch" ⟶ Console
		.	.	Typewriter
		.	.	
		HLT		Stop to allow operator action
		ICØZ		Remove inhibition on clearing overflow indicator
		JMP	START$	Jump to repeat the computation

The next two examples illustrate programming around overflow.

Example

The coding which produces a quotient even if overflow occurs is shown below. If potential overflow is detected, the Divide instruction is repeated after the dividend has been automatically shifted one position right. The dividend is assumed to be in memory location 3968 and the divisor in 3969.

L	LOCATION	COMMAND	ADDRESS AND REMARKS	
		TMA	3968$	Dividend → A
	DIVIDE	DA	3969$	Divide[shifts(A)Reg.R one bit if overflow detected]
		JØF	DIVIDE$	Jump to DIVIDE if the overflow
		⋮	⋮	indicator is set to one.

Additional coding to count the number of shifts of the dividend is required. This count indicates the number of positions necessary to shift the quotient left to maintain the same scale factor. The coding will be illustrated in Chapter VIII under the subject of Index Registers.

Example

If overflow occurs due to an error in the data, it is desirable to detect it and jump to a part of the program to correct the error. For example, the coding below will jump to CORRECT if (3968) plus (3969) cause overflow.

L	LOCATION	COMMAND	ADDRESS AND REMARKS	
		TMA	3968$	
		AM	3969$	
		JØF	CØRRECT$	

Additional Features of PHILCO 2000 Arithmetic

Multiplication. In multiplication, rounding is accomplished by adding one to the most significant bit of the Q Register which contains the minor half of a double length product. The original contents of Q are then restored. The product in multiplication will be negative if the signs of the two operands are different. It will be positive if the signs are the same. In unrounded multiplication, the sign bits in the A and Q Registers will be the same.

Division. The results of division, when no overflow is detected, are shown on the following table:

DIVIDEND IS:	POSITIVE		NEGATIVE		POSITIVE		NEGATIVE	
DIVISOR IS:	POSITIVE		NEGATIVE		NEGATIVE		POSITIVE	
IF DIVISION IS NOT EXACT								
	RESULTS		RESULTS		RESULTS		RESULTS	
	SIGN	VALUE	SIGN	VALUE	SIGN	VALUE	SIGN	VALUE
QUOTIENT IS:	+	True	+	True	-	One's complement	-	One's complement
REMAINDER IS:	+	True	-	True	+	True	-	True
IF DIVISION IS EXACT								
QUOTIENT IS:	+	True	+	One less than the true value in the least significant bit	-	One's complement	-	True (two's complement)
REMAINDER IS:	+	Zero	-	Same as divisor (two's complement)	+	Zero	-	Two's complement of divisor

The table shows that

a. The sign of the remainder is always the same as the sign of the dividend.

b. Negative quotients are normally produced in their one's complement form.

Two special cases arise from these characteristics of division.

a. In cases of exact division involving a negative dividend, a negative remainder of zero cannot be formed, as zero is a positive number. In such a case, the value in the A Register (remainder) is made equal to the two's complement of the absolute value of the divisor. During this process the value of the quotient is made smaller by one in the least significant bit position.

b. The last step noted above causes the quotient in an exact division involving a negative dividend and negative divisor to be one less than its true value. In an exact division involving a negative dividend and a positive divisor, this last step causes the quotient, which would otherwise be in a one's complement form, to be expressed as a two's complement--thereby representing its true value.

Scaling (FIXED POINT ARITHMETIC)

Representing Whole and Mixed Numbers. Despite the arithmetic section's interpretation that all numbers are less than one or greater than or equal to minus one, the programmer is not restricted to this range. He may assume a binary point anywhere within a word or outside of a word. Having made this choice, the programmer must keep track of the assumed point throughout all subsequent arithmetic operations. This process of representing any desired number by selecting an appropriate binary point is called SCALING, and the number of positions between the computer's point and the assumed point is called the SCALE FACTOR. The scale of a number is that power of two which, when multiplied by the computer number, produces the desired number.

For example, to represent the number 4, using the computer number, .5, the scale factor must be 3, i.e.,

The computer number .5 = 0.10000...

with a scale factor of 3 $0.100_{\wedge}00...$

is equal to $.5 \times 2^3 = .5 \times 8 = 4 = 100$,

where the caret, $_\wedge$, indicates the assumed binary point.

In TAC notation this number would be written as

4.0 B3.

B3 means that the position of the assumed binary point is three places to the right of bit position zero. (A complete description of TAC notation will be found in a subsequent section of this chapter.)

A positive scale factor indicates that the assumed point is to the right of bit position zero. A scale factor of zero indicates that the assumed point coincides with the arithmetic section's point. A negative scale factor indicates that the assumed point lies to the left of bit position zero. Note that a negative scale factor must indicate a fractional number because it specifies a binary point which precedes the number. Similarly, except for minus one, a zero scale factor indicates a fractional number. For example:

```
4.5₁₀ = 4.4₈       4.5 B4   =   0.0100‸10...
                  .25 B-1  =  ‸0.100000...
                   .5 B0   =   0.100000...
                                ‸
```

Manipulation of scaled numbers only requires that the programmer keep track of the assumed point which may move or have to be moved for arithmetic operations. How an assumed binary point moves will become evident by reviewing the computer's interpretation of numbers being multiplied and divided.

```
                           PHILCO 2000 Multiplication

        .5                    0.10000...0
  x     .5                    0.10000...0
  ----------               ------------------------
       .25        =        0.01000...0 000000...0

       .75                    0.11000...0
  x    .25                    0.01000...0
  ----------               ------------------------
       375        =        0.00110...0 000000...0
      150
  ----------
     .1875
```

If the numbers of the first example are scaled to represent 4.0 B3 and 2.0 B2, respectively, the multiplication may be considered as

```
       0.100‸000...0
       0.10‸0000...0
     ------------------
   0.01000‸000000...0 = 8.0 B5.
```

That is, $(.5 \times 2^3) \times (.5 \times 2^2) = .25 \times 2^5$.

From this it can be seen that the assumed binary point has now moved to the right to a position five places from bit position zero. This result may be generalized as follows:

a. In multiplication, the scale factor of the product is equal to the sum of the scale factors of the multiplier and the multiplicand.

b. In division, the scale factor of the quotient is equal to the difference between the scale factors of the dividend and the divisor.

For example, consider the division of 1/4 by 1/2. In the PHILCO 2000 this division produces the following result:

$$\left\{\frac{.25}{.5}\right\} = \frac{0.01000\ldots0}{0.10000\ldots0} = 0.10000\ldots0 = .5.$$

If the numbers are scaled so that the dividend represents 8.0 B5 and the divisor represents 2.0 B2, the quotient must be 4.0 B3 or

$$\frac{8.0\ \mathrm{B}5}{2.0\ \mathrm{B}2} = \frac{0.01000_{\wedge}\ldots0}{0.10_{\wedge}000\ldots0} = 0.100_{\wedge}00\ldots0 = 4.0\ \mathrm{B}3.$$

c. When performing addition and subtraction, the scale factors must be the same; that is, the binary points must be aligned. Thus, to add

$$\begin{array}{ccl} 9.0\ \mathrm{B}4 & = & 0.1001_{\wedge}0\ldots0 \\ \text{and} & & \\ 3.5\ \mathrm{B}2 & = & 0.11_{\wedge}100\ldots0 \end{array}$$

the second number must be shifted right so that the assumed point is at B4. That is,

$$\begin{array}{rcl} 9.0\ \mathrm{B}4 & & 0.1001_{\wedge}0\ldots0 \\ +\ 3.5\ \mathrm{B}4 & & 0.0011_{\wedge}10..0 \\ \hline 12.5\ \mathrm{B}4 & = & 0.1100_{\wedge}10..0 \end{array}$$

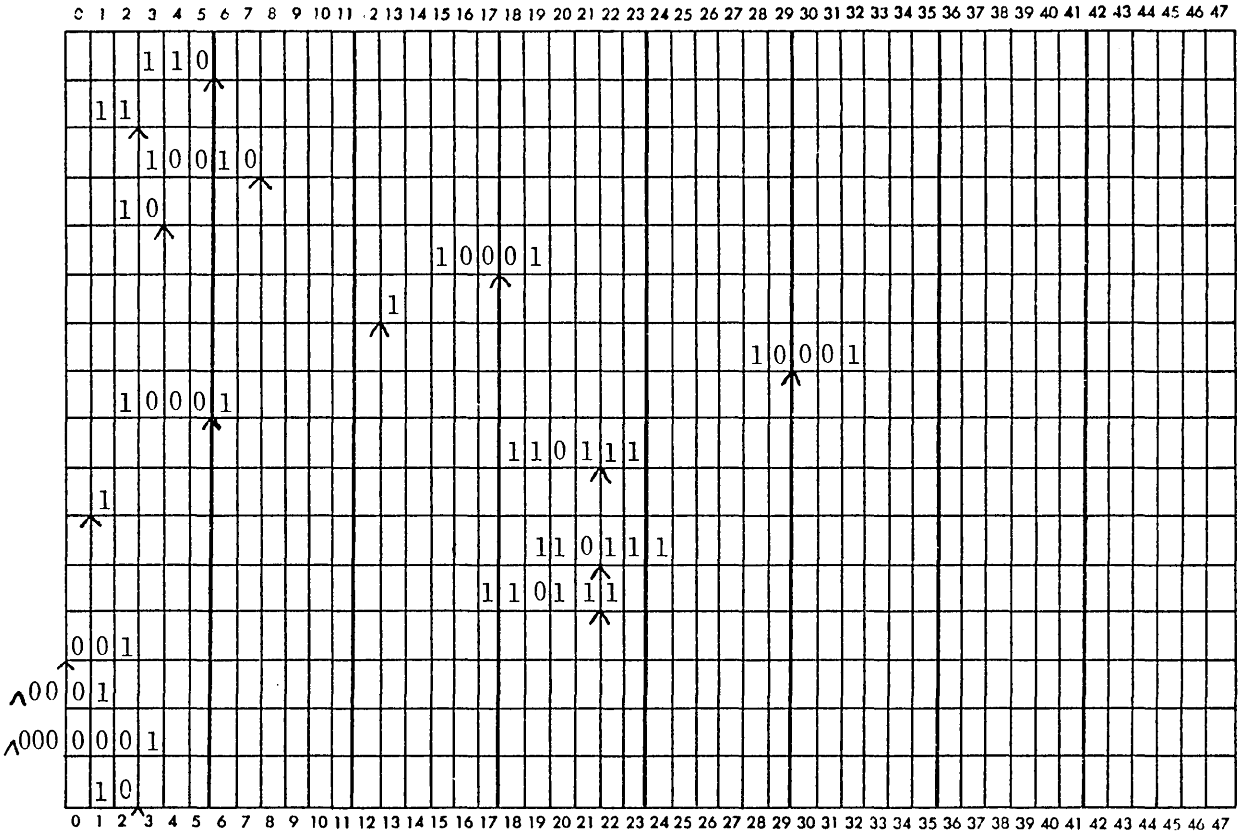
Sometimes it will be convenient to work with whole numbers rather than mixed numbers. For example, instead of dollars and cents, all figures and computations can be expressed in cents. Then the only time that a distinction between dollars and cents need be made is when the numbers are to be printed or punched in cards or paper tape.

Examples of scaling

Number 1 is to be multiplied or divided by Number 2.

Number 1	Number 2	Product	Quotient
6.0 B5	3.0 B2	18.0 B7	2.0 B3
4.25 B17	.5 B12	2.125 B29	8.5 B5
13.75 B21	.5 B0	6.875 B21	27.5 B21
.125 B-1	.0625 B-3	.0078125 B-4	2.0 B2
a Bx	b By	ab B(x+y)	a/b B(x-y)

These numbers would have the following appearance in PHILCO 2000 words. Note: Blank positions represent zeros.



	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47
6.0 B5				1	1	0																																										
3.0 B2		1	1																																													
18.0 B7				1	0	0	1	0																																								
2.0 B3			1	0																																												
4.25 B17																1	0	0	0	1																												
.5 B12														1																																		
2.125 B29																													1	0	0	0	1															
8.5 B5			1	0	0	0	1																																									
13.75 B21																			1	1	0	1	1	1																								
.5 B0		1																																													
6.875 B21																				1	1	0	1	1	1																							
27.5 B21																		1	1	0	1	1	1																									
.125 B-1	0	0	1																																													
.0625 B-3 ∧00	0	1																																														
.0078125 B-4 ∧000	0	0	0	1																																												
2.0 B2		1	0																																												

Exercises

155 [illegible] 41.5[illegible]

1. Perform the following binary arithmetic and determine the decimal equivalents:

   a. Add and subtract 1101101 ± 0011011 10110.001101 ± 01101.010110 .

   $$\begin{array}{r} 1101101 \\ \pm\ 0011011 \\ \hline \end{array} \qquad \begin{array}{r} 10110.001101 \\ \pm\ 01101.010110 \\ \hline \end{array}$$

   -15.[illegible] 10.6[illegible]

   b. Use two's complement subtraction in part a.

   c. Multiply

   $$\begin{array}{r} 10110.101 \\ \times\ 1.01011 \\ \hline \end{array}$$

2. Determine the products and quotients of the following pairs of numbers:

   16.25 B34, 3.5 B10
   6.4 B10, .125 B0
   .375 B0, 2.0 B3

3. Show how the above operands and results would appear in PHILCO 2000 words. Stop converting a result to its binary form if it is a repeating fraction.

TRANSLATOR-ASSEMBLER-COMPILER CONSTANTS

Use of Constants

The preceding section was concerned with the PHILCO 2000 representation of numbers, letters, and symbols. This section is concerned with the Translator-Assembler-Compiler representation of constants.

The term "constant" is derived from the fact that, unlike data, constants generally do not vary but are fixed. A constant may be a word, part of a word, or a number of words. A constant usually applies to many units of data but, rather than be repeated in the data, is included once in the program. This saves magnetic tape space and tape running time.

Constants were used in several examples in Chapter III. The last exercise in that chapter required such constants as the dollar amount, $144.00, and the percentage, 3.00%. All constants in Chapter III were assumed to be stored in certain memory locations. No mention was made of how they got there or what their format was. These points are the subject of this section.

TAC constants are not instructions but are written on coding paper along with instructions. They may be written in place of instruction address parts or in place of entire instructions. An identifying letter followed by a slash precedes the TAC constant and distinguishes it from an address or instruction. For example, to represent the alphanumeric constant, MAY Δ 1958, the following notation will be used:

A/MAY Δ 1958

A/ identifies the characters that follow as alphanumeric. TAC, of course, deletes the letter and slash before placing the constant in the program.

Pool and Non-Pool Constants

A constant written in the address column is placed in the constant "pool" by TAC, and its address is inserted in the instruction referring to it. By definition, such a constant is called a pool constant and the part of the program allocated to such constants is called the constant pool. A constant beginning in the command column, however, appears in the sequence in which it was coded rather than in the constant pool. This type of constant is called a non-pool constant.

There are the following distinctions between pool constants and non-pool constants:

a. Pool Constants

1. The constant is placed in the constant pool.
2. The length of the constant is limited to one word.
3. TAC insures that no constant of the same type is unnecessarily repeated.

b. Non-Pool Constants

1. The constant appears in the program where written.

2. Some non-pool constants may be any size and occupy as many successive memory locations as necessary.

3. Repetition of constants may occur.

The programmer determines the location of the constant by considering the above points. If a constant is to be modified, for example, it should not be a pool constant. Similarly, it is simpler to make a message or sentence, that is to be printed on the Console Typewriter, a non-pool constant. In most other cases, however, it is simpler to write the constant as a pool constant.

Note that when constants are written in the Command column, the Central Computer does not distinguish between them and instructions. Therefore, it is the programmer's responsibility to ensure that a constant is not executed as an instruction. This is done by placing constants after Jump or machine Halt instructions or by writing constants as addresses.

In order for TAC to distinguish the end of the address part from the beginning of the remarks section, the dollar sign ($) must follow the address. This symbol may be written each time or omitted if it is understood that the keypunch operator will punch the sign after every address. (This assumes that the program is keypunched onto cards. It is also possible to punch the program onto paper tape; however, the same convention applies.)

A constant written both as a pool constant and as a non-pool constant is shown below. The two TMA instructions produce the same result.

L	LOCATION	COMMAND	ADDRESS AND REMARKS
		TMA	A/MAYΔ1958$
		TMA	CØNST$
		.	.
		.	.
		.	.
	CØNST	A/MAYΔ19	58$

Types of Constants

TAC constants and their formats are listed below. The lower case letters represent characters or digits which vary according to the particular constant. The upper case letters are a part of every constant. The configuration, xx...xx, indicates a variable number of characters.

Fixed point decimal number:	D/Number EcBd	— binary point position
Floating point number:	F/Number Ec	— power of 10
Word constant:	W/xxxxxxxx	
Alphanumeric:	A/xx...xx	
Octal:	Ø/xx...xxTd	terminal position of field
Hexadecimal:	H/xx...xxTd	
Numeric:	N/xx...xxTd	
Binary:	Decimal/Binary Td	

(Other specialized constants are described in subsequent chapters and in the TAC manual)

For the first three types of constant--Floating Point, Fixed Point, and Word constants--TAC produces only one full word. For the remaining types, TAC produces a constant which may be a part of a word, or a full word, or in the case of alphanumeric constants, a number of words. If a constant requires only a part of a word, the programmer may specify that other constants occupy the same word. The number of constants which may be combined in a word is limited by the size of each constant and the size of the word.

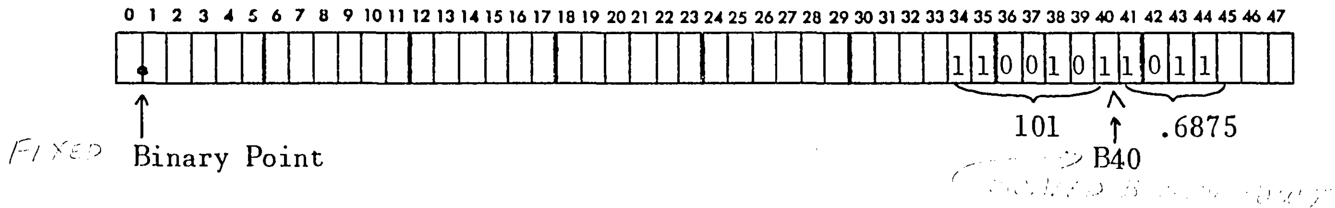
A combination of constants is specified by writing all the desired constants and separating them by semicolons. Thus, the following notation refers to a word which will be composed of an octal constant, a hexadecimal constant, and a binary constant:

Ø/425; H/976; 27/1

Fixed and Floating Point Decimal Constants

In the listed format, Number represents the decimal number, Ec represents the power of ten, positive or negative, by which it is multiplied, and Bd represents the binary scale factor or position of the binary point for a fixed point decimal number.

For example, D/.1016875 E3 B40 would represent the number 101.6875 placed in a PHILCO 2000 word with the binary point positioned 40 places to the right of bit position zero. That is



This constant could be written as an address

COMMAND	ADDRESS AND REMARKS
T M A	D / . 1 0 1 6 8 7 5 E3 B40 $

or in place of an instruction

COMMAND	ADDRESS AND REMARKS
D / . 1 0 1 6 8	7 5 E 3 B 4 0 $

In writing a decimal constant, the principal part (number) is written first and is followed by the exponent and the binary scale factor. If a scale factor is zero, B0 must be written. When the principal part is a whole number, the decimal point may be omitted. In the special case where both the binary scale factor and the decimal point are omitted, the constant is treated as an integer with scale factor B47. Note that in this case, a fractional part of a number would fall outside the 48 bits of the word and would be lost.

D/15	≡	15 B47
D/15 E3	≡	15 E3 B47
D/15 E-3	≡	0 (since the fractional part is lost)

Floating point decimal constants may or may not contain a decimal point and do not contain a binary scale factor, i.e.,

F/15.0 ≡ Floating point number 15

For both forms of decimal constants a positive number is indicated by a plus sign or it may be omitted; a negative number is always indicated by a minus sign.

Word Constants

A special type of alphanumeric constant is the Word constant which is designated by the symbols W/ followed by any combination of eight computer-acceptable characters. The constant must contain exactly eight binary-coded characters to be stored in the order that they are written. Word constants permit the use of such symbols as the semicolon and the dollar sign.(Refer to alphanumeric constant.)

Examples of Word constants are:

COMMAND								ADDRESS AND REMARKS								
T	M	A						W	/	;	/	A	)	$	6	R Δ $
W	/	;	/	A	)	$	6	R	Δ	$						

A/Pool Constant = 1 to 8 characters; fill with zeros.

Alphanumeric Constants

A/Non Pool Constant = no limit of characters; fill with blanks.

Alphanumeric constants are composed of binary-coded characters. When written as an address, a maximum of eight characters, that is, one word, may be used. If less than eight characters are written, they are placed in the high order positions of a word and zeros fill the remainder of the word provided that no other constant shares the word.

Alphanumeric constants in place of instructions may be any length and may be continued from line to line. This is a unique feature of alphanumeric constants. The first instruction line begins in the Command column and may contain up to 62 characters. Subsequent lines begin in the Address column and may contain up to 56 characters. TAC places every eight character group in a consecutive word of memory. If the number of characters written is not a multiple of eight, TAC fills the remainder of the last word with non-printing "space" symbols.

Alphanumeric constants are written with the symbols A/ immediately preceding the constant. Each alphanumeric constant is terminated by a dollar sign, $, unless this type of constant is only one part of a constant word. As a result, no alphanumeric constant may contain a dollar sign, semicolon, or right parenthesis. This type of constant will be described later.

	COMMAND								ADDRESS AND REMARKS								
1)	T	M	A						A	/	C	D	E	1	2	3	4 5 $ An eight character pool constant
2)	A	/	C	D	E	1	2	3	4	5	$						An eight character non-pool constant
3)	T	M	A						A	/	C	D	E	$			A three character pool constant
4	A	/	P	R	I	C	E	Δ	Q	U	Ø	T	A	T	I	Ø	N Δ TØDAY Δ IS $ A three word non-pool constant (space is a character)

These constants would appear in PHILCO 2000 words as follows:

1+2

C	D	E	1	2	3	4	5

(3)

C	D	E	0	0	0	0	0

(4)

P	R	I	C	E	Δ	Q	U
O	T	A	T	I	O	N	Δ
T	O	D	A	Y	Δ	I	S

where each character is represented by six bits.

Octal Constants

Octal constants have the form Ø/xx...xx Td. The x's represent up to 16 octal digits (one word) and the T entry indicates the termination position of the constant. The T is analogous to the B in fixed point decimal constants.

Octal constants are always considered to be integers by TAC. These constants are converted directly to binary form, three bits per octal digit. Octal constants are positioned within a PHILCO 2000 word by the T entry, which indicates the position of the least significant digit of the octal constant.

If no termination is indicated, an octal constant fills the PHILCO 2000 word from left to right, high order to low order. If sixteen digits are not written and no other constant is to share the word, the low order positions of the word are filled with zeros. Similarly, if a constant is written which doesn't specify digits for the high order positions, they too are filled with zeros.

The following are examples of octal constants and their binary equivalents:

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47
Ø/75	1	1	1	1	0	1																																										
Ø/75 T23																			1	1	1	1	0	1																								

Note: Blank positions indicate zeros.

Hexadecimal Constants

Hexadecimal constants are indicated by the symbols H/ followed by the constant. The constant may consist of up to 12 hexadecimal digits. A hexadecimal digit is a number 0 through 9 or a letter A through F. Each of the digits requires four bits to represent it according to the following table.

Hexadecimal Digit	Decimal Equivalent	Binary Equivalent
0	0	0000
1	1	0001
2	2	0010
3	3	0011
4	4	0100
5	5	0101
6	6	0110
7	7	0111
8	8	1000
9	9	1001
A	10	1010
B	11	1011
C	12	1100
D	13	1101
E	14	1110
F	15	1111

The format for hexadecimal constants is H/xx...xx Td. The x's represent up to 12 (one word) hexadecimal digits. In all other aspects hexadecimal constants are treated in the same way as are octal constants.

The following are examples of hexadecimal constants:

H/1234ABCD78EF

H/AB

H/DEF T16

Numeric Constants

Numeric constants are designated by the symbols N/. The decimal number following these symbols must be positive and integral in value. The format for this constant is N/Number Td. A termination indicator, Td, must follow the decimal number to indicate the right boundary. The decimal number will then be converted to its binary equivalent and inserted in the word in the proper position. For example, N/1149 T35 would become

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47
																									1	0	0	0	1	1	1	1	1	0	1												

1149_{10}

$= 2115_8$

in a PHILCO 2000 word.

Sufficient space in the word must be allocated to contain the binary equivalent of the decimal integer. If the termination indicator is too small, the number will "overflow" and the high-order bits will be lost. For example, N/1149 T9 would become

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47
1	0	0	0	1	1	1	1	1	0	1																																						

and the most significant bit would be lost.

Binary Constants

Pure binary constants are written by specifying the desired number of binary ones, zeros, or combinations of ones and zeros in the following format:

Decimal Number/Binary Number Td.

The decimal number may have any value from 1 to 48 and the binary number represents the desired binary configuration. The decimal number indicates the number of groups of this configuration to be placed in the constant. The right boundary may be specified by a termination indicator.

For example, 20 binary ones terminating in the 19th position, 20/1 T19, would appear in a PHILCO 2000 word as 20 bits = 0 —19

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47
1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1																												

20/1

Since the semicolon is used to separate constants, 20 binary ones, followed by two groups of 101, followed by 22 ones would be written as follows:

20/1; 2/101; 22/1

This constant would appear as

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47
1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	1	1	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1

20/1 2/101 22/1

When a number of constants are to occupy a word, they are written in the order that they are to appear in the word. Unspecified parts of a word are filled with zeros. A termination indicator (T entry) may be used to determine the right boundary of each constant (except alphanumeric constants). Otherwise it will be determined by its own size requirement. The left boundary of a constant is determined by the preceding constant. If a constant is too long to fit within the existing boundaries, its high order bits are lost. To avoid this loss, the programmer must maintain a count of the bit positions used per constant, so as not to exceed 48 bits.

For example, the following binary information might be written as:

L	LOCATION	COMMAND	ADDRESS AND REMARKS
		TMA	A/AB;Ø/77;H/99;10/1 T4 5 $
	or		
		A/AB;Ø/7	7;H/99;10/1 T4 5 $
	or		
		A/AB;Ø/7	7T17;H/99;10/1 T4 5 $

and would appear in a PHILCO 2000 word as

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47
0	1	0	0	0	1	0	1	0	0	1	0	1	1	1	1	1	1	1	0	0	1	1	0	0	1	0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1	1	0	0
A						B						7			7			9				9														10/1 T45											

SUMMARY

a. Because of the economy and efficiency of binary manipulation, most computers use a form of binary representation.

b. The rules of binary arithmetic are:

Addition

$$\begin{array}{cccc} 0 & 0 & 1 & 1 \\ +0 & +1 & +0 & +1 \\ \hline 0 & 1 & 1 & 10 \end{array}$$

$\nwarrow$ Carry

Subtraction

Borrow

$$\begin{array}{cccc} 0 & \curvearrowright 0 & 1 & 1 \\ -0 & -1 & -0 & -1 \\ \hline 0 & 1 & 1 & 0 \end{array}$$

Multiplication

$$\begin{array}{r} 0 \\ \times 0 \\ \hline 0 \end{array} \qquad \begin{array}{r} 0 \\ \times 1 \\ \hline 0 \end{array} \qquad \begin{array}{r} 1 \\ \times 0 \\ \hline 0 \end{array} \qquad \begin{array}{r} 1 \\ \times 1 \\ \hline 1 \end{array}$$

c. Conversion from one number system to another with a lower base is accomplished by successive divisions by the lower base. The remainder digits make up the converted number. Conversion from the lower base number to the higher base number is accomplished by forming a sum of the products of the digits in the number by the appropriate power of the base.

d. An octal number is a number with a base of eight and may be converted directly to binary by converting each octal digit to three binary digits.

e. Hexadecimal numbers have a base of 16. Each hexadecimal digit is converted to four bits.

f. All numbers in the arithmetic section are considered to be less than one and greater than or equal to minus one. Negative numbers are represented in two's complement form.

g. An arithmetic result which falls outside of the above limits sets the overflow indicator and produces a sign which is the opposite of the correct one.

h. The instructions used in the detection of overflow are

ICØS: Inhibit Clearing of Overflow indicator

ICØZ: Remove Inhibition on Clearing of Overflow indicator

JØF: Jump if Overflow is indicated

JNØ: Jump if No Overflow is indicated

i. Despite the computer number range, any number may be represented in the Central Computer by scaling. When arithmetic is performed, the scale factors must be considered for the positioning of the binary point. The necessary factors are as follows:

1. Addition and Subtraction

   The binary points must be aligned.

2. Multiplication

   The scale factor of the product is the sum of the scale factors of the multiplier and multiplicand.

3. Division

   The scale factor of the quotient is the difference between the scale factors of the dividend and the divisor.

j. TAC constants, except for those to be described in subsequent chapters, are listed below:

1. One to a word

Fixed Point:	D/Number Ec Bd - if omitted, B47 assumed
Floating Point:	F/Number Ec
Word:	W/xxxxxxxx

2. More than one to a word permissible (Field Constant)

Alphanumeric:	A/xx...xx
Octal:	Ø/xx...xx Td optional
Hexadecimal:	H/xx...xx Td optional
Numeric:	N/Number <u>Td</u> mandatory
Binary:	Decimal/Binary Td optional

k. All constants may be pool or non-pool constants. Pool con-constants are written in place of addresses; non-pool con-constants are written in place of instructions. A dollar sign is used to separate the remarks from the instruction or constant. Semicolons are used to separate combined constants.

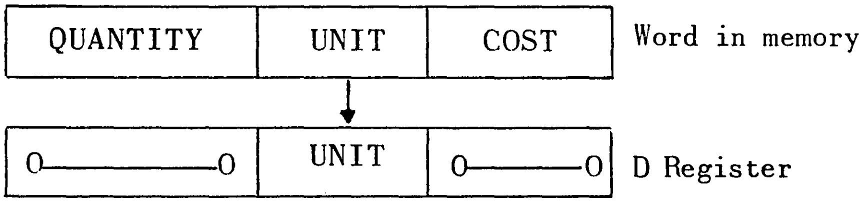
CHAPTER VI

DATA MODIFICATION

MODIFYING WORDS

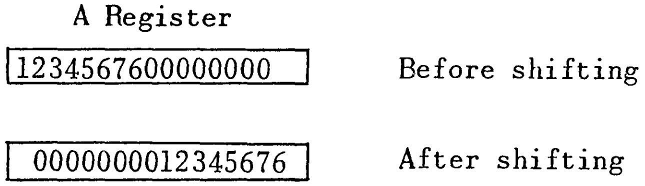
Prior to this chapter, all data modification was accomplished by transferring entire words to the adder network. The modification performed was that of arithmetic. In this chapter, methods will be described which allow the programmer to modify parts of words and to alter the position of data within words. These functions are called extracting and shifting and can be illustrated by the following diagrams:

a. Extracting: Extract the unit designation from a word in memory to the D Register.



The primary purpose of extracting, is to select one of several data elements which have been packed in one word.

b. Shifting: Shift the number in the A Register to the right.



Shifting establishes the position of the binary point of numbers before and after arithmetic operations. Since the value of a number is relative to the position of the binary point in a word, shifting is equivalent to multiplying or dividing by a power of two. Shifting is also used to align alphanumeric data which is to be compared.

SHIFTING

Types of Shifts

Only words in the arithmetic registers may be shifted. They may be shifted individually, or they may be shifted as one double length unit, as in the case of words in the A and Q Registers.

Three types of shifts are possible in the Central Computer:

a. ordinary (logical)

b. numerical (preserves sign)

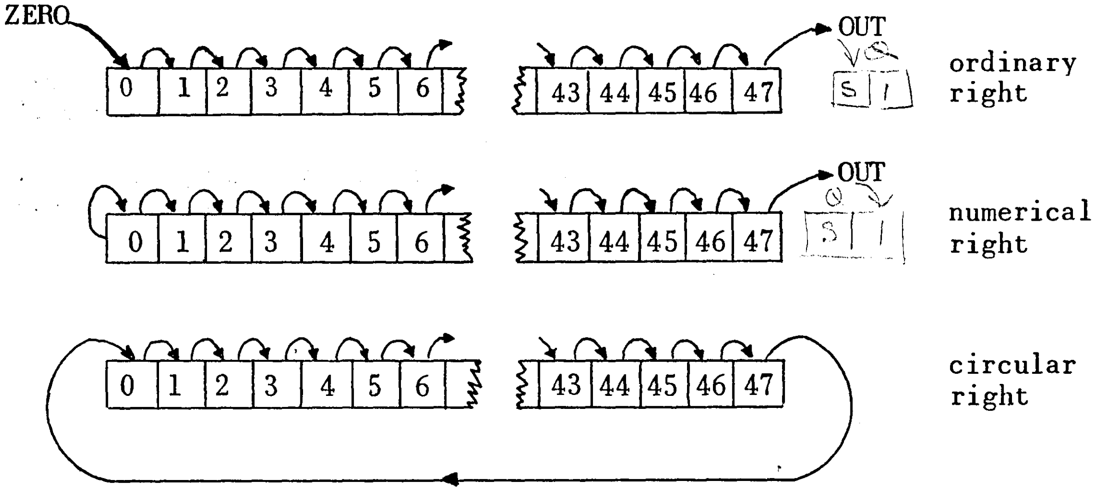
c. circular. (D register and conditional Q-jumps only)

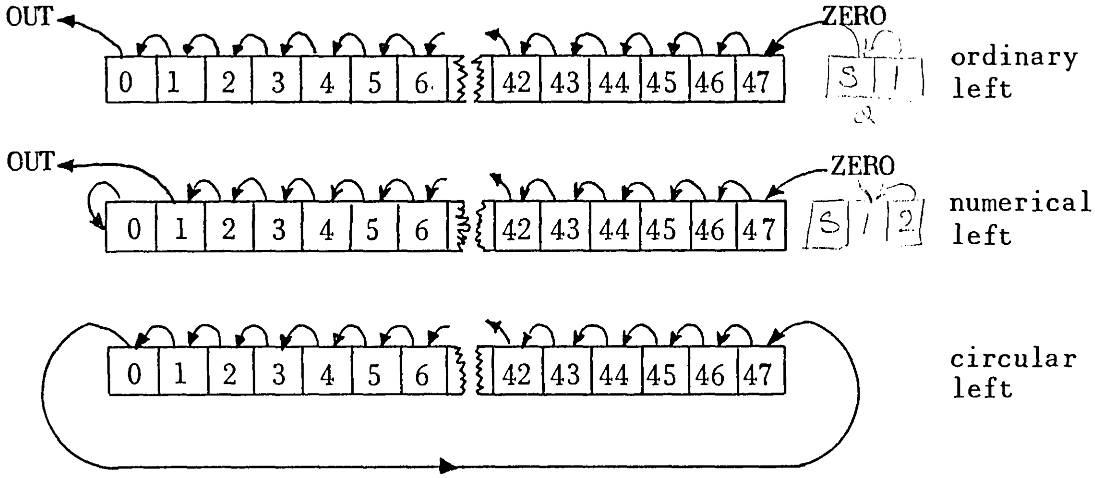
Ordinary shifts treat every bit in the affected register or registers alike and are usually used for nonnumeric data. Numerical shifts treat words in such a way as to preserve the signs of numbers. A circular shift is an ordinary shift except that bits shifted out one end of the register are returned at the other end of the register.

For each bit position shifted in each type of shift, one bit is shifted out of one end of the register and one bit is introduced at the other end. In ordinary shifts and numeric left shifts the bits introduced are zeros. However, in numeric right shifts the bits introduced to the right of the sign bit are the same as the sign.

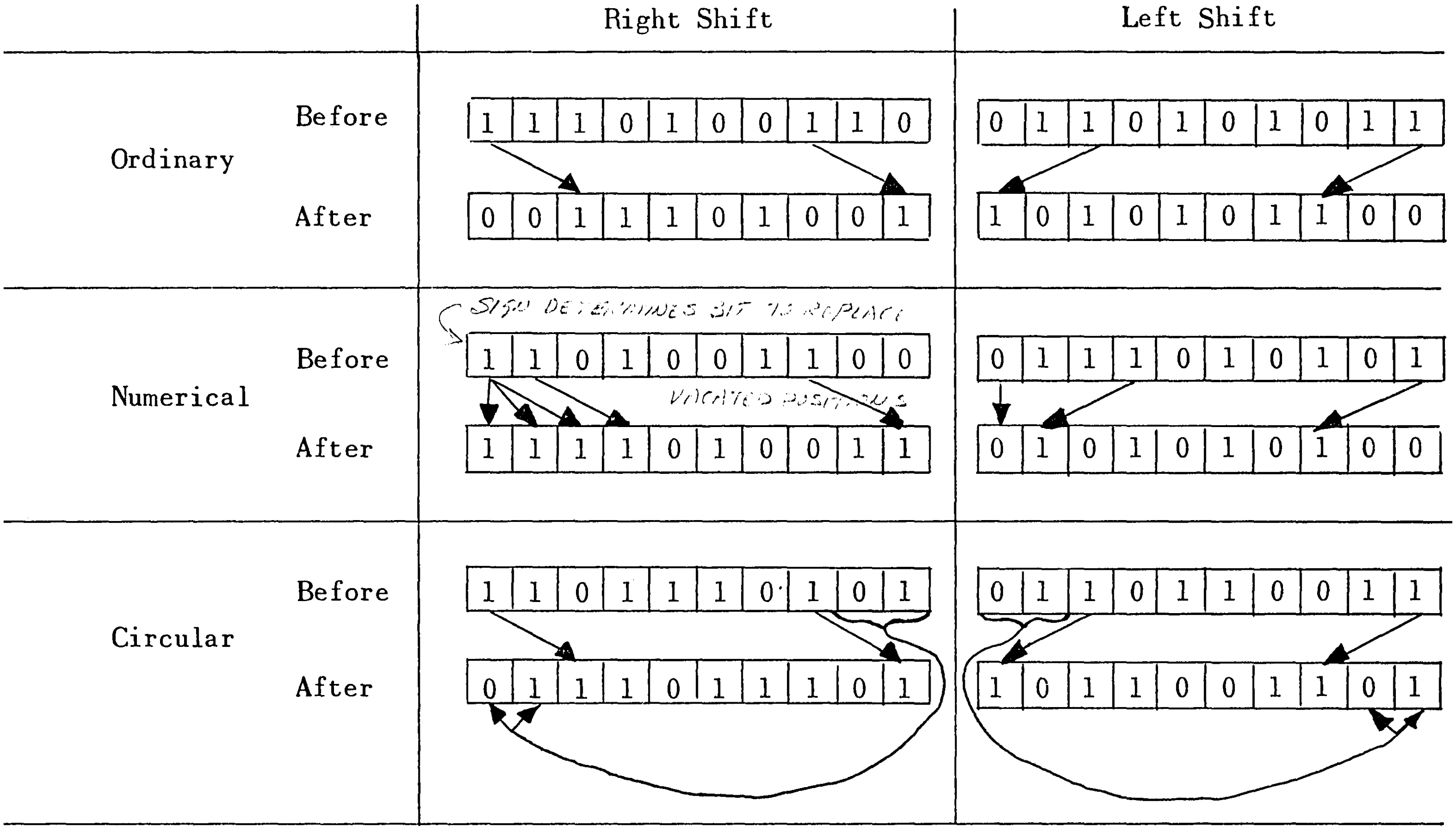
Overflow will occur if ordinary left shift causes a different bit to be shifted into sign than that shifted out of the sign position.

The following diagrams illustrate the three types of shifts: (Arrows indicate the movement of one bit.)





