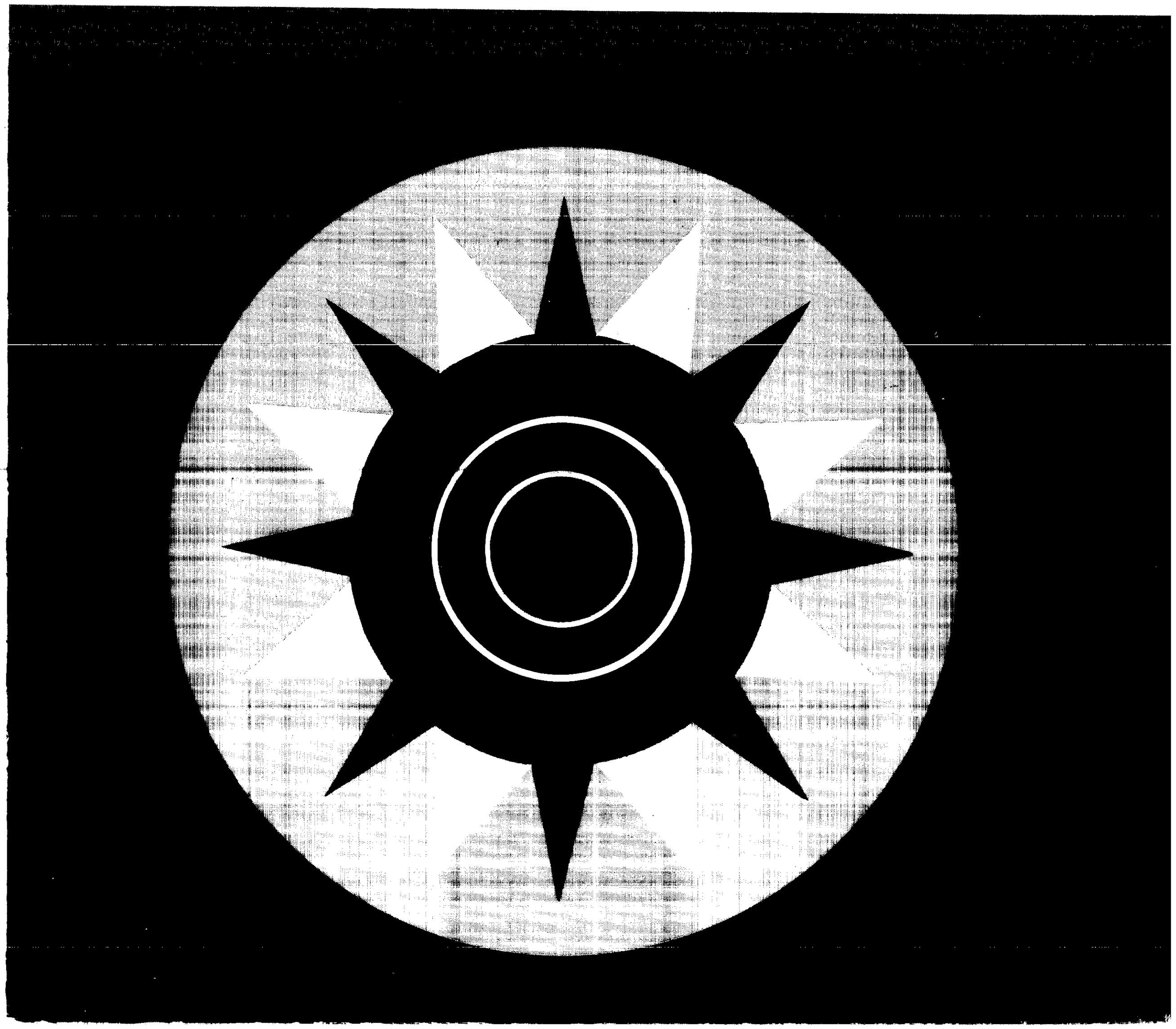
IBM 360

Programming and Computing

JAMES T. GOLDEN
RICHARD M. LEICHUS



IBM 360

Programming and Computing

PRENTICE-HALL INTERNATIONAL, INC., *London*
PRENTICE-HALL OF AUSTRALIA, PTY. LTD., *Sydney*
PRENTICE-HALL OF CANADA, LTD., *Toronto*
PRENTICE-HALL OF INDIA PRIVATE LTD., *New Delhi*
PRENTICE-HALL OF JAPAN, INC., *Tokyo*