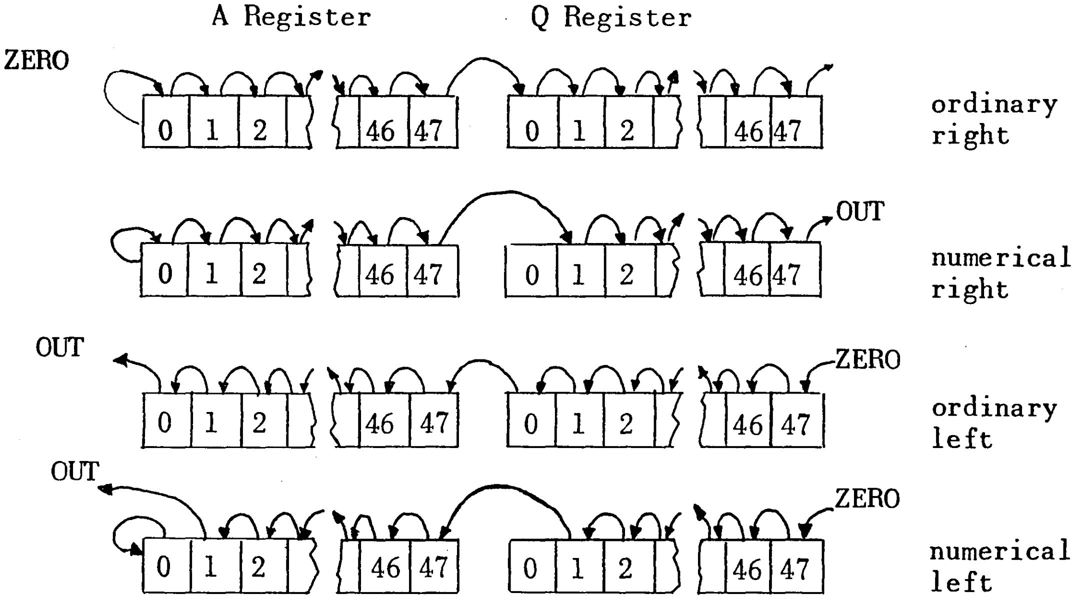
The following diagrams illustrate shifts of two-bit positions: (For simplicity each register is shown as if it had a ten-bit capacity.)



		Right Shift	Left Shift
Ordinary	Before	1 1 1 0 1 0 0 1 1 0	0 1 1 0 1 0 1 0 1 1
	After	0 0 1 1 1 0 1 0 0 1	1 0 1 0 1 0 1 1 0 0
Numerical	Before	1 1 0 1 0 0 1 1 0 0	0 1 1 1 0 1 0 1 0 1
	After	1 1 1 1 0 1 0 0 1 1	0 1 0 1 0 1 0 1 0 0
Circular	Before	1 1 0 1 1 1 0 1 0 1	0 1 1 0 1 1 0 0 1 1
	After	0 1 1 1 0 1 1 1 0 1	1 0 1 1 0 0 1 1 0 1

(Note: Left circular shifts are only possible with the conditional jump instructions described later in this chapter.)

The following diagrams show the additional effects of shifting the contents of the A and Q Registers together:



The shift commands have the following format:

OPERATION	REGISTER	OPTION
Shift'Left Shift Right	Register A or Q or D* or A and Q	Numerical (blank for ordinary)
Shift Circular	D*	None
SL SR	A Q D* AQ	N
SC	D*	None

*The contents of the D Register may only be shifted to the right.

The complete list of shift commands follows: (The shift is ordinary unless stated otherwise.)

COMMAND	EXPLANATION
SLA	Shift Left A
SRA	Shift Right A
SLAN	Shift Left A Numerically
SRAN	Shift Right A Numerically
SLQ	Shift Left Q
SRQ	Shift Right Q
SLQN	Shift Left Q Numerically
SRQN	Shift Right Q Numerically
SRD	Shift Right D
SRDN	Shift Right D Numerically
SCD	Shift Circular D (right)
SLAQ	Shift Left A and Q
SRAQ	Shift Right A and Q
SLAQN	Shift Left A and Q Numerically
SRAQN	Shift Right A and Q Numerically

The number of binary positions shifted is specified by the address part of the instruction and may be from zero to 63 positions. (The reason for this limit is that the control section determines the number of positions to shift by the right six bits of the address. The largest number represented by six bits is 63 which is the first sum of·powers of two exceeding 48. Usually there is no reason for shifting more than 48 positions, i.e., an entire word length. However, if it is necessary to shift more than 48 positions, it is faster to use a Transfer instruction for the first 48 and then a Shift instruction.

A zero position shift has no effect on the computer. No shift occurs and the computer does not stop. If the address part of a Shift instruction specifies a shift of more than 63 positions, the number of places shifted is determined by the value of the right six bits of the address. This number will be equal to the address written minus a multiple of 64. For example, if 143 is specified, the number of positions shifted will be 143 - (64 x 2) = 15.

Another characteristic of Shift instructions is that the overflow indicator is cleared before they are executed. Also, left shifts may set the overflow indicator if the bit in the sign position differs from the next bit for one of the positions shifted. This allows the programmer to determine if the sign had changed in an ordinary shift or if a significant bit were lost in a numerical shift.

These conditions are shown in the following diagrams:

After Shifting Left One Bit

Before Shifting			Ordinary				Numerical				Overflow Indicator
0	1	2	0	1	2		0	1	2		
1	1	X	1	X		No sign change	1	X		No significant bit lost	Not set
0	0	X	0	X			0	X			Not set
1	0	X	0	X		Sign change	1	X		Significant bit lost	Set
0	1	X	1	X			0	X			Set

Sign bit position

SYMBOLIC ADDRESSING

Up to this point, symbolic addressing has only been used in reference to instructions. Because of its convenience and ease of use, symbolic addressing may also be used in reference to data.

The programmer assigns a meaningful symbol, name, or abbreviation of up to 21 characters to the first word of a data area in memory. All words following this word may be labeled by adding one to each preceding symbol. For example, a payroll data area might be labeled as follows:

PAY	First word of area
PAY + 1	Second word of area
PAY + 2	Third word of area
.	.
.	.
.	.

As in all symbolic addressing, TAC substitutes an actual or computer address for each symbolic address.

(Chapter V should now be reviewed for representation of numbers and constants and for the placement of the binary point in PHILCO 2000 arithmetic.)

Example 1: Aligning Values for Addition, Subtraction

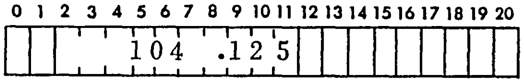
Add the numbers 104.125 B8 and 749.5 B24.

To add the two numbers, their binary points must be aligned; i.e., they must occupy the same relative positions within their respective words.

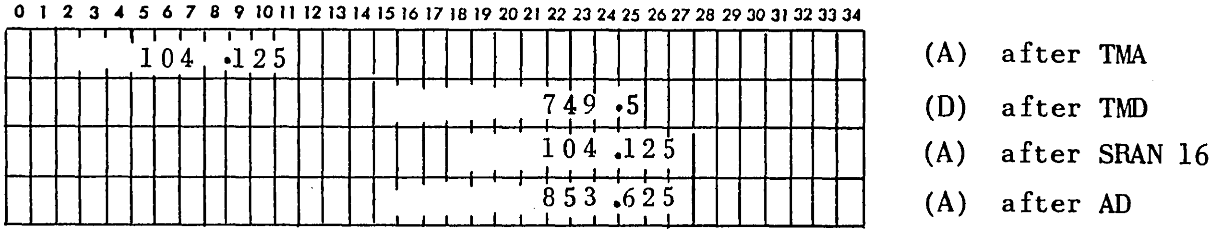
COMMAND	ADDRESS AND REMARKS
TMA	D/104.125B8 $
TMD	D/749.5B24 $
SRAN	16$
AD	

To align the points at B24, the number 104.125 is shifted right 16 positions.

The notation used is



where the number 104.125 is in a word in binary positions 2 through 11, inclusive. Since the binary point is placed 8 positions to the right of bit position zero, the scale factor is 8 or B8. The diagram that follows shows the contents of the registers before and after shifting:



For two reasons, the number 749.5 cannot be shifted left to align the points. First, the contents of the D Register cannot be shifted left. Second, since ten bits should be allocated to a three decimal digit number, part of the number would be lost in aligning the points even if the word in D could be shifted left.

Example 2: Aligning Words for Comparisons

Memory location ALPH1 contains a four character alphanumeric serial number. Memory location ALPH2 contains another four character serial number. Compare them and jump to MATCH if (ALPH1) ⩾ (ALPH2). The serial number in ALPH1 occupies the first four binary-coded character positions and the serial number in ALPH2 occupies the last four binary-coded character positions (the first four positions contain zeros).

The coding for this example follows:

L	LOCATION	COMMAND	ADDRESS AND REMARKS
		TMA	ALPH1$
		SRA	24$
		TMD	ALPH2$
		JAGD	MATCH$

The following diagram shows the status of the registers:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47	
			A					B							C						D																											(A) after TMA
																											A						B					C							D			(A) after SRA
																											W						X					Y							Z			(D) after TMD

In the above example, Q could have been used with JAGQ even though the data is alphanumeric.

Example 3: Multiplication and Division by Shifting

Given the number 14: 1110

Shift it one position right: 111 = 7

or shift the number 14 one position left: 11100 = 28

Each right shift of one position is, in effect, a division by two without remainder, and each left shift of one position is equivalent to multiplying by two.

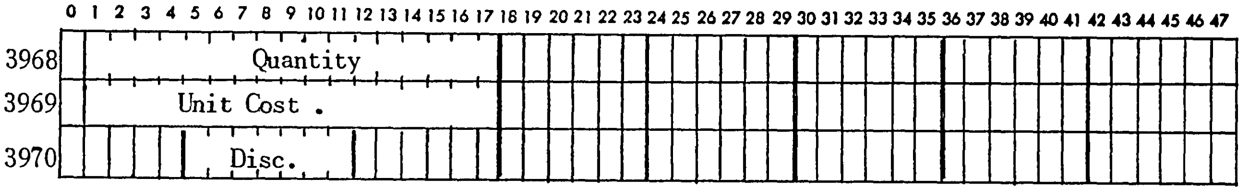
Multiplication by numbers which are not powers of two may be done by shifting and adding. For example, shifting a number three places left (i.e., multiplying by eight) and then adding the result to the original number is equivalent to multiplying by nine.

Example: Multiply 3 by 9 by shifting.

Given the number 3:	0011
Shift it three positions left:	11000 = 24
Add the shifted number to the original:	11011 = 27

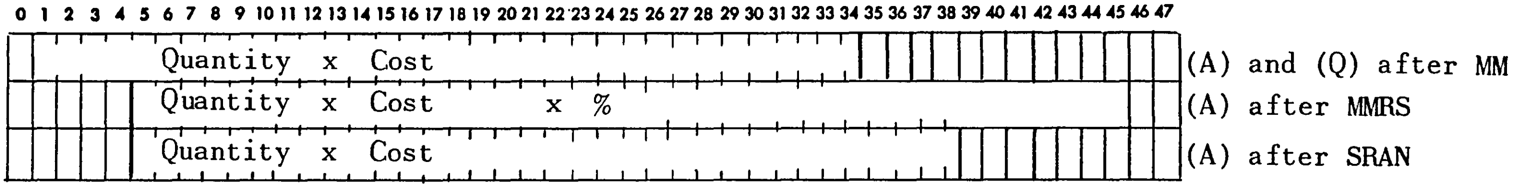
Example 4: Shifting by Multiplication

Given the following data format, perform the calculations required in the example in Chapter III, page 42. Assume five decimal digits for quantity and cost and two decimal digits for the percentage discount.



COMMAND	ADDRESS AND REMARKS	
TMQ	3968$	Quantity x Cost → A
MM	3969$	
TAQ		Quantity x Cost → Q
MMRS	3970$	Quantity x Cost x % → A, D, M
TQA		Quantity x Cost → A
SRAN	4$	Align binary points
SMS	3970$	Quantity x Cost - Quantity x Cost x %
		→ A, D, M

The status of the registers during this operation are as follows:



Circular Shifts of the Q Register

Four conditional Jump instructions cause circular shifts of the contents of the Q Register.

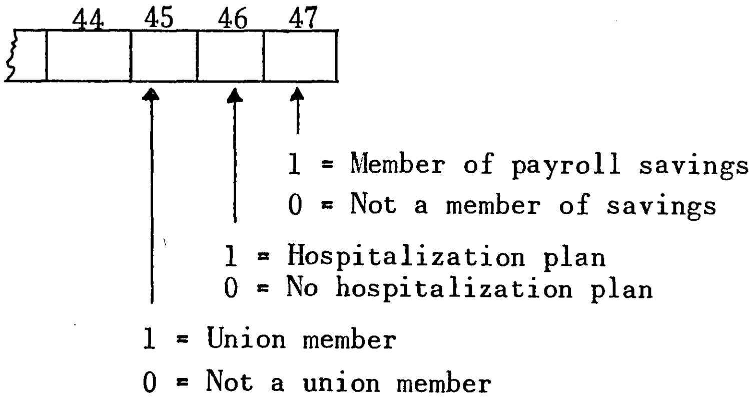
JQP	Jump if (Q) are positive	Left shift
JQN	Jump if (Q) are negative	
JQO	Jump if (Q) are odd	Right shift
JQE	Jump if (Q) are even	

Each time JQP or JQN is executed, the contents of the Q Registers are circular shifted one position to the left regardless of conditions. Each time JQO or JQE is executed, the contents of the Q Register are circular shifted one position to the right regardless of conditions.

Unlike the Shift instructions, JQP, JQN, JQO, and JQE have no effect on the overflow indicator. These instructions have many uses, including determining the sign of a quotient, counting, and testing individual bits which represent yes or no codes; i.e., 1 corresponds to yes and 0 to no. (The use of these instructions for counting will be illustrated in Chapter VIII.)

Example

The right three bits of the word in memory location CODE represent yes or no codes as follows:


44
45
46
47
1 = Member of payroll savings
0 = Not a member of savings
1 = Hospitalization plan
0 = No hospitalization plan
1 = Union member
0 = Not a union member

Code the instructions necessary for each code to be examined and jump to SAVE, and/or HOSP, and/or UNION, depending on the combination of ones and zeros present. Note that if the first code is a one, a jump to SAVE is effected. In order to have the next two codes examined it is necessary to store the contents of the Q Register in memory after the jump. Then, after the SAVE operations are completed, the code word must be replaced in Q and a jump, to the instruction JQO HOSP, executed. The same procedure must be provided for each Jump instruction. These provisions are not shown in the coding but will be covered in Chapter VII under subroutines.

The required coding is as follows:

COMMAND								ADDRESS AND REMARKS						
T	M	Q						C	Ø	D	E			Code word ———▸Q
J	Q	Ø						S	A	V	E			If last bit is one, jump to SAVE
J	Q	Ø						H	Ø	S	P			If next to last bit is one, jump to HØSP.
J	Q	Ø						U	N	I	Ø	N		If third from last bit is one,jump to UNIØN.

To illustrate the effect of the above instructions, assume that each code is a one. Because the remaining bits of the word are not significant to this illustration, they are left blank. The contents of Q would then have the following appearance after each of the above instructions.

0	1	2	3	4	5	6	7	8	9	10	11	35	36	37	38	39	40	41	42	43	44	45	46	47	
																						1	1	1	(Q) initially
1																							1	1	after JQO SAVE
1	1																							1	after JQO HOSP
1	1	1																							after JQO UNION

Inclusive OR

The following command performs an inclusive OR operation with two words - one in D and the other in the memory location designated by the address portion of the instruction:

COMMAND	EXPLANATION
DORMS	D or M and Store: A composite word is formed in which there are binary ones in every position for which there is a one in the D Register OR in the specified Memory location. This is equivalent to transferring all the ones from the word in memory to the D Register without changing the remaining bits of D. The resulting word in D is then Stored in the specified memory location.

The rules that this operation follow are

0	1	0	1	D Before execution
0	0	1	1	M Before execution
0	1	1	1	D and M After execution.

For example:

Before executing DORMS

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47	
	1	0	1	0	1	1	1	0	0	1	1	0	0	0	1	0	1	0	1	0	0	0	0	1	1	1	1	1	1	0	0	1	1	1	0	0	0	0	1	1	1	0	1	0	1	1	0	1	Memory location
	1	1	0	0	0	1	0	0	1	1	1	0	1	1	0	1	1	0	1	0	1	1	0	0	1	0	0	1	0	0	0	0	1	0	1	1	0	0	1	1	0	1	0	1	1	0	0	0	D Register

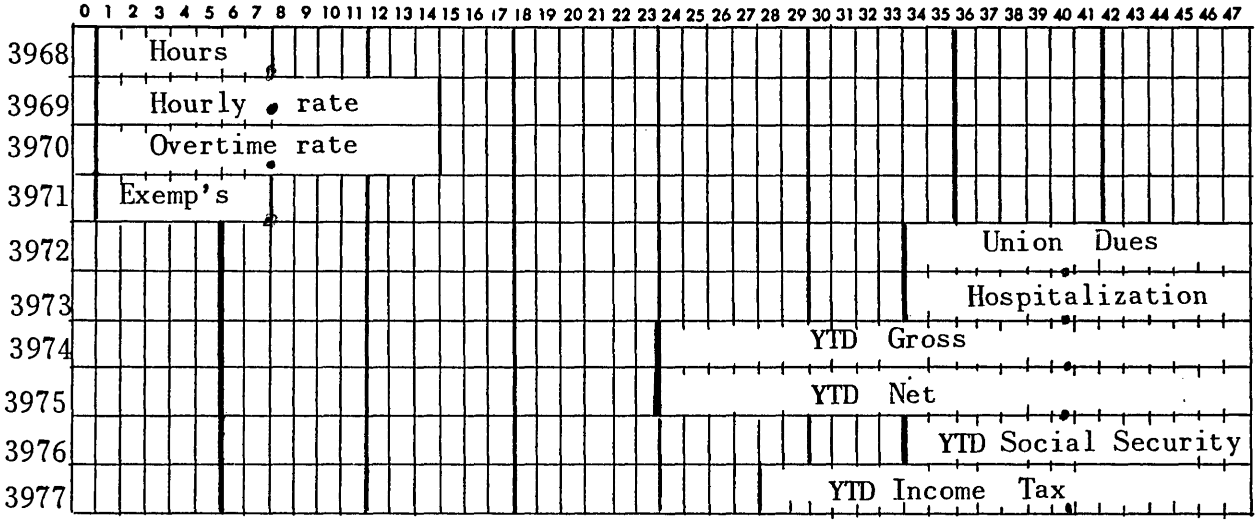
After executing DORMS

1	1	1	0	1	1	1	0	1	1	1	0	1	1	1	1	1	0	1	0	1	1	0	1	1	1	1	1	1	0	0	1	1	1	1	1	0	0	1	1	1	1	1	1	1	1	0	1	Memory location
1	1	1	0	1	1	1	0	1	1	1	0	1	1	1	1	1	0	1	0	1	1	0	1	1	1	1	1	1	0	0	1	1	1	1	1	0	0	1	1	1	1	1	1	1	1	0	1	D Register

These two instructions are used for combining fields in words, for Boolean operations, and when it is necessary to preserve particular bits in words.

Exercise

Using the data format illustrated below, recode the exercise in Chapter III, page 62. The constants which were stored in memory locations 3000-3004 are to be allocated by the reader.



EXTRACTING

Records, Fields, .and Record Layouts

In data processing a "record" is defined as the unit of information which completely describes one member of a class of data. Within the record are fields which describe each element of the record.

Typical data processing records are the following:

a. master employee record

b. inventory record

c. stock transaction record.

Typical fields in an inventory record are the following:

a. stock number

b. unit of issue

c. on-hand amount

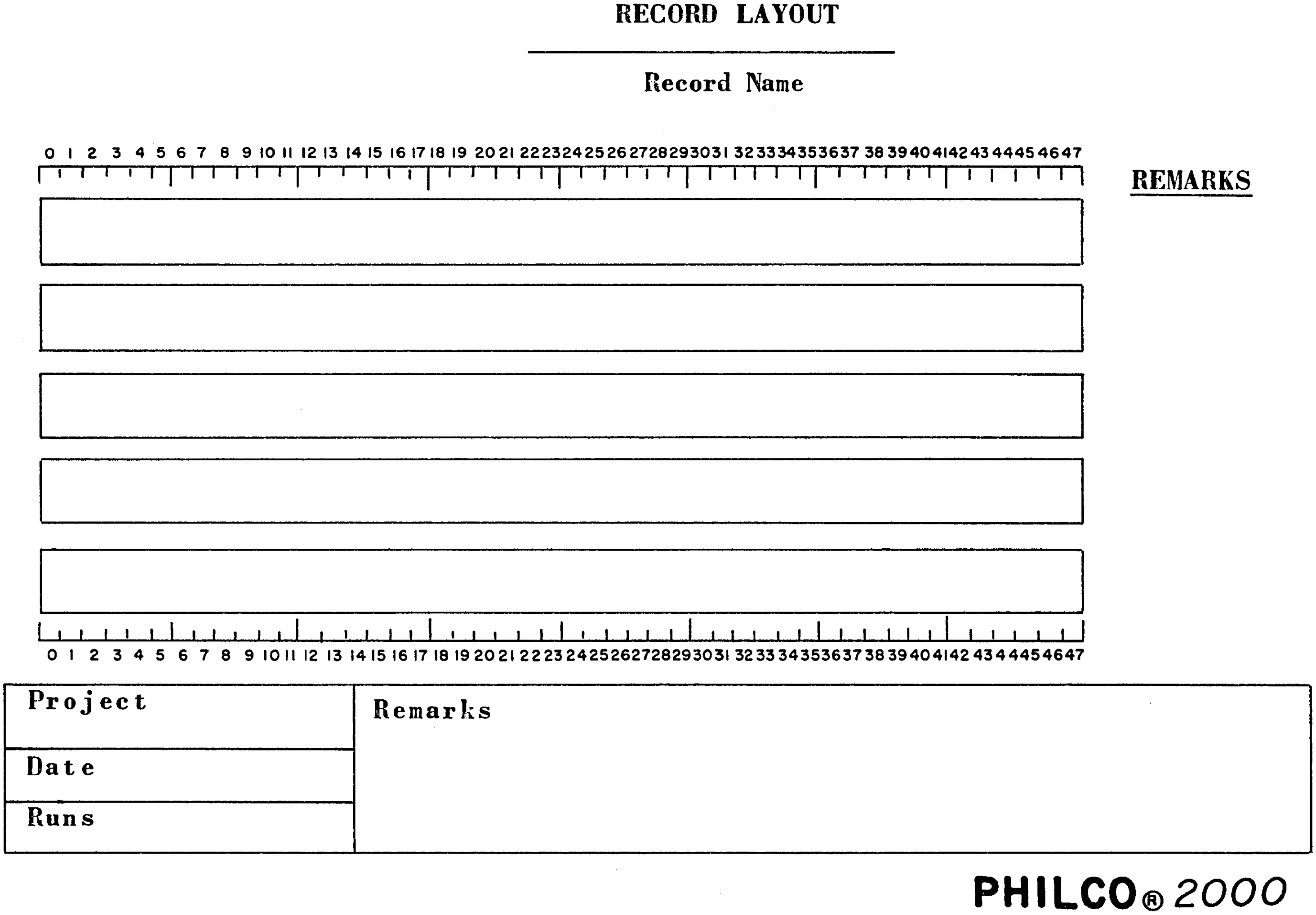
d. low level amount.

A record, then, is the totality of all its fields. (A file is the totality of all the records which are common to a given subject. The above mentioned records are part of a master employee file, an inventory file, and a stock transaction file.)

Extracting may be required when more than one field occupies a PHILCO 2000 word or when a field is not the sole occupant of a word.

Before starting to program, all pertinent records must be defined. This definition should include a layout which shows the disposition of every field of a record as that field will appear in the computer. The layout is done on a record layout sheet which shows the location of all fields in terms of bits, characters, and words.

The format of a PHILCO 2000 Record Layout is shown below.



RECORD LAYOUT

Record Name

REMARKS

Project

Date

Runs

Remarks

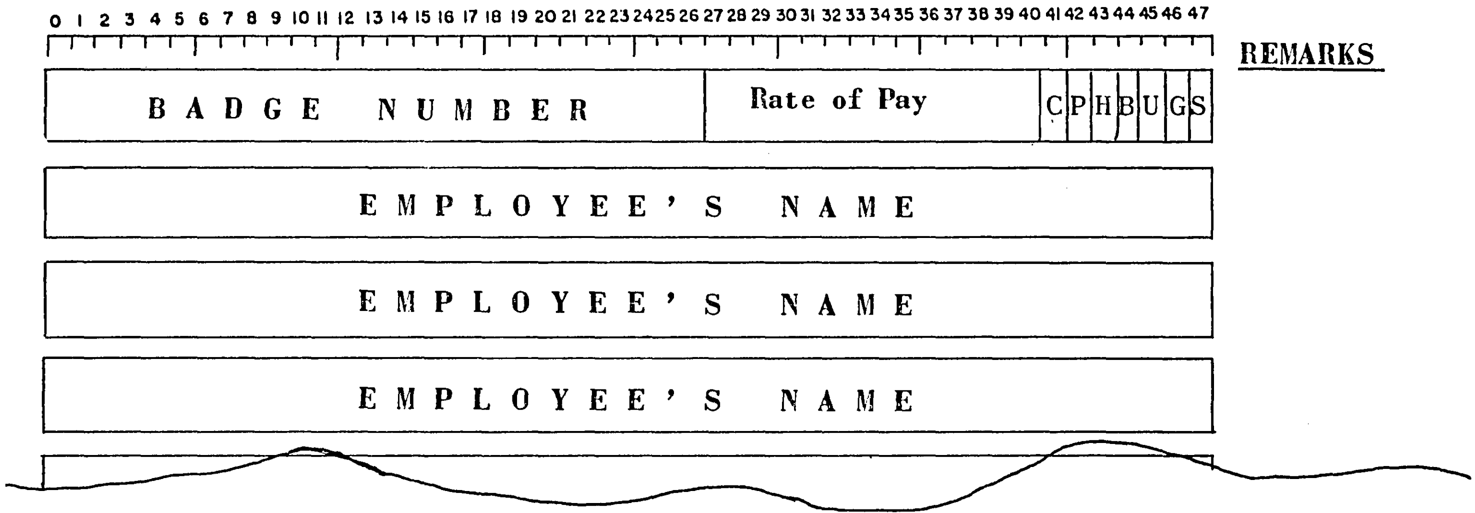
PHILCO® 2000

The numbers at the top and bottom of the record layout specify the bit positions in a word. To simplify the allocation of binary-coded characters, heavy vertical marks separate every six bit positions. Since each rectangle represents a PHILCO 2000 word, a record layout contains as many rectangles as there are words in the record.

A part of a master employee record is shown below.

RECORD LAYOUT

Record Name



Field	Number of Bits	Decimal Equivalent
Badge Number	27	8
Rate of Pay	14	4
Company Club Code	1	-
Pension Code	1	-
Hospitalization Code	1	-
Bond Deduction Code	1	-
Union Membership Code	1	-
Gender Code	1	-
Salary Type Code	1	-
Name (alphanumeric)	3 words	24 characters

The reader must be careful, from this point on, to use the following terms as they are defined:

a. character: one of the 64 six-bit representations for numbers, letters, punctuation symbols, etc.

b. digit: one position or element of a number. In this text, it may be binary, octal, decimal, or hexadecimal.

c. bit: a binary digit.

d. number: a group of digits. The sign of the number is assumed to be positive unless otherwise noted.

To program an application, a tentative record layout which represents the programmer's best estimate of an efficient layout is prepared. Then the program is begun. From this point on, the program and the layout are revised together to make each as efficient as possible. Since one depends on the other, a final record layout cannot be prepared until the program which utilizes the layout has been completed.

Two factors which should be considered when preparing a record layout are the following:

a. Fields which are used together should occupy the same relative positions in words.

b. Fields should be so placed in words as to minimize the amount of shifting and extracting needed to use them.

Note that in order for the final program to operate, all of the data must be recorded on magnetic tape, paper tape, or punched cards in the form specified by the record layout.

The following steps are necessary to prepare original master data in the final record layout form:

a. The final record layout is determined as stated above.

b. Then a record layout is prepared for recording the original data. In it, all fields will be in binary-coded form and most fields will be in separate words.

c. From the layout described in step b, the original data is then recorded on tape via punched cards.

d. From the final record layout, a program is prepared which converts the original data on tape (step c) to the final tape record format. This master data tape is then used in subsequent data processing.

Extract Instructions

The Extract instructions cause the transfer of a desired field without the transfer of extraneous data which surrounds the field. This is accomplished by a masking transfer, i.e., a transfer in which part of the word is masked or covered.

For example, to utilize the Rate of Pay field in the record described previously, it is necessary to isolate it from the Badge Number and the single bit codes. An extraction or masking transfer will mask the latter fields and allow only the Rate of Pay field to be transferred from memory. This is accomplished by placing a mask, composed of binary ones and zeros, in the Q Registèr. The ones must correspond to the Rate of Pay field; the zeros, to the unwanted fields. The Extract instruction causes the masked word to be transferred to the D Register where the Rate of Pay will appear and all other positions will be zero.

This is shown by the following diagrams which illustrate the extraction of the Rate of Pay field. The original contents of the D register are not shown because they are replaced:

Before Extracting

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47	
BADGE NUMBER																											RATE OF PAY														C	P	H	B	U	G	S	Memory location
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	Q Register

After Extracting

BADGE NUMBER																											RATE OF PAY														C	P	H	B	U	G	S	Memory location
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	Q Register
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	RATE OF PAY														0	0	0	0	0	0	0	D Register

The following Extract instructions may be used to transfer desired information:

COMMAND	EXPLANATION
ETD	Extract transfer to D Extract from the specified memory location according to the mask in the Q Register and transfer to the D Register.
ETA	Extract Transfer to A Extract from the specified memory location according to the mask in the Q Register and transfer to the A Register. The D Register receives and retains the extracted field(s).
EA	Extract Add Extract from the specified memory location according to the mask in the Q Register and add to the contents of the A Register. The sum is placed in the A Register. The D Register receives and retains the extracted field(s). (For a floating point addition, the command is FEA.)

ES	Extract Subtract Extract from the specified memory location according to the mask in the Q Register and subtract from the contents of the A Register. The difference is placed in the A Register. The D Register receives and retains the extracted field(s). (For a floating point subtraction the command is FES.)

The following two Extract instructions operate in a slightly different manner. As before, field(s) from a word in memory are extracted and transferred to the D Register. Then the extracted part, corresponding to the ones in the Q Register, is inserted in the A Register without disturbing the remaining positions of the A Register. The composite word in A may then be stored in memory.

COMMAND	EXPLANATION
EI	Extract Insert Extract from the specified memory location and insert in the A Register. The D Register receives and retains the extracted field(s).
EIS	Extract Insert and Store The same as EI. The result is stored in the specified memory location after passing through the D Register.

To illustrate the use of the Extract Insert instruction, assume that the aforementioned employee has a rate of pay change and that the first word of his record has been transferred to the A Register. The new rate must be extracted from some memory location, say RATE, and inserted into the A Register. The status of the registers before and after the Extract Insert instruction (EI RATE) is shown in the following diagrams:

Before executing EI RATE

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26	27 28 29 30 31 32 33 34 35 36 37 38 39 40	41	42	43	44	45	46	47	
OTHER DATA	NEW RATE OF PAY	OTHER DATA							RATE
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0	1 1 1 1 1 1 1 1 1 1 1 1 1 1	0	0	0	0	0	0	0	Q
BADGE NUMBER	OLD RATE OF PAY	C	P	H	B	U	G	S	A

After executing EI RATE

OTHER DATA	NEW RATE OF PAY	OTHER DATA							RATE
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0	1 1 1 1 1 1 1 1 1 1 1 1 1 1	0	0	0	0	0	0	0	Q
BADGE NUMBER	NEW RATE OF PAY	C	P	H	B	U	G	S	A
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0	NEW RATE OF PAY	0	0	0	0	0	0	0	D

With the same initial conditions, the A Register, the D Register, and memory location RATE would all contain the same word after the instruction, EIS RATE, is executed.

Another way to accomplish the insertion of the new rate of pay field is shown in the following diagrams which show the effect of executing the instruction EIS RATE.

Before executing EIS RATE

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26	27 28 29 30 31 32 33 34 35 36 37 38 39 40	41 42 43 44 45 46 47	
B A D G E N U M B E R	OLD RATE OF PAY	C P H B U G S	RATE
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1	0 0 0 0 0 0 0 0 0 0 0 0 0 0	1 1 1 1 1 1 1	Q
O T H E R D A T A	NEW RATE OF PAY	OTHER DATA	A

After executing EIS RATE

B A D G E N U M B E R	NEW RATE OF PAY	C P H B U G S	RATE
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1	0 0 0 0 0 0 0 0 0 0 0 0 0 0	1 1 1 1 1 1 1	Q
B A D G E N U M B E R	NEW RATE OF PAY	C P H B U G S	A
B A D G E N U M B E R	NEW RATE OF PAY	C P H B U G S	D

The reader should review the octal notation for writing masks that was described in Chapter V. The two masks used in the Extract instructions, EI RATE and EIS RATE, are represented octally as 0000000007777600 and 7777777770000177, respectively. Because of the positioning features of TAC constants, 0000000007777600 may be written as 0/77776T41, or as 27/0; 14/1, or as 14/1T40.

Logic of Extract Instructions

The Extract instructions are basically logical bit-by-bit multiplications between the contents of Q and the corresponding bits of the designated memory location. This is actually a logical AND operation. The rules for this operation are

0	1	0	1	Q	Before and after execution
0	0	1	1	M	Before and after execution
0	0	0	1	D	After execution.

This operation forms the basis for all Extract instructions.

OTHER LOGICAL OPERATIONS

Exclusive OR

The following command performs a bit-by-bit exclusive OR operation with two words - one in A and the other in the memory location designated by the address portion of the instruction.

COMMAND	EXPLANATION
AWCS	Add Without Carry and Store: The contents of the A Register are added to the contents of the specified memory location. The sum, without carries, is placed in the D Register and is then transferred to M. The original contents of the A Register are unaltered.

The rules for Addition Without Carry are

0	1	0	1	A	Before and after execution
0	0	1	1	M	Before execution
0	1	1	0	M and D	After execution.

This instruction may be used to alter single bit codes, such as C, P, H, U, G, and S in the preceding illustration. For example, if the employee is a member of the company club (C = 1) and he resigns, a one may be added to the C position, bit position 41, using AWCS. Adding without carry changes C to zero without affecting the next field.

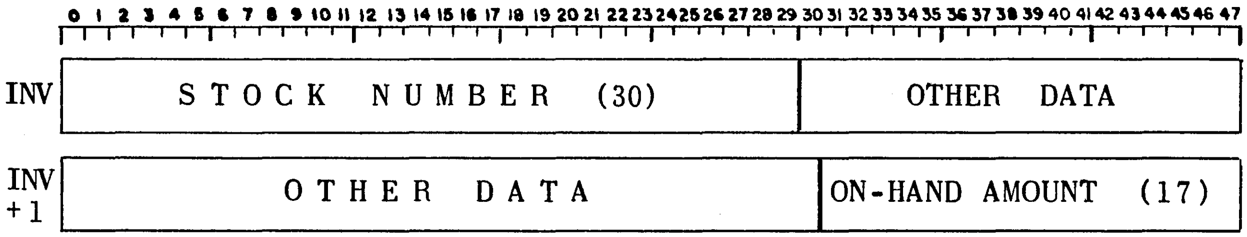
Example 1: Inventory Problem

Parts of an inventory record and a transaction record are stored in memory and have the following format:

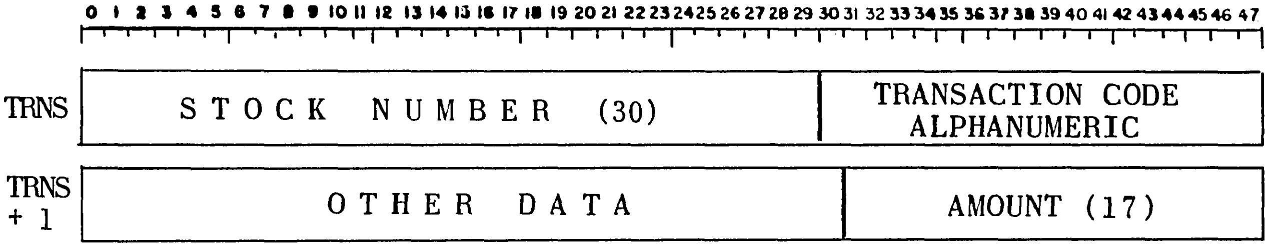
Memory location

Contents

Inventory record



Transaction record



The numbers in parentheses indicate the number of bits in each field. The following processing is to be performed:

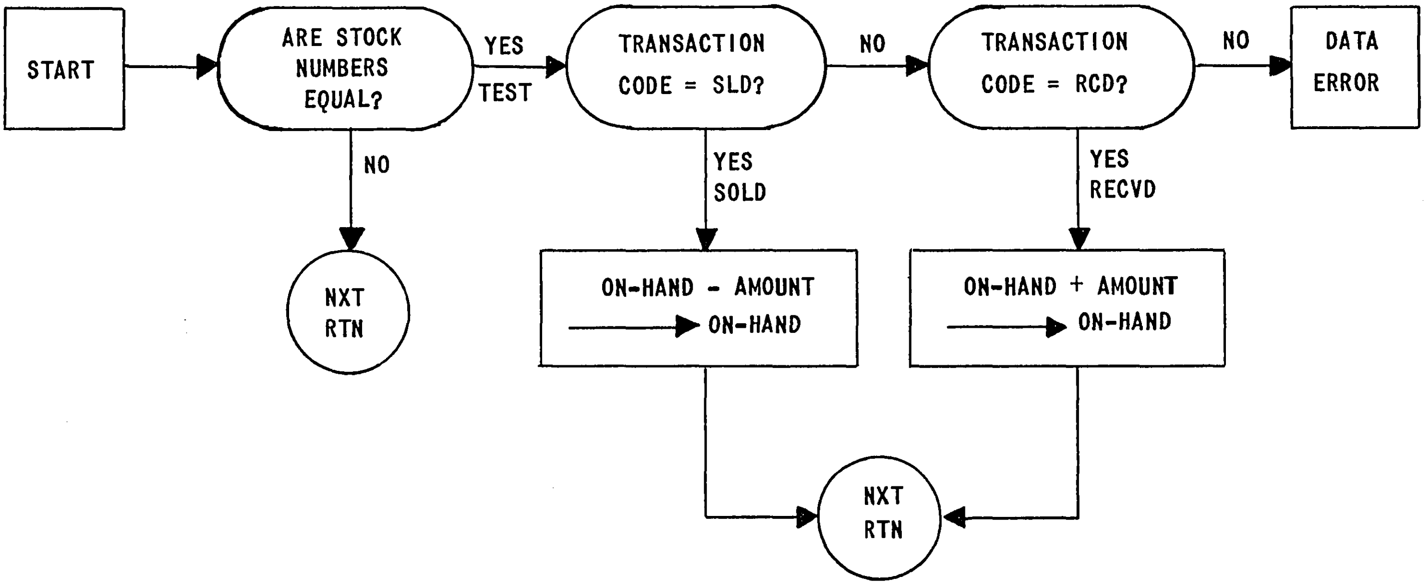
a. Determine if the stock numbers are equal; if they are not, go to NXTRTN.

b. If they are equal, test the transaction code to determine if the transaction indicates an amount sold (SLD) or received (RCD). If the transaction code is not RCD or SLD, the data is in error.

c. For an amount sold, subtract the transaction amount from the on-hand amount. Then go to NXTRTN.

d. For an amount received, add the transaction amount to the on-hand amount. Then go to NXTRTN.