IBM 360

Programming and Computing

JAMES T. GOLDEN
Product Programs Manager, IBM Corporation

RICHARD M. LEICHUS
Advisory Systems Engineer

Prentice-Hall, Inc., Englewood Cliffs, New Jersey

Current printing (last digit):
10 9 8 7 6 5 4 3 2 1

Library of Congress Catalog Number 67-25861
Printed in the United States of America

PREFACE

This book is written as an introduction to programming and operating systems. It may be used for self-study or as a text on computing and programming. A large number of worked examples are included to facilitate both of these objectives. Rather than develop a treatment of a hypothetical computer, we have selected a widely representative system, the IBM 360. In this way, in addition to learning the major concepts of computing and programming, the reader acquires a skill with direct, practical relevance.

Our guidelines in treating the important subject of operating systems were twofold. First, we show the reader why a particular facility is needed. Second, we introduce the reader to the programming techniques needed by this operating system facility, often by coding a representative sample of the system. After setting the stage in this way, we then describe, by means of simple control cards, how the facility is used. In this way, we hope to build a broader and deeper perspective for the reader. By treating an operating system as something more than just a "black box" with certain properties, we believe that the reader can begin to develop a sophistication in his own programming. We have selected the 360 Disk Operating System as the example operating system because of its widespread usage. The concepts developed here are directly transferable to Operating System/360. The reader who has an understanding of the material in Chaps. 9-12 should be able to learn OS/360 from its reference manuals on a self-study basis.

The reader who is familiar with another computing system should skim over Chaps. 1 and 2 and begin in Chap. 3. For a minimum introduction to the 360, we suggest Chaps. 1, 2, 3, Sec. 4.2, Chaps. 8, 9, 10 and 11. The worked examples are an integral part of the text and should be studied carefully. In addition, we recommend strongly that as many as possible of the problems which follow most chapters be worked out.

We wish to thank the International Business Machines Corporation for permission to include copyrighted material in this book.

J. T. Golden
R. M. Leichus

TABLE OF CONTENTS

Chapter 1

INTRODUCTION TO COMPUTING

1.1 Introduction

As a general statement, the material advance of man has come about because he has discovered ways of doing things with less effort, or for the same effort, getting more done. This is true whether we speak of human energy or animal or machine energy. The more important among these discoveries have caused immense changes. The invention of agriculture by paleolithic food gatherers made it possible for the first time for the tribe to produce more food than it could consume. With the surplus of labor this freed, the rise of civilization began. Much later, the steam engines of Newcomen and Watt, which perhaps more than any other invention made the Industrial Revolution fruitful or even possible, were in their earliest stages able to deliver only about 100 times the power of a single man, that is, two orders of magnitude more. They produced about 10 horsepower whereas a man working a winch or a hand pump, can deliver about one-tenth of a horsepower. There are many other examples where an advance of an order of magnitude or two have brought along great change. The automobile is approximately 10 times faster than a man walking, a jet plane, about 100 times faster. It is hardly necessary to comment on the extent of change since 1900 caused by these two innovations alone. Today, the electronic computer offers a far greater range of amplification of human abilities, for it is not 10 or 100 or even 1000 times faster than a human being. Present day computers and their extensions which are within the grasp of today's technology are able to do arithmetic 1, 000, 000 to 1, 000, 000, 000 times faster than a human being. Now this is not to say that computers are this much faster than people in every area of intellectual endeavor. What the human brain lacks in speed of operation, it more than makes up for in complexity of organization and information holding, or storage capacity. As a result, there are many everyday human activities which are beyond the capabilities of a computer. However, there are still a large number of human activities which can be handled by a computer provided we can describe them to the computer in a suitable fashion. Given the speed ratios above, and continuing progress in understanding how to solve problems on computers, it is clear that we have a tool which offers significant potential, for amplifying man's intellectual resources as well as extending his control over his environment--a potential which may offer far more than those innovations of the past which multiplied mechanical energy only.