The following is a flowchart of this processing:



The coding for these operations follows:

L	LOCATION	COMMAND	ADDRESS AND REMARKS	
		TMQ	30/1$	30-bit mask → Q
		ETA	INV$	Extract inventory stock number → A
		ETD	TRNS$	Extract Transaction Stock Number → D
		JAED	TEST$	Jump to TEST if (A) = (D)
		JMP	NXTRTN$	
	TEST	TMQ	18/1T47$	18-bit mask → Q
		ETA	TRNS$	Extract Transaction code → A
		TMD	A/00000S	L D $ SLD constant → D
		SRQ	1$	Shift mask for next operations
		JAED	SØLD$	Jump to SLD for sale transaction
		TMD	RCD$	RCD constant → D
		JAED	RECVD$	Jump to RECVD for receipt transaction
		JMP	ERRØR$	If not SLD or RCD, the data is in error
	SØLD	TMA	INV+1$	On-hand → A
		ES	TRNS+1$	Extract and subtract transaction amount
		TAM	INV+1$	New on-hand → memory location INV + 1
		JMP	NXTRTN$	
	RECVD	TMA	INV+1$	On-hand → A
		EA	TRNS+1$	Extract and Add Transaction amount
		TAM	INV+1$	New on-hand → memory location INV + 1
		JMP	NXTRTN$	
	RCD	A/00000R	CD$	

To understand the coding, the reader should recall the methods used by TAC to handle constants. From the instruction TMQ 30/1, TAC produces a constant of 30 ones and 18 zeros. This eventually goes into the constant pool in memory as part of the running program. After the constant is produced, its address is inserted in the TMQ instruction.

The address part of the instruction TMQ 18/1T47 causes TAC to produce a constant which has 18 ones. Because the termination position, T47, is given the 18 ones appear in the right hand part of the word. The remaining positions are zeros.

TMD A/00000SLD is another instruction whose address part is not an address. In this case because of the A/, the code, 00000SLD, is incorporated in the program in binary-coded, alphanumeric form.

TMD RCD is illustrative symbolic addressing in the same way as are JAED TEST, JMP NXTRTN, JAED RECVD, etc. TAC assigns an address to the location RCD and inserts the address in the TMD instruction. The contents of location RCD is the alphanumeric constant 00000RCD.

In a similar manner TAC assigns addresses to symbolic locations INV, INV + 1, TRNS, and TRNS + 1. The addresses are then inserted in the appropriate instructions.

Example 2: Zero Suppression

Memory location WORD contains a number to be printed. It is, therefore, in binary-coded form and contains 8 decimal digits. The number contains at least one non-zero decimal digit, and zeros precede the first such digit. For example, the number might be 00000004, or 01234567, or 00045678, or 87654123.

The number is to be edited for printing without leading zeros. That is, the numbers are to be printed as 4, 1234567, 45678, or 87654123. The non-printing of leading zeros is called zero suppression. To accomplish this, leading zeros must be replaced by non-printing space symbols(Δ: 110000). That is, before the above numbers are printed, they must be changed to Δ Δ Δ Δ Δ Δ Δ 4,Δ 1234567, and Δ Δ Δ 45678. The fourth number is correct as written.

Zero suppression is achieved by comparing each decimal digit (six bits) to zero and then replacing all zeros by spaces. This is accomplished by successive extraction of sequential decimal digits and comparison of the extracted word with zero. When the extracted word becomes non-zero, the number of leading zeros has been determined and is replaced by spaces.

To extract the first decimal digit a mask containing six ones followed by zeros is used. If the first decimal digit is zero, the mask is numerically shifted right six positions, thereby producing a mask with 12 ones. Through the use of this mask, the first two decimal digits are extracted. Since it has already been established that the first decimal digit is zero, the second comparison tests ,only the second digit.

This process continues until a non-zero digit is detected, at which time the mask corresponds to the leading zeros and the first non-zero decimal digit. In order to insert space symbols into the positions containing zeros, the mask is shifted left 6 positions and is then used in the extraction.

The following figure is a flowchart for zero suppression:

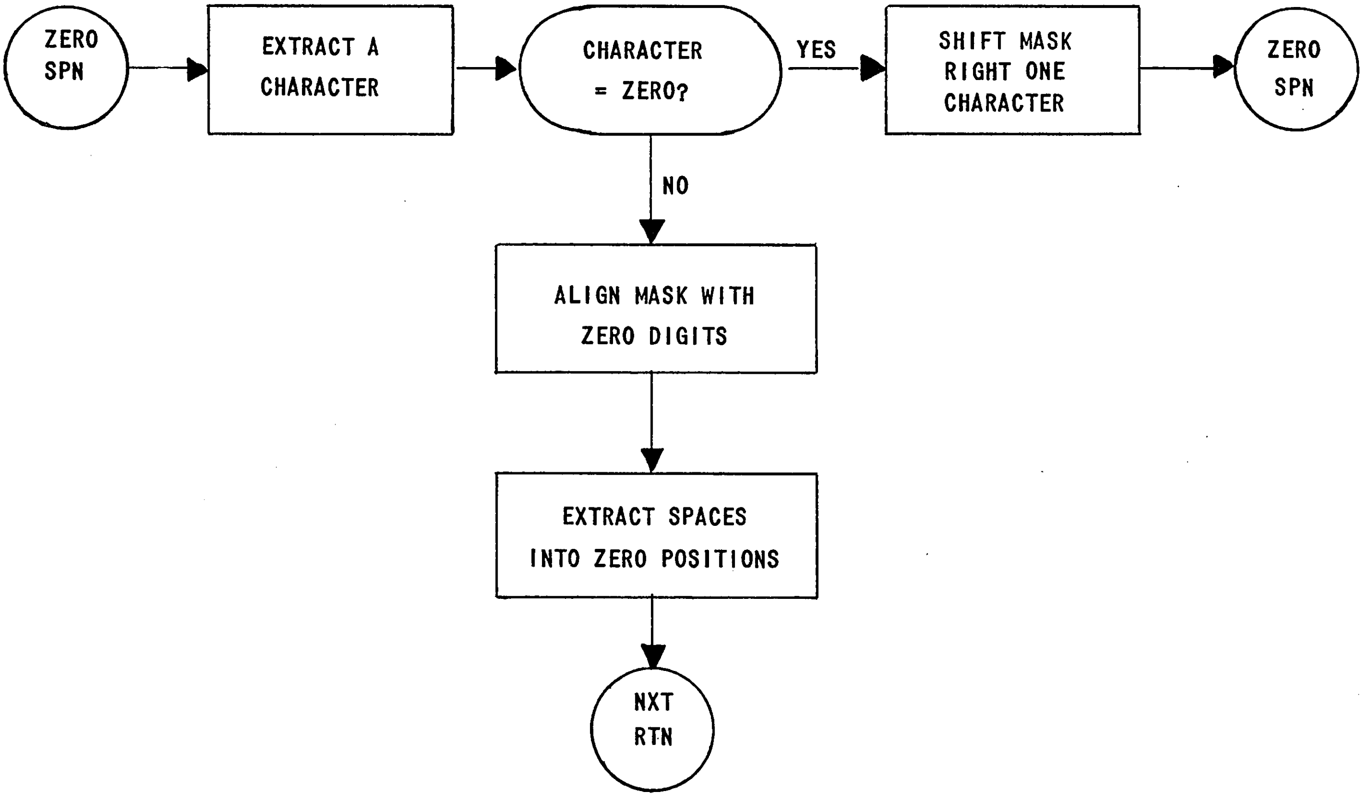

ZERO
SPN
EXTRACT A CHARACTER
CHARACTER = ZERO?
YES
SHIFT MASK RIGHT ONE CHARACTER
ZERO
SPN
NO
ALIGN MASK WITH ZERO DIGITS
EXTRACT SPACES INTO ZERO POSITIONS
NXT
RTN

Figure 7. Flowchart of Zero Suppression

Zero suppression is accomplished by the following coding:

L	LOCATION	COMMAND	ADDRESS AND REMARKS		
		TMQ	6/1$	Six-bit mask → Q	
		JMP	EXTRACT	$ Jump to extract the first digit	
	SHIFT	SRQN	6$	Shift the mask for next digit	creates
	EXTRACT	ETA	WØRD$	Extract a digit → A	the
		JAZ	SHIFT$	Jump if the digit is zero	proper
		SLQ	6$	Align the mask with the zero digits	mask
		TMA	WØRD$	Transfer the number → A	
		EI	SPACES$	Extract spaces into the zero positions	
		TAM	WØRD$	Store the zero suppressed number	
		JMP	NXTRTN$		
	SPACES	A/ΔΔΔΔΔΔ	ΔΔ$		

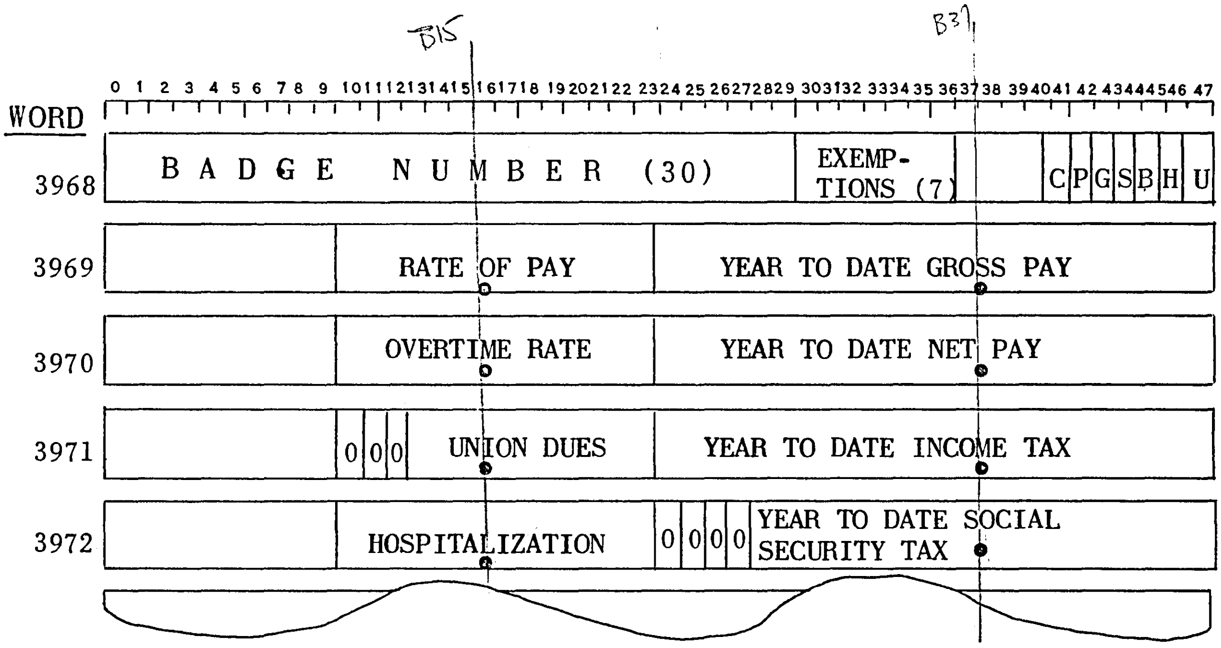
(Note: The instructions SRQN, ETA, and JAZ form a "loop". That is, they are executed as if the instructions were inscribed on a loop with SRQN following JAZ. The topic of loops will be treated in Chapter VIII.)

Two alternative methods may be used to place the space symbols in the word to be printed. The following coding may replace the instructions following the SLQ 6 instruction:

L	LOCATION	COMMAND	ADDRESS AND REMARKS	
		ETD	SPACES$	Spaces corresponding to zeros → D
		DØRMS	WØRD$	Spaces and Number → WORD
		JMP	NXTRTN$	
	SPACES	A/ΔΔΔΔΔΔ	ΔΔ$	
	or			
		ETA	SPACES$	This produces the same result,
		AMS	WØRD$	by addition
		JMP	NXTRTN$	
	SPACES	A/ΔΔΔΔΔΔ	ΔΔ$	

Exercise

Using the following data format, code the Exercise in Chapter III, page 62. All blank fields contain other data.



The hospitalization and union dues deductions are only made if the H and U codes are 1. Assume the Hours Worked to be in word 3840 and any necessary constants to start in word 3000. The positions within words are to be allocated by the reader.

SUMMARY

Shift Instructions

Shifting is necessary to align fields which must be added, subtracted, or compared. It is also necessary for editing, i.e., to place digits and fields in the desired positions for printing, storing on tape, punching cards, etc. Shifting may also be used when multiplying or dividing by a power of two.

The Shift instructions cause the contents of the arithmetic registers to be shifted to the right or to the left. The shift modes are ordinary, numerical, and circular. The chart below represents the types of shifts possible in the Central Computer.

Register	Modes	Directions
A Q A,Q	Ordinary Numeric	Left Right
D	Ordinary Numeric Circular	Right
Q	Circular (conditional jumps)	Left Right

Extract Instructions

Extracting is necessary to isolate a field from one or more other fields in the same PHILCO 2000 word.

The Extract instructions transfer a word from memory to the D Register. All positions are masked except those corresponding to binary ones in the Q Register. The masked positions are made zero. The extracted word may then be added, subtracted, inserted, or transferred to the A Register. The instructions which do this are the following:

ETD:	Extract Transfer to D
ETA:	Extract Transfer to A
EA:	Extract Add
ES:	Extract Subtract
EI:	Extract Insert
EIS:	Extract Insert and Store

Other Logical Operations

Instructions AWCS and DORMS have functions which are similar to the Extract instructions.

AWCS: Add Without Carry and Store

(A) + (M) ——> M, without carries.

DORMS: D or M Stored: a binary one in D or M causes a binary one to be placed in D and M.

Record Layout

A record layout should be made for every record. When preparing the layout, the programmer should attempt to align all fields used together and, whenever possible, to place all fields in positions which minimize shifting and extracting. The record layout then indicates the location of each field and the format for the necessary masks.

Rules of Thumb for Shifting and Extracting

a. Use numerical shifts

   1. on all numbers
   2. whenever it is desired to preserve the sign bit
   3. whenever a mask is to be "generated."

b. Use the Q Register conditional Jump instructions to test single bit codes. Put all single bit codes together at one end of a word in the order of their use.

c. Whenever possible, use a single bit code rather than a code of more than one bit. This will save space in the record and also allow for simple testing of the codes. Establish or modify the codes with instruction AWCS.

d. Align all fields to be used together to reduce the number of masks necessary and the number of times a mask must be placed in the Q Register.

e. Place numbers which are to be multiplied or divided in such positions that the products or quotients do not require shifting.

f. If it is necessary to execute ETA and ETD consecutively, execute ETA first.

g. Placing more than one field in a word saves space on tape but requires additional computer time for extracting. Thought should be given to the placement of fields to minimize the overall processing time. Factors to consider include the frequency of using a field and the amount of magnetic tape saved by packing fields in words.

CHAPTER VII

SUBROUTINES AND PROGRAM SWITCHES

SUBROUTINES

Introduction to Subroutines

In programming, a routine is defined as a series of instructions or operations arranged in the sequence necessary to perform a major function. Typical data processing routines perform payroll operations, inventory operations, data reductions, and simulation studies.

A subroutine is a part of a routine which performs a specified function within the routine. Although this is a simple statement, the use of subroutines in flowcharting and coding is significant in reducing programming time and effort, program testing time, and computer time.

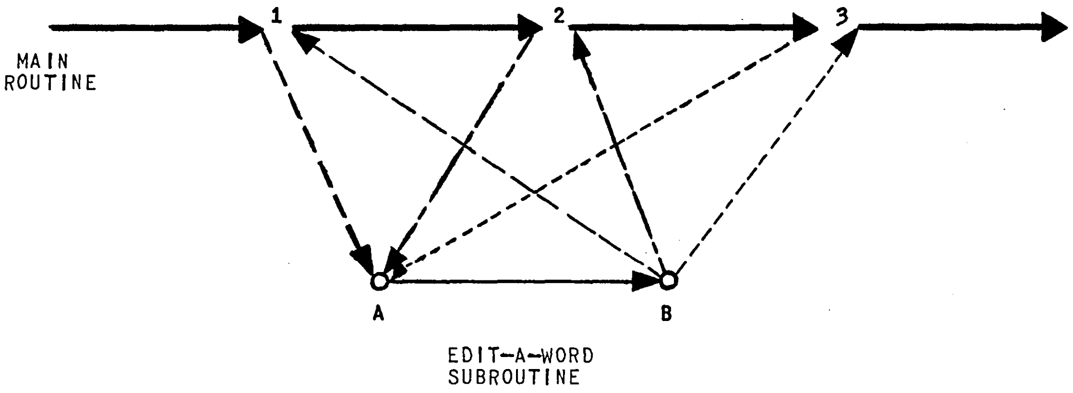
Furthermore, through subroutines, a more logical and convenient approach can be taken towards programming. The programmer can concentrate on the major processing path of a routine and defer programming a minor function by making it a subroutine. The subroutine may then be programmed at a more convenient time.

Typical subroutines in a program calculate the sine of an angle, compute net pay, perform data validity checks, and edit words for printing.

A subroutine which can be used repeatedly from program to program, such as the Calculate a Sine subroutine, is usually recorded on magnetic tape as part of a tape library. Then whenever a program requires a sine calculation, the Sine subroutine need not be coded but merely withdrawn from the tape library. Such subroutines are called library subroutines. The Translator-Assembler-Compiler has a library of subroutines and the facility to incorporate the desired ones in every program.

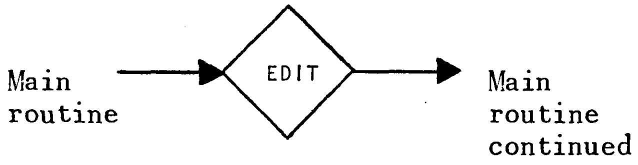
For the purpose of this chapter, subroutines will be restricted to those which may be required in several parts of the same routine or program. Examples of this type of subroutine include Edit-a-word-to-be-printed and Read-a-record subroutines.

The desired end in programming is to program the subroutine once, be able to jump to it whenever its function is to be performed, and enable the subroutine to jump back to the proper place in the program. A graphic illustration of this is shown in the following chart. The heavy line repre represents the main routine. At points 1, 2, and 3 it is necessary to perform an editing function. The broken lines indicate the jumps to and from the subroutine. Point A is the entrance to the subroutine and point B is the exit from the subroutine.

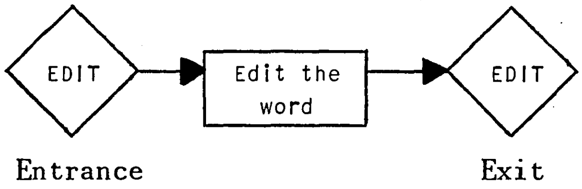


The method of "telling" the subroutine where to return to the main program will be covered when the coding is explained. The flowcharting conventions for subroutine use are shown in the following diagram:

Execute the Edit subroutine.
This diagram corresponds to points 1, or 2, or 3 in the preceding diagram and is a Jump instruction.



The Edit subroutine:



When a subroutine is coded, its name is usually placed in the location column of the first instruction of the subroutine. This location is the entrance or "jump to" location of the subroutine. The last instruction of the subroutine, the exit, is usually an unconditional Jump instruction, JMP. When the exit jump is coded, its address part may be left blank since it will be fabricated each time the subroutine is performed.

The following example illustrates the use of subroutines in flowcharting:

Example

In addition to calculating gross pay, income tax, net pay, etc. in the exercise in Chapter III, Page 62, zero suppress each of these quantities (see Chapter VI, Example 2: Zero Suppression). The flowchart, with a minimum of detail, is shown on the following page. Note: "Convert Word" in the subroutine indicates the conversion from binary to binary-coded form.

Even from this abbreviated flowchart it should be evident that space consuming repetition of the editing function is avoided by incorporating the editing as a subroutine.

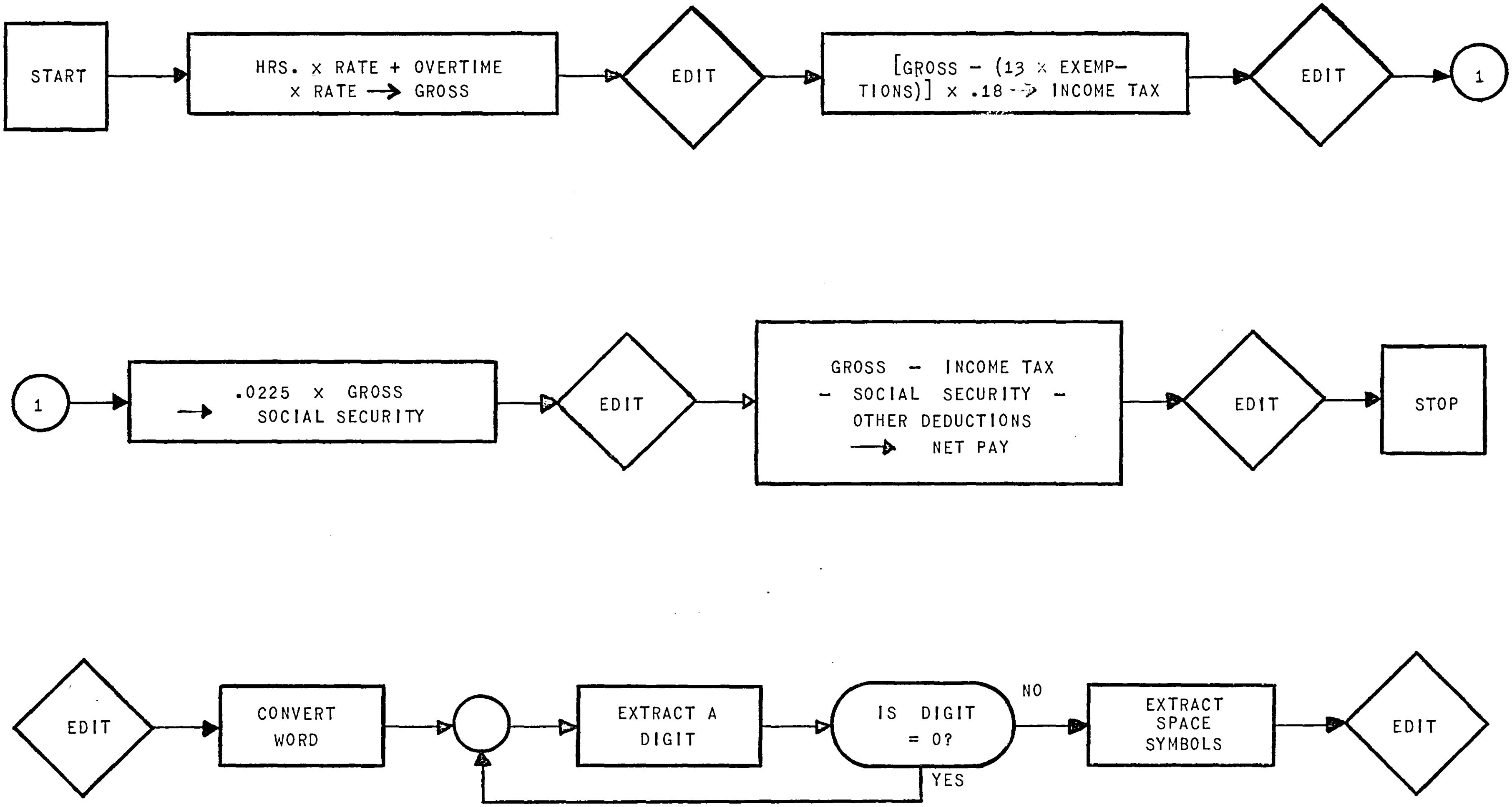

START
HRS. x RATE + OVERTIME x RATE → GROSS
EDIT
[GROSS − (13 × EXEMPTIONS)] x .18 → INCOME TAX
EDIT
1
1
.0225 x GROSS → SOCIAL SECURITY
EDIT
GROSS − INCOME TAX − SOCIAL SECURITY − OTHER DEDUCTIONS → NET PAY
EDIT
STOP
EDIT
CONVERT WORD
EXTRACT A DIGIT
IS DIGIT = 0?
NO
YES
EXTRACT SPACE SYMBOLS
EDIT

Figure 8. Flowchart of Subroutines

The Jump Address Register

Subroutine exits are fabricated by taking advantage of the fact that before every Jump instruction is executed, the address of the next sequential instruction is placed in the JA Register. This Jump Address is the one needed by the subroutine to return to the appropriate place in the program. Thus, the subroutine exit, or return jump, is fabricated by merely storing the contents of JA as the address part of a Jump instruction.

The coding necessary to calculate gross pay and income tax, excluding the zero suppression subroutine, is shown below. It assumes that the subroutine EDIT will convert a number in the A Register to binary-coded form (for printing), edit it, and place the edited number in the A Register. The coding below will store the edited numbers in the successive output data locations OUT and OUT + 1.

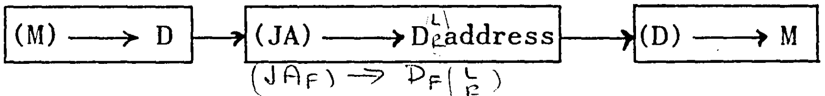
COMMAND	ADDRESS AND REMARKS		
TMQ	3968$	Hours x	
MM	3969$	hourly rate +	Calculate
TMQ	3978$	overtime hours x	Gross
MAD	3970$	overtime rate	Pay
TAM	3979$	→ 3979.	
JMP	EDIT$	Edit gross pay	
TAM	ØUT$	Edited gross pay → OUT	
TMQ	3971$	Exemptions x	
MM	3001$	13	
TAQ		→ Q	Calculate
TMA	3979$	Gross Pay	Income
SQ		- (Exemptions x 13) → A	Tax
TMQ	3002$	18 → Q	
MA		Income tax → A	
TAM	3980$	Income tax → 3980	
JMP	EDIT$	Edit income tax	
TAM	ØUT+1$	Edited income tax → OUT + 1	

The subroutine, EDIT, has to return to the instruction TAM OUT (the seventh instruction) after the subroutine is used for the gross pay. It has to return to the instruction TAM OUT + 1 (the last instruction shown) after the subroutine is used for the income tax. JA receives the addresses of the two TAM instructions when the jumps to EDIT are executed.

The first instruction of the EDIT subroutine establishes the subroutine exit by storing the Jump Address as the address part of its exit jump. The instruction which does this is

TJM: Transfer (JA) to Memory

Transfer the contents of the Jump Address Register to an address part of the word in the specified memory location. After executing TJM the D Register contains the changed word. The flowchart for this instruction is



.

For example, the instruction TJM EXIT in the following coding stores the Jump Address in the address part of the instruction JMP at location EXIT. The following coding is for the subroutine EDIT. The conversion from binary to binary-coded numbers is excluded.

L	LOCATION	COMMAND	ADDRESS AND REMARKS	
	EDIT	TJM	EXIT$	Store the Jump Address in EXIT
		.	.	Convert the number in the
		.	.	A Register to binary-coded form
		.	.	and store it in STRA
		TMQ	6/1$	
		JMP	EXTRACT$	
	SHIFT	SRQN	6$	
	EXTRACT	ETA	STRA$	
		JAZ	SHIFT$	
		SLQ	6$	
		TMA	STRA$	
		EI	A/ΔΔΔΔΔΔΔΔ$	
	EXIT	JMP	....	(Edit Subroutine Exit)

(Note: Instead of coding the conversion in the Edit subroutine a second subroutine, CONVERT, could be specified. This would illustrate the occurrence of a subroutine within a subroutine.)

To illustrate the TJM command in terms of computer code, the code produced by the Translator-Assembler-Compiler assumes that the instructions to compute the gross pay begin in memory location 0100. (See coding on following page.) The first JMP EDIT will appear as the right half instruction of memory location 0102. The corresponding Jump Address is 0103, left half. The next JMP EDIT will appear as the right half instruction of memory location 0107. The corresponding Jump Address is 0108, left half. The Jump Addresses will automatically be placed in JA. Then the TJM EXIT instruction in the subroutine will cause the exit jump to become JMP 0103, left, the first time and JMP 0108, left, the second time.

Note that if a Jump instruction occurs as a left half instruction, the Jump Address will be the same as the address of the Jump instruction but it will be right half. Thus, if the Jump instruction is 1432, left half, the Jump Address is 1432, right half. (See Figure 9.)

In the example illustrating the JQO instruction in Chapter VI, SAVE, HOSP, and UNION could have been subroutines. The coding that follows shows some of the instructions necessary to enable them to function as subroutines.

(Note that these subroutines are probably not called upon from any other point in the program. When the programmer detects a situation like this, he may eliminate the TJM instruction and specify the address of the exit jump.) (See Figure 10.)

PROGRAM SWITCHES

Logical Program Switches

The logical decisions of Chapter III were characterized by the following sequence of steps:

a. A comparison was made.

b. Based on the comparison, one of two operations was performed.

Situations arise in every program in which it is desirable to make comparisons and then to perform some intervening processing between Steps a and b. That is

a. A comparison is made.

b. Some intervening processing is performed.

c. Based on the comparison, one of two operations is performed.

L	LOCATION	COMMAND	ADDRESS AND REMARKS
	0100	TMQ	3968$
		MM	3969$
	0101	TMQ	3978$
		MAD	3970$
	0102	TAM	3979$
		JMP	EDIT$ [First Jump EDIT]
	0103	TAM	ØUT$
		TMQ	3971$
	0104	MM	3001$
		TAQ	
	0105	TMA	3979$
		SQ	
	0106	TMQ	3002$
		MA	
	0107	TAM	3980$
		JMP	EDIT$ [Second Jump EDIT]
	0108	TAM	ØUT+1 $
•	• • • • • • •	• • • • • • • •	• •
	EDIT	TJM	EXIT$
		:	:
		:	:
		TMQ	6/1$
		JMP	EXTRACT$
	SHIFT	SRQN	6$
	EXTRACT	ETA	STRA$
		JAZ	SHIFT$
		SLQ	6$
		TMA	STRA$
		EI	A/ΔΔΔΔΔΔΔΔ$
	EXIT	JMP	• • • • • • • •

Figure 9. Examples of Jump Instructions

L	LOCATION	COMMAND	ADDRESS AND REMARKS	
		TMQ	CØDE$	
		JQØ	SAVE$	
		JQØ	HØSP$	
		JQØ	UNIØN$	
		:	:	Continue processing payroll
		:	:	
	SAVE	TJM	SAVEEX$	Store the Jump Address
		TQM	STRQ$	Store the code word
		:	:	
		:	:	Coding for payroll savings
		:	:	
		TMQ	STRQ$	Replace code word in Q
	SAVEEX	JMP	• • • • • •	Exit from SAVE
	HØSP	TJM	HØSPEX$	Store the Jump Address
		TQM	STRQ$	
		:	:	
		:	:	Coding for hospitalization plan
		:	:	
		TMQ	STRQ$	
	HØSPEX	JMP	• • • • •	Exit from HOSP
	UNIØN	TJM	UNIØNEX$	Store the Jump Address
		TQM	STRQ$	
		:	:	
		:	:	Coding for union members
		TMQ	STRQ$	
	UNIØNEX	JMP	• • • • • •	Exit from UNION

Figure 10. Uses of Jump Instructions

In order that the result of the comparison be "remembered" at Step c, it must be stored after Step a. That is

a. A comparison is made.

b. The result of the comparison is stored.

c. Some intervening processing is performed.

d. Based on the stored result, one of two operations is performed.

These four steps are analogous to a train traveling a length of track which branches at a switch.

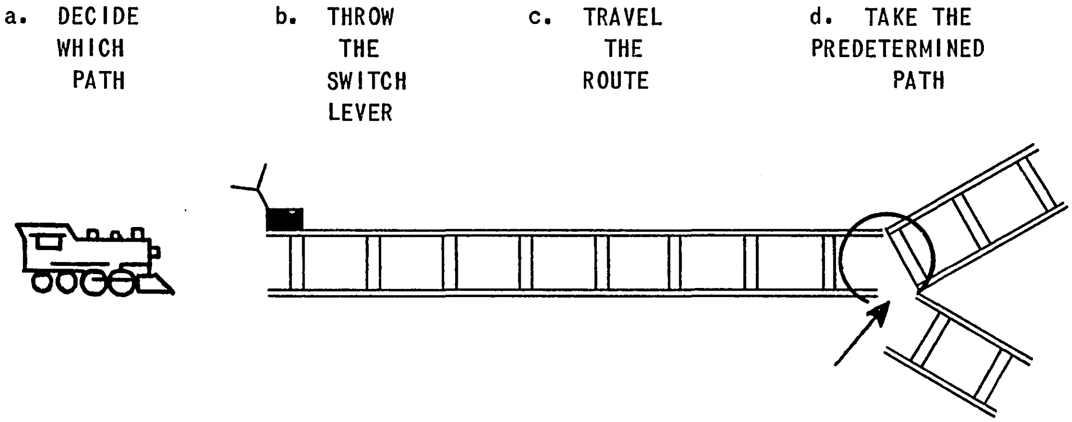
a. Before the train reaches the switch someone must decide which branch the train will take.

b. Accordingly, a lever in the control tower is thrown which places the switch in the proper position.

c. The train travels down the track towards the switch.

d. Finally, it branches according to the setting of the switch.

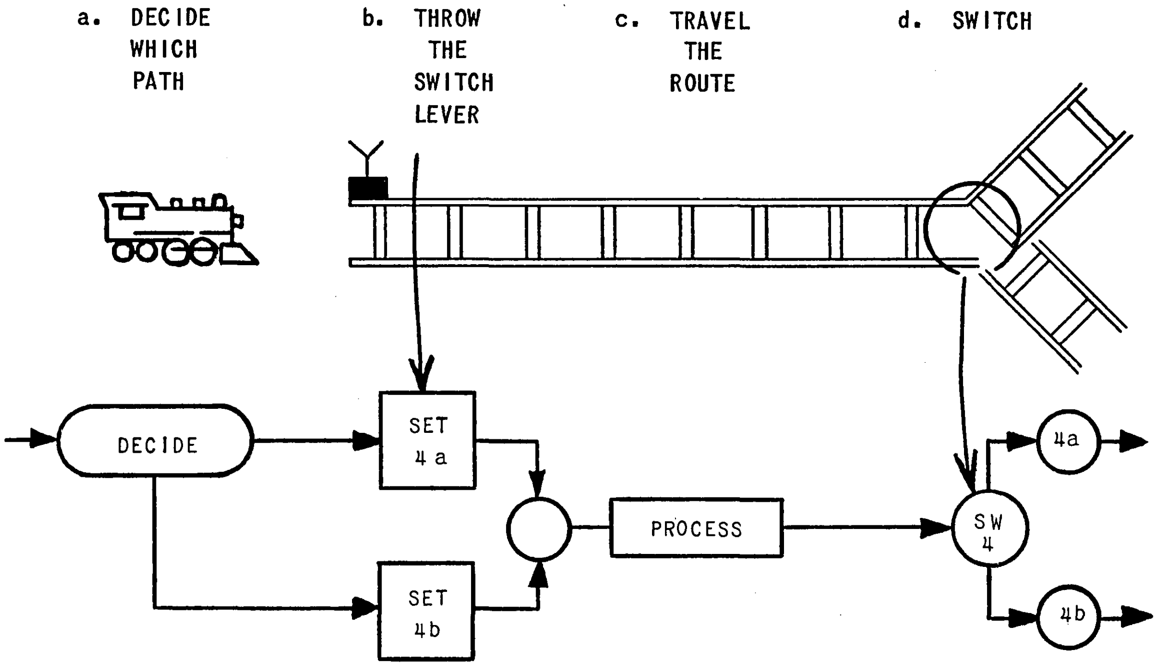
These steps are illustrated by the following diagram:



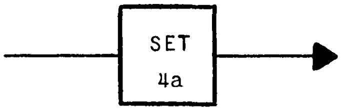
Note that at Step a as many decisions are possible as there are paths from the switch and levers to activate it.

Flowcharting Program Switches

Because of the similarity to the railroad switch the programming counterpart is called a Program Switch. The flowcharting notation parallels the previous railroad illustration as shown below:

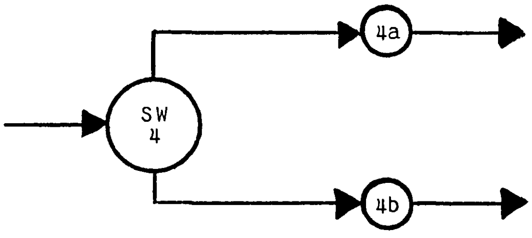


At the point in the flowchart where the "lever" is thrown, a square is drawn to indicate the setting of a switch



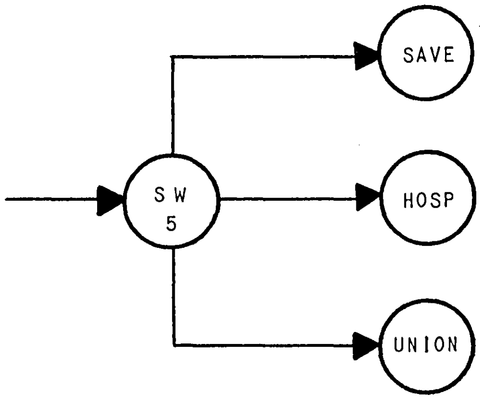
SET SWITCH 4 TO THE "a" PATH.

The switch itself is generally shown as

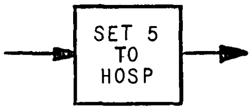


Thus, when the process flow passes through the box Set 4a, the path from Switch 4 is established, and the process flow, upon arriving at the switch, follows path 4a.

In many cases the switch is a Jump instruction. Setting the switch is accomplished by providing the Jump instruction with one of a number of "jump to" addresses. For this reason, the following flowchart symbology is very helpful:



This means that after passing through Switch 5 the process flow proceeds to location SAVE, HOSP, or UNION, depending on the setting of the switch. Switch settings of this type are shown as



Coding Program Switches

The coding for this method of setting switches places an address in the JA Register (without a Jump instruction) and then transfers the address from JA to the address part of a Jump instruction.

The instruction which places an address in JA is

TIJ: Transfer the Instruction address to JA

The address part of the TIJ instruction is placed in JA and replaces the original contents of JA (which may be a Jump Address).

As in subroutine use, the contents of JA are stored by the instruction TJM.

Note that prior to this, all transfer instructions transferred the contents of a register. TIJ transfers a part of a word from a register - the Program Register. Furthermore, in previous transfers which specified a memory address, the contents of that location were transferred or replaced. In TIJ a memory address is specified but the memory location is not affected.

For example, the following instructions cause the Jump instruction at location SWITCH to Jump to location SAVE:

	L	LOCATION	COMMAND	ADDRESS AND REMARKS	
			TIJ	SAVE$	The address SAVE ⟶ JA
			TJM	SWITCH$	(JA) ⟶ address part of SWITCH
*	L	SWITCH	JMP	••••••	This becomes JMP SAVE

*Note that if the TJM and JMP instructions are assembled in the same word by TAC, JMP SAVE is not executed immediately following TJM. (The reader should now review the operations of the control unit in Chapter II.) When TJM is executed in this example, the instruction JMP SAVE is formed in the memory, but the Jump instruction executed is in the Program Register with TJM and will not be JMP SAVE.

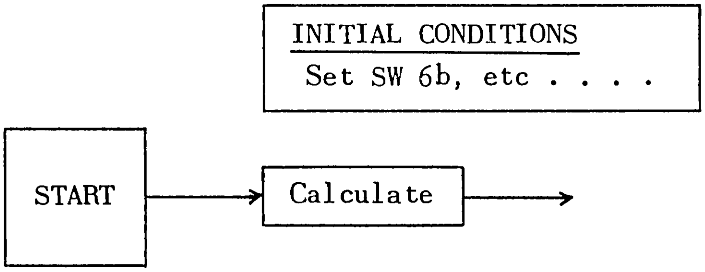
PROGRAM REGISTER

Left half	Right half
TJM SWITCH	JMP ....
↑ This instruction will affect memory location SWITCH, by forming JMP SAVE.	↑ This instruction, the next instruction to be executed, is in PR and will not be affected by TJM.