In the brief span of less than 20 years since their introduction in the late 1940's and early 1950's, digital computers have progressed greatly. In speed, they have increased by a factor of over 10, 000, 000 when compared to the relay operated Mark I; in storage size, they have grown to almost 100, 000 times larger than their earlier counterparts. Today, well over a million people are engaged in operating computers, programming and preparing input to them, manufacturing, selling and servicing them. In ways ranging from calculating golf handicaps to auditing income tax returns, computers have entered the lives of all of us. In this context, it is interesting to note that the electronic digital computer is probably the most complicated device ever to come into such widespread usage. Yet this complexity is largely illusory. From the point of view of someone preparing a problem for a computer, most of the computer's intricacies are hidden from view. They are covered up, so to speak, by advances that have been made in communicating with computers--by advances in their programming. This book will describe the overall functioning of a widely representative computer--the IBM 360--and will introduce the reader to the general subject of programming as well as developing his ability to write programs for the 360.

Let us begin by discussing the functional organization of an electronic digital computer as shown in Fig. 1-1. The four principal parts of a computer are its control, input, output and working storage.

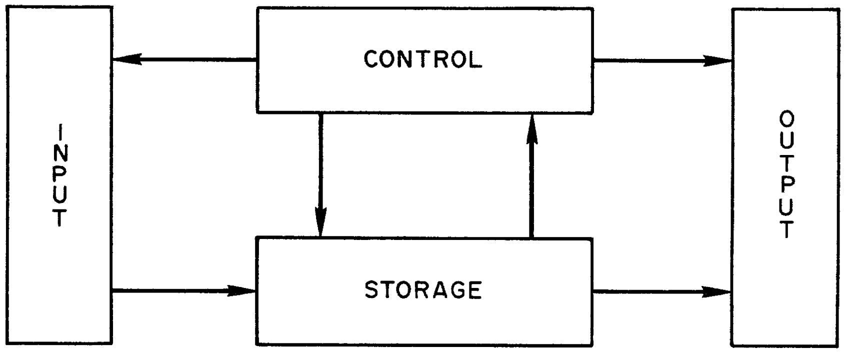


Fig. 1-1 Functional Components of a Digital Computer

We can better understand their functions and interrelationships by comparison with a human analogy--a highly trained and precise, but totally unimaginative, clerk. Our hypothetical clerk will have an in-basket where documents will be piled up awaiting processing by him. This is analogous to the computer's input device, usually a punched-card reader or a magnetic tape-drive. He will also have an out-basket where his results will be distributed to the people who will use them. This is analogous to the printer attached to a computer which prints its output. The control of the process, in the sense of deciding what to do next and how to do it resides in the clerk's memory and he is probably backed up by books or tables as an aid to his memory. The analogy here is to the working storage of the computer which we will describe later. (Indeed, because of this close analogy, computer storage is often termed