Therefore, at least one instruction should separate TJM from the JMP instruction. Another solution, as shown in the coding, would be to place an L in the Label column of the JMP instruction. This forces TAC to place JMP . . . . in the left half of a word, thereby making certain that TJM and JMP cannot appear in the same word. Other uses of the Label column will be explained in Chapter VIII.

Other methods of setting switches include modifying "jump to" addresses by addition and subtraction, and replacing one instruction pair by another. Refer to Chapter IX.

Sometimes a switch has an initial setting. That is, at the start of a program, the process flow is to take a particular path from the switch. This fact is indicated by a box over the flowchart. If Switch 6 of a program is to be initially set to the "b" path, the flowchart would contain the following notation:

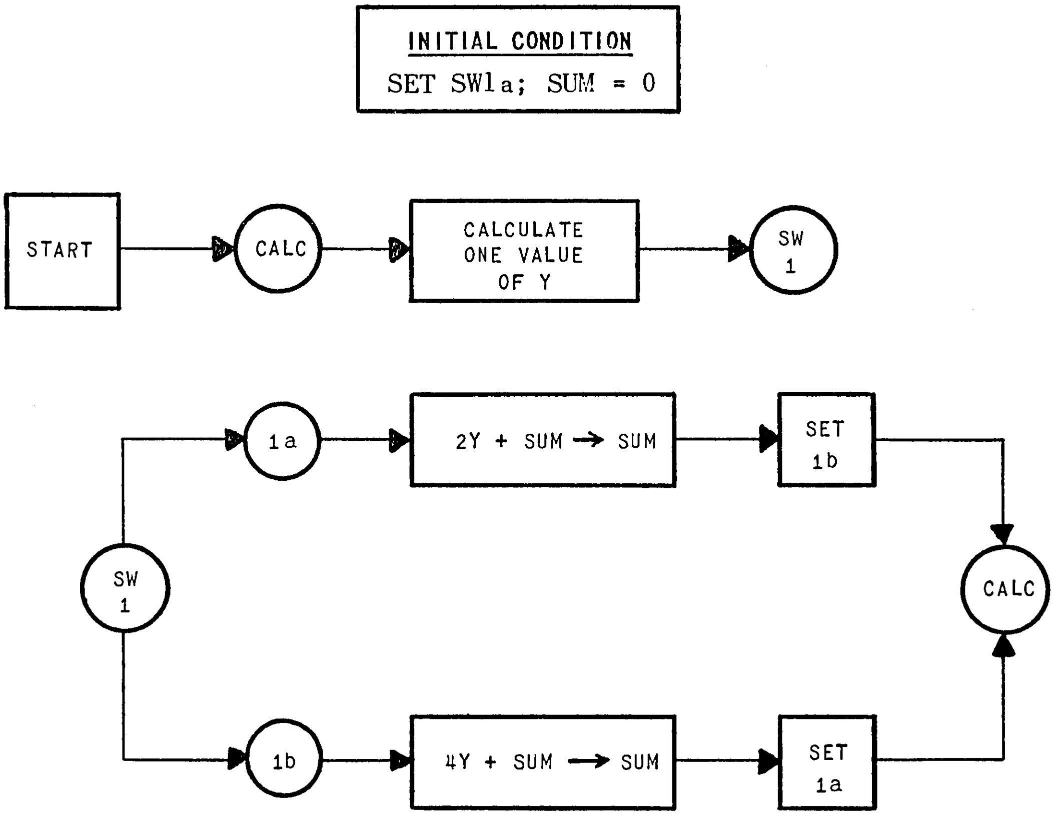


Example 1

The values Y_1, Y_2, Y_3, Y_4, etc.. are to be calculated. (The actual calculation is of no concern to this example.) Then the following summation is to be performed:

$$2Y_1 + 4Y_2 + 2Y_3 + 4Y_4 + \ldots +$$

A program switch is used to alternate between a multiplier of 2 and a multiplier of 4. Note that here the decisions are not made in the program but have been made beforehand.



Note that because of the paper limitation the first SW 1 symbol is used as a connector. Note also that this program has no end. Ending a loop of this kind will be explained in Chapter VIII.

The following coding assumes that the calculated value of Y is placed in the A Register, and the numbers 2 and 4 have their binary points at B4.

L	LOCATION	COMMAND	ADDRESS AND REMARKS	
	CALC	:		
		:		Calculated value of Y
		:		⟶ A Register
		:		
	SW1	JMP	SW1A$	The address of JMP may also be SW1B
	SW1A	TMQ	D/2.0B4$	
		MA		2Y + (SUM) ⟶ SUM
		AMS	SUM$	
		TIJ	SW1B$	Set address at location SW1 to SW1B
		TJM	SW1$	
		JMP	CALC$	
	SW1B	TMQ	D/4.0B4$	
		MA		4Y + (SUM) ⟶ SUM
		AMS	SUM$	
		TIJ	SW1A$	Set address at location SW1 to SW1A
		TJM	SW1$	
		JMP	CALC$	

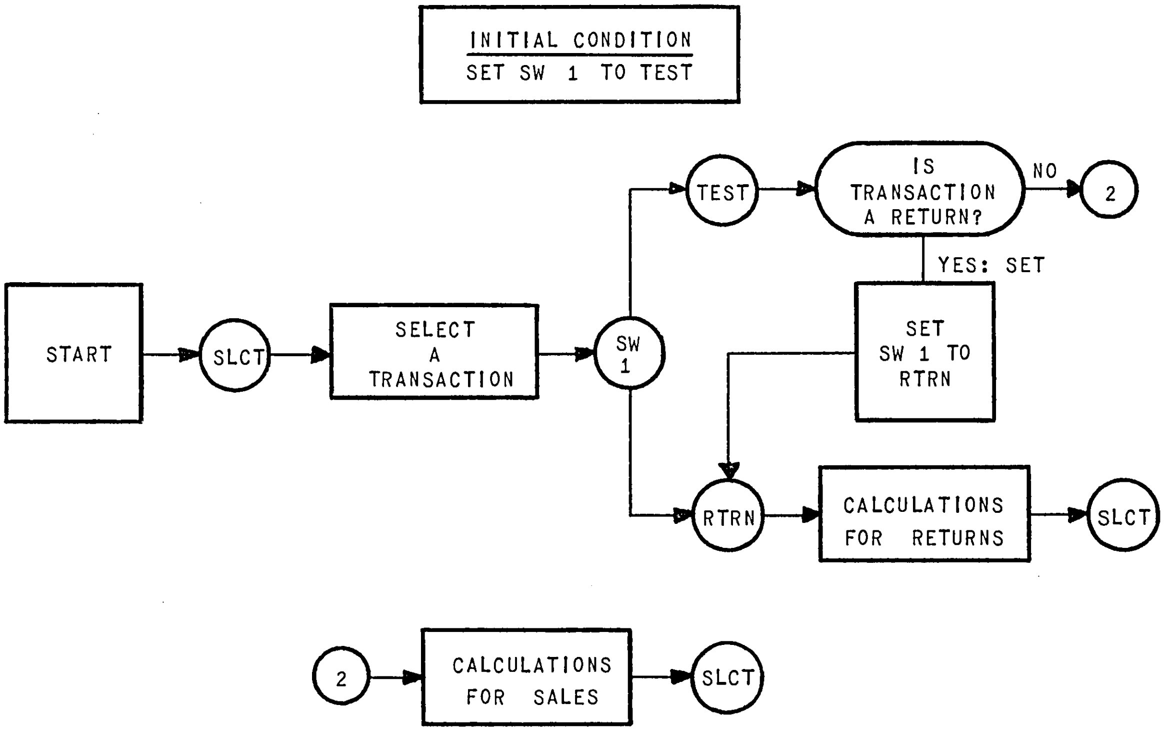
Note that in the example the instructions MA, AMS, TJM, and JMP CALC are repeated in the SW1A path and the SW1B path. These instructions could be grouped together in a subroutine to be used by both switch paths. Although no substantial saving of memory space would be realized in this small example, it illustrates common operations which may be grouped together. In a larger example, common operations should be detected in the flowchart and grouped together for a subsequent saving of memory space.

Example 2

A large number of inventory transaction records, representing quantities sold or returned, are to be processed. A code word in each record distinguishes a sale from a return transaction. Numerous calculations are to be performed for each record and it is desirable to avoid testing every one to determine if it is a sale or return. Therefore, the transactions are grouped so that all of the returns follow all of the sales.

A program switch can be used to "eliminate" a part of a program after it is no longer needed. In this case the test to determine Sale or Return is eliminated when all of the sales have been processed.

An abbreviated flowchart of this process follows:



The initial loop is from SLCT to SW1 to TEST to 2 and back to SLCT. When all of the sales records have been processed, the loop becomes SLCT to SW1 to RTRN and back to SLCT. No exit from the loop is shown.

The coding which accomplishes the switching follows:

L	LOCATION	COMMAND	ADDRESS AND REMARKS		
	SLCT	:	:		
		:	:	Coding to select a	
		:	:	transaction	
		:	:		
	SW1	JMP	TEST$		
	TEST	TMA		Transaction code → A	Is
		TMD		Return code → D	transaction
		JAED	SET$	Compare	a return?
		:	:		
		:	:	Process the sale	
		JMP	SLCT$		
	SET	TIJ	RTRN$	Set the switch	
		TJM	SW1$		
	RTRN	:	:		
		:	:	Process the return	
		JMP	SLCT$		

Note: Methods of selecting a transaction will be explained in the chapter on index registers, Chapter VIII.

One common programming technique combines the use of a program switch and a number of subroutines. The necessity for this combination arises when one of a number of subroutines is to be executed depending on an indicator, key, or code in the data.

For example, a change key may indicate that the master file or a master file record be changed in the following ways:

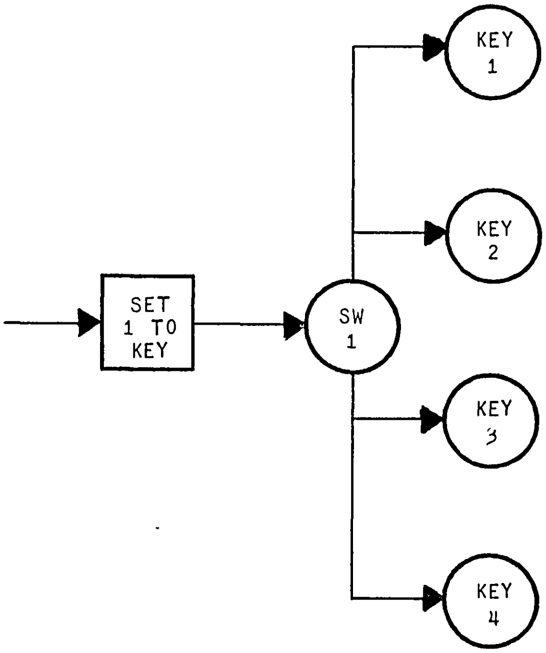
Key 1. substitute new information in a record

Key 2. add or subtract an amount from a record

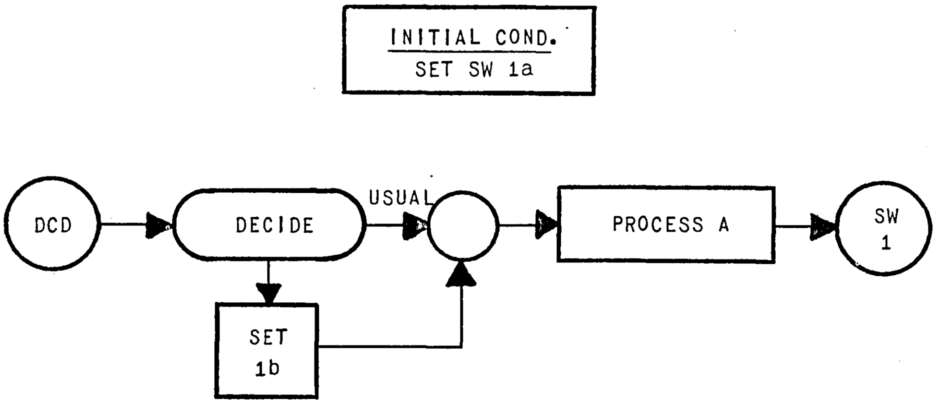
Key 3. delete an entire record

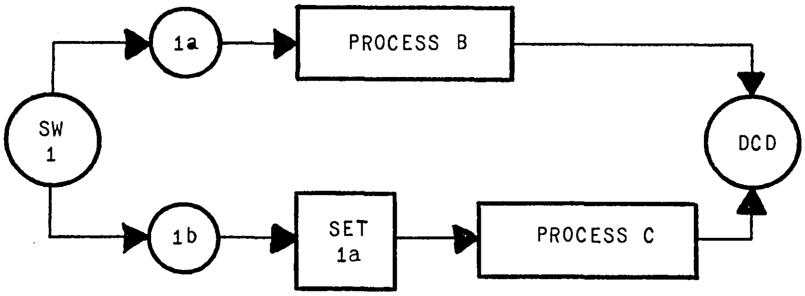
Key 4. place a new record in the file.

This can be accomplished by a Function Table Lookup of Subroutines. The flowchart notation for a four subroutine lookup is as follows:



Another switching method assumes that one path is to be used more frequently than another. The following flowchart illustrates this method:





In general, because fewer instructions are required for a given program, program switches save program running time and/or memory space. Most program switches permit a particular processing path to be followed for a specific interval of time after which a new path is taken, thereby altering the nature of the processing or discontinuing it. As stated in the introduction to program switches, the switch stores or "remembers" the appropriate processing path.

Exercise

Over 100 orders resulted from a promotional scheme by a department store. The amounts of each order have been stored in memory locations ORD, ORD + 1, ORD + 2, ..., etc. Each of the first 100 orders are to be discounted by 10%.

Flowchart a procedure which will process all of the orders and discount only the first 100.

Code only the parts of the process concerned with changing the procedure after the first 100 orders have been processed.

SUMMARY

Subroutines and Program Switches

a. A subroutine is a part of a program which performs a well-defined function.

b. A subroutine is usually entered by a Jump instruction. The first instruction of the subroutine fabricates the exit jump with a TJM instruction.

TJM: Transfer (JA) to Memory

The steps in this operation are:

$$\boxed{(M) \rightarrow D} \rightarrow \boxed{(JA) \rightarrow D\ \text{Address}} \rightarrow \boxed{(D) \rightarrow M}$$

c. Program switches control the flow of processing in a program in a manner similar to that of conditional jump operations. They save time and/or memory space.

d. The simplest method of coding switches is to make the switch a Jump instruction. Setting the switch is accomplished by inserting an address in the address part of the Jump instruction. This is accomplished by the pair of instructions, TIJ and TJM.

TIJ: Transfer the Instruction address to JA

The address part of the TIJ instruction is placed in JA and replaces the original contents.

Rules of Thumb for Subroutines and Program Switches

a. Flowchart and code the main path of a routine and defer all parts which may be prepared in subroutine form.

b. If the subroutine is not entered from more than one point in the program, it may be coded in the main body of the coding. This is called an open subroutine.

c. If several parts of a routine require that the subroutine employ different parameters, the parameters should be in registers just prior to entering the subroutines. If more parameters are needed than there are registers, use memory locations.

d. As a safeguard, perform TJM instructions as soon as possible after the JA Register receives the desired address. The first instruction of a closed subroutine, one which is called from more than one place in the program, should be a TJM instruction.

e. Set switches, which are Jump instructions, by TIJ - TJM pairs of instructions.

f. The TJM instruction functions as described above when it is used to provide addresses for Jump instructions. As will be explained in Chapter IX, TJM will not function as expected when providing addresses for other types of instructions.

Computers which have an Auto-Control Unit employ two special jump instructions: JL and JR. These instructions permit a left or right unconditional jump which does not affect the contents of the JA Register. Because of this facility, the original contents of the JA Register can be restored by the Auto-Control Executive Routine and a return can be made to the main program through use of a JL or JR instruction.

If either of these instructions is used with a computer which does not have an Auto-Control Unit, a Command Fault will occur.

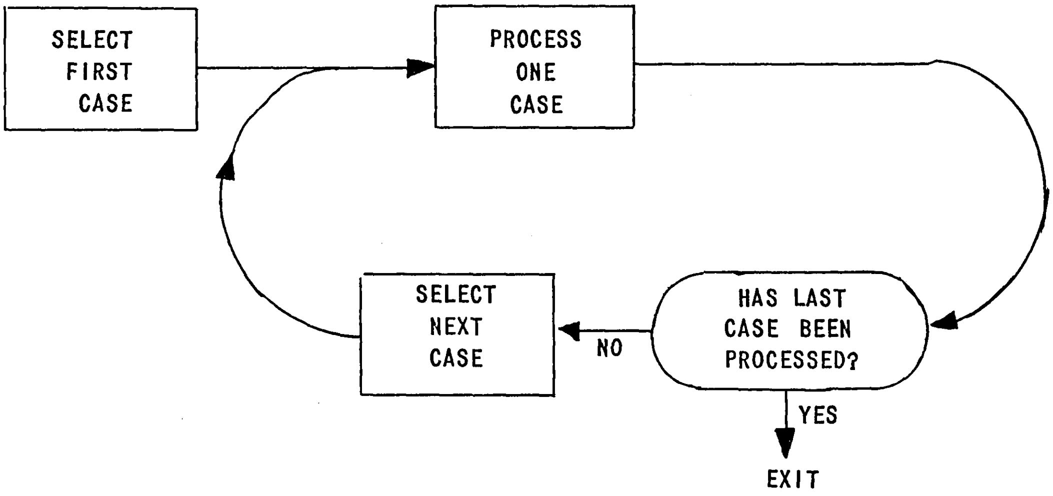
NOTES

CHAPTER VIII

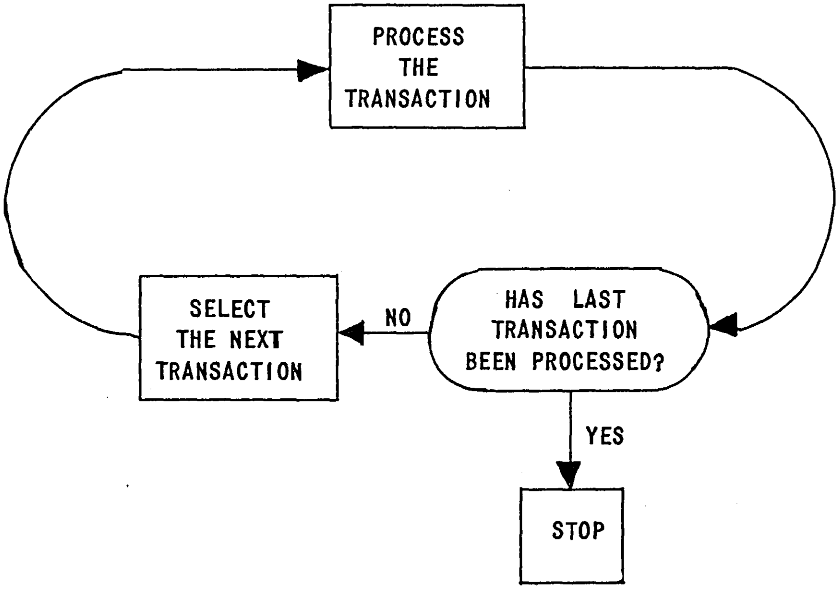
THE LOOP

INTRODUCTION

In the preceding chapters, several references were made to loops. Simply defined, a loop is a group of operations which applies to and is repeated for a number of similar records, words, characters, unknowns, parameters, values, etc. The name "loop" stems from the repetition of operations. This can be shown by a general flowchart of all loops:

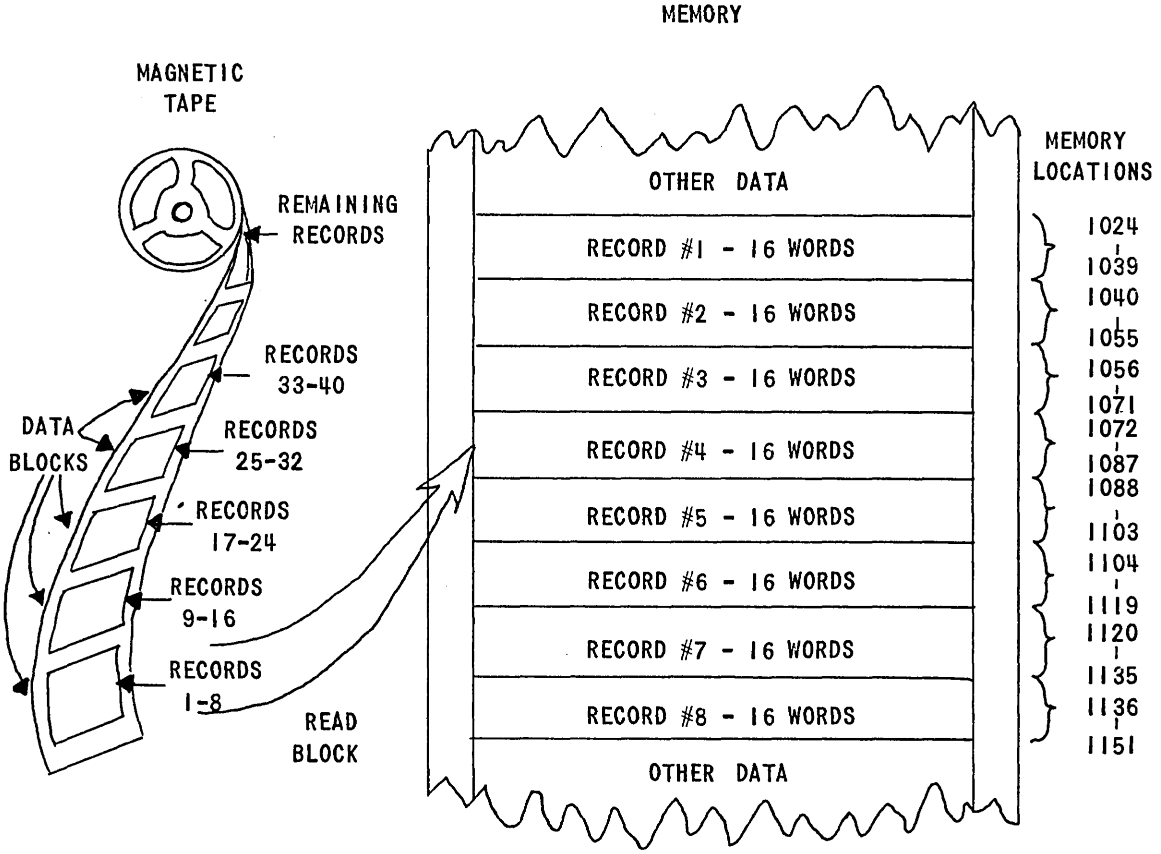


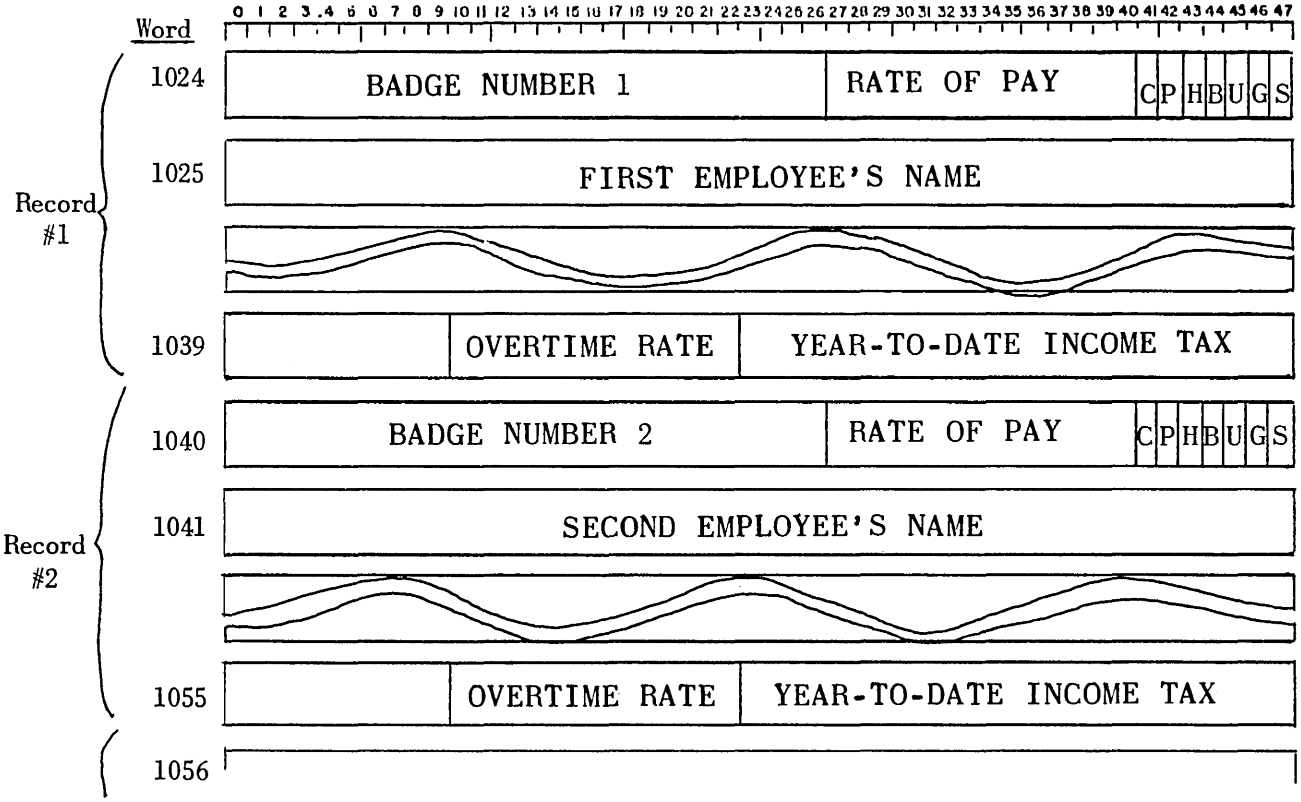
For example, the loop in Example 2, page 211, can be condensed as follows:


PROCESS THE TRANSACTION
HAS LAST TRANSACTION BEEN PROCESSED?
NO
SELECT THE NEXT TRANSACTION
YES
STOP

In order to understand the programming requirements for loops, one must consider the nature of data handling in data processing. Before data can be processed by the PHILCO 2000, it must be recorded on punched cards, punched paper tape, or magnetic tape. For simplicity, and because this chapter is not concerned with input-output, it will be assumed that all data to be processed is recorded on magnetic tape.

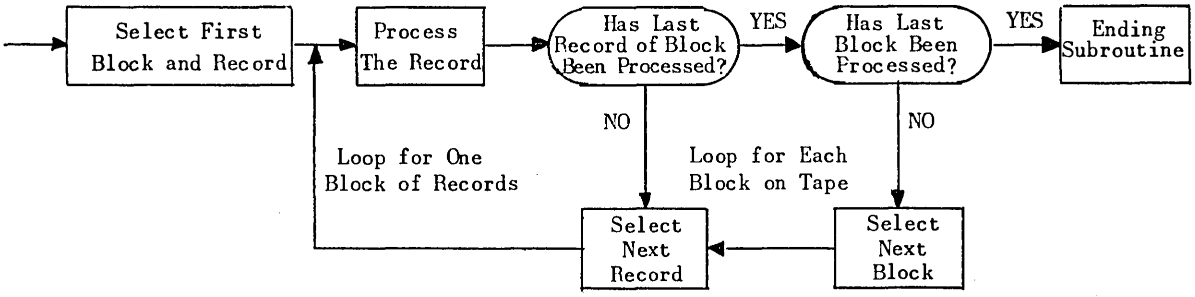
This data must then be read, or transmitted, into the memory. It is read in blocks of 128 words, the block being the unit of data on magnetic tape. In memory the block will occupy 128 consecutive locations. If, for example, a payroll file composed of records of 16 words each were to be processed, one block would contain 8 records. The relations between blocks, words, and records, are shown in the following two diagrams:





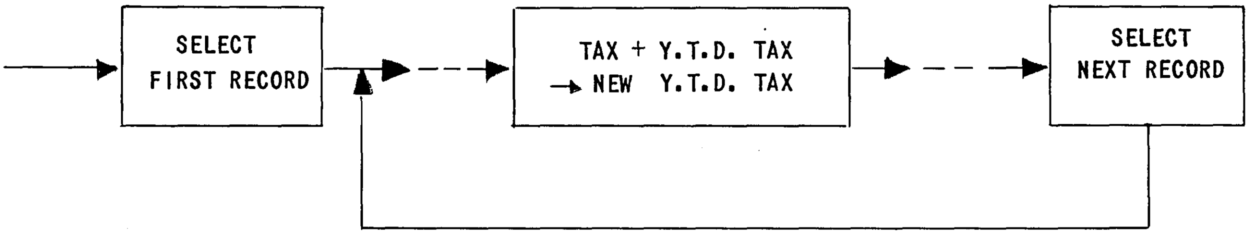
The processing for this data would begin with record number one, which relates to the first employee. Then record number two would be processed, and so on to record number eight. After the eighth record has been processed, the second data block with eight more records, numbered nine to sixteen, is read into memory. Then record nine is processed, record ten, and so on to record sixteen; whereupon the procedure repeats itself for the next block, and the next, until every data block has been processed.

The general flowchart for processing blocks of records has two loops and is as follows:


Select First Block and Record
Process The Record
Has Last Record of Block Been Processed?
YES
Has Last Block Been Processed?
YES
Ending Subroutine
NO
NO
Loop for One Block of Records
Loop for Each Block on Tape
Select Next Record
Select Next Block

Assume that part of the payroll processing is to add this week's income tax, which will be properly aligned in the A Register for each record, to the year-to-date income tax. Also assume that the year-to-date total cannot exceed its allotted 25 bits. The latter assumption avoids the necessity of extracting.

The appropriate parts of the flowchart for this procedure are



The instruction necessary to perform the addition for the first record is AMS 1039, and the necessary instruction for the second record is AMS 1055.

The instructions for the remaining records in the block are the following:

AMS	1071
AMS	1087
AMS	1103
AMS	1119
AMS	1135
AMS	1151

One way to code a program which executes these instructions is to code all of the processing for the first record, then code all of the processing for the second, and so on to the eighth record.

This method, called straight line coding, requires the most memory space and in many data processing situations would require more memory than is available. Note that the instructions are the same for each record but that the addresses differ by a factor of 16 which is the number of words in the record.

Another method is to code the processing for the first record. For all subsequent records, the address parts of all instructions which refer to the first record would be modified by addition. This is illustrated by the following additions:

```
      AMS 1039          Instruction coded for the first record
   +        16
     ----------

      AMS 1055   \  (   Instructions for the second and
   +        16    > {   third records produced by
     ----------  /  (   addition.
      AMS 1071

          .                          .

          .                          .

          .                          .
```

This method may require less memory space but is more time consuming, since an addition must be performed for every instruction which refers to a record in memory.

Still another method is to transfer every record to some area in memory from which it will be processed. This area is called a working storage area. Then the processing is coded for a record in the working storage area. The only instruction addresses which would require modification are those which transfer each subsequent record to the working storage area. This method is preferable to the previous two.

However, the procedure which should be followed wherever possible is to use index registers for address modification.

INDEX REGISTERS

The main purpose of index registers is to provide instructions with the proper addresses in a minimum amount of time and with a minimum of memory space being used.

All the functions of index registers can be summarized as follows:

a. Instruction Address Modification: to modify instruction addresses by addition, as applied to the loops previously mentioned

b. Counting: to count operations that have been performed and to address successive memory locations automatically

c. Instruction Address Substitution: to substitute one address for another in the same way as is done by the TJM instruction. This function will be discussed in Chapter IX.

Index registers are "address sized"; that is, they have a capacity of up to 15 bits and can accommodate addresses and numbers up to 32,767 depending on the memory size of a particular PHILCO 2000 system. The capacity or length of index registers is shown in the following table:

Size of Memory (words)	Capacity of Index Registers (bits)
4096	12
8192	13
16,384	14
32,768	15

Each index register has a counter indicator bit associated with it. The index registers -- there may be from 8 to 32 in one system -- are designated as X_0, X_1, X_2, X_3, ..., X_{31}; and the counter bit for each is designated as X_c.

Almost every instruction can have its address part modified by the addition of the contents of an index register. The sum is called the effective address part and will be an actual or effective memory address or a number, such as the number of places to shift. Instructions which can have their addresses modified in this manner are called indexable. The effective address part of an indexable instruction whose S bit is one is the sum of the contents of a specified index register and the V field of the (computer) instruction. The symbolic notation for this definition is

$$\text{Effective Address Part} = I_V + (X).$$

This sum does not alter the contents of the index register or the instruction in memory.

The instructions which cannot be index register modified are the Repeat, Skip if no Fault, Skip Check and those instructions which have an X in the mnemonic command. All of these instructions will be described later.

If the specified index register is set so that it functions as a counter, it will automatically increase its contents by one every time it is specified by an indexable instruction. An exception to this feature is that counting will not take place if the indexable instruction is executed under the Repeat Mode which will be explained later.

Note that when an index register is not specified by an instruction, its effective address part is the address written, which may be from 0 to 32,767, depending on the size of the memory. In this case the N and V fields of the instruction are combined into a 15-bit field.

If an index register is specified, the maximum address that can be written is limited by the number of bits of the V field. (The bits of the N field are used to address the selected index register.) In turn, the size of the V field is dependent upon the number of index registers in the system. For example, a system having 32 index registers has a five-bit N field and ten-bit V field. Therefore, the maximum address that can be written for an instruction, when an index register is specified, is 1023.

The TAC notation for specifying an index register is a comma after the number or address written in the address part of an instruction, followed by the specified index register number -- 0, 1, 2, 3,..., 31.

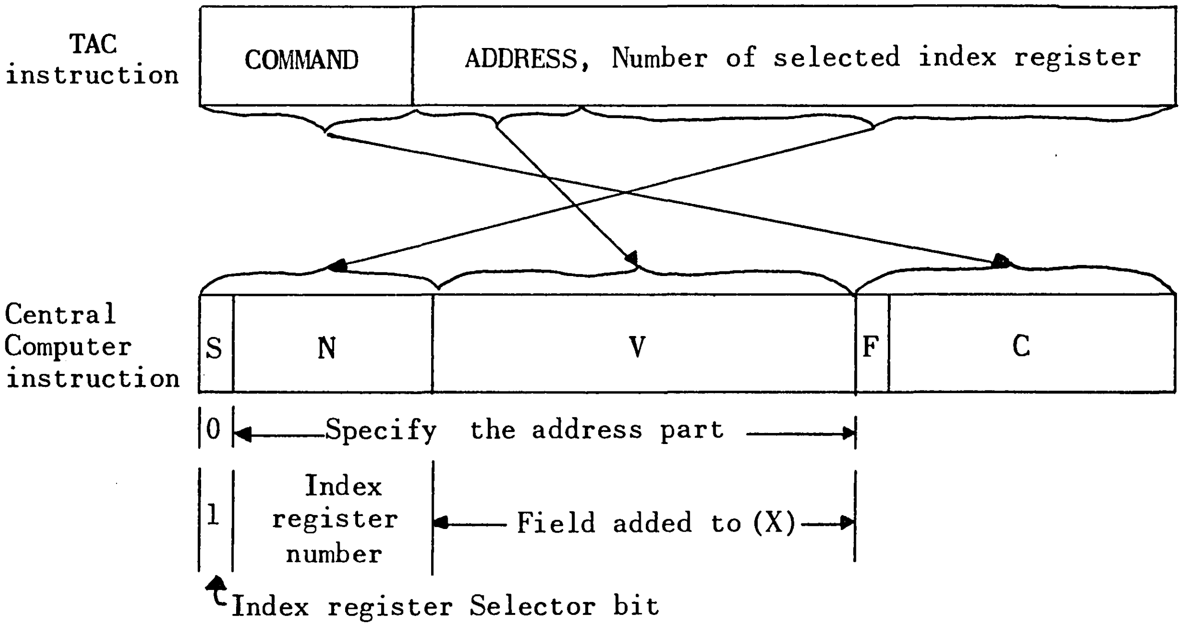
For example, the instruction

TMA 1024,2

specifies that the contents of a memory location, whose address is the sum of 1024 and the contents of Index Register 2, is transferred to the A Register. Thus, the effective address of the TMA instruction is

$$1024 + (X_2).$$

The TAC instruction is related to its Central Computer counterpart as shown below:


TAC instruction
COMMAND
ADDRESS, Number of selected index register
Central Computer instruction
S
N
V
F
C
0
Specify the address part
1
Index register number
Field added to (X)
Index register Selector bit

As indicated in the preceding diagram, if $S = 0$, the N and V fields together specify the address part of the instruction. If $S = 1$, an index register will be selected. In this case, N is the number of the selected index register, and V is the field added to the contents of the index register to produce the effective address. The selector bit S is made one when the TAC instruction calling upon an index register is converted to the corresponding computer instruction.

In the previous example, if Index Register 2 contained the number 16 and the instruction

TMA 1024,2

were executed, the effect would be the same as executing

TMA 1040.

That is,

$$\begin{array}{lll} & \text{TMA} & 1024 \\ & & \underline{+\ \ 16} \\ \text{is equivalent to} & \text{TMA} & 1040. \end{array}$$

If X_2 contains the address 1024, the instruction

TMA 16,2

is also equivalent to TMA 1040.

If the counter indicator of Index Register 2 had been set to one, i.e., to count, it will contain the address 1025 after executing TMA 16,2.

If the index register already contains the desired address, the address part of the instruction is either left blank or a zero is written. Thus if X_2 contains the address 1024, either of the following instructions:

L	LOCATION	COMMAND	ADDRESS AND REMARKS
		TMA	,2$
		or	
		TMA	0,2$

would have the same effect as TMA 1024.

Note that the sum $I_V + (X)$ must be less than or equal to the largest memory address. The sum of the largest memory address and one, therefore, is zero. Thus in a system with 8192 words of memory, the address $8191 + 1 \equiv 0$, the address $8191 + 2 \equiv 1$, etc. This type of memory addressing is usually referred to as being cyclic.

The general method of utilizing index registers will now be explained in relation to processing the payroll records described earlier. The programming procedure should be as follows:

a. Place the address of the first word of the first record of the block in an index register. This address can be written as either an absolute or a symbolic address.

b. The address part of each instruction which refers to a word in the record must contain the number of the word in the record. That is, the number of the first word is zero; therefore, the address part of an instruction referring to it is made zero or left blank. For an instruction referring to the second word, i.e., word number one, its address part would be one. Therefore, for this illustration, address parts of instructions will have values from 0 to 15.

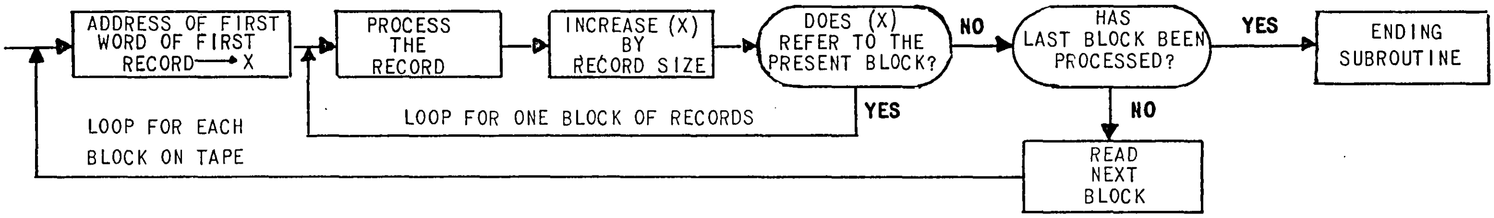
c. After a record has been processed, increase the contents of the index register, i.e., the address of the first word of the present record, by the size of the record -- in this case 16.

d. Test the new contents of the index register to see if the last record of the block has been processed.