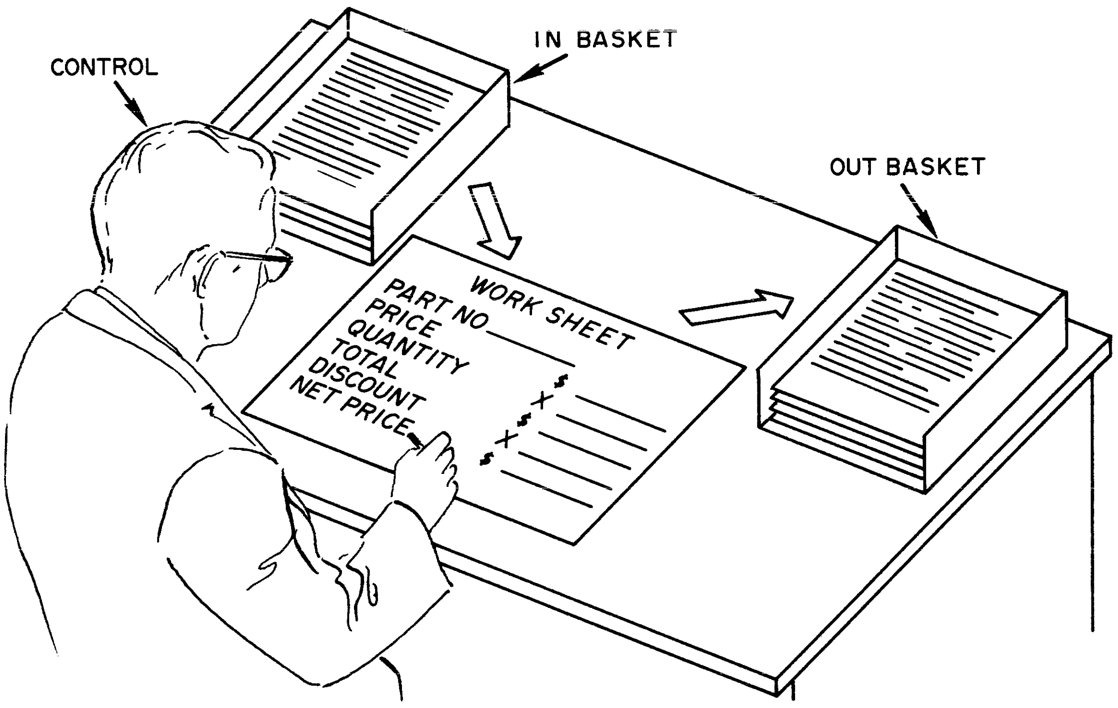


Fig. 1-2 Human Analogue of a Digital Computer

"memory".) We can think of our clerk as being programmed, or instructed, to do a certain set of tasks repeatedly according to certain rules. Let us assume that this clerk has been trained to handle only one type of transaction, pricing. He will have a pricing worksheet which we can think of as "working storage" since it contains the numbers he will work on. Figure 1-2 shows our clerk at work. Figure 1-3 shows a schematic of his instructions, or program, in the form of a flow chart. As we shall see, flow charting is an important aid in describing how a problem is to be solved by a computer. Note the almost extreme degree to which the clerk's tasks are spelled out explicitly. If there are no orders in the in-basket, the clerk is programmed to sit there and wait for an order. When one does reach his in-basket, his next instruction is to fetch it and check to see that the order is valid. That is, it should be legible and properly filled out. If anything is wrong, he is instructed to stop and ask for help. Next, he checks to see if the part number on the order sheet is a valid one, that is, it must appear in the price book. If it does not, he stops again to ask for help and so on. Note that this job definition in Fig. 1-3 is one which we expect our clerk to adhere to rigorously and never make any changes to. Now this inflexible approach would be an unrealistic way to train a clerk. We have allowed no room for imagination or individual initiative. Our clerk would price an order for 100-gross of

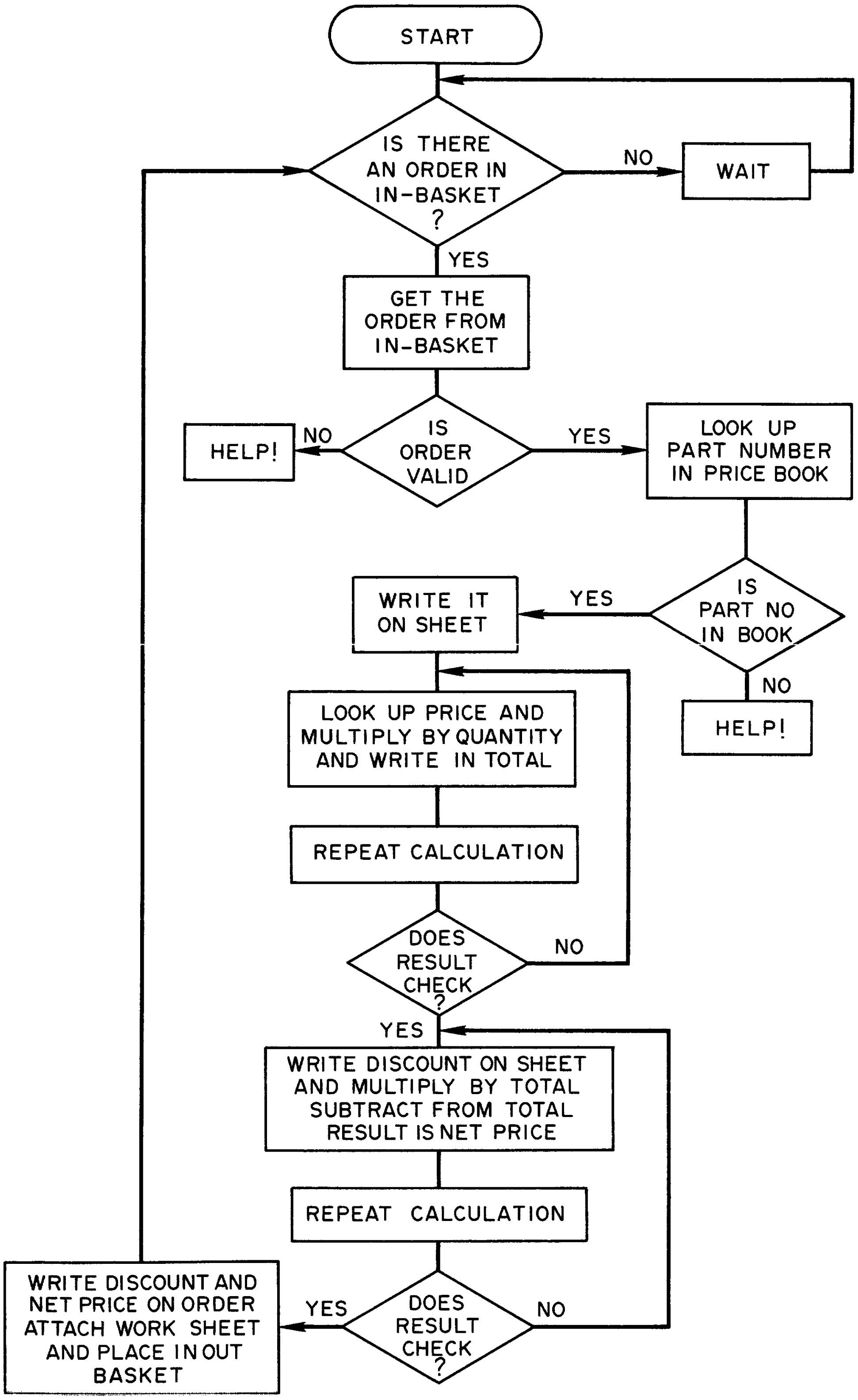

START
IS THERE AN ORDER IN IN-BASKET ?
NO
WAIT
YES
GET THE ORDER FROM IN-BASKET
IS ORDER VALID
NO
HELP!
YES
LOOK UP PART NUMBER IN PRICE BOOK
IS PART NO IN BOOK
YES
WRITE IT ON SHEET
NO
HELP!
LOOK UP PRICE AND MULTIPLY BY QUANTITY AND WRITE IN TOTAL
REPEAT CALCULATION
DOES RESULT CHECK ?
NO
YES
WRITE DISCOUNT ON SHEET AND MULTIPLY BY TOTAL SUBTRACT FROM TOTAL RESULT IS NET PRICE
REPEAT CALCULATION
DOES RESULT CHECK ?
NO
YES
WRITE DISCOUNT AND NET PRICE ON ORDER ATTACH WORK SHEET AND PLACE IN OUT BASKET

Fig. 1-3 Pricing Flow Chart

Carriage Bolts as readily as he would another order for 100-gross of Double Overhead Garage Doors, without suspecting that the latter one probably contains a typographical error since this quantity may represent several years' production. Again, our clerk would be stopped in his tracks if he had an order, say for 10-24 x 3/4 screws, which was missing the catalog part number. A real-world clerk might conclude that he still had a valid order since the only 10-24 x 3/4 screws in the catalog have part number 5816.