   1. If it has been processed, read the next block of data and return to Step 1a.

   2. If it hasn't been processed, process the record and return to Step 3c.

A flowchart of this procedure follows:


ADDRESS OF FIRST WORD OF FIRST RECORD ⟶ X
PROCESS THE RECORD
INCREASE (X) BY RECORD SIZE
DOES (X) REFER TO THE PRESENT BLOCK?
NO
HAS LAST BLOCK BEEN PROCESSED?
YES
ENDING SUBROUTINE
YES
NO
READ NEXT BLOCK
LOOP FOR ONE BLOCK OF RECORDS
LOOP FOR EACH BLOCK ON TAPE

For the payroll records, the address 1024, or an equivalent symbolic address, would be placed in an index register, for instance X_3. The relation between the instruction address parts, the numbers of the words in the record, and the corresponding effective addresses for one record is shown in the following example in which $(X_3) = 1024$.

Instruction Address Part	Word Number	Effective Address: $I_V + (X_3)$
0, 3	0	1024
1, 3	1	1025
2, 3	2	1026
3, 3	3	1027
.	.	.
.	.	.
.	.	.
15, 3	15	1039

As illustrated, all of the address parts written are relative to the contents of the specified index register; i.e., when X_3 contains 1024, the instructions with the above address parts refer to record number 1.

After the contents of X_3 have been increased by 16 to 1040, the instructions refer to the second record. When X_3 contains 1056, the instructions refer to the third record, and so on. After eight records have been processed (the contents of X_3 will have become 1152), the next block of data is read into the same area of memory, i.e., locations 1024 to 1151, and the address 1024 is again placed in the index register. Now instruction address parts (0,3 to 15, 3) refer to the first record of the second block. This procedure is repeated until all the blocks have been processed.

However, before any processing can begin, the desired address must first be placed in the selected index register. Instructions which place addresses in index registers and which manipulate the contents of index registers are called Index Register Instructions. Index register instructions are distinguished from indexable instructions in the following ways:

a. Some index register instructions modify the contents of index registers; no indexable ones do (except for the counting function and under Repeat control).

b. Index register instructions cannot have their addresses modified by the contents of index registers; indexable ones can.

c. Some index register instructions store the contents of index registers in the D Register and JA; no indexable ones do.

PROGRAMMING FOR INDEX REGISTERS

Index Register Instructions

Two index register instructions used to place addresses and numbers in index registers are TIX and TDX.

TIXc: Transfer Instruction address to Index register

The instruction address part, I_V, is transferred to the specified index register. If S is written for c, the counter indicator bit is set to one. If Z is written for c, the counter bit is made zero.

The following coding and diagrams illustrate the effects of this instruction:

L	LOCATION	COMMAND	ADDRESS AND REMARKS	
		TIXS	1024,3$	1024 → X_3, 1 → X_c
		TMA	,3$	(1024) → A register
		TIXZ	,3$	0 → X_3, 0 → X_c

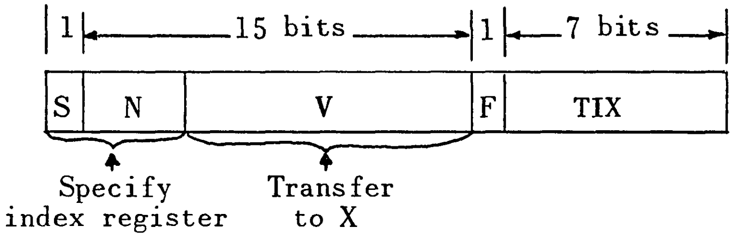
	X_3	X_c
Initial contents of X_3 (assumed)	32 760	0
After executing TIXS 1024,3	01024	1
After executing TMA, 3 [which performs (1024) → A]	01025	1
After executing TIXZ, 3	00000	0

Note that the effect of the last instruction is to clear the index register to zero. This is another use of the TIX instruction.

In the preceding examples, absolute addresses were used to illustrate the mechanics of index registers. However, as stated in a previous chapter, the programmer will most often use symbolic addresses. Thus, the instruction TIXS 1024,3 would normally be written as

```
TIXS    PAYROLL, 3
```

The size of the address part, I_V, transferred by TIX depends on the number of index registers in the system. The reason for this can be seen by examining the format of the computer instruction corresponding to the TIX command. The following chart shows the sizes of the N and V fields.



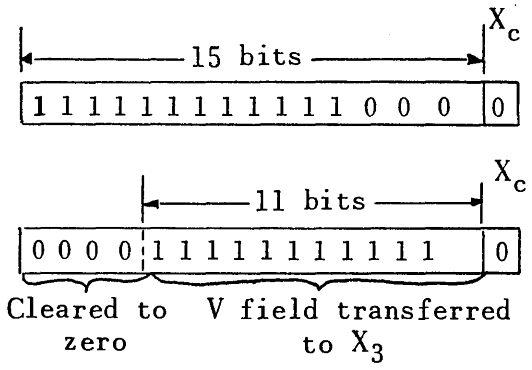
Number of Index Registers in System	Number of Bits in N	Number of Bits in V, Transferred by TIX	Maximum Equivalent Address for I_V
8	3	12	4095
16	4	11	2047
32	5	10	1023

For example, if a PHILCO 2000 system had 16 index registers, the largest address that could be transferred by TIX to the specified index register would be 2047, i.e., an 11 bit V field.

If V bits are less than the capacity of the index register, the high order or leftmost bits of the index register will be cleared to zero. For example, assuming a PHILCO 2000 system having a memory capacity of 32,768 words and 16 index registers, the instruction TIXZ 2047, 3 would have the following effect:

Initial contents of X_3 (assumed to be 32,760)

After executing TIXZ 2047,3



When it is necessary to transfer to an index register an address which requires more bits than are available from I_V, the TIX instruction cannot be used. For example, in the above system, an address greater than 2047 could not be transferred to X_3 by the TIX instruction. Instead, the D Register and the TDX instruction must be used.

Because the D Register is used to place addresses in index registers and to receive addresses from index registers, it is necessary to think of D as containing a pair of instructions. Thus, it will be common to refer to the address part of one half of D and to refer to other parts such as D_S, D_F and so on. Special constants, which will be explained later, are used to specify the address parts of D and the associated F bits (the command parts are of no concern for this use). The TDX instruction is written as follows:

TDXhc: Transfer a D address to Index register

The address part of the "h" half (L or R for left or right) of the contents of the D Register is transferred to the specified index register. If C is written for "c", the counter bit of the index register is replaced by the corresponding F bit of the D Register. The counting function is determined by the F bit. If the "c" entry is left blank, the F bit is not transferred to X_c and the counting function remains the same. The address part of the TDX instruction is ignored by the PHILCO 2000 except for the index register specification.

The following are two examples of the TDX instruction:

COMMAND								ADDRESS AND REMARKS								
T	D	X	L	C				,	2	$						Left half D address ⟶ X_2, D_{LF} ⟶ X_{2c}
T	D	X	R					,	3	$						Right half D address ⟶ X_3

The preceding section was concerned with the methods of loading addresses and numbers into index registers in preparation for their use. The reverse of these operations, transfers from index registers to the D Register, are performed by the TXD instruction.

TXDhc: Transfer from Index to D Register

A field in the specified index register is transferred to the address part of the "h" half (L or R for left or right) of the word in the D Register. If C is written for "c", the counter bit is transferred to the F bit position of the specified half of D. The address part of the TXD instruction is ignored by the PHILCO 2000, except for the index register specification. Only the specified address part of D is affected -- the remaining parts are unaltered.

In addition to these effects of the TXD instruction, it must also be noted that the JA Register is affected. The transfer is actually

$$(X) \longrightarrow JA;\ X_c \longrightarrow JA_F; \quad (JA) \longrightarrow D \text{ address.}$$

If the counter bit is specified by the instruction, the following transfer is made:

$$JA_F \longrightarrow D_F$$

This feature of the TXD instruction will be utilized in Chapter IX under the subject of instruction modification. The TJM instruction is also used to modify addresses. Examples of the TXD instruction are

COMMAND								ADDRESS AND REMARKS								
T	X	D	R	C				,	2	$						$(X_2) \rightarrow$ JA, $X_{2c} \rightarrow JA_F$; (JA),$JA_F \rightarrow$ D right address
T	X	D	L					,	3	$						$(X_3) \rightarrow$ JA; $X_{3c} \rightarrow JA_F$; (JA) $\rightarrow$ left D address

The next two instructions modify the contents of an index register (using the D Register) by addition or subtraction.

ADXh Add a D address to Index register

The address part of the "h" half of the word in the D Register is added to the contents of the specified index register. The sum replaces the original contents of the index register. The "h" may be L or R to specify left or right. Except for the index register specification, the address part of ADX is ignored.

SDXh Subtract a D address from Index register

The address part of the "h" half of the word in the D Register is subtracted from the contents of the specified index register. The difference replaces the original contents of the index register. The "h" may be L or R to specify left or right. Except for the index register specification, the address part of SDX is ignored.

The following four instructions modify the contents of an index register by addition or subraction and then cause a comparison:

AIXJ Add Instruction address to Index register and Jump

The instruction address part, I_V, is added to the contents of the specified index register. The sum replaces the original contents of the index register and is then compared to the address part of the left half of the word in the D Register. If the two are not equal, a jump is effected to the location specified by the address part of the right half of the word in the D Register. If the two are equal, the next instruction selected is the next sequential instruction.

In either case, the address of the next sequential instruction is placed in JA.

SIXJ Subtract Instruction address from Index register and Jump

The instruction address part, I_V, is subtracted from the contents of the specified index register. The difference replaces the original contents of the index register and is then compared to the address part of the left half of the word in the D Register. If the two are not equal, a jump is effected to the location specified by the address part of the right half of the word in the D Register. If the two are equal, the next instruction selected is the next sequential instruction.

In either case, the address of the next sequential instruction is placed in JA.

AIXOh Add Instruction address to Index register and set Overflow

The instruction address part, I_V, is added to the contents of the specified index register. The sum replaces the original contents of the index register and is then compared to the address part of the "h" half of the word in the D Register; "h" may be L or R. If the two are equal, the overflow indicator is set to one. Prior to executing AIXO,the overflow indicator is cleared to zero.

SIXOh Subtract the Instruction address from Index register and set Overflow

The address part of the instruction, I_V, is subtracted from the contents of the specified index register. The difference replaces the original contents of the index register and is then compared to the address part of the "h" half of the word in the D Register; "h" may be L or R. If the two are equal, the overflow indicator is set to one. Prior to executing SIXO,the overflow indicator is cleared to zero.

When using these instructions, the reader should recall the cyclic nature of memory addressing; i.e., in a system with 4096 words of memory, the address 4095 + 1 ≡ 0, and the contents of an index register, 2036, minus 2040 is equivalent to 4092. That is,

$$2036 - 2040 \equiv 4092 = 4096 + 2036 - 2040$$

The following are four examples of the effects of the SIXJ, AIXOR, and SIXOL instructions:

Example 1

	X_1	X_c	S	N	V	F	C	S	N	V	F	C
Initial Conditions	02048	0	0	01919		0	0	0	PROCESS		0	0
					D_L					D_R		
After SIXJ 32,1	02016	0	Jump to PROCESS.									

Example 2

	X_1	X_c	S	N	V	F	C	S	N	V	F	C
Initial Conditions	DATA	0	0	00000		0	0	0	DATA + 128		0	0
After AIXOR 8,1	DATA + 8	0	The overflow indicator is cleared to zero, and it remains zero.									

Example 3

	X_1	X_c	S	N	V	F	C	S	N	V	F	C
Initial Conditions	DATA + 120	0	0	00000		0	0	0	DATA + 128		0	0
After AIXOR 8,1	DATA + 128	0	The overflow indicator is set to 1.									

Example 4

	X_1	X_c	S	N	V	F	C	S	N	V	F	C
Initial Conditions	32767	0	0	32639		0	0	0	00000		0	0
After SIXOL 4,1	32763	0	The overflow indicator is cleared to zero, and it remains zero.									

The normal procedure is to follow an AIXO or SIXO instruction with one of the overflow Jump instructions to determine a course of action based on AIXO or SIXO.

The last index register instruction is TCX which is used when it is desired to establish or alter only the counter indicator of an index register.

TCXci   Transfer Counter to Index register

The counter indicator of the specified index register is set to one if S is written for "c" and zero if Z is written for "c". The "i" may be C or omitted and may only be used with TCXS. TCXSC sets the counter indicator to one and immediately increases the contents of the index register by one.

Note that in Appendix D, the instructions TCXZ and TCXS are not listed with the other index register instructions but are listed in the Special column. TCXSC is not listed at all but its command configuration is the same as that of TCXS. The difference between the two is that the S bit of the instruction is zero for TCXS and one for TCXSC.

TAC AND INDEX REGISTERS

TAC Constants

Before most index register instructions can be used, the programmer must place in the D Register a constant which looks like an instruction word. The constant can be one of two types -- a Location or a Command constant. Either of the two types can be a pool or a non-pool constant. Location constants are used with the instructions TDX, ADX, SDX, AIXO, and SIXO. Command constants may be used with the preceding five instructions and also with the AIXJ and SIXJ instructions.

A Location constant is a word which contains two identical address parts, two like F bits, and zeros everywhere else. The format for this constant is L/LOCATION where LOCATION can be either a symbolic or absolute address which is stored in the address part of both halves of the word.

As a pool constant, this would be written as follows:

L	LOCATION	COMMAND	ADDRESS AND REMARKS
		TMD	L/LØCATIØN$

As a non-pool constant, it would appear as follows:

L	LOCATION							COMMAND								ADDRESS AND REMARKS								
								T	M	D						C	Ø	N	S	T	$			
											⋮													
											⋮													
											⋮													
	C	Ø	N	S	T			L	/	L	Ø	C	A	T	I	Ø	N	$						

The symbolic address, LOCATION, is converted to its actual address by TAC and the resulting constant would appear in memory as follows:

S	N,V	F	C	S	N,V	F	C
0	Actual address of LOCATION	0	0 0 0 0 0 0 0	0	Actual address of LOCATION	0	0 0 0 0 0 0 0

Note that this illustration assumes an F bit of zero.

When a Location constant is used by an index register instruction, it is sometimes necessary to consider the F bits of the constant. The reason for this is that the F bit establishes the counter bit of an index register for the instruction TDXC.

Normally, a symbolic data address, like LOCATION, would be a left half address and D_F would therefore be zero. Note that the F bit of an address indicates left or right half. When F is zero, the address is left half. When F is one, the address is right half. Therefore, to make D_F a one, which would cause a TDXLC instruction to make the index register count, the constant should be written as L/LOCATION + 1H.

LOCATION + 1 is the address of the word following LOCATION.
LOCATION + 1H is the address of the half word following LOCATION.
When LOCATION is a left half address, LOCATION + 1H is a designation for the right half of LOCATION.

The second type of constant placed in the D Register to be used with index register instructions is the Command constant. This is a half-word constant composed of a complete TAC instruction whose parts are separated by commas. An example of a Command constant is C/JMP, NXTRTN, 2.

If this is the complete constant, the Command constant occupies the left half of a computer word and the right half of the word is all zeros. When two constants are written, a semicolon is used to separate them as in the following example:

C/TMA, DATA, 2;C/MP, NXTRTN

The constants occupy a whole word and appear as follows:

S	N	V	F	C	S	N and V	F	C
1	2	Actual address of DATA	0	TMA	0	Actual address of NXTRTN	0	JMP
Left half					Right half			

Usually the programmer is not concerned with the S bit because a combined N and V field of the D Register is used, in which case the S bit is zero. When it is necessary to specify an S bit of one, the Command constant written must be an instruction calling upon an index register.

The C fields of the D Register are not significant when used in index register instructions. However, when an F bit of D must be specified, as in a TDXLC instruction, it is convenient to use instructions such as HLTL or HLTR and JMPL or JMPR in the appropriate Command constant. An L then specifies an F bit of zero and an R specifies an F bit of one. Other instructions with this facility may be found in Appendix C. It should be noted that because of symbolic addressing, L and R are not normally used except in constants. Further use of Command constants will be illustrated in Chapter IX.

To place the address PAYROLL in Index Register 3 and to set the index register to count, the following instructions may be used:

L	LOCATION	COMMAND	ADDRESS AND REMARKS
		TMD	C/HLTR,P AYRØLL $
		TDXLC	,3$
		or	
		TMD	CØNST$
		TDXLC	,3$
		.	.
		.	.
		.	.
	CØNST	C/HLTR,P	AYRØLL$

The use of these index register instructions and their effects upon the registers involved are illustrated in the following paragraphs. The following coding is illustrative only:

COMMAND								ADDRESS AND REMARKS								
T	M	D						C	./	H	L	T	L	,	I	NPUT; C/HLTL, 16 $ Constant → D
T	D	X	L	C				,	1	$						Left D address → X_1; D_{LF}, 0, → X_c
A	D	X	R					,	1	$						(X_1) + Right D address → X_1
T	X	D	L					,	1	$						(X_1) → Left D address part
T	D	M						S	T	R	D	$				(D) → Memory location STRD

The effects on Index Register 1 and the D Register are shown in the following diagrams. Note that TAC converts the symbolic addresses and mnemonic commands before executing the program. They would never appear in the registers as shown.

	Selected Index Register		D Register									
	X_1	X_c	S	N	V	F	C	S	N	V	F	C
Initial conditions	28761	1										
After TMD	28761	1	0	INPUT		0	HLT	0	00016		0	HLT
After TDXLC	INPUT	0	0	INPUT		0	HLT	0	00016		0	HLT
After ADXR	INPUT + 16	0	0	INPUT		0	HLT	0	00016		0	HLT
After TXDL	INPUT + 16	0	0	INPUT + 16		0	HLT	0	00016		0	HLT

Condition of JA:

JA	JA_F
INPUT + 16	0

The four instructions, AIXO, AIXJ, SIXO, SIXJ, enable the programmer to test the contents of an index register against some predetermined limit and then to select one of two processing paths according to the result of the comparison. For example, if the payroll records described earlier were to be processed using index registers, the instruction AIXJ could be used to add to the address in an index register and to determine when one block of records had been processed.

Assuming that the first word of the block is called PAYROLL, the coding for this procedure, following the steps outlined earlier, would be as follows:

L	LOCATION	COMMAND	ADDRESS AND REMARKS
		TMD	L/PAYRØLL $ PAYROLL to address parts of D
		TDXLC	,1$ Left half address of D → X_1, 0 → X_c
	PRØCESS	.	.
		.	.
		.	. All the coding to process one record
		.	.
		.	.
		TMD	C/HLT,PAYRØLL+ 128; C/HLT, PROCESS $
		AIXJ	16,1$ $(X_1)+16 \rightarrow X_1$; Jump to PROCESS if
		.	. $(X_1) \neq$ PAYROLL + 128
		.	. Coding executed after one block
		.	. of records is processed.

Analysis of the Coding

TMD L/PAYROLL transfers to the D Register a constant which has in both address parts the actual computer address represented by PAYROLL. The remainder of the word contains zeros.

TDXLC, 1 places the left half address part of D, PAYROLL, in Index Register 1. The counter indicator is made zero because D_F is zero.

After one record has been processed, 16 is to be added to the address in the index register so that the processing coding will refer to the next record. This is the first function of AIXJ 16,1.

After the addition, the new contents of the index register are compared to the address part of the left half of the D Register. This address is PAYROLL + 128. When the two are not equal, as will be the case for the first seven records, a jump is effected to PROCESS, the address part of the right half of the word in D.

After the eighth record has been processed, the AIXJ instruction causes the contents of X_1 to be increased to PAYROLL + 128. Then the jump of AIXJ is not effected because PAYROLL + 128 equals the left half address part of the word in the D Register.

The status of X_1 during the processing of the block of records is as follows:

	Contents of X_1
Prior to processing 1st record	PAYROLL
After processing 1st record	PAYROLL + 16
After processing 2nd record	PAYROLL + 32
After processing 3rd record	PAYROLL + 48
After processing 4th record	PAYROLL + 64
After processing 5th record	PAYROLL + 80
After processing 6th record	PAYROLL + 96
After processing 7th record	PAYROLL + 112
After processing last record	PAYROLL + 128

TAC and the S Bit of Instructions

The S bit of all instructions specifying an index register is made one by TAC, except for the TCXS instruction. The N bits of any index register instruction specify which index register is to be used. The reduced address field (I_V) is involved as long as the S bit remains one. If the S bit is a zero, the N bits continue to specify which index register is to be used, but the full address field, including possibly some of the N bits, are used by the instruction.

An index register is specified for an instruction by a comma followed by the number of the index register desired. This configuration is written in the address field. If an S bit of zero is desired, no comma or index register number should be written. The most significant bits of the address may then fill the N bits if the address is large enough.

Example 1

In Chapter V potential overflow in division was described and part of the coding to produce a quotient in such a case was shown. Omitted from the example was the coding to count the number of shifts of the dividend. This count is necessary, for example, if the quotient is to be used as an operand in another arithmetic operation. If an addition is to be performed, the other operand must be shifted right to align the binary points. Caution must be exercised to avoid losing significant bits when the shift is effected.

An index register is used as a counter to count the shifts of the dividend and then to provide the effective address for a shift instruction. The instruction NOP, No Operation, is merely a filler instruction whose address part is ignored; it is used simply to cause Index Register 2 to count.

Should the division proceed normally, i.e., if there is no potential overflow, the right shift will be a shift of zero positions. This shift has no effect.

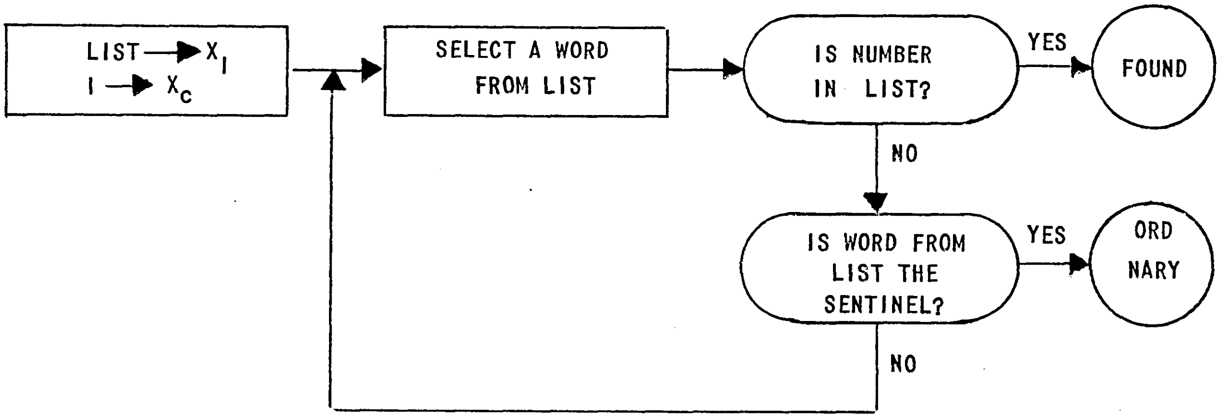
The coding to perform an addition after the division follows:

L	LOCATION	COMMAND	ADDRESS AND REMARKS	
		TIXS	0,2$	Clear X_2 to zero and set it to count
		TMA	DIVIDEND	$ Dividend → A
	DIVIDE	DA	DIVISØR$	Divide
		JNØ	NØØFLØW$	If overflow doesn't occur, jump
		NØP	,2$	$(X_2) + 1 \rightarrow X_2$: Automatic counting
		JMP	DIVIDE$	Jump to divide again
	NØØFLØW	TMA	ØPRAND$	Operand → A
		SRA	,2$	Shift (A) right to align points
		AQ		Perform the addition

Example 2

A list of special account numbers is stored in successive locations in memory such that the first number is in location LIST. There is one account number consisting of eight alphanumeric characters in each location. The size of the list is not known so the sentinel word, END-LIST follows the last account number in the list. An account number is in memory location NUMBER. This example illustrates the technique called table look-up.

The problem is to jump to FOUND if the account in location NUMBER is a special account, i.e., is in the list, or to jump to ORDINARY if the account number is not in the list. A counting index register is used to address the successive locations in the list. A flowchart of this problem follows:



The coding for the problem follows:

L	LOCATION	COMMAND	ADDRESS AND REMARKS	
		TMD	C/HLTR,LIST$	Place address of list in
		TDXLC	,1$	X_1 and set it to count
		TMQ	A/END△LIST$	Sentinel word → Q
	SEARCH	TMA	,1$	One word from list → A
		TMD	NUMBER$	Account number → D
		JAED	FØUND$	Jump if account number is in list
		JAEQ	ØRDNARY$	Jump if word in list is sentinel
		JMP	SEARCH$	Jump to examine next word

After examining the entire list, the index register contains the address of the word following the sentinel because of the counting operation.

If it were desired to place an account number, not in the list, at the end of the list, it must be placed in the sentinel position. The address of the sentinel may be found by subtracting one from the contents of the index register. This may be done by the instructions SIXO or SDX. SIXO is usually more convenient -- assuming that the possible setting of the overflow indicator is of no concern.

The instructions which place the new account number at the end of the list, and then place the sentinel word after it, are shown in the following diagram:

L	LOCATION	COMMAND	ADDRESS AND REMARKS	
	ØRDNARY	SIXØ	1,1$	Account number to last position
		TMD	NUMBER$	in list; $(X_1) + 1 \rightarrow X_1$ (counting)
		TDM	,1$	
		TQM	,1$	END Δ LIST to location following
				last word

If the problem were the same except that two or more words in the list applied to one special account, AIXO or ADX would be used to increase the index register which would not be set to count.

The most frequent type of index register use will be to address successive words or records in a specific area of memory, such as a data block from magnetic tape.

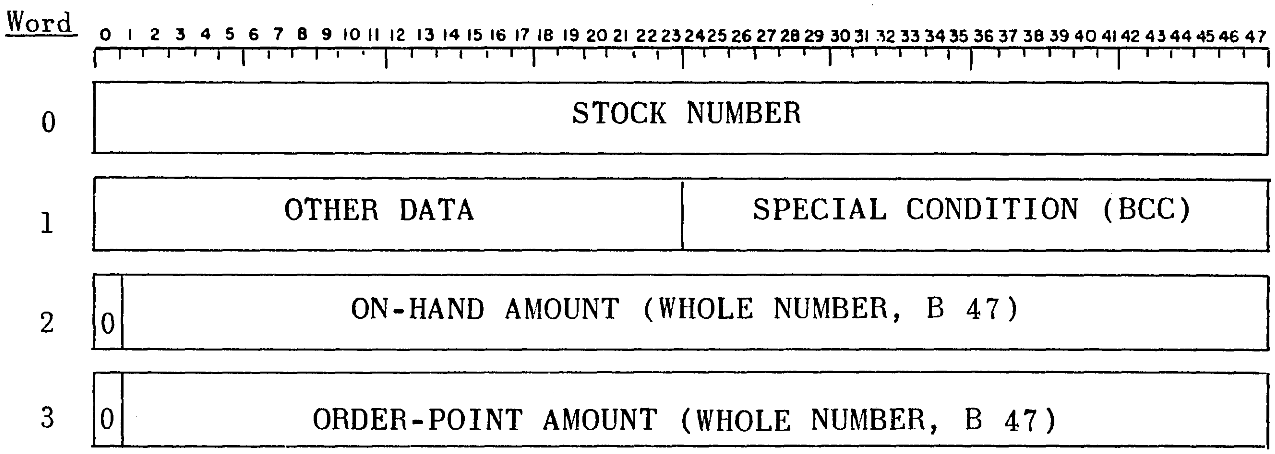
The preceding table look-up for a two-block (256 words) table without a sentinel is coded as follows:

L	LOCATION	COMMAND	ADDRESS AND REMARKS	
		TMD	L/LIST$	List → X_1
		TDXLC	,1$	0 → X_c
		TMQ	NUMBER$	
	SEARCH	TMA	,1$	
		JAEQ	FØUND$	
		TMD	C/HLT,LIST+ 256; C/HLT, SEARCH $	
		AIXJ	1,1$	
				Coding for number
				not in the list

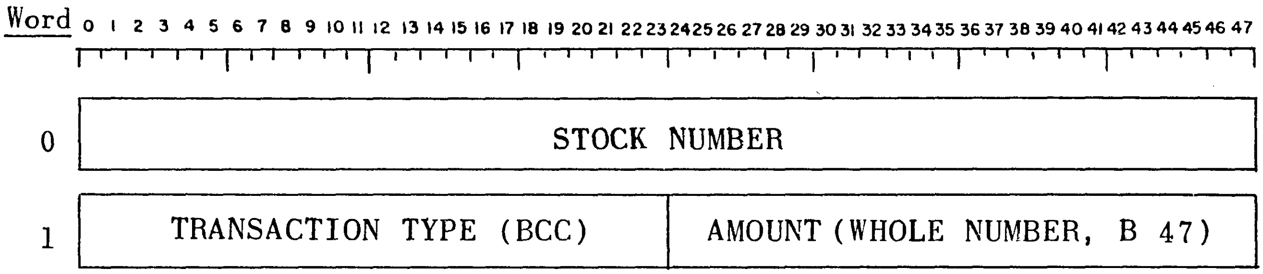
Exercises

1. Using AIXO and an overflow jump instruction, recode the two block table look-up. on preceding page.

2. A data block beginning at location DATA contains 32 inventory records, each of which has the following format:



Another data block beginning at location TRANS contains 64 transaction records, each of which has the following format:



The Special Condition can be HOLD or SHIP and the Transaction Type can be SALE or BACK. All records are in numerical order within the blocks according to the stock numbers.

Match the transactions with the inventory records having the same stock number. There need be no transaction for a given inventory record but there must be an inventory record for every transaction record. However, both matching records need not be in memory at one time; i.e., one may not yet have been read from magnetic tape. Also there may be any number of transaction records to be applied successively to a given inventory record.

When a match is found, determine the transaction type. If the type is SALE, check the Special Condition, and if it is SHIP, subtract the transaction amount from the On-Hand Amount, and replace the old On-Hand Amount with the new. Do not subtract if the Special Condition is HOLD (see below). Then, determine if the new On-Hand Amount is less than or equal to the Order-Point Amount. If this is the case, execute the subroutine ORDER (no coding necessary). In any case, the next operation is to select the next transaction and continue processing.

Change the special condition from SHIP to HOLD.

If the Special Condition is HOLD, execute the subroutine SPECIAL (no coding necessary); by-pass any and all transaction records which apply to the inventory record, and select the next inventory record.

If the Transaction Type is BACK, add the transaction amount to the On-Hand Amount and replace the old On-Hand Amount with the new. Then select the next transaction and continue processing.

— Change the special condition from HOLD to SHIP.

When all inventory records in memory have been processed, execute the subroutine INPUT-I (no coding necessary), which will record the updated inventory records on magnetic tape and place the next block of records, from magnetic tape, in the 128 locations beginning at DATA. When all transaction records in memory have been processed, execute the subroutine INPUT-T (no coding necessary), which will place the next block of transaction records, from magnetic tape, in the 128 locations beginning at TRANS.

Exercise

3. Under <u>certain</u> circumstances the programmer may have an effective address formed by subtracting a number from the contents of an index register. This can be accomplished by using the complement of the number as the instruction address part. For example, in some cases a V part of an instruction equal to 4095 will form an effective address which is one less than the contents of an index register. In other cases a V part of an instruction equal to 8191 will form an effective address which is one less than the contents of an index register.

What are the requirements for two PHILCO 2000 systems where a complement in V is possible; two more where it is not possible?

THE REPEAT INSTRUCTION

The Repeat instruction, RPT, performs a small loop of one or two instructions a specified number of times up to 4095. It is very valuable in transferring records from one area of memory to another, in table look-up, and in sorting.

The definition of the Repeat instruction follows:

RPT — The next instruction or instruction pair following the Repeat instruction is performed the number of times specified by the address part of the Repeat instruction. If the Repeat instruction is a left half instruction, the right half instruction in that word is performed. If the Repeat instruction is a right half instruction, the next pair of instructions is performed. An L or R in the label column specifies the Repeat instruction as a left or right half instruction. The address part can be any number up to 4095. If this number is zero, the Repeat instruction has no effect and the otherwise repeated instructions are ignored. This causes one or two instructions to be skipped.

The Repeat instruction, itself, cannot be index register modified.

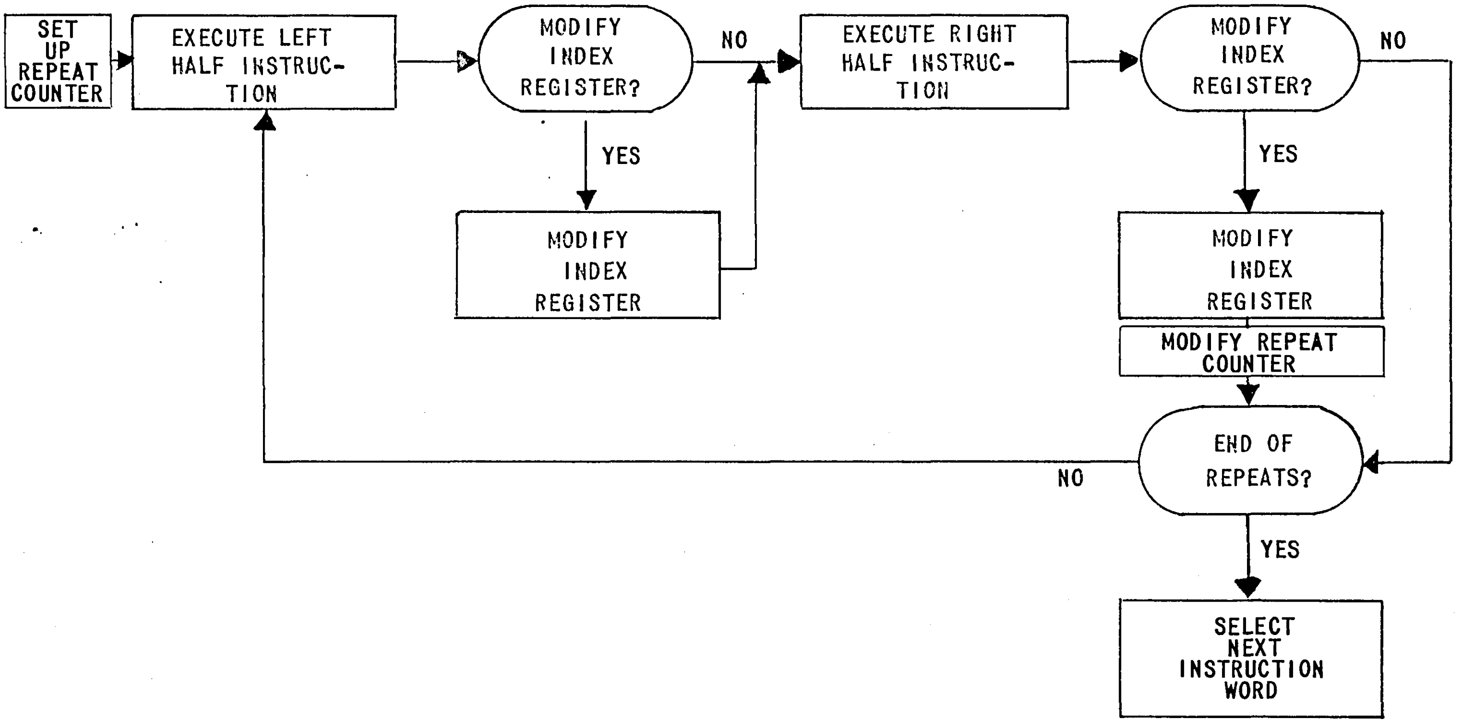
However, the instruction(s) repeated can specify index registers for address modification of either or both addresses in the normal mode, as explained earlier in this chapter, or in the Repeat mode.

Under the Repeat mode the effective address of a repeated instruction is the contents of an index register, if that instruction is under the A or S option (refer to the following page) of the RPT. After the repeated instruction is executed, its address part, I_V, is added to or subtracted from the contents of its index register; the result replaces the original contents of the index register. If an index register is specified under Repeat mode modification the counting function of a specified index register is bypassed, even if the counter bit is set to one.

The following is the format of the two possible Repeat commands:

L	LOCATION							COMMAND							
L								R	P	T	r				
					a	n	d								
R								R	P	T	l	r			

where l and r specify any Repeat mode modification for the left and right half repeated instructions. Note that l and r each may be N, A, or S for No Repeat mode modification. Add I_V to the contents of the index register or Subtract I_V from the contents of the index register. If N is specified, the repeated instruction may specify normal index register use. The counter will be operative. The following flowchart illustrates the loop formed by a Repeat instruction, when two instructions are being repeated:



If an index register is not specified by the repeated instruction, the full address field of that instruction is used as the effective address.

If the A or S option is selected, the computer uses the contents of the index register specified by the N bits of the repeated instruction as the effective address for that instruction. It then modifies the designated index register by that instruction's reduced address field I_V. Note that the computer assumes that if the A or S option of the Repeat is used, an index register is involved. If the repeated instruction does not specify an index register by a 1 in its S bit, the effective address for that instruction becomes 00000. Moreover, the index register designated by the N bits of that instruction is modified by the full address field of the instruction.

Following the completion of the number of performances of an instruction specified by a RPT instruction, the next instruction word is brought to the Program Register, and the program proceeds.

If a RPT with an address field of zero is executed the instruction or instructions which would normally fall under the Repeat are skipped, and the program proceeds with the next instruction word.

If a jump instruction that is located in the left half of an instruction word is being repeated, the number of times remaining to be repeated may be determined from the contents of the JA register. The Repeat Counter is counted by passing its contents through the Control adder, into the MA register, and back into the Repeat Counter. Therefore, an indication of the count is in the MA register every time (except the first) that the repeated instructions are performed. Since a left jump always causes the contents of the MA register to be transferred to the JA register, this indication of the count appears in the JA register after the jump instruction is performed. Therefore, if a jump occurred from the left half of an instruction word being repeated, the number of times remaining to perform the right half instruction may be determined by storing the contents of the JA register, and then solving the formula (JA)-4096 + N, where N is the address portion of the RPT instruction.

If the right half of an instruction word being repeated is a jump instruction, the JA register contains the address of the instruction word which follows the instruction word being repeated. This occurs because a right half jump causes the contents of the PA register to be transmitted to the JA register.

If the repeated instruction is an Index Register instruction, an A or S option on the RPT for that instruction has no effect. The effect is the same as if the N option had been selected. The command parts of the TCX instructions, however, have a bit configuration different from that of other Index Register instructions (refer to Appendix D) and behave, under repeat control, as indexable rather than Index Register instructions.

The following instructions illustrate two Repeat instructions:

L	LOCATION	COMMAND	ADDRESS AND REMARKS	
L		RPTA	48$	Repeat next instruction 48 times; its
				specified index register is increased.
R		RPTNS	16$	Repeat next pair of instructions 16
				times; no modification of an index
				register that may be specified by the left half
				instruction takes place; the index
				register of the right half instruction
				is decreased.

Example 1: Testing the Q Register

One of 48 possible courses of action can be selected by placing a word in the Q Register, in which every bit position represents a different course of action, only one of which is significant, i.e., not zero. Forty-eight Jump instructions are necessary and are stored in consecutive memory locations starting at JUMP. The following coding examines such a word and jumps to the proper one of 48 subroutines.

L	LOCATION	COMMAND	ADDRESS AND REMARKS	
		TIXZ	JUMP,1$	JUMP→X_1, 0→X_c
R		RPTA	48$	Repeat JQO 48 times
		JQØ	1,1$	Jump if (Q) are odd
		NØP.	.	
		.	.	
		.	.	
L	JUMP	JMP	RTN1$	
L		JMP	RTN2$	
L		JMP	RTN3$	
.		.	.	
.		.	.	
.		.	.	
L		JMP	RTN48$	This is the last jump instruction.