It is this extreme literal-mindedness, which people find constraining, when applied to their activities, that characterizes the way a computer operates. It will do precisely what it is told, tirelessly and without error, but not one iota more. If there are exceptions in a given problem, they must be recognized and programmed for explicitly, if the computer is to produce meaningful results.

The job description chart, or flow-chart, in Fig. 1-3 describes the logic of pricing in a way that is satisfactory as a description of a computer program. Minor changes, only, have to be made to take into account the different equipment available to the computer. The new flow-chart is shown in Fig. 1-4. The symbols used there are commonly accepted and will be used throughout this book.

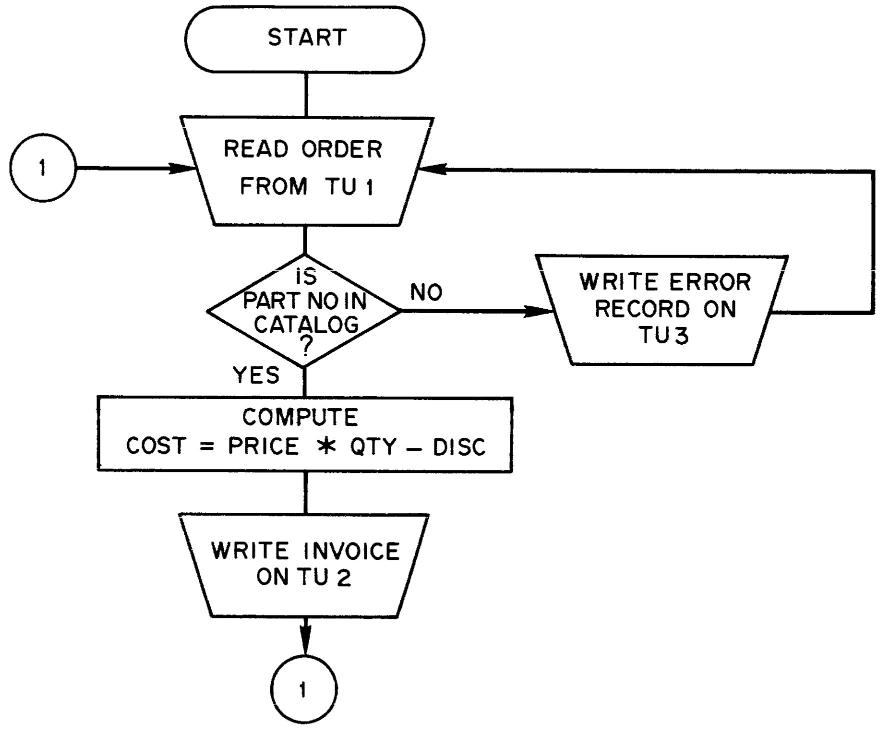
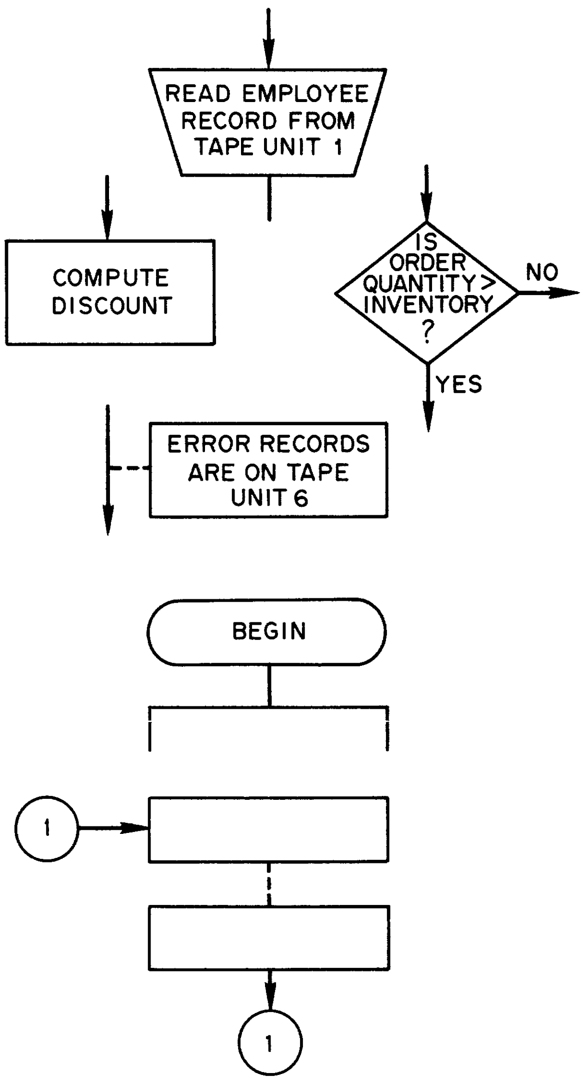