Index Register 1 contains the address JUMP initially, JUMP + 1 for the second execution of JQO, JUMP + 2 for the third execution of JQO, and so on up to JUMP + 47. Note that to use TIXZ JUMP, the address of JUMP must be represented in V bits or less.

Example 2: Transfer of Sequential Words

This type of operation is a frequent one in data processing. It is often used to transfer an input record which has been processed to an output data area prior to recording an output block on tape or cards. At other times, when a record may expand or contract because of processing, the input record may be transferred to a working storage area. After the record is processed in working storage, it is transferred to the output data area.

The following coding illustrates the transfer of a 16-word record which starts in location INV and continues to the area starting with location OUT:

L	LOCATION	COMMAND	ADDRESS AND REMARKS	
		TMD	C/HLTL, INV;C/HLTL, ØUT $	Addresses
		TDXLC	,1$	INV ⟶ X_1
		TDXRC	,2$	OUT ⟶ X_2
R		RPTAA	16$	Repeat next 2 instructions 16 times
		TMD	1,1$	(M) ⟶ D, Increment X_1
		TDM TMD	1,2$	(D) ⟶ M, Increment X_2

Each of the Repeat instructions increase the contents of its index register by one so that the effective instructions executed are TMD INV, TMD OUT; TMD INV + 1, TDM OUT + 2; etc. After 16 repetitions of the TMD-TDM pair of instructions, X_1 contains the address INV + 16, and X_2 contains the address OUT + 16.

Example 3: Table Look-Up Under Repeat Control

Assume that a table in memory consists of 64 two-word entries. The first word of the entry is an identifying key and the second word is an associated data field. A key in question is stored in KEY and the table begins at location TABLE. Jump to FOUND if the key in question is in the table.

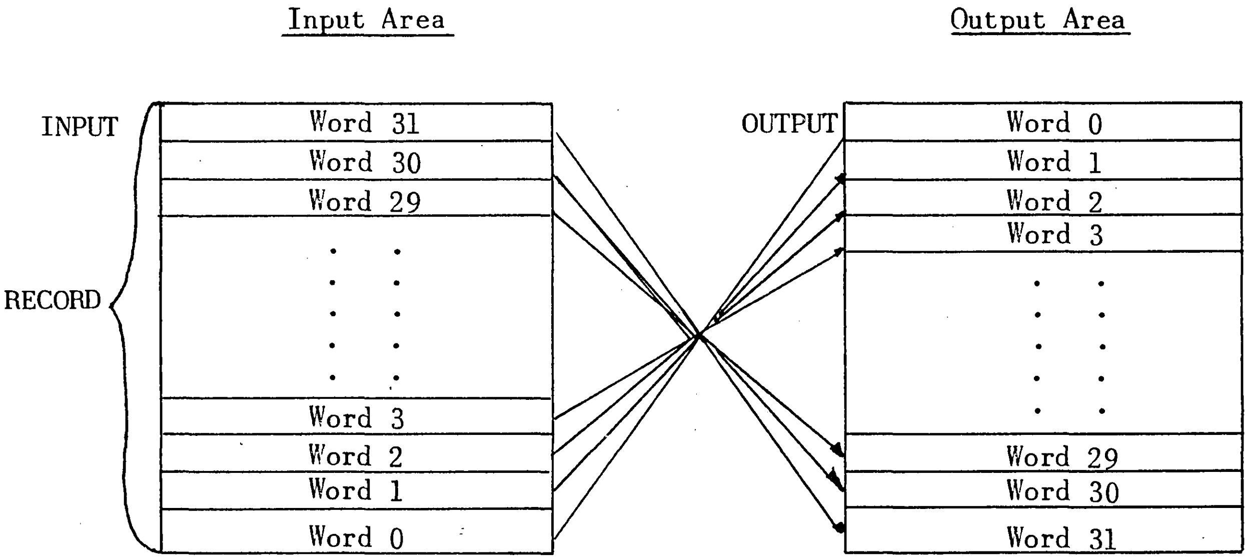
The coding to accomplish this follows:

L	LOCATION	COMMAND	ADDRESS AND REMARKS	
		TMD	L/TABLE$	Address, TABLE ⟶ X_1
		TDXL	,1$	
		TMA	KEY$	Key in question ⟶ A
R		RPTAN	64$	
		TMD	2,1$	Successive keys from table ⟶ D
		JAED	FØUND $	Jump if (A) = (D)
		.	.	
		.	.	If key is not in the table
		.	.	

If the instructions were performed 64 times the effective address of TMD would be TABLE, TABLE + 2, TABLE + 4, and so on up to TABLE + 128 which would be the effective address after the last execution. The instruction would not be performed 64 times if a jump was effected. After an entry in the table is found, the index register contains the address of the first word of the entry, plus two. To utilize the entry, it is necessary to subtract from the contents of the index register. However, table lookup may also be performed by searching the table backwards, i.e., starting with the last key rather than the first. When an entry is found, the index register contains the address of the first word of the entry minus the size of the entry. To utilize the entry, the contents of the index register need not be corrected by an extra instruction. Instead, the address part of the indexable instructions utilizing the entry will contain normal but compensating increments.

Exercise

A record composed of 32 words is to be transferred from one area of memory to another. The order of words in the record is backwards and is to be reversed during the transfer. The following diagram illustrates the memory locations concerned before and after the required transfers:



NOTES ON THE ADDRESS PARTS OF THE D REGISTER

The size of the address part of one half of the word in the D Register is determined in the same way as is the address part of a computer instruction, i.e., by the S bit. An S bit of one indicates that the V part is to be used, and an S bit of zero indicates that the combined V and N parts are used. With an S bit of zero, only as much of N is used as is necessary to represent the largest memory address in the system.

In any system, the combined V and N parts would be as follows:

a. 12 bits for 4096 words of memory

b. 13 bits for 8192 words of memory

c. 14 bits for up to 16,384 words of memory

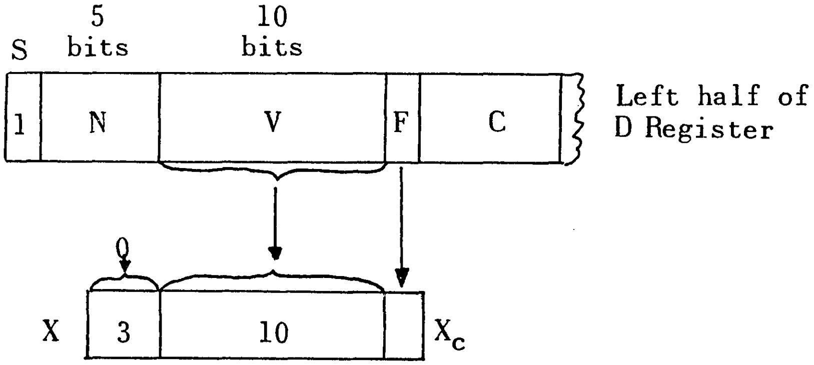
d. 15 bits for up to 32,768 words of memory

Thus, if D_S were one and a TDX instruction were executed, only a V part would be transferred to the index register. Similarly, if a TXD instruction were executed, only V bits would be transferred from the index register to the D Register. When the number of bits transferred to an index register is less than the capacity of the index register, the leftmost bit positions of the index register are cleared to zero.

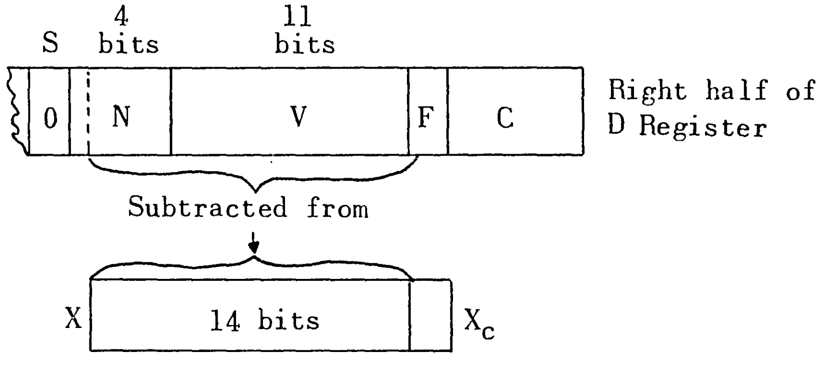
It should be recalled from the preceding examples and exercises that in the majority of cases it is not necessary to be concerned with the size of the D address. However, with some instructions, such as the TXD instruction, the programmer can vary D_S to regulate the size of the address to be placed in the index register. He will usually want to ensure that D_S is zero (by clearing the D Register) in order to transfer the entire contents of the index register. With AIXJ, SIXJ, AIXOh, and SIXOh instructions, the number of bits from the appropriate half of D to be compared to the contents of the specified index register is determined by the S bit of that half of D. Also, with AIXJ or SIXJ, the address to which a jump may be made is designated by the full address field of the right half of D if the S bit of that half is zero or by the reduced address field (with no index register modification) if the S bit is one. ADXH and SDXH modify an index register with the number of bits designated by the D_S bit of the appropriate half of the D Register.

These points are illustrated by the following diagrams and illustrations:

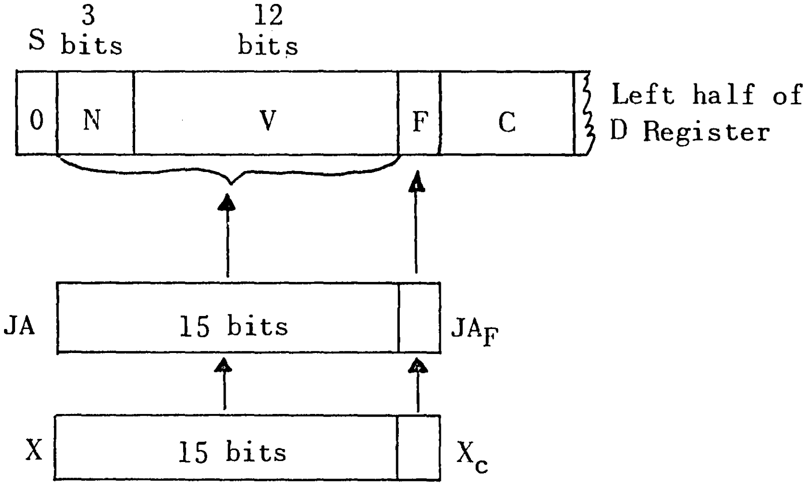
a. TDXLC in a system with 32 index registers and 8192 words of memory (D_{LS} = 1):



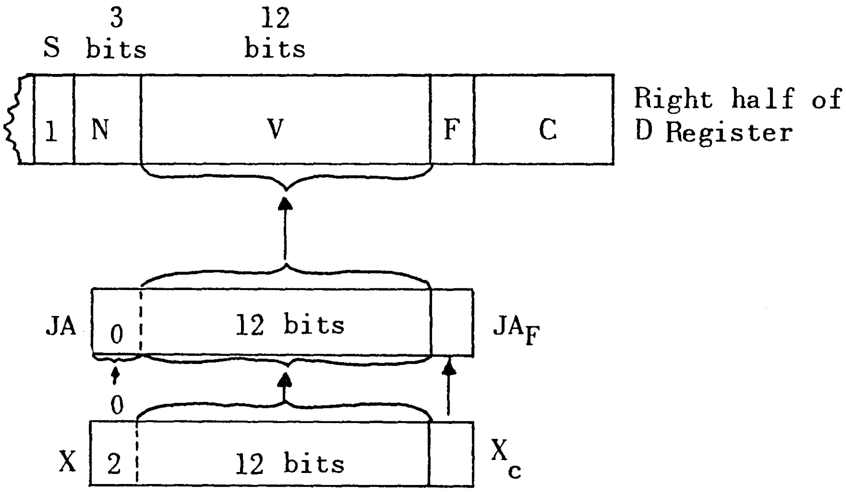
b. SDXR in a system with 16 index registers and 16,384 words of memory (D_{RS} = 0):



c. TXDLC in a system with eight index registers and 32,768 words of memory (D_{LS} = 0):

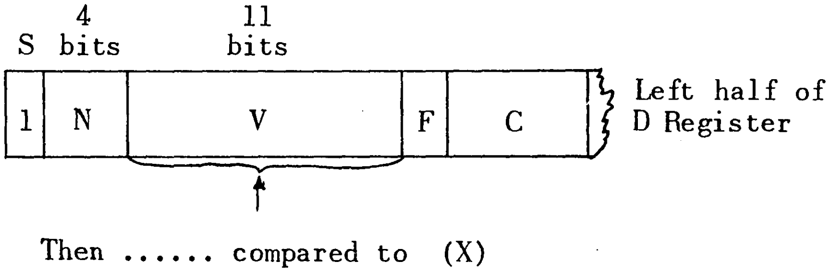


d. TXDR in a system with eight index registers and 16,384 words of memory (D_{RS} = 1):



e. AIXOL in a system with 16 index registers (D_{LS} = 1);

$I_V + (X) \longrightarrow X$:



Although the addressing characteristics in the preceding description were related to index register instructions, they also apply to the instructions TJM and INCA. Further description of these instructions will be found in Chapter IX.

SUMMARY

Index Registers

A Loop is a sequence of operations which is repeated for a group of similar cases.

Index registers provide a convenient method of modifying the addresses of instructions. In a loop the modification is necessary so that the repeated instructions apply to successive records or units of data in memory. All instructions except RPT, SKC, SKF, and the Index Register instructions may utilize index registers for address modification. The modification determines the effective address part of the instruction as $I_V + (X)$.

This sum does not alter the contents of the index register or the instruction in memory.

Index registers have a capacity equivalent to the largest memory address in a system, and a system may have up to 32 index registers. Each index register has a counter indicator which may be set to one or zero. When set to one, the contents of the index register are increased by one every time the register is specified by an indexable instruction and TCXSC, except for instructions executed under the Repeat mode.

The basic functions of the Index Register instructions are the following:

a. transfer the contents of an index register to the D Register

b. transfer the specified address part of the D Register to an index register

c. transfer the address part of the index register instruction to an index register

d. add or subtract the address in the D Register to or from the contents of an index register

e. add or subtract the address part of an Index Register instruction to or from the contents of an index register. Then the new contents of the index register are compared with an address in the D Register; several options are then possible.

The following are the Index Register instructions:

COMMAND	EXPLANATION
TCXci	Transfer Counter to Index For $c = S$, $1 \longrightarrow X_c$; $(X) + 1 \longrightarrow X$ if $i = C$ For $c = Z$, $0 \longrightarrow X_c$.
TIXc	Transfer Instruction address to Index $I_V \longrightarrow X$. For $c = S$, $1 \longrightarrow X_c$; For $c = Z$, $0 \longrightarrow X_c$.
TDXhc	Transfer D address to Index h half D address $\longrightarrow X$. If $c = C$, $D_F \longrightarrow X_c$. h may be L or R.
TXDhc	Transfer from Index to D $(X) \longrightarrow$ h half D address. If $c = C$, $X_c \longrightarrow D_F$. h may be L or R.
ADXh	Add D address to Index h half D address $+ (X) \longrightarrow X$. h may be L or R.
SDXh	Subtract D address from Index $(X) -$ h half D address $\longrightarrow X$. h may be L or R.
AIXJ	Add Instruction address to Index and Jump $(X) + I_V \longrightarrow X$. If (X) do not equal the left half D address, a jump is effected to the location specified by the right half D address.
SIXJ	Subtract Instruction address from Index and Jump $(X) - I_V \longrightarrow X$. If (X) do not equal the left half D address, a jump is effected to the location specified by the right half D address.

COMMAND	EXPLANATION
AIXOh	Add Instruction address to Index and set Overflow $(X) + I_V \rightarrow X$. The overflow indicator is set to one if (X) equal the h half D address. h may be L or R.
SIXOh	Subtract Instruction address from Index and set Overflow $(X) - I_V \rightarrow X$. The overflow indicator is set to one if (X) equal the h half D address. h may be L or R.

Repeat Instruction

The Repeat instruction permits the repetition of one or a pair of instructions up to 4095 times. It is very useful for record transfers, a table look-up, and internal sorting.

RPT lr	The next instruction or instruction pair is performed the number of times specified by I_V. A left half RPT repeats one instruction, whereas a right half RPT repeats a pair of instructions. The letters l and r indicate the type of Repeat mode index register modification to be effected with the left and right Repeat instructions. l and r may be N, A, or S for No modification, Add to the index register, or Subtract from the index register.

TAC Constants

The following two constants are used primarily to load the D Register for use with the Index Register instructions:

Location Constant:	L/xx...xx This creates a word with xx...xx in the N and V parts of both halves of the word.
Command Constant:	C/Command, Address, Index. This creates a half word identical to the computer instruction corresponding to the mnemonic form written.

D Register: S Bit, Address Portion, and F Bit

The size of the address part of D is determined by the S bit (first bit) of the appropriate half word in D. If D_S is 1, the address part is D_V; if D_S is 0, the address part is the size of the index register. Most of the time little or no consideration is given to D_S. A TAC constant is usually written to be placed in D; its most important part is a symbolic address, and its S bit is generally zero.

The F-bit positions of the D Register are of concern only with TDXC and TXDC because the F bit corresponds to X_c. The F bit of D_R specifies left or right for the jump in AIXJ and SIXJ and is established by TAC if symbolic addresses are employed. If necessary, a command constant can be written with instructions, such as HLTL or HLTR, to specify an F bit of zero or one.

Programming Rules of Thumb

a. Place the base or reference memory address in the index registers. Then place the increments or decrements to the base in the address parts of the instructions specifying the index register.

b. Whenever it is necessary to store the contents of an index register in memory, clear the D Register first with the CD command to insure a transfer of the entire contents of the index register.

c. For a general program, place addresses in index registers with TDX rather than TIX because of the addressing limitations of TIX. Use TIX to clear index registers to zero.

d. Use the Location constant if an address is needed for one index register and the counter is not to be set. Otherwise, use a Command constant.

CHAPTER IX

PROGRAMMING TECHNIQUES

INSTRUCTION MODIFICATION

As explained in the preceding chapter, index registers should be used wherever possible for instruction address modification. In some programs, however, situations may arise when an index register isn't available for address modification, or the instruction to be modified is not indexable, or the command is to be modified. In these cases, the programmer must employ techniques of programmed instruction modification utilizing such instructions as TJM and the Extract instructions. This type of modification differs from the index register type of Chapter VIII in that it occurs before the modified instruction is executed rather than while it is executed.

Whenever such instruction modification is to be effected, the programmer must be certain that he is thoroughly familiar with the binary structure of the instruction to be altered and with the details of the instruction which will cause the modification. In general, it is advisable to use Extract instructions rather than TJM to cause instruction address modification.

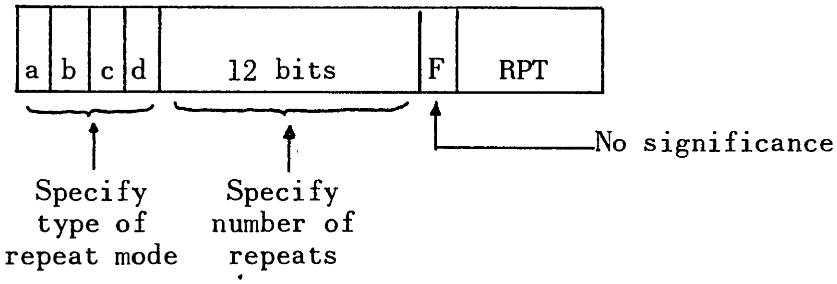
Programmed Address Modification By Substitution

Instruction addresses may be modified by substitution or by arithmetic. When modifying addresses by substitution the most straightforward method is to use the Extract instructions. Other methods, to be described shortly, use either of the pairs of instructions: TXD-TJM and TIJ-TJM.

The introductory remarks and the advisability of using Extracts rather than TJM will become evident by considering the address modification of a Repeat instruction.

Since the Repeat instruction is not indexable, it can only be modified by programmed address modification. For example, in the transfer of variable sized records within memory, the number of repeats varies according to the size of the record and must be determined during the running of the program. In order to perform the transfer of such a record, one field in it must specify its size in words. The programmer will insert this field in the address part of a Repeat instruction which repeats a TMD-TDM pair of instructions.

Before doing this, however, he must be acquainted with the format of the Repeat instruction. This format is as follows:



The type of repeat mode modification (N,A and S in mnemonic form) is specified by the first four bits, such that a and b apply to the left half Repeat instruction and c and d refer to the right half Repeat instruction.

The following are the effective configurations of a, b, c, and d and the corresponding type of modification (x indicates not significant):

TAC		PHILCO 2000			
		Left		Right	
Left	Right	a	b	c	d
N	N	0	x	0	x
A	A	1	0	1	0
S	S	1	1	1	1
N	A	0	x	1	0
A	N	1	0	0	x
N	S	0	x	1	1
S	N	1	1	0	x
A	S	1	0	1	1
S	A	1	1	1	0

Note that TAC makes "a" and "b" zero for any Repeat instruction of the form RPTr, because it is assumed that only a right half instruction is being repeated. However, left half Repeat instructions take the form RPTlr to make "a" and "b" any combination of ones and zeros.

Because the "a" bit is the S bit of the Repeat instruction and because the S bit determines the size of the field transferred from the JA Register by TJM, caution must be exercised when using TJM to modify a Repeat address. Specifically, TJM will transfer a V-sized field from JA if the S bit is one, or it will transfer the entire contents of JA (the number of bits to represent the largest memory address) if the S bit is zero. Thus, if the "a" bit of a Repeat instruction is zero, TJM may alter the repeat mode bits. For example, if the size of the memory is greater than 4096 words, more than 12 bits will be transferred. Similarly, the maximum number of repeats, 4095, cannot be transferred from JA if the "a" bit is one and the size of the V field is less than 12 bits.

Therefore, for most general usage, it is preferable to modify the address of the Repeat instruction by extracting, as shown in the following example:

Example

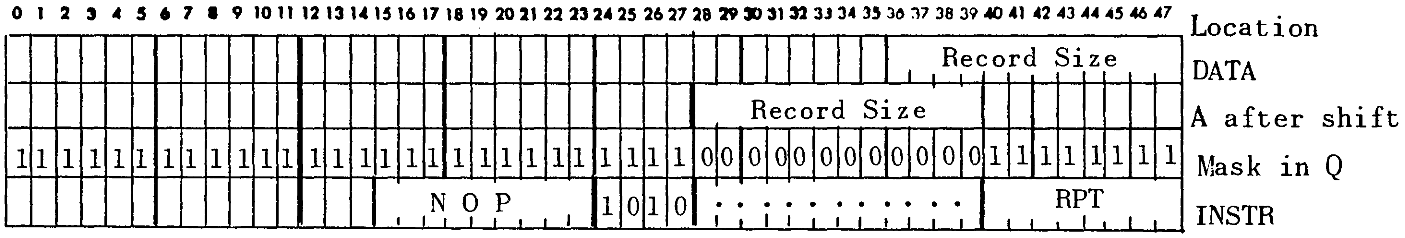
Assume that the word in DATA is the first word of a record and that it contains the record size in its rightmost twelve bit positions. Twelve bits were allocated because this is the maximum size of the address part of the Repeat instruction. Assume also that the entire record is in memory and that it is desired to transfer it to the locations beginning with OUTPUT.

The following coding will accomplish this:

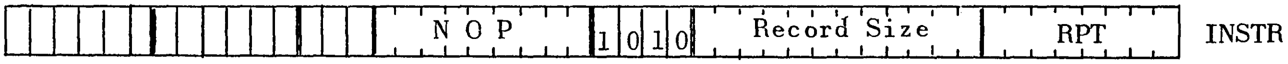
L	LOCATION	COMMAND	ADDRESS AND REMARKS	
		TMD	C/HLTL,DATA; C/HLTL, ØUTPUT $	Set up index registers
		TDXLC	,1$	$DATA \rightarrow X_1$, $0 \rightarrow X_{1C}$
		TDXRC	,2$	$OUTPUT \rightarrow X_2$, $0 \rightarrow X_{2C}$
		TMA	,1$	Record size field to right half address part of A Register
		SLA	8$	
		TMQ	28/1;12/0; 8/1 $	Extract Insert record size field in address part of RPT
		EIS	INSTR$	
L	INSTR	NØP		
		RPTAA	. . . .	Transfer the record
		TMD	1,1$	
		TDM	1,2$	

The following diagram shows the formats of the significant words in this example:

Before EIS



After EIS



As mentioned in Chapter VIII, index registers can be used in conjunction with TJM instructions to substitute addresses in instructions in memory. The steps to follow when using index registers for this purpose are the following:

a. Place the desired address in an index register.

b. Execute a TXD instruction to place the address in the JA Register. Care must be exercised to ensure that D_S is zero if the entire contents of the index register are to be transferred.

c. Execute a TJM instruction for each substitution.

In a similar manner, address substitution may be effected with a TIJ-TJM pair of instructions. This combination was used in Chapter VII, and the substitution provided addresses for Jump instructions. At that time no consideration was given to the fact that JA has an F bit associated with it and that TJM transfers the F bit. This F bit may be zero or one and replaces the F bit of the instruction being altered. If the JA Register contains a Jump instruction, the F bit is automatically set to indicate which half of an instruction word is the next to be executed (zero for a left half, one for a right half) if the Jump is not effected. If the JA Register contains a TIJ instruction, the F bit of the TIJ instruction is transferred to JA_F. The F bit of the TIJ instruction may be set to zero if TIJL is written or to one if TIJR is written.

The F bit need not be considered as long as TIJ and TJM instructions utilize symbolic addresses and are applied to Jump instructions. The reason for this is that TAC will make the proper F-bit assignment when symbolic addresses are used. In other situations, the programmer must be aware of the effect of transferring an F bit to an instruction to insure that TJM does not alter the instruction receiving JA_F, unless an alteration is desired.

The following list illustrates the use of the F bit with some of the instructions:

Instruction Class	Purpose of F bit	State of F bit
Arithmetic	Arithmetic Mode	0: Fixed point 1: Floating point
Jumps	Left or Right	0: Left 1: Right
Index Register	Counter or Half of D	0: Left or No counter 1: Right or Set counter
Arithmetic Transfers Shifts	Part of Command	0 1

Some instructions, such as RPT, HLT, NOP, operate in the same manner regardless of the F bit. Other instructions, such as the Extract instructions, require eight bits including the F bit, to define the command.

Also to be considered with TJM is the fact that the size of the address transferred from JA is determined by the S bit of the instruction <u>receiving</u> the contents of JA. Thus, if $I_S = 0$, the largest address sized field is transferred and if $I_S = 1$, a V sized field is transferred.

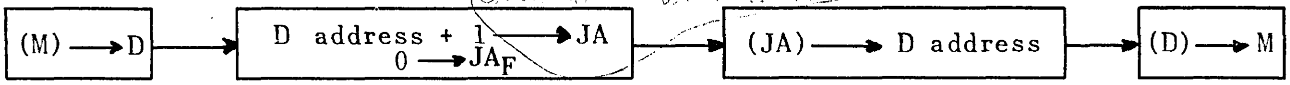
Programmed Address Modification By Addition

If it is necessary to increase an address by one without using index registers, the instruction INCA may be used.

INCAh   Increase Address in memory

A one is added to the address part of the instruction at the specified memory location.

The execution of this instruction uses JA and the D Register as follows:



If INCA addresses an instruction symbolically, "h" may be omitted. If INCA addresses an instruction word, "h" specifies the half to be modified: L or R, for left or right. The F bit of the half of the word addressed is not altered.

As previously stated, the size of the address part of the instruction being modified depends on its S bit. For S = 1, the V field is modified, and for S = 0, the field modified is the size of the largest memory address.

If a number of instructions are to contain the same address parts, TJM instructions can follow INCA for the necessary substitutions.

Example

Assume a 128-word table starting at location TABLE, and a key to be looked up in location KEY. Jump to FOUND if the table entry equals the word in KEY. Jump to FOUND if the key in question is in the table or to NOENTRY if it is not. In the preceding chapter this example used a counting index register to address successive memory locations. INCA is used in the following coding for the same purpose:

L	LOCATION	COMMAND	ADDRESS AND REMARKS	
L	SEARCH	TMA	KEY$	Jump to FOUND if the table entry equals
		TMD	TABLE$	the word in KEY
		JAED	FØUND$	
		TMA	SEARCH$	Test to see
		TMD	C/TMA.,KEY; C/TMD, TABLE + 127 $	if last table
		JAED	NØENTRY$	entry used
		INCAR	SEARCH$	Add 1 to TMD TABLE
		JMP	SEARCH$	Jump for next table test
	NØENTRY	TMD	C/TMA,KEY; C/TMD, TABLE $	Restore original
		TDM	SEARCH$	instructions

Analysis of the Coding

The instruction in location SEARCH is specified as a left half instruction which makes TMD TABLE a right half instruction. Because the computer compares whole words, the pair of instructions in location SEARCH must be compared with the pool constant shown, to determine when the address part of TMD TABLE has been increased to TABLE + 127, the last word in the table. If the two words are not equal, INCAR adds one to the address part of TMD TABLE and the next table comparison is made. The instructions at NOENTRY restore the original word at memory location SEARCH. Note that the address part of the TMD TABLE instruction can also be restored by the instructions TIJ TABLE, TJMR SEARCH. Note also that it is usually preferable to preceed a subroutine like SEARCH with the instructions at NOENTRY.

This method may be extended to instruction address modification by amounts other than one, in which case a single INCA cannot be used. Instead, the programmer may write a constant with the necessary address increment and perform an addition to the instruction word to be altered.

Example

The following coding will perform the same table lookup as in the preceding example if the entries consist of two words:

L	LOCATION	COMMAND	ADDRESS AND REMARKS	
L	SEARCH	TMA	KEY $	
		TMD	TABLE $	Table look-up
		JAED	FØUND $	
		TMA	C/HLTL,0; C/HLTL, 2 $	Add 2 to address of
		AMS	SEARCH $	TMD TABLE
		TMD	C/TMA,KEY; C/TMD, TABLE + 128 $	Test to see if
		JAED	NØENTRY $	last entry used
		JMP	SEARCH $	Jump for next table test
	NØENTRY	TMD	C/TMA,KEY; C/TMD, TABLE $	Restore original
		TDM	SEARCH $	instructions

Programmed Command Modification

The final type of instruction modification concerns the modification of commands rather than addresses. Although situations requiring this type of modification occur less frequently than the preceding types, the programmer should be aware of their possibilities.

There will be instances, for example, when one quantity is to be added to a total and another is to be subtracted from the total. This may be a program switch situation and can be satisfied by modifying a command. The alteration may be performed by extracting or by instructions such as AWCS, which modifies by addition. In either case, the programmer who performs command modification must be fully aware of the bit structures of the commands involved.

INTERNAL SORTING, MERGING, AND MATCHING

Sorting

In most data processing operations, the order or sequence of records in files is important. Sequence is important to minimize the time necessary to locate records in a large file. If the records were in random order, i.e., no particular sequence, it would be necessary to search through half the file, on the average, to find a given record. The time to search for many records would, of course, be prohibitive. Because of the importance of data sequence, it is necessary to consider the requirements of establishing the sequence -- a process known as SORTING.

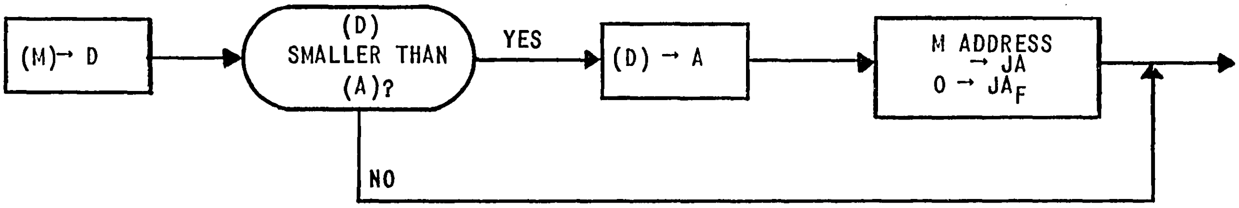
Internal sorting is the process of arranging records <u>in memory</u> in an ordered sequence. One method of performing an internal sort begins by comparing the identifying keys of a group of records against one another and selecting the record with the smallest key, if an ascending sequence is desired. This record is transferred to another area of memory. The remaining keys are compared and the record with the next smallest key is placed behind the one with the smallest. The process is continued until of the records have been transferred, in ascending order, to the other area of memory. If a descending sequence is desired, records will be selected according to the larger keys.

The PHILCO 2000 was designed to facilitate internal sorting of the above type by incorporating the following two special instructions:

SWD Smaller Word

The word from the specified memory location is transferred to the D Register. Then the words in the A and D Registers are compared in the alphanumeric sense. If the word from <u>memory</u> is <u>smaller</u> than the word in A, it is transferred to A and its address is placed in the JA Register. The F bit of JA is made zero. If the word from memory is larger than or equal to the word in A, the instruction has no effect other than that the comparison is made.

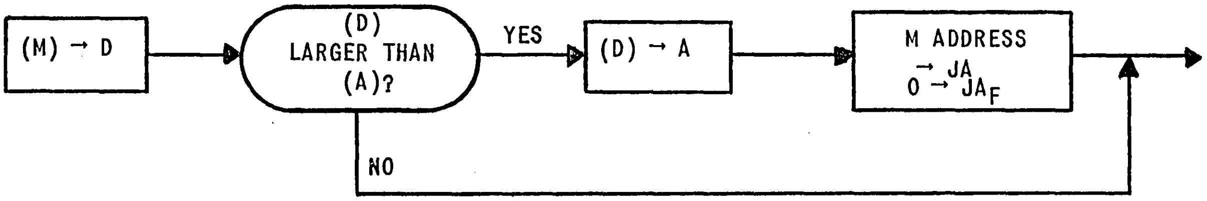
A micro-flowchart of this instruction follows:



LWD Larger Word

The word from the specified memory location is transferred to the D Register. Then the words in the A and D Registers are compared in the alphanumeric sense. If the word from memory is larger than the word in A, it is transferred to A and its address is placed in the JA Register. The F bit of JA is made zero. If the word from memory is smaller than or equal to the word in A, the instruction has no effect other than that the comparison is made.

A micro-flowchart of this instruction follows:



The SWD instruction is used to produce an ascending sequence and LWD is used to produce a descending sequence. Prior to the use of either instruction, the A Register and the JA Register should contain the first key and its address, respectively.

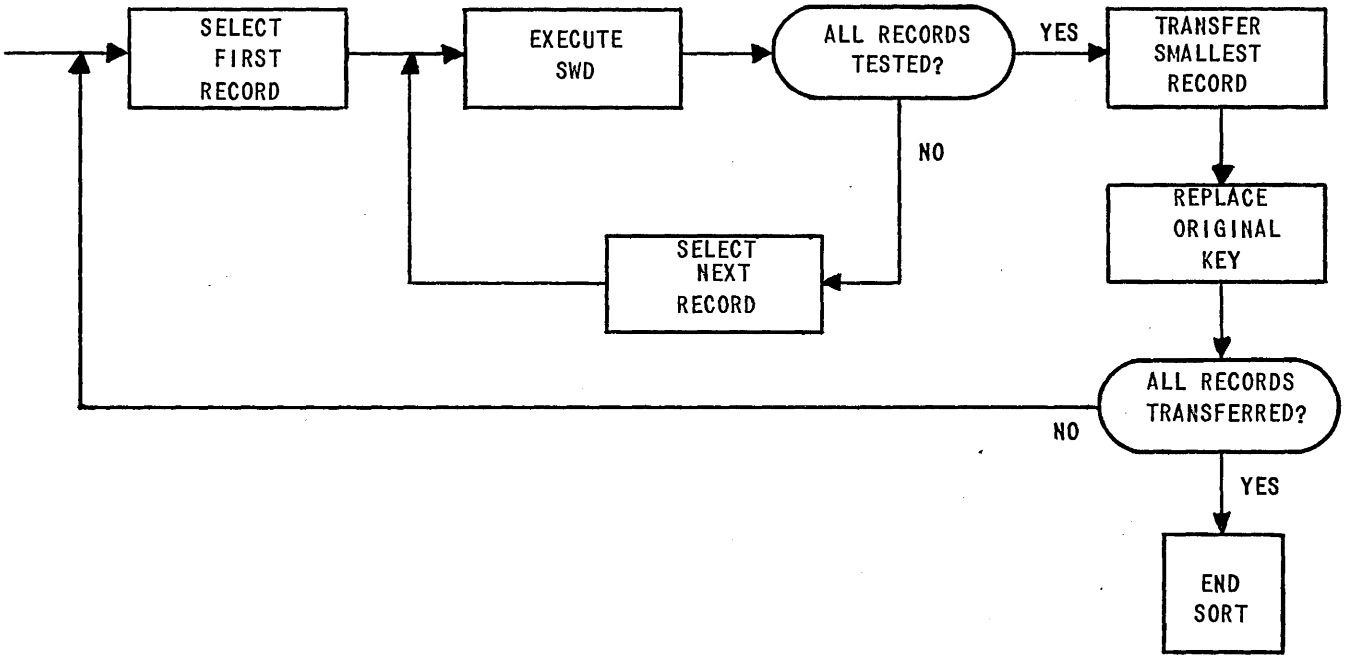
An equally valid procedure is to first place in the A Register the largest possible word, all ones, for SWD, or the smallest possible word, all zeros, for LWD. These two procedures will insure the validity of the first execution of SWD or LWD.

For example, if it is desired to select the smaller of two words and SWD is used, the following instructions can be written:

COMMAND	ADDRESS AND REMARKS	
TIJ	WØRD1$	Address of word 1 → JA
TMA	WØRD1$	Word 1 → A
SWD	WØRD2$	Smaller word → A, its address → JA

Thus, if word 1 is smaller than word 2, SWD has no effect; the A Register contains the smaller word, and JA contains its address. If word 2 is smaller, it is transferred to A and its address is placed in JA. In either case, A contains the smaller word and JA contains its address.

To sort a group of records, LWD or SWD is used under control of the Repeat instruction. This combination tests a key against all other keys in the group. After the tests have been made, the record with the smallest key, for an ascending sequence, is transferred to the sorted area; a dummy key which is larger than any real key is substituted for the original, and the process is repeated until all records have been sorted. A flowchart for this operation follows:



Example

Assume a block of two-word records beginning in location PAY. The first word of each record contains a key. Sort the records into ascending order by the key and store the sorted records starting at location SORTED.