Fig. 1-4 Flow Chart for Pricing by Computer (TU1 is Tape Unit 1, etc.)

The trapezoid is used to indicate input-output operations--card reading and punching, printing, reading and writing magnetic tapes, for instance.

A rectangle is used to indicate calculations and a diamond-shaped box shows the decision points in the flow-chart. The results of a test, or decision, are indicated on the exits of the test.

The direction of flow is from top to bottom and from left to right unless otherwise indicated by arrows. If comments are in order about a particular section of the flow-chart, they can be placed in a box attached by a dashed line to one of the flow lines. The beginning and end of the flow chart can be indicated by appropriately labeled ovals; distant points which must be connected can be indicated by placing identifying numbers in circles as shown. As a practical point, we recommend that flow charts be drawn by putting down the descriptive information first and then enclosing it in the appropriate figure.


READ EMPLOYEE RECORD FROM TAPE UNIT 1
COMPUTE DISCOUNT
IS ORDER QUANTITY > INVENTORY ?
NO
YES
ERROR RECORDS ARE ON TAPE UNIT 6
BEGIN
1
1

1-2 Computer Components

In this section, we will describe the major components of a computer and their functions. Let us begin with the most common computer input medium, the punched card. Figure 1-5 shows a card punched with the digits 0-9, the letters A-Z and a sampling of what are called special characters (+, - . %, etc.). Each symbol, or alpha-numeric character, has a unique set of punches associated with it and takes one column of an 80-column card. If a number, such as an account number or employee identification number, requires six digits, it will be punched into six successive card columns (cc). A group of related columns such as this is termed a field. Note that the card in Fig.1-5 is punched in 12 rows. Ten of these rows are marked 0 through 9 and correspond to the digit positions for punching. The topmost row on the card is termed the 12-row and the next lower one, the 11-row. These rows are used in combination with the 0-9 rows for letters and special characters. In addition, a single 11-punch in the units column of a field indicates that the number in that field is negative. As an illustration, if the number -12345 is to be punched into cc 1-5, column five, the units position of the field, will contain an 11-punch as well as a 5-punch. The punches are created by a typewriter-like device called a card punch or keypunch. A keypunch has a keyboard similar to a typewriter's and card-feeding and punching mechanisms. The data punched in a deck of cards can be verified for correctness in several ways. First, if the volume of cards is small, they may be sight checked by reading the printing on top of the card. For large volumes of data,