The coding to accomplish this is as follows:

L	LOCATION	COMMAND	ADDRESS AND REMARKS	
		TMD	L/SØRTED$	Set up X_1 for output of sort
		TDXLC	,1$	
		TMQ	48/1$	Dummy key, all ones, → Q
	SØRT	TMD	L/PAY$	Set up X_2 for first record of block
		TDXLC	,2$	
		TQA		Largest word, i.e., dummy key → A
L		RPTA	64$	Place smallest of 64 keys in A,
		SWD	2,2$	its address in JA
		TJM	DUMMY$	Address of smallest key → TQM instruction
		TDXLC	,3$	Address of smallest key → X_3; 0 → X_{3c}
R		RPTAA	2$	
		TMD	1,3$	Record with smallest key → SORTED area
		TDM	1,1$	
	DUMMY	TQM	. . . .	Dummy key replaces smallest key
		TMD	C/HLTL,SØRTED + 128; C/HLT, SØRT $	Test if
		AIXJ	,1$	SORTED area
				is complete

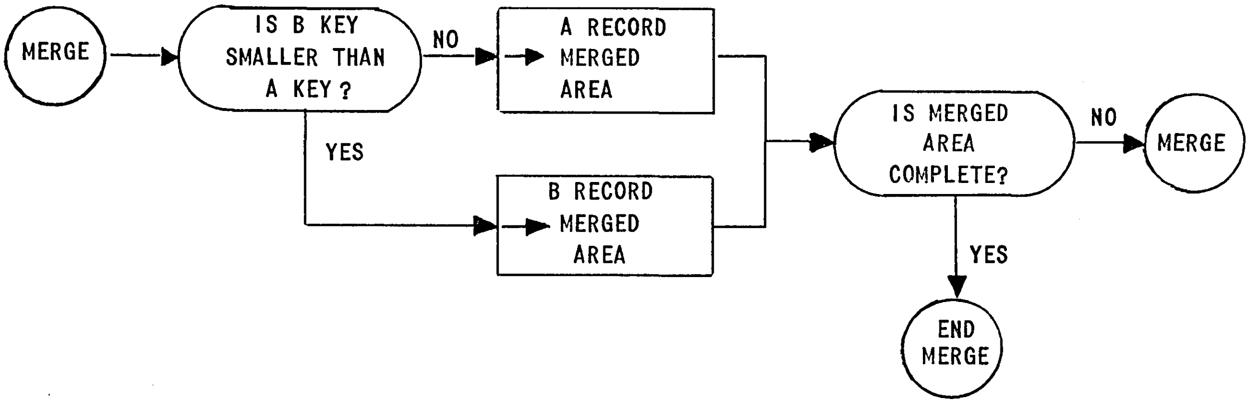
Merging

Merging is the operation which produces one group of records in sequence from two or more groups of records, each of which is in sequence. Other names for merging are collating and interfiling. Merging is necessary in internal sorting techniques and in the sorting of large quantities of data. Its function is to produce larger and larger groups of sorted records. When all of the records to sort do not fit in memory at one time, merged groups of records are written on magnetic tape. The remaining requirements for sorting include a knowledge of magnetic tape input-output.

Example

It is desired to merge two blocks of records. Assume that they are four-word records, that the first word of each is a key, that the first block begins in ADATA, the second begins in BDATA, and that the merged records are to begin at MERGED.

An abbreviated flowchart, which doesn't indicate index register use for selecting the next record, follows:



The coding for the merge follows:

L	LOCATION	COMMAND	ADDRESS AND REMARKS	
		TMD	C/HLTL,ADATA; C/HLTL, BDATA $	
		TDXLC	,1$	Set up Index
		TDXRC	,2$	Registers
		TMD	L/MERGED $	1, 2, 3 - no
		TDXLC	,3$	counting
	MERGE	TMA	,1$ Compare keys:	
		TMD	,2$ Jump if A key ≧ B key	
		JAGD	TRFRB$	
R		RPTAA	4$ Transfer A record (4 words)	
		TMD	1,1$ to MERGED area	
		TDM	1,3$	
	ENDTEST	TMD	C/HLTL,MERGED + 256;C/HLTL, MERGE $	MERGED
		AIXJ	,3$	complete?
		JMP	ENDMRGE$ Jump after all records merged	
R	TRFRB	RPTAA	4$	
		TMD	1,2$ Transfer B record (4 words)	
		TDM	1,3$ to MERGED area	
		JMP	ENDTEST$ Jump to the ending test	

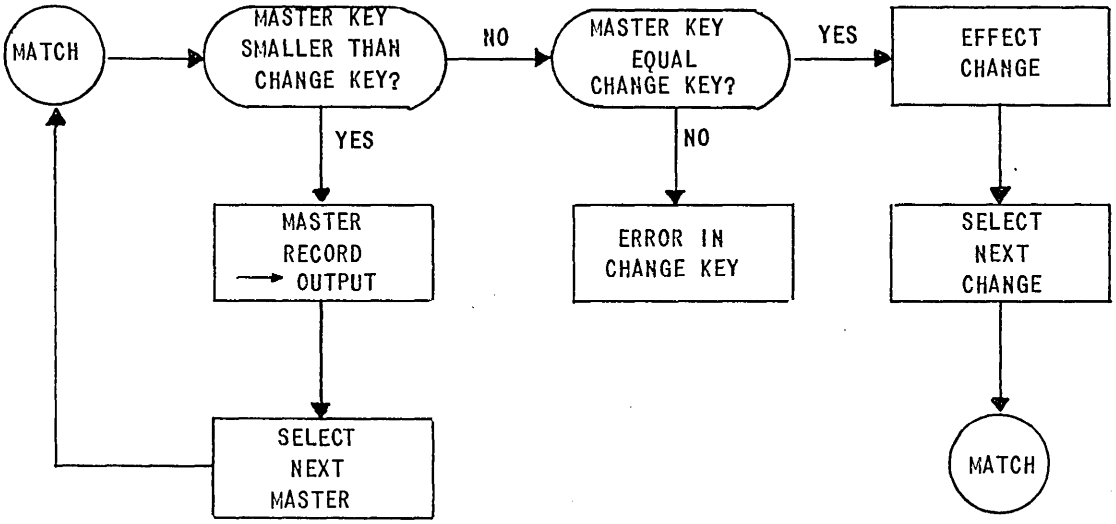
Note that in order to merge the last record in each of the two blocks, a word consisting of all ones would have to follow each block. Normally, however, it is more common to test for the last record of every block.

Matching

Matching is the operation used to determine if one record applies to another by comparing their keys.

If it were necessary to change various records of a master file, for example, it would be accomplished by preparing one change record for each record to be changed. The change record would contain the key of the record to be changed and some indication as to what is to be changed. This type of matching has been illustrated throughout the manual. A realistic example in which transactions were applied to inventory records was included in Chapter VII, page 211.

An abbreviated flowchart for the matching required in File Maintenance is shown below. File Maintenance requires that a master file incorporating all changes be reproduced.



EXTERNAL PROGRAM CONTROL

Several means are provided to allow the programmer to introduce or examine small amounts of data from the computer. The Console Typewriter permits typing or transmission of information on punched paper tape into and out of the computer. The Toggle Register allows for the entrance of one word in binary form. The Breakpoint switch allows the program to be halted at appropriate times. These devices provide programs with small amounts of data during the running of a program, permit manual options in the program, enable the program to print control totals, and allow for printing directions for the computer operator.

Console Typewriter

The Console Typewriter accepts punched paper tape or keystroke input and punches paper tape and produces typed copy. The keystroke input through the Console Typewriter enters the computer in binary-coded form. Although there are less than 64 keys on the Console Typewriter, all 64 Philco characters may be represented by preceeding certain characters with a shift to lower case. This shift to lower case has its own six-bit representation, and certain characters transmitted through the console typewriter will therefore be represented by 12 bits -- six for the lower case shift and six more for the character of which it is the lower case. The Console Typewriter will then remain in the lower case until a shift back to upper case (with its own specific six-bit code) is made. These special lower case characters are clearly indicated on the keyboard.

The two instructions for the Console Typewriter are:

TCM: Transfer from Console Typewriter to Memory

TDC: Transfer from D to Console Typewriter

The TCM instruction transfers one character from paper tape or the keyboard to the rightmost six-bit positions of the D Register, without altering the remaining positions of D. The word in the D Register is then transferred to the specified memory location.

The TDC instruction transfers the character in the leftmost six-bit positions of the D Register to the Console Typewriter. The character is then typed or punched in paper tape. To transmit more than one character to or from the Console Typewriter, a loop must be programmed. This loop is usually under Repeat control and shifts the contents of the D Register to make the next character or character position ready for the next transfer. The right circular shift, SCD is used regardless of which typewriter operation is required, and in both cases it will be a shift of 42 positions.

The instructions necessary to type eight characters from the keyboard to memory location WORD are the following:

L	LOCATION	COMMAND	ADDRESS AND REMARKS	
R		RPTNN	8$	
		SCD	42$	Shift seven characters
		TCM	WØRD$	One character → D, (D) → WORD

The following drawings illustrate the effects of these instructions when used to enter the word PHILCOΔΔ. Only the D Register is shown because the contents of WORD are the same.

	D Register	
Initial Condition	1 2 3 4 5 6 7 8	
After SCD	2 3 4 5 6 7 8 1	1st repeat
After TCM	2 3 4 5 6 7 8 P	←From typewriter
After SCD	3 4 5 6 7 8 P 2	2nd repeat
After TCM	3 4 5 6 7 8 P H	←From typewriter
After SCD	4 5 6 7 8 P H 3	3rd repeat
After TCM	4 5 6 7 8 P H I	←From typewriter
After SCD	5 6 7 8 P H I 4	4th repeat
After TCM	5 6 7 8 P H I L	← From typewriter

and so on until the last repeat:

After SCD	P H I L C O Δ 8	
After TCM	P H I L C O Δ Δ	←From typewriter

The program that follows illustrates a loop to type instructions to the operator from memory. It will cause the Console Typewriter to print the contents of the three consecutive memory locations starting at location OPRTR.

L	LOCATION	COMMAND	ADDRESS AND REMARKS	
		TMD	L/ØPRT'R$	Set up X_1 to address the words to
		TDXLC	,1$	to be typed
	TYPEØUT	TMD	,1$	One word to be typed ⟶ D
R		RPTNN	8$	
		TDC		Type eight characters
		SCD	42$	
		TMD	C/HLT,ØP	RTR + 3; C/HLT, TYPEØUT $
		AIXJ	1,1$	
	ØPRTR	A/SETΔTH	EΔØVERFL	Ø W Δ SWITCH. $

The type out operation for the first word has the following effects on the D Register:

```
                          D Register
Initially          | S  E  T  Δ  T  H  E  Δ |
                     ↓
After   TDC "S" is typed                        \
                                                 } 1st repeat
After   SCD        | E  T  Δ  T  H  E  Δ  S |   /
                     ↓
After   TDC "E" is typed                        \
                                                 } 2nd repeat
After   SCD        | T  Δ  T  H  E  Δ  S  E |   /
                     ↓
After   TDC "T" is typed
```

and so on until eight repeats complete the first word.

Toggle Register

The Toggle Register is a 48-bit register composed of 48 switches on the control console. Each switch may be placed in the "on" or "off" position to correspond to a binary one or zero, respectively. After a word has been established in the Toggle Register the instruction

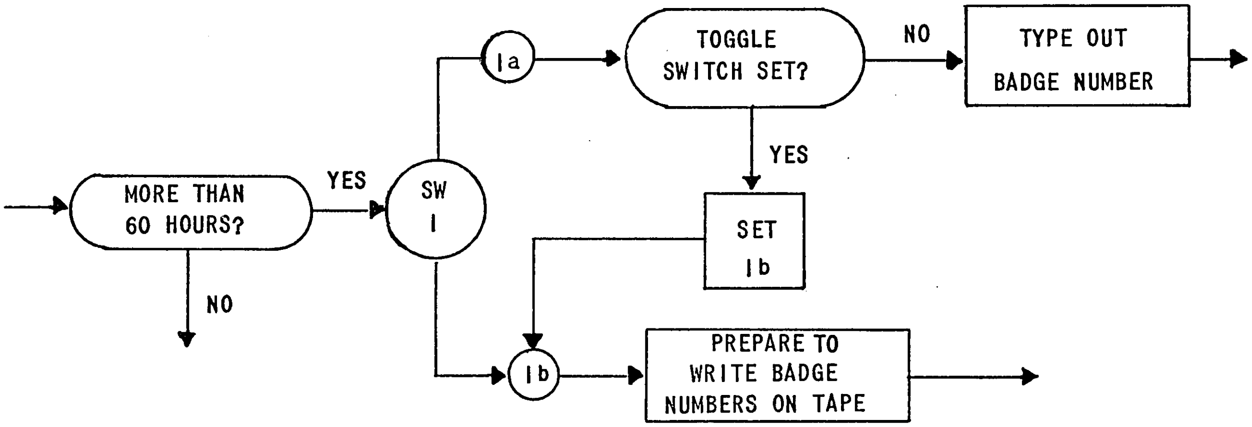
TTD: Transfer from Toggle Register to D

transfers the word to the D Register. Then the programmer can use the word as a series of yes/no codes, as a mask, or as a control key or total. An advantage of the Toggle Register over the Console Typewriter is that the switches can be set while the computer is in operation. Thus, the time to enter such information is merely the time to execute the TTD instruction.

The Toggle Register may be used to allow for an option in the running of a program. For example, in a payroll operation it may be desired to have the computer "inform" the payroll department when a record indicates that an employee has worked more than 60 hours. This may be achieved in two ways. Either the badge numbers of such employees may be typed on the Console Typewriter, or they may be recorded on tape for subsequent printing. To accommodate an option like this, coding for both possibilities must be included in the program, and a program switch provided to select the option. Then the program switch may be set according to the setting of a Toggle Register switch.

For example, the program may initially cause the badge numbers to be printed on the Console Typewriter. If the operator decides that too much time is being taken because of many typeouts, he will stop the computer and depress the rightmost toggle switch. Then the computer is started again and the program interprets the setting of the toggle switch to set the program switch which determines the disposition of "over 60-hour badge numbers."

The following is a flowchart for these operations. Switch 1 is initially set to 1a.



The coding to cause the examination of the Toggle Register and the setting of 1b follows:

L	LOCATION	COMMAND	ADDRESS AND REMARKS	
	SW1	JMP	SW1A$	Switch 1 set to 1a
	SW1A	TTD		Word in toggle switches ——▶D
		TDQ		(D) ——▶Q
		JQE	TYPEØUT	$ Type badge number if toggle switch is not set
		TIJ	SW1B$	Set 1b if
		TJM	SW1$	toggle switch is set
	SW1B			
				Prepare badge numbers for tape

This type of option is usually referred to as a breakpoint option. Breakpoint is defined as the point in the program where a manual operation is performed which may break the normal sequence.

Breakpoint Switch

Another method of exercising a breakpoint option is to use the Breakpoint Jump instruction:

JBT: Halt, Jump, or Proceed depending on the setting of the Breakpoint switch.

Associated with this instruction is a three position locking switch on the control panel. The switch positions are labeled HALT, JUMP, and IGNORE. JBT is interpreted as an HLT instruction when the switch is on HALT, as a JMP instruction when the switch is on JUMP, and as a NOP instruction when the switch is on IGNORE. This gives the programmer many options. For example, when a program is being tested, JBT instructions jump to subroutines to print the contents of specific locations if the switch is placed on JUMP. During the normal running of the program, the switch is placed on IGNORE, and the JBT instruction acts as a NOP.

Exercises

1. Code the program switch which is an arithmetic order to add or subtract the contents of WORD from the contents of the A Register. Set the switch to add or subtract by modifying the command. (AMS: 01000001; SMS: 01010001).

2. The block of memory beginning at FILE-A contains 32 records -- 4 words each -- in descending order according to the key in the first word of the records. The block in memory beginning at FILE-B contains a similar group of records. Furthermore, the words in each record are in reverse order. Thus the last four words of a block contain the record with the smallest key in the block, and the last word of the block contains this key. The problem is to merge the two blocks to produce an ascending sequence starting at location MERGE. The words in each record should be in the proper order.

SUMMARY

Instruction Modification

Instruction modification is the alteration of an instruction either before or during its execution. Most frequently, the part of the instruction modified is the address part. The simplest address modification is performed by index registers during the execution of the instruction. Other types of modification occur before the execution of the instruction and require the use of such instructions as TJM or the Extract instructions. The latter instructions are more straightforward to use.

Instruction address modification may be performed either by substitution or by addition. When performing address modification by substitution,the Extract instructions or either of the pairs of instructions, TXD-TJM and TIJ-TJM, can be used. Address modification by addition may be accomplished by using the instruction INCA.

INCAh	Increase Address in memory
	The address of the "h" half of the word at the specified location is increased by one.

Before attempting instruction modification, however, the programmer must be thoroughly familiar with the binary instruction format and the mode of operation of all the instructions. Several pertinent facts on this subject follow:

a. The JA Register has an F bit associated with it and the TJM instruction transfers it to memory.

b. The S bit of the instruction to be modified determines the size of its modified address. This applies to the instructions TJM and INCA.

c. The address part of the RPT instruction is always 12 bits; the first four bits are concerned with the Repeat instructions, and the F bit is not significant.

d. When symbolic addresses are used, TAC inserts the appropriate F bits in such instructions as TIJ, TJM, and Jump instructions.

e. During INCA, JA_F is zero.

Sorting, Merging and Matching

Sorting places records in an ordered sequence according to a key or number in the record. Merging produces one sequence of records from two or more sorted groups of records. Matching associates records with identical keys from several files.

Two instructions which are useful in sorting are LWD and SWD.

LWD Larger Word

If the contents of the specified memory location are larger than the contents of the A Register, they are transferred to A and the address of the word is placed in JA. The F bit of JA is made zero.

SWD Smaller Word

If the contents of the specified memory location are smaller than the contents of the A Register they are transferred to A and the address of the word is placed in JA. The F bit of JA is made zero.

External Program Control

A program may require information which is not a part of the original program or the data. This information can be provided by the Console Typewriter and the Toggle Register. Similarly, the program may communicate with the operator via the Console Typewriter. The instructions for these operations are TCM, TDC, and TTD.

TCM Transfer from Console to Memory

One character, typed or from paper tape, is transferred to the right six-bit positions of the D Register. The contents of D are then transferred to memory.

TDC Transfer from D to Console

The character in the left six-bit positions of the D Register is typed on the Console Typewriter or punched in paper tape.

Manual program options are exercised at Breakpoints in the program. The transfer from Toggle Register and the Breakpoint Jump instruction may be used to specify options.

TTD Transfer from Toggle Register to D

The word created by setting the Toggle Register switches is transferred to the D Register.

JBT Breakpoint Jump:

If the Breakpoint switch is set at Halt, the computer stops. If set at Jump, an unconditional jump is effected. If set at Ignore, the command is treated as a NOP.

NOTES

APPENDICES

APPENDIX A

BINARY AND DECIMAL EQUIVALENTS

(1) Maximum Decimal Integral Value	Number of (2) Decimal Digits	(3) Number of Bits	(4) Maximum Decimal Fractional Value
1		1	.5
3		2	.75
7		3	.875
15	1	4	.937 5
31		5	.968 75
63		6	.984 375
127	2	7	.992 187 5
255		8	.996 093 75
511		9	.998 046 875
1 023	3	10	.999 023 437 5
2 047		11	.999 511 718 75
4 095		12	.999 755 859 375
8 191		13	.999 877 929 687 5
16 383	4	14	.999 938 964 843 75
32 767		15	.999 969 482 421 875
65 535		16	.999 984 741 210 937 5
131 071	5	17	.999 992 370 605 468 75
262 143		18	.999 996 185 302 734 375
524 287		19	.999 998 092 651 367 187 5
1 048 575	6	20	.999 999 046 325 683 593 75
2 097 151		21	.999 999 523 162 841 796 875
4 194 303		22	.999 999 761 581 420 898 437 5
8 388 607		23	.999 999 880 790 710 449 218 75
16 777 215	7	24	.999 999 940 395 355 244 609 375
33 554 431		25	.999 999 970 197 677 612 304 687 5
67 108 863		26	.999 999 985 098 838 806 152 343 75
134 217 727	8	27	.999 999 992 549 419 403 076 171 875
268 435 455		28	.999 999 996 274 709 701 538 085 937 5
536 870 911		29	.999 999 998 137 354 850 769 042 968 75
1 073 741 823	9	30	.999 999 999 068 677 425 384 521 484 375
2 147 483 647		31	.999 999 999 534 338 712 692 260 742 187 5
4 294 967 295		32	.999 999 999 767 169 356 346 130 371 093 75
8 589 934 591		33	.999 999 999 883 584 678 173 065 185 546 875
17 179 869 183	10	34	.999 999 999 941 792 339 086 532 592 773 437 5
34 359 738 367		35	.999 999 999 970 896 169 543 266 296 386 718 75
68 719 476 735		36	.999 999 999 985 448 034 771 633 148 193 359 375
137 438 953 471	11	37	.999 999 999 992 724 042 385 816 574 096 679 687 5
274 877 906 943		38	.999 999 999 996 362 021 192 908 287 048 339 843 75
549 755 813 887		39	.999 999 999 998 181 010 596 454 143 524 169 921 875
1 099 511 627 775	12	40	.999 999 999 999 090 505 298 227 071 762 084 960 937 5
2 199 023 255 551		41	.999 999 999 999 545 252 649 113 535 881 042 480 468 75
4 398 046 511 103		42	.999 999 999 999 772 626 324 556 767 940 521 240 234 375
8 796 093 022 207		43	.999 999 999 999 886 313 162 278 383 970 260 620 117 187 5
17 592 186 044 415	13	44	.999 999 999 999 943 156 581 139 191 985 130 310 058 593 75
35 184 372 088 831		45	.999 999 999 999 971 578 290 569 595 992 565 155 029 296 875
70 368 744 177 663		46	.999 999 999 999 985 789 145 284 797 996 282 577 514 648 437 5
140 737 488 355 327	14	47	.999 999 999 999 992 894 572 642 398 998 141 288 757 324 218 75
281 474 976 710 655		48	

This chart provides the information necessary to determine:

a. The number of bits needed to represent a given decimal number. Use columns one and three or four and three.

b. The number of bits needed to represent a given number of decimal digits (all nines). Use columns two and three.

c. The maximum decimal value represented by a given number of bits, use columns one and three or three and four.

APPENDIX B

REGISTER CONTENTS ALTERED BY INSTRUCTIONS

The following chart indicates which registers have their contents altered by particular instructions. The reader is advised to refer to the manual if the reason for an alteration is not apparent. An "A" indicates always altered; a number refers to a note following the chart.

Instruction or Instruction Class	Register Contents Altered					
	A	Q	D	JA	X	O.F.
Addition	A		A*			A
Subtraction	A		A*			A
Multiplication	A	1	A			A
Division	A	A	A			A
Transfers and Clears	2	2	A*			
Shifts	3	3	3			A
Jumps				A		
JQØ, JQE, JQP, JQN		A		A		
JAEQ, JAGQ, JAGQF			A	A		
JØF, JNØ				A		A
Index Register						
TCXS					A	
TIX, TDX, ADX, SDX					A	
TXD			A	A		
AIXJ, SIXJ				A	A	
AIXØ, SIXØ					A	A
Extract			A			
ETA, EI, EIS	A		A			
EA, ES	A		A			A
DØRMS, AWCS			A			
LWD, SWD	4		A	4		
TJM			A			
TIJ				A		
INCA			A	A		
TCM, TTD			A			

NOTES

1. Q will contain the minor half of a double length product.
2. (A) and (Q) are altered by transfers to A and Q, respectively.
3. (A), (Q), and (D) are altered by shifts involving A, Q, and D, respectively.
4. (A) and (JA) are altered when LWD or SWD finds a larger or smaller word, respectively.

Instructions under Repeat control can modify the contents of index registers. (MA) are altered whenever the memory is accessed. MA may not be accessed by the program but may be used in conjunction with the Memory Preset switches on the control panel. PR and PA cannot be accessed by the program. However, PA is altered whenever a jump is effected.

* Except if the operand is in D originally.

APPENDIX C

PHILCO 2000 INSTRUCTIONS

The following chart lists the instruction codes by illustrating the three necessary parts of each instruction class. The first part of the instruction code is the operation, such as: A for add, S for subtract, etc., and may be one to five letters. The second part of the instruction code represents an affected register, or a condition, or a particular half word, or another operation. The second part contains one or two letters. The final part contains options.

A particular command code is formed by selecting one of the entries in the Operation column, followed by one of the entries (if there is one) from the next column, followed by any or none of the options except where otherwise noted.

An abbreviated description of each operation is included in the chart. Although not specifically stated in each description, the D Register plays a part in all instructions which involve memory access, arithmetic operations, and transfers of data between arithmetic registers.

PHILCO 2000 INSTRUCTIONS

Instruction Class	Instruction	Mnemonic Code			Description of Operation	Notes	Code Example
		Operation	Register or Condition	Option			
Addition	Add	A	M Q	A,S	1. (A) + Operand → A 2. When result is stored: (A) → M (and D). The operand is from M or Q and may be in absolute value. Before step 1, A is cleared to zero for Clear Add.	Options: A = absolute operand S = result stored F = floating point Overflow: The overflow indicator is set when the result ≥ 1 or < -1.	AM AMA CAQS CAQAS
	Clear Add	CA					
	Add D	AD			(A) + (D) → A		
Subtraction	Subtract	S	M Q	A,S	1. (A) - Operand → A 2. When result stored: (A) → M (and D). The operand is from M or Q and may be in absolute value. Before step 1, A is cleared to zero for Clear Subtract.	Options: A = absolute operand S = result stored Overflow: The overflow indicator is set when the result ≥ 1 or < -1.	SM SMS SQA CSQAS
	Clear Subtract	CS					
	Subtract D	SD			(A) - (D) → A		
Multiplication	Multiply	M	A M	A R S	1. Operand x (Q) → A, Q or A rounded.* 2. When result stored: (A) → M (and D). The operand is from M or A and may be in absolute value.	Options: A = absolute operand R = rounded product S = result stored Overflow: when the result = 1 * (Q) are unaltered when rounded.	MAR MMRS
	Multiply and Add	MAD			[(M) x (Q)] + (A) → A	The product is rounded. (Q) are unaltered.	
	Multiply and Subtract	MSU			[(M) x (Q)] - (A) → A		
Division	Divide A register	DA		S	1. [(A) or (A,Q)] ÷ (M) → Q, remainder → A. 2. When result stored: (Q) → M (and D).	Option: S = result stored. Potential overflow is detected if |M|<|A| or if |M| = |A| and (A) are positive.	DA DAS DAQ DAQS
	Divide A and Q registers	DAQ					
Transfer	Clear	C	M,A,Q,D		0 → M or A or Q or D		CM
	Transfer	T	M,A,Q,D	M,A,Q,D*	Transfer [(M) or (A) or (Q) or (D)] to [M or A or Q or D].	*These are not optional. One letter must be selected. TMM, TAA, TQQ and TDD are not permitted.	TMA
Shift	Shift Left	SL	A AQ Q D	N	Shift the contents of the register(s) the number of places specified by the address. A numerical shift will preserve the sign of a word.	Option: N = numerical shift. No option specifies ordinary shift. (D) may only be shifted right.	SLA SRQN
	Shift Right	SR					
	Shift Circular (D)	SCD			Shift (D) in circular mode right		
Jump	Jump	JMP			Unconditional Jump	1. Address of next instruction → JA. 2. Effective address → PA. *Shift (Q) in circular mode left (for P or N) or right (for O or E) one position regardless of conditions. In these cases, positive is defined as sign bit = 0; negative as sign bit = 1. ‡See notes for NOP and TJM. †JAGD treats words as alphanumeric. For A and Q comparisons, (Q) → D. Then (A) are compared to (D). In JAGQF the numbers should be normalized.	
	Breakpoint Jump	JBT			Stop if breakpoint switch set, jump if not		
	Jump if Overflow	JOF			Jump if overflow indicator is set		
	Jump if No Overflow	JNO		L‡ or R‡	Jump if overflow indicator is not set		
	Jump if (D) are Positive	JDP			Jump if (D) are positive		
	Jump if (A) are +, -, 0	JA	P,N,Z		Jump if (A) are positive or negative, or zero		JAP
	*Jump if (Q) are +, -, odd, even	JQ	P,N,O,E		Jump if (Q) are positive or negative, or odd or even		JQE
	Jump if (A) Equal (D) or (Q)	JAE	D,Q		Jump if (A) equal (D) or (Q)		JAED
	†Jump if (A) are Greater than or equal to (D) or (Q) or (Q)-floating point	JAG	D,Q,QF		Jump if (A) are greater than or equal to (D), or (Q), or (Q) if (A) and (Q) are floating point numbers.		JAGQ
	Jump (Auto-Control Unit Only)	J			Unconditional Jump — JA not affected		

APPENDIX C (cont'd)

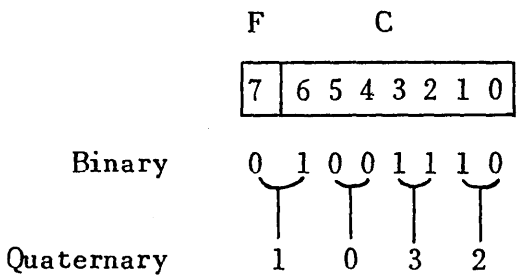
PHILCO 2000 INSTRUCTIONS

Instruction Class	Instruction	Mnemonic Code			Description of Operation	Notes
		Operation	Operation or Half Word	Option		
Index Register	Transfer Counter to Index	TCX	S,Z		$1 \rightarrow X_c$ if S, $0 \rightarrow X_c$ if Z.	Option: C = Counter indicator is transferred.
	Transfer Instruction address to Index	TIX			$I_v \rightarrow X$; $1 \rightarrow X_c$ if S, $0 \rightarrow X_c$ if Z.	
	Transfer from D to Index	TDX	L,R	C	D address → X.	L and R specify left or right half of D Register.
	Transfer from Index to D	TXD			(X)→D address via JA.	
	Add (D) to Index	ADX	L,R		(X) + D address → X.	
	Subtract (D) from Index	SDX			(X) - D address → X.	
	Add Instruction address to Index and Jump	AIXJ			$(X) + I_v \rightarrow X$ } Jump to D right address if (X) ≠ D left address	
	Subtract Instruction address from Index and Jump	SIXJ			$(X) - I_v \rightarrow X$ }	
	Add Instruction address to Index and set Overflow	AIXO	L,R		$(X) + I_v \rightarrow X$ } Set overflow if (X) = D address	
	Subtract Instruction address from Index and set Overflow	SIXO			$(X) - I_v \rightarrow X$ }	
	Repeat Repeat	RPT	N,A,S	N,A,S*	The next instruction or instruction pair is repeated the number of times specified by the address part of the RPT. If RPT is left half instruction, next instruction is repeated; if RPT is right half instruction, next pair of instructions is repeated.	*N,A, and S are not optional and specify no modification, add to, and subtract from the index register(s) specified by the repeated instruction(s).
Extract	Extract from memory and Transfer to D	ETD			Extract: bit by bit logical multiply (M) · (Q)→D; or mask (M)→D according to (Q). 1. e.g. M · Q → D 1 0 0 0 0 0 1 1 1 0 1 0 2. (D)→A or (A) ± (D)→A.	Floating point mode is possible with EA and ES but is only in effect after the extraction.
	Transfer to A	ETA				
	Add	EA				
	Subtract	ES				
	Insert	EI			1. M Q A A after 2. When result stored 1 0 x x (A)→D, (D)→M. 0 0 x x 1 1 x 1 0 1 x 0	After EI: (D) = (M) · (Q).
	Insert and Store	EIS				
Special	Larger Word	LWD			*If (M) > (A), (M)→A and M address → JA, $0 \rightarrow JA_F$.	*Actually, (M)→D and (D) and (A) are compared in the alphanumeric sense.
	Smaller Word	SWD			*If (M) < (A), (M)→A and M address → JA, $0 \rightarrow JA_F$.	
	No Operation	†NOP			No operation	
	Halt	†HLT		L or R	Stop computation	†L or R specifies I_F as 0 or 1.
	Transfer (JA) to Memory	‡TJM			(JA)→ M address, $JA_F \rightarrow M_F$ via D.	
	Transfer Instruction address to JA	‡TIJ			Effective address JA; $0 \rightarrow JA_F$ if L, $1 \rightarrow JA_F$ if R.	‡L or R specifies the particular half word of M.
	Increase Address in memory	‡INCA			1 + M address → M address via D and JA; $0 \rightarrow JA_F$.	
	Inhibit Clearing of Overflow indicator	ICOS			Inhibits clearing of O.F. indicator	
	Inhibition on Clearing of Overflow indicator made Zero	ICOZ			Removes the inhibition on clearing the overflow indicator set by an ICOS	
	Transfer from Console Typewriter to Memory	TCM			One character → six right bit positions of M and D.	#The Skip instructions have a number of options described elsewhere in connection with the Input-Output instructions.
	Transfer from D to Console Typewriter	TDC			Transfer left (six bits) character of D → typewriter	
	Transfer from Toggle register to D	TTD			Word set up with toggle switches → D.	
	Transfer control to Input-Output	TIO			(D) → input-output control; execute this I-O instruction.	
	Skip if no Fault	#SKF	I-O		If no fault exists, the next instruction is skipped.	
	Skip Check	#SKC	I-O		Skip the next instruction if $I_V \geqq$ the number in the specified input counter.	
	(D) or (M) bit by bit Stored	DORMS			Binary ones from (D) or (M) → D, M (1+0=0+1=1+1=1;0+0=0).	
	Add Without Carry and Store	AWCS			(M) + (A) without carries → D, M (1+1=0+0=0;0+1=1+0=1).	

APPENDIX D

QUATERNARY REPRESENTATION OF PHILCO 2000 COMMANDS

The quaternary representation of PHILCO 2000 commands uses four digits in place of the 8 binary digits. The instruction CAQA, Clear Add Q in Absolute value, for example, is represented as follows:



The commands are listed in quaternary order to facilitate locating a binary command. Any quaternary configuration not shown is a command fault. Commands with an asterisk (*) are listed twice and differ <u>only</u> by their F bits, not in effect.

APPENDIX D (cont'd)

Quaternary	Mnemonic	Quaternary	Mnemonic
SPECIAL AND OTHER		JUMPS-LEFT	
0000	HLTL*	0200	JMPL
0001	JBTL	0201	JAZL
0002	ICOS	0202	JNOL
0003	NOPL*	0203	JOFL
0010	TIO	0210	JAPL
0011	TCM	0211	JANL
0012	SKC	0212	JAEQL
0013	TCXZ	0213	JAEDL
0020	TJML	0220	JQPL
0021	INCAL	0221	JQNL
0022	TIJL	0222	JQEL
0023	RPT*	0223	JQOL
0030	ETD	0230	JDPL
0031	DORMS	0231	JAGQFL
0032	EI	0232	JAGQL
0033	LWD	0233	JAGDL
TRANSFERS		INDEX-LEFT	
0100	CM	0300	TDXL
0101	TMA	0301	TDXLC
0102	TMQ	0302	TXDL
0103	TMD	0303	TXDLC
0110	TAM	0310	ADXL
0111	CA	0311	SDXL
0112	TAQ	0321	TIXZ
0113	TAD	0330	AIXJ
0120	TQM	0331	SIXJ
0121	TQA	0332	AIXOL
0122	CQ	0333	SIXOL
0123	TQD		
0130	TDM		
0131	TDA		
0132	TDQ		
0133	CD		

APPENDIX D (cont'd)

Quaternary	Mnemonic	Quaternary	Mnemonic
ADDITION		MULTIPLICATION	
1000	AM	1200	MM
1001	AMS	1201	MMS
1002	CAM	1202	MMR
1003	CAMS	1203	MMRS
1010	AMA	1210	MMA
1011	AMAS	1211	MMAS
1012	CAMA	1212	MMAR
1013	CAMAS	1213	MMARS
1020	AQ	1220	MA
1021	AQS	1221	MAS
1022	CAQ	1222	MAR
1023	CAQS	1223	MARS
1030	AQA	1230	MAA
1031	AQAS	1231	MAAS
1032	CAQA	1232	MAAR
1033	CAQAS	1233	MAARS
SUBTRACTION		DIVISION AND SPECIAL ARITHMETIC	
1100	SM	1300	DAQ
1101	SMS	1301	DAQS
1102	CSM	1302	DA
1103	CSMS	1303	DAS
1110	SMA	1320	MAD
1111	SMAS	1321	MSU
1112	CSMA	1322	EA
1113	CSMAS	1323	ES
1120	SQ	1330	AD
1121	SQS	1331	SD
1122	CSQ		
1123	CSQS		
1130	SQA		
1131	SQAS		
1132	CSQA		
1133	CSQAS		

APPENDIX D (cont'd)

SPECIAL AND OTHER

Quaternary	Mnemonic
2000	HLTR*
2001	JBTR
2002	ICOZ
2003	NOPR*
2010	TTD
2011	TDC
2012	SKF
2013	TCXS
2020	TJMR
2021	INCAR
2022	TIJR
2023	RPT*
2030	ETA
2031	AWCS
2032	EIS
2033	SWD

SHIFTS

Quaternary	Mnemonic
2100	SLAQ
2101	SRAQ
2102	SLAQN
2103	SRAQN
2110	SLA
2111	SRA
2112	SLAN
2113	SRAN
2120	SLQ
2121	SRQ
2122	SLQN
2123	SRQN
2130	SCD
2131	SRD
2132	SCD
2133	SRDN

JUMPS-RIGHT

Quaternary	Mnemonic
2200	JMPR
2201	JAZR
2202	JNOR
2203	JOFR
2210	JAPR
2211	JANR
2212	JAEQR
2213	JAEDR
2220	JQPR
2221	JQNR
2222	JQER
2223	JQOR
2230	JDPR
2231	JAGQFR
2232	JAGQR
2233	JAGDR

INDEX-RIGHT

Quaternary	Mnemonic
2300	TDXR
2301	TDXRC
2302	TXDR
2303	TXDRC
2310	ADXR
2311	SDXR
2321	TIXS
2330	AIXJ
2331	SIXJ
2332	AIXOR
2333	SIXOR

FLOATING POINT ARITHMETIC

Quaternary	Mnemonic
3000	
.	
.	
.	
3331	

APPENDIX E

FLOATING POINT ARITHMETIC

FLOATING POINT DECIMAL ARITHMETIC

Almost everyone is familiar with the representation of very large or very small numbers by the convenient device of separating such numbers into three parts: the mantissa, the base, and the characteristic or exponent. For example, the very small decimal number: 0.00000000625 can be compactly represented as:

$$\underbrace{6.25}_{\substack{\text{MANTISSA}\\ \text{(Significant Digits)}}} \times \underset{\substack{\uparrow \\ \text{BASE OF NUMBER} \\ \text{SYSTEM}}}{10}{}^{-9}_{\;\uparrow\text{EXPONENT}}$$

Equally valid representations are

$$625 \times 10^{-11}$$

$$62.5 \times 10^{-10}$$

or

$$.625 \times 10^{-8}$$

Note that each of these numbers was obtained from the first representation by shifting the decimal point and adding or subtracting from the exponent the number equal to the number of shifts. The name floating point arises from the fact that the decimal point "floats" in reference to the mantissa -- its position being determined by the exponent. Note also that both the mantissa and the exponent of a floating point number can be either positive or negative, and that the exponent must be an integral power of ten.

Normalized Floating Point Values

Floating point values are often expressed in normalized form. A normalized decimal floating point number is one whose most significant digit immediately follows the decimal point. Thus, of the above four floating point numbers only the last number, $.625 \times 10^{-8}$, is in normalized form. Normalization allows for the maximum number of significant digits in a given number of digit positions.

Multiplication and Division

In multiplication the mantissas are multiplied and the exponents added.

$$\begin{array}{l} \left.\begin{array}{r} 62.5 \times 10^{-10} \\ \times\ 2.5 \times 10^{\ 5} \end{array}\right\} \text{Adding exponents} \\ \hline \ \ 3125 \\ 1250 \\ \hline 156.25 \times 10^{-5} \end{array}$$

Normalized, the answer becomes

$$.15625 \times 10^{-2} .$$

In division the mantissas are divided and the exponents subtracted

$$\frac{62.5 \times 10^{-10}}{2.5 \times 10^{\ 5}} = \frac{62.5}{2.5} \times 10^{-10-5} = 25 \times 10^{-15}$$

or $.25 \times 10^{-13}$ normalized.

Addition and Subtraction

In the preceding two examples the multiplication and division proceeded without regard to the relative values of the exponents -- they were simply added or subtracted, respectively. However, addition and subtraction cannot be performed unless the exponents are the same. For example, the following addition:

$$\begin{array}{r} 1.25 \ \ \times 10^{-7} \\ +\ 0.375 \times 10^{-5} \\ \hline \end{array}$$

cannot be performed until the exponents are equalized. This can be accomplished by moving the decimal point of the addend two places right and subtracting two from its exponent, after which the addition proceeds

$$\begin{array}{r} 1.25 \times 10^{-7} \\ +\ 37.5 \ \ \times 10^{-7} \\ \hline 38.75 \times 10^{-7} \end{array}$$

or

$.3875 \times 10^{-5}$ normalized.

An equally valid procedure would have been to adjust the value of the augend and then perform the addition.

The similarity between the treatment of exponents in floating point arithmetic (performed manually) and that of scale factors in fixed point arithmetic is shown by the following chart:

		EXPONENTS	SCALE FACTORS
Multiplication	:	Added	Added
Division	:	Subtracted	Subtracted
Addition and Subtraction	:	Equalized	Equalized

FLOATING POINT BINARY ARITHMETIC

Just as numbers can be represented in decimal floating point form so can they be represented in binary floating point form. The essential difference is that the exponent is an integral power of the base two rather than ten. Of course, both mantissa and exponent are expressed in binary.

For example, the number, 127, can be represented in decimal form as $.127 \times 10^3$.

It could also be expressed in binary form as

$$\frac{127}{2^7} \times 2^7 \text{ or } \frac{127}{128} \times 2^7$$

which is also normalized.

The treatment of the exponents in floating point binary arithmetic is the same as that of floating point decimal arithmetic. In multiplication, exponents are added; in division, they are subtracted. Before addition or subtraction can proceed exponents must be made equal. Equalizing exponents is accomplished by multiplying or dividing the mantissa of one operand by a power of two and subtracting or adding to its exponent, respectively. The arithmetic operations themselves are performed in two's complement arithmetic as described in Chapter V.

Addition

$$\begin{array}{rlcrl} 106.0 & \times 2^0 & = & 106.0 & \times 2^0 \\ +\ 6.25 & \times 2^1 & = & +\ 12.5 & \times 2^0 \\ \cline{1-2} \cline{4-5} & & & 118.5 & \times 2^0 \end{array}$$

$$\begin{array}{rlcrl} -127.0 & \times 2^0 & = & -127.0 & \times 2^0 \\ +\ 6.25 & \times 2^1 & = & +\ 12.5 & \times 2^0 \\ \cline{1-2} \cline{4-5} & & & -114.5 & \times 2^0 \end{array}$$

Subtraction

$$\begin{array}{rcr} 63.25 \times 2^3 & = & 63.25 \times 2^3 \\ -\ 5.75 \times 2^4 & = & -11.50 \times 2^3 \\ \hline & & 51.75 \times 2^3 \end{array}$$

$$\begin{array}{rcr} 64.00 \times 2^{-1} & = & 8.00 \times 2^2 \\ -(-5.75 \times 2^{2}) & = & +\ 5.75 \times 2^2 \\ \hline & & 13.75 \times 2^2 \end{array}$$

Multiplication

$$\begin{array}{rl} 12.5 \times 2^3 & \\ \times\ 5.25 \times 2^2 & \text{Adding exponents} \\ \hline 625 & \\ 250 & \\ 625 & \\ \hline 65.625 \times 2^5 & \longleftarrow \end{array}$$

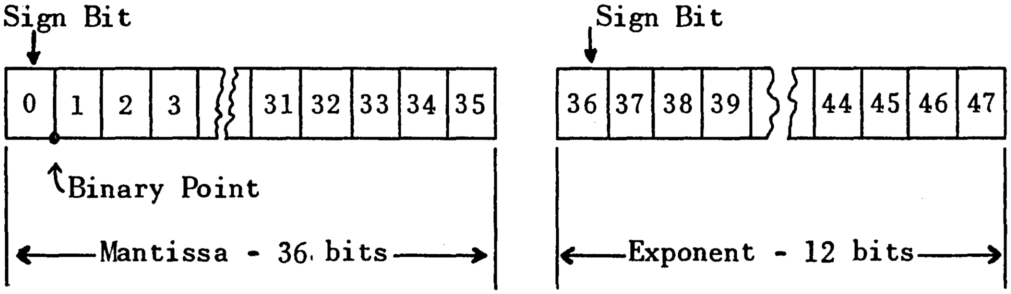
Division

$$\frac{62.5 \times 2^3}{2.5 \times 2^2} = \frac{62.5}{2.5} \times \overset{\substack{\text{Subtracting exponents} \\ \downarrow}}{2^{3-2}} = 25 \times 2^1$$

FLOATING POINT ARITHMETIC IN THE PHILCO 2000

For the PHILCO 2000, floating point arithmetic is specified by writing an F in front of the mnemonic command of any of the fixed point arithmetic instructions. The advantages of using floating point arithmetic are that the programmer is relieved of the necessity of scaling and that a greater range of values can be expressed in computer words.

During the execution of floating point arithmetic operations, all arithmetic registers are treated as if they were divided into two parts: a 36-bit mantissa and a 12-bit exponent, as shown below.



The mantissa is considered to be fractional, but the exponent represents an integral power of two. Both can be either positive or negative as indicated by their sign bits.

The number

$$127 = \frac{127}{2^7} \times 2^7 = \frac{127}{128} \times 2^7 ,$$

has the following PHILCO 2000 floating point form:

0	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	1	1	1
0	1	2	3	4	5	6	7	8	35	36	37	38	39	43	44	45	46	47

Mantissas of negative numbers and negative exponents are represented in two's complement form. Thus, the number

$$- \frac{127}{128} \times 2^{-7}$$

would be represented in floating point form as the following PHILCO 2000 word:

1	0	0	0	0	0	0	1	0	0	1	1	1	1	1	1	0	0	1
0	1	2	3	4	5	6	7	8	35	36	37	38	39	43	44	45	46	47

Arithmetic operations performed on the mantissas of floating point numbers are essentially the same as those of fixed point arithmetic. The differences between the two types of arithmetic lay in the treatment of the exponents and the handling of overflow and normalization which will be covered shortly.

Addition and Subtraction

In addition and subtraction, the mantissa of the value with smaller exponent is shifted to the right the number of places equal to the difference between the exponents. If the value with the smaller exponent is the addend or subtrahend, its mantissa is shifted in the D register after the original value has been rewritten in memory, if necessary. If the absolute value of the difference is greater than 35, the operand with the smaller exponent is made floating point zero, and no time is taken up for shifting. If there is no difference between exponents, the mantissa of the addend or subtrahend is shifted in the D Register one place to the right before the arithmetic starts. However, since the arithmetic proceeds with the unshifted value of the addend or subtrahend from the slave register of D, the shift in D does not affect the result.

If the value in the D Register does not have to be shifted or changed and the Store option is not used, the value in D is rewritten into memory while the mantissa of the value in the A Register is being shifted and the arithmetic is taking place. This time saving function can take place only if the exponent of the augend (or minuend) is smaller than that of the addend (or subtrahend). In any case, the addition or subtraction proceeds with the mantissas being added or subtracted and the exponent of the result set equal to the larger exponent.

For example, the following numbers are to be added:

$$1/2 \times 2^3 + 1/2 \times 2^2$$

The difference between the exponents is 1. Therefore, multiplying and dividing the number with the smaller exponent by 2^1 (equivalent to setting the smaller exponent equal to the larger and shifting the mantissa to the right) yields

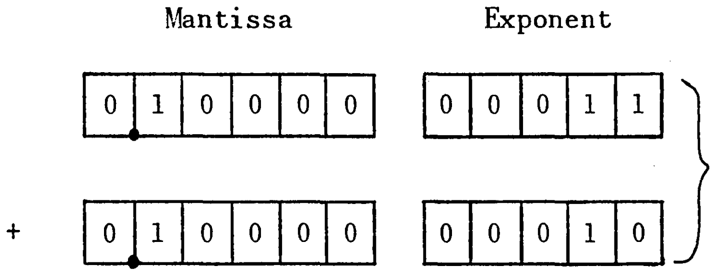
$$\left(\frac{1}{2 \cdot 2^1}\right) \times \ (2^2 \cdot 2^1) \ = \ 1/4 \times 2^3$$

and the addition is performed:

$$1/2 \times 2^3 + 1/4 \times 2^3 = 3/4 \times 2^3 = 6.$$

For convenience, in this example and those to follow, eleven bit registers will be shown rather than 48-bit registers as in the PHILCO 2000. Also, to emphasize the split function of the registers in floating point arithmetic, they will be shown in two parts.

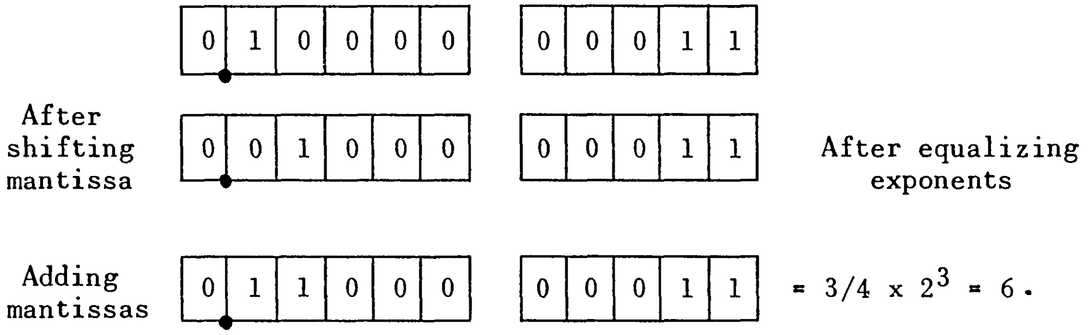
Expressed in floating point binary form, the preceding example becomes:



	Mantissa	Exponent
	0 1 0 0 0 0	0 0 0 1 1
+	0 1 0 0 0 0	0 0 0 1 0

Difference between exponents equals 1.

Before the addition is performed, the mantissa of the addend is numerically shifted one place right (i.e., divided by 2), and its exponent is assumed to be equal to three (although the exponent of the value in D is not changed):



Another example is

$$1/2 \times 2^{3} + 1/2 \times 2^{-1}.$$

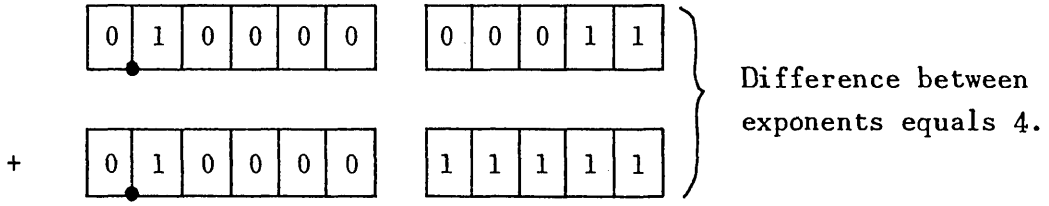
The difference between the exponents is 4. Therefore, multiplying and dividing by 2^4, the addend becomes

$$\left(\frac{1}{2 \cdot 2^{4}}\right) \times (2^{-1} \cdot 2^{4}) = \frac{1}{2^{5}} \times 2^{3} = \frac{1}{32} \times 2^{3}$$

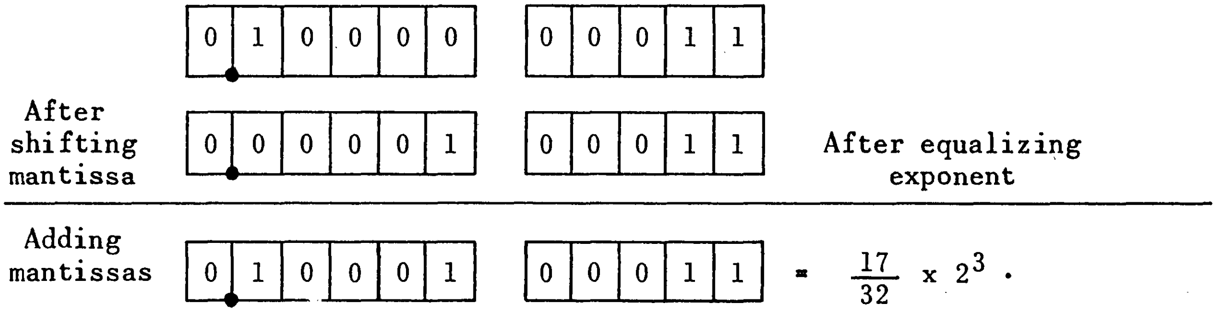
and the addition is performed:

$$(1/2 \times 2^{3}) + (1/32 \times 2^{3}) = \frac{17}{32} \times 2^{3} = 4.25.$$

In binary, the example becomes



Since division by 2^4 is equivalent to a right shift of four places, the mantissa of the addend is numerically shifted right four places and its exponent is assumed to be equal to three.



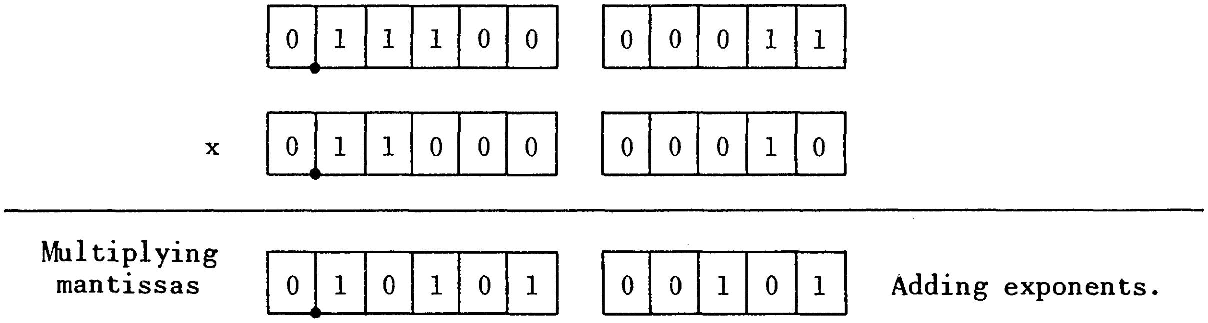
Except for the fact that the complement of the mantissa of the subtrahend is added to the mantissa of the minuend, floating point subtraction is performed in the same way.

Multiplication and Division

When multiplying two floating point numbers, the computer adds the exponents and multiplies the mantissas. For example, the following numbers are to be multiplied:

$$(7/8 \times 2^3) \times (3/4 \times 2^2) = 21/32 \times 2^5 = 21$$

Expressed in floating point binary the example becomes


0 1 1 1 0 0   0 0 0 1 1
x   0 1 1 0 0 0   0 0 0 1 0
Multiplying mantissas   0 1 0 1 0 1   0 0 1 0 1   Adding exponents.

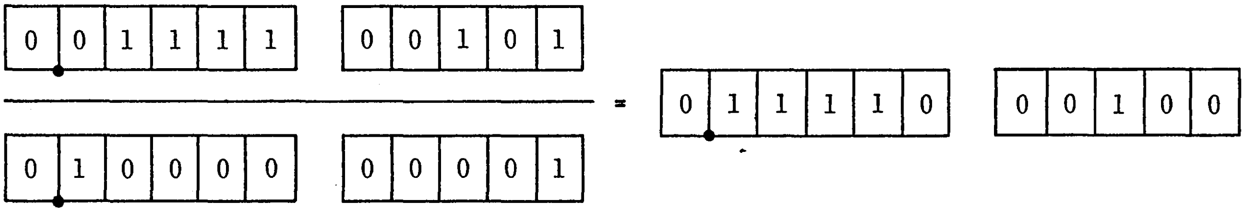
Rounding in floating point multiplication is accomplished by adding one to the most significant bit of the mantissa of the Q Register, which contains the minor half of a double length floating point product. The original contents of the Q Register are then restored. In unrounded multiplication, the exponents of both halves of the product are the same. (Note that, except for the split registers, the functions of the arithmetic registers are the same as those in fixed point arithmetic. Refer to Chapter III, page 52, Functions of Arithmetic Registers in Arithmetic Operations.)

In floating point division, the arithmetic section, subtracts the exponent of the divisor from the exponent of the dividend and divides the mantissa of the dividend by the mantissa of the divisor. In double length division, the exponent of the dividend used is only that in the A Register; the exponent in the Q Register is ignored.

Example:

$$(15/32 \times 2^5) \div (1/2 \times 2^1) = 15/16 \times 2^4 = 15.$$

In floating point form the operands are:



Multiplication and division begin immediately after the read from memory is completed, except in the case of FMAD and FMSU. The rewrite time is thereby overlapped by the arithmetic time. For FMAD and FMSU, the multiplicand is restored before arithmetic begins.

Floating Point Number Range

The largest representable mantissa is a zero in the sign position followed by 35 ones; the largest exponent is a zero followed by 11 ones.

Mantissa

0	1	2	3	4	31	32	33	34	35
0	1	1	1	1	1	1	1	1	1

Exponent

36	37	38	39	40	45	46	47
0	1	1	1	1	1	1	1

This number is equivalent to .9999999... x 2^{2047} , which is very close to but not equal to +1 x 2^{2047}. This value is equal to slightly more than 10^{616}.

Similarly, the smallest computer number is equal to minus one times the largest exponent, i.e., -1 x 2^{2047}.

Mantissa

0	1	2	3	4	31	32	33	34	35
1	0	0	0	0	0	0	0	0	0

Exponent

36	37	38	39	40	45	46	47
0	1	1	1	1	1	1	1

However, the normalized non-zero floating point value which is smallest in magnitude is

Mantissa

0	1	2	3	4	31	32	33	34	35
0	1	0	0	0	0	0	0	0	0

Exponent

36	37	38	39	40	45	46	47
1	0	0	0	0	0	0	0

.

This value is equivalent to .5 x 2^{-2048} (or 1 x 2^{-2049}) which is slightly less than 10^{-617}.

The range of non-zero magnitudes in floating point representation, therefore, is from slightly more than 10^{616} to slightly less than 10^{-617}.

Mantissa Overflow

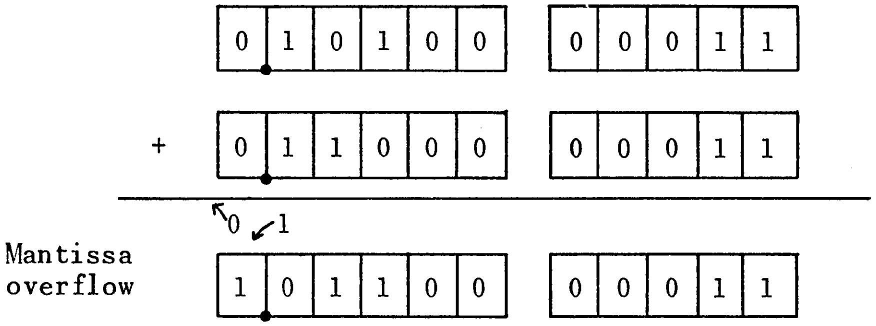
Since mantissas of floating point numbers represent binary fractions in the arithmetic section, mantissa overflow results when the computer attempts to produce a mantissa equal to plus one or less than minus one. Unlike fixed point arithmetic, the overflow indicator is not set although it is cleared prior to each floating point operation. Instead, mantissa overflow is automatically corrected, except in the case of division. In division, potential overflow is detected and prevented before the division is performed.

When mantissa overflow occurs during an addition, subtraction, or multiplication, the mantissa of the result is shifted right one place and the exponent is increased by one.

For example, the following numbers are to be added:

$$5/8 \times 2^3 + 3/4 \times 2^3 = 11/8 \times 2^3.$$

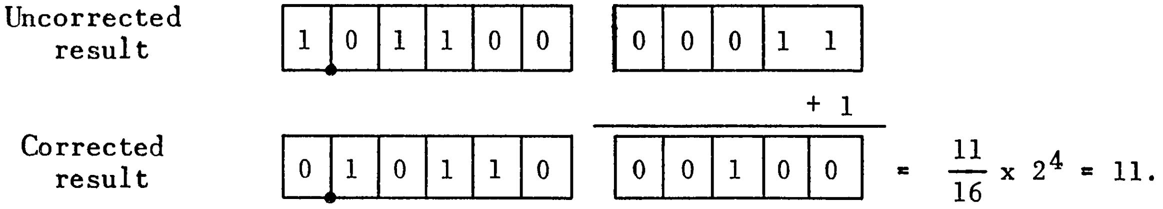
Expressed in binary, these numbers are



```
                 0.1 0 1 0 0    0 0 0 1 1

            +    0.1 1 0 0 0    0 0 0 1 1
         ------------------------------------
                 ↖0 ↙1
Mantissa
overflow         1.0 1 1 0 0    0 0 0 1 1
```

As shown above, mantissa overflow resulted. The computer then shifts the mantissa one place right and adds one to the exponent, as follows:



```
Uncorrected      1.0 1 1 0 0    0 0 0 1 1
  result
                                      + 1
                               -----------
 Corrected       0.1 0 1 1 0    0 0 1 0 0   =  11/16 x 2^4 = 11.
  result
```

In floating point division, the mantissa is tested for overflow before division is performed. If overflow is detected, the dividend is shifted right one place; the exponent is increased by one, and another attempt is made to perform the division. This process is repeated until either the dividend has become smaller in absolute value than the divisor, or the dividend has shifted 36 places. Any time the dividend has been shifted 36 places, division by zero was attempted. The Exponent Fault neon is lighted; a jump to memory location 00000 is effected, and the address of the next instruction word is placed in JA. The F bit of JA is set to 0 if the fault occurred in a left half instruction or to 1 if the fault occurred in a right half instruction.

The following is an example of potential overflow in division:

$$(81/128 \times 2^7) \div (9/16 \times 2^4) = 9/8 \times 2^3.$$

or in floating point binary

0	1	0	1	0	0	0	1	0	0	0		0	0	1	1	1	
					0	1	0	0	1	0		0	0	1	0	0	= Potential overflow.

Since the dividend, 81/128, is greater than 9/16, the test for divisibility indicates potential overflow. Therefore, the mantissa of the dividend is numerically shifted right one place and its exponent is increased by one.

Corrected dividend	0	0	1	0	1	0	0	0	1	0	0		0	1	0	0	0	$= \frac{81}{256} \times 2^8$
						0	1	0	0	1	0		0	0	1	0	0	$= \frac{9}{16} \times 2^4$

The division can now be performed since $\frac{81}{256}$ is less than $\frac{9}{16}$. The result is as follows:

0	1	0	0	1	0		0	0	1	0	0	$= \frac{9}{16} \times 2^4 = 9.$

Normalization

A floating point number is in normalized form if the most significant digit of the mantissa immediately follows the binary point. This is equivalent to requiring that the sign bit of the mantissa and the adjacent bit be different. Thus, of the following four numbers only two are normalized, as indicated:

$\frac{3}{4} \times 2^3 =$	0	1	1	0	0	0		0	0	0	1	1	Normalized
$-\frac{5}{8} \times 2^3 =$	1	0	1	1	0	0		0	0	0	1	1	Normalized
$\frac{7}{16} \times 2^3 =$	0	0	1	1	1	0		0	0	0	1	1	Not normalized
$-\frac{5}{16} \times 2^3 =$	1	1	0	1	1	0		0	0	0	1	1	Not normalized

Although original operands need not be in normalized form, the computer will always attempt to normalize the result of a floating point arithmetic operation. The reason for normalizing is to allow for the maximum number of significant digits in arithmetic results. The method by which the computer normalizes numbers is as follows:

The first two bits of the result are examined. If they are alike, the mantissa of the result is numerically shifted one place left and the exponent is reduced by one. This procedure is repeated until either a normalized value results or 36 shifts have been made. If the result is still unnormalized at the end of 36 shifts, the result is made floating point zero, which is represented as

0	0	0	0	≀	0	0	0		1	0	0	0	0	0	0	0	0	0	0	0

$= 0 \times 2^{-2048}$.

Complete normalization takes place in the following two examples:

Example 1

$$(1/2 \times 2^3) - (1/2 \times 2^2) = (1/2 \times 2^3) - (1/4 \times 2^3) = 1/4 \times 2^3$$

In floating point form the operands are

	0	1	0	0	0	0		0	0	0	1	1
+	1	1	0	0	0	0		0	0	0	1	0

Unequal exponents.

After shifting and equalizing, the operands are

	0	1	0	0	0	0		0	0	0	1	1
+	1	1	1	0	0	0		0	0	0	1	1
	0	0	1	0	0	0		0	0	0	1	1

$= \frac{1}{4} \times 2^3$.

Since the first two bits of the mantissa are the same, the result is not in normalized form. Normalization is accomplished by numerically shifting the mantissa left one place and decreasing the exponent by one.

```
 0 0 1 0 0 0     0 0 0 1 1  \
                             > Subtracting 1
Shifting left               |
 one place     + 1 1 1 1 1  /
               -------------
 0 1 0 0 0 0     0 0 0 1 0
```

$= \frac{1}{2} \times 2^2 = 2$

Example 2

$$(17/32 \times 2^5) \times (1/8 \times 2^3) = 17/256 \times 2^8$$

In floating point form the operands are

```
     0 1 0 0 0 1     0 0 1 0 1  \
                                 > Adding
                                |  exponents
   x 0 0 0 1 0 0     0 0 0 1 1  /
---------------------------------------------
0 0 0 0 1 0 0 0 1 0 0     0 1 0 0 0
```

$= \frac{17}{256} \times 2^8$.

The result is normalized by shifting the mantissa left three times and by subtracting 1 from the exponent three times. The normalized result is

```
0 1 0 0 0 1 0 0 0 0 0     0 0 1 0 1
```

$= \frac{17}{32} \times 2^5 = 17.$

Exponent Overflow and Underflow

In the arithmetic section, exponent overflow occurs when an attempt is made to produce a floating point number which would have an exponent greater than +2047. This is the largest possible exponent and is represented as follows:

```
Sign bit
   |
   v
   0 1 1 1 1 1 1 1 1 1 1 1   = +2047.
          Exponent
```

By definition, exponent overflow results whenever the carry in to the exponent sign bit is one and the carry out is zero.

Exponent overflow may occur during multiplication or division or during a correction cycle for mantissa overflow. If it occurs during the operation, the overflow may disappear after the result has been normalized. If exponent overflow still exists after normalizing or if it had occurred during a mantissa correction cycle, the Exponent Fault neon is lighted; a jump to memory location 00000 is effected, and the address of the next instruction word is placed in JA. The F bit of JA is set to 0 if the fault occurred in a left half instruction or to 1 if the fault occurred in a right half instruction.

Exponent underflow occurs when an attempt is made to produce an exponent smaller than -2048 and is defined as a carry into the sign bit of zero and a carry out of one. In the computer, -2048 is the smallest possible exponent and is represented as follows:

Sign bit

1	0	0	0	0	0	0	0	0	0	0	0

= -2048.

Exponent

Exponent underflow may occur during multiplication or division or during any normalization cycle. If it occurs prior to normalization, the result is made zero. If it occurs during a normalization cycle, the underflow may correct a previous exponent overflow. If there had been no previous exponent overflow and underflow occurs during a normalization cycle, the cycle is discontinued and processing continues.

Exponent overflow is automatically corrected in the following example:

$$(1/2 \times 2^{2047}) \times (3/8 \times 2^{1}) = 3/16 \times 2^{2048}$$ Exponent overflow.

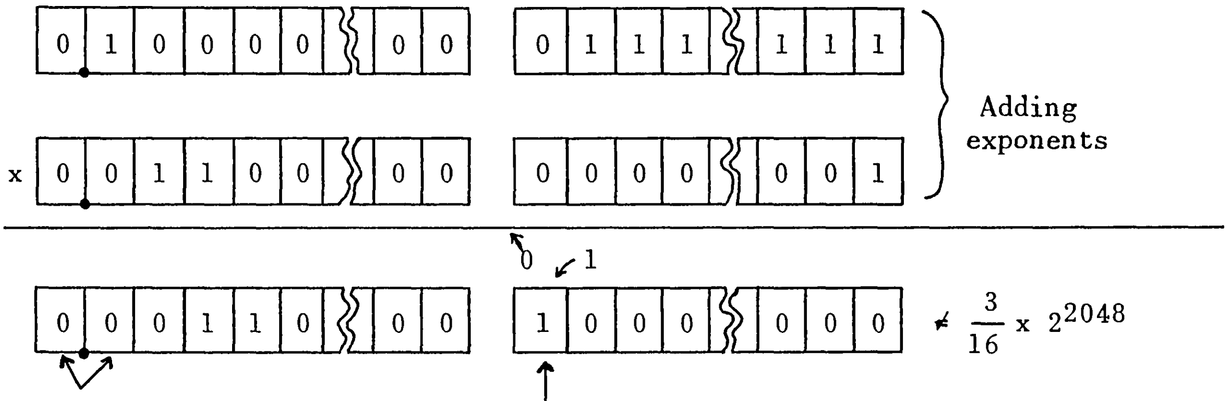
The result is normalized as follows:

$$3/16 \times 2^{2048} = 3/8 \times 2^{2047} = 3/4 \times 2^{2046}.$$

Exponent overflow corrected

Whenever, however, floating point operation, [illegible] is needed, floating point [illegible] automatically [illegible]. The [illegible] normally [illegible] normalization. Floating point arithmetic will therefore, always yield normalized results.

The multiplication in binary of the preceding example would proceed as follows:

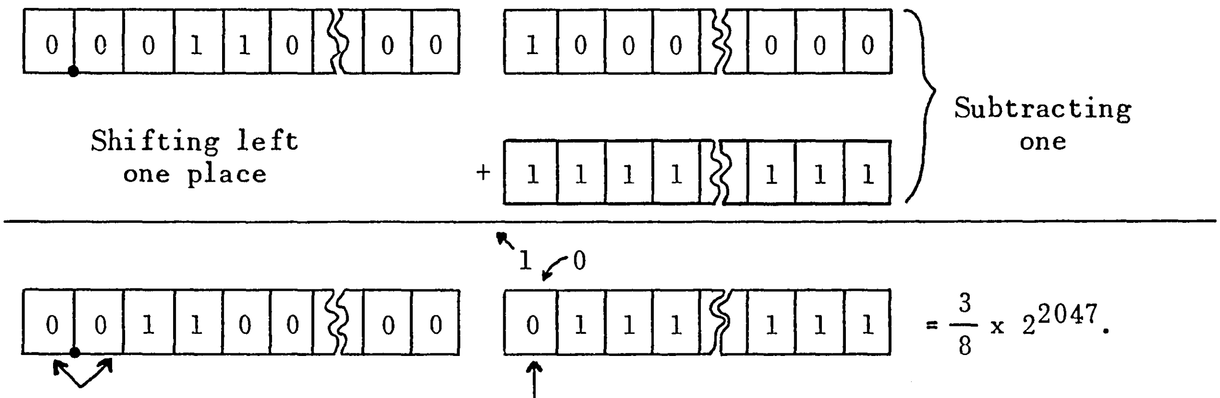


```
  0.1 0 0 0 0 ... 0 0     0 1 1 1 ... 1 1 1  ⎫
                                             ⎬ Adding exponents
x 0.0 1 1 0 0 ... 0 0     0 0 0 0 ... 0 0 1  ⎭
----------------------------------------------
                         0 1
  0.0 0 1 1 0 ... 0 0     1 0 0 0 ... 0 0 0   ≠ 3/16 x 2^2048
```

Mantissa unnormalized

Exponent overflow

Since the result is unnormalized, the mantissa is shifted left one place and the exponent is decreased by one as follows:

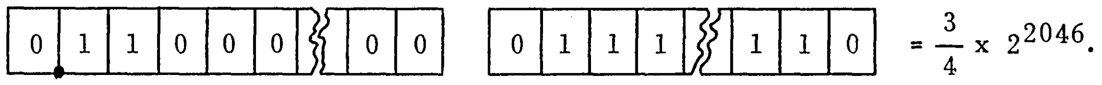


```
  0.0 0 1 1 0 ... 0 0     1 0 0 0 ... 0 0 0  ⎫
                                             ⎬ Subtracting one
  Shifting left         + 1 1 1 1 ... 1 1 1  ⎭
  one place
----------------------------------------------
                         1 0
  0.0 1 1 0 0 ... 0 0     0 1 1 1 ... 1 1 1   = 3/8 x 2^2047.
```

Mantissa unnormalized

Exponent overflow corrected

A second shift left is performed; one is subtracted from the exponent, and the normalized answer is obtained as follows:



```
  0.1 1 0 0 0 ... 0 0     0 1 1 1 ... 1 1 0   = 3/4 x 2^2046.
```

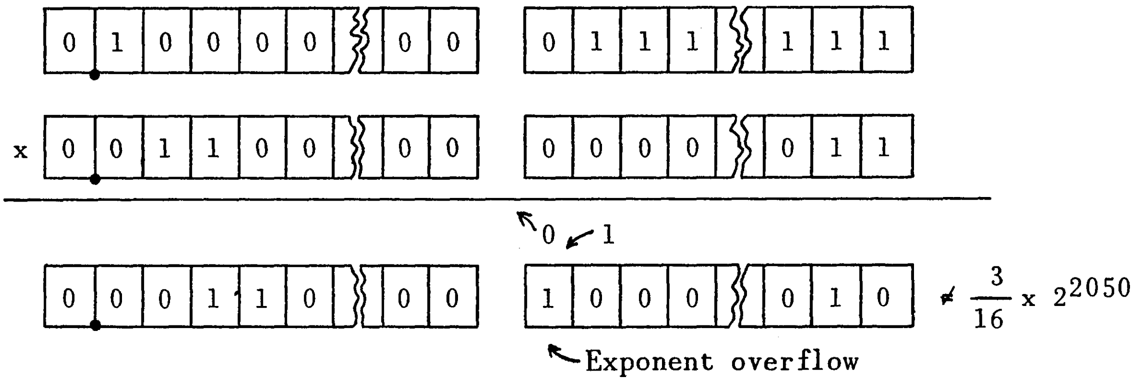
Exponent overflow can not be automatically corrected in the following example:

$(1/2 \times 2^{2047}) \times (3/8 \times 2^3) = 3/16 \times 2^{2050}$ ← Exponent overflow.

The result is normalized as follows:

$3/16 \times 2^{2050} = 3/8 \times 2^{2049} = 3/4 \times 2^{2048}$ ← Exponent overflow remaining.

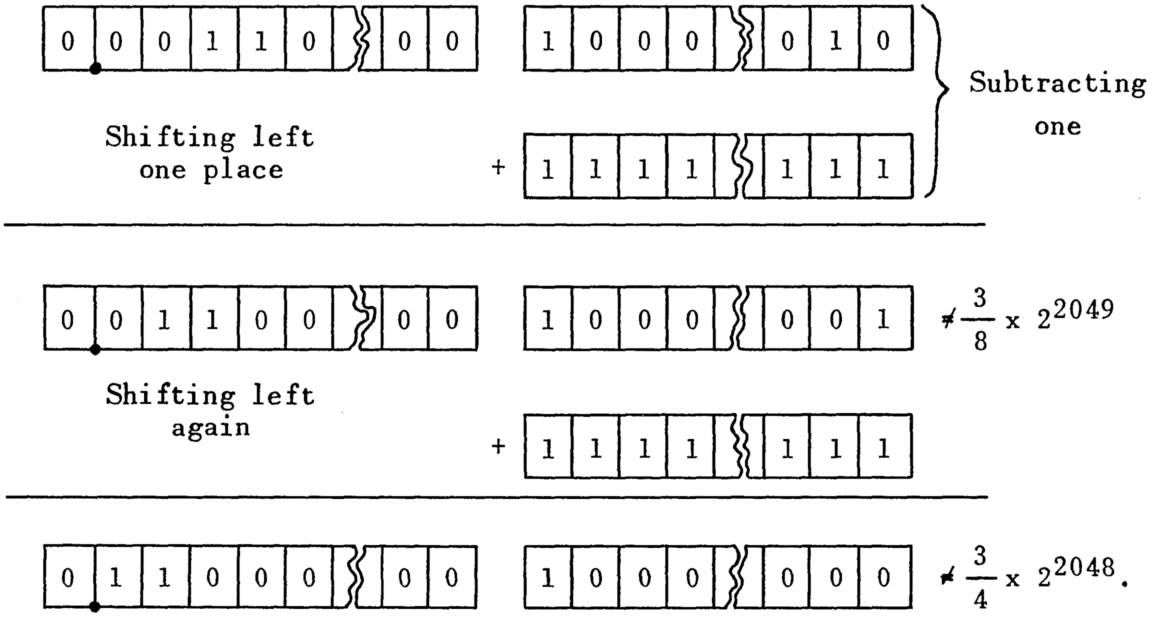
The multiplication in binary of the preceding example would proceed as follows:



```
  0.1 0 0 0 0 ... 0 0     0 1 1 1 ... 1 1 1
x 0.0 1 1 0 0 ... 0 0     0 0 0 0 ... 0 1 1
-------------------------------------------
                           0   1
  0.0 0 1 1 0 ... 0 0     1 0 0 0 ... 0 1 0
```

$\neq \frac{3}{16} x\ 2^{2050}$

The result is normalized as follows:



```
  0.0 0 1 1 0 ... 0 0     1 0 0 0 ... 0 1 0
                        + 1 1 1 1 ... 1 1 1
-------------------------------------------
  0.0 1 1 0 0 ... 0 0     1 0 0 0 ... 0 0 1
                        + 1 1 1 1 ... 1 1 1
-------------------------------------------
  0.1 1 0 0 0 ... 0 0     1 0 0 0 ... 0 0 0
```

$\neq \frac{3}{8} x\ 2^{2049}$

$\neq \frac{3}{4} x\ 2^{2048}.$

In the preceding example, the normalization process did not eliminate the prior exponent overflow. Consequently, the Exponent Fault neon is lighted, a jump to memory location 00000 is effected, and the address of the next instruction word is placed in JA. The F bit of JA indicates whether the fault occurred in the right or left half of the instruction word before the one whose address is in JA.

PROGRAMMING FLOATING POINT OPERATIONS

The programming for floating point arithmetic is essentially the same as that for fixed point arithmetic, except that the numbers used must be in floating point form. TAC converts decimal data to binary floating point form and provides floating point constants.

Floating point constants may be pool or non-pool constants and have the form, F/Number Ec. This formula represents a decimal number multiplied by some power of ten, Ec.

For example, to express the decimal number $.127 \times 10^3$ in floating point binary form the programmer writes

F/.127 E3 or F/127

and TAC will produce the normalized floating point number

$$\frac{127}{128} \times 2^7$$

which would appear as the following PHILCO 2000 word:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47
0	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1

Mantissa

Exponent

In addition to the previously described fixed point instructions, the instruction JAGQF compares the magnitudes of floating point numbers.

JAGQF Jump if (A) are greater than or equal to (Q) in the Floating point sense.

The first step of this instruction transfers the contents of the Q Register to the D Register. The contents of A and D are then compared. As in all Jump instructions, the address of the next sequential instruction is placed in the JA Register. Both numbers must be in normalized from prior to the comparison.

Note that all floating point numbers produced by TAC and most of the results of arithmetic operations will be in normalized form. Should some situation arise in which the programmer must normalize a floating point number, he may use one of the floating point Clear Add instructions, such as FCAM.

Example

Add the floating point numbers in locations Y1 and Y2. Jump to COMPUTE if the number in Y3, which is normalized, is less than the sum.

COMMAND	ADDRESS AND REMARKS	
TMA	Y1	Floating point addition
FAM	Y2	Y1 + Y2 → A Register
TMQ	Y3	Y3 → Q Register
JAGQF	CØMPUTE	Jump if sum ≥ Y3

Rules of Thumb

a. If maximum speed is desired when executing FAM and FSM instructions and the magnitudes of the operand are known, place the smaller operand in the A Register. Then the equalizing of exponents can take place during the memory restore cycle of the transfer of the larger operand to the D Register.

b. Note that whenever floating point number systems are used, the normal laws of associativity are not always valid. When a mantissa is shifted right to equalize exponents, significant digits may be lost. In the example below, when 10^{-50} is added to 10^{+50} , all of the significance of 10^{-50} is lost.

For example,

$$(10^{-50} + 10^{+50}) -10^{+50} = 0$$

because

$$10^{-50} + 10^{+50} = 10^{+50}.$$

PHILCO
Famous for
QUALITY
the World Over

PHILCO CORPORATION

GOVERNMENT AND INDUSTRIAL GROUP — COMPUTER DIVISION

3900 Welsh Road
Willow Grove, Penna.