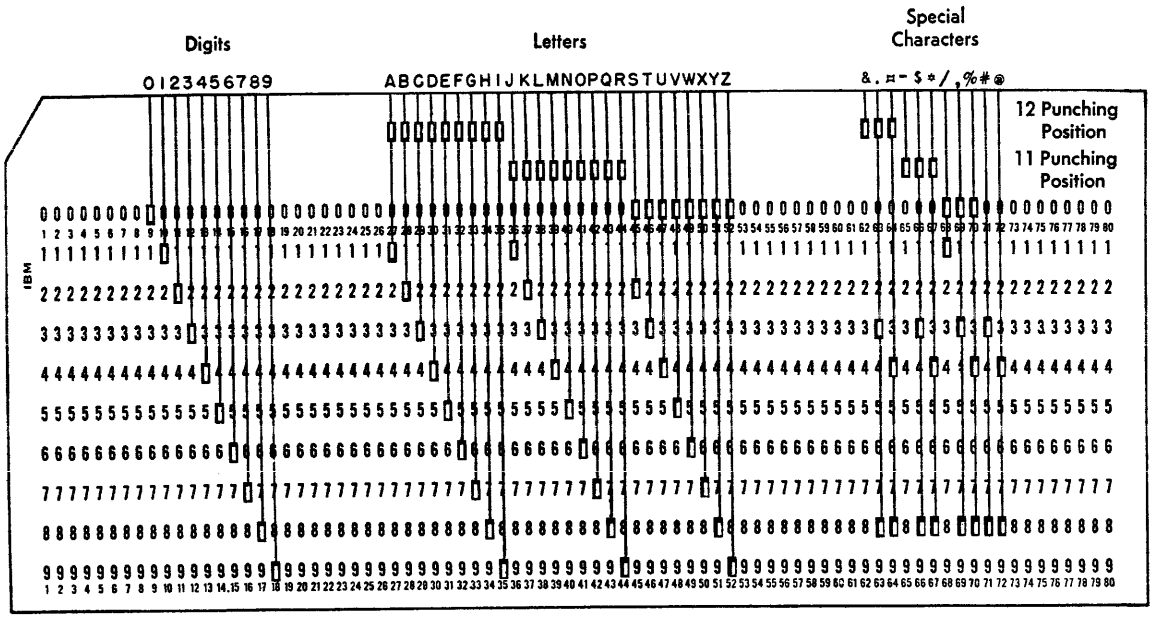


Fig. 1-5 Punched Card

a card verifier is used. This is identical to a keypunch except that instead of punching holes in a card, it senses the holes that have been punched previously. The verifier operator uses the same source documents used by the keypunch operator and if there is a mismatch between what the verifier operator keys and what has been punched, the verifier will signal the operator, usually by locking the keyboard and turning on a red light. In this way, punching error can be detected and corrected.

At this point, the card deck is ready for input to the computer. The data on the cards are read into the computer through its card-reader at speeds of up to 1,000 cards per minute. The holes in the card are detected either by a wire brush located over each column making electrical contact through a hole in the card to a metal plate below, or photoelectrically with an "electric-eye" positioned over each column. The control circuitry in the card reader translates the punches into an internal code and places the results in the working storage of the computer. When the results of a computation are ready to be printed, the control circuitry of the printer can be activated to fetch them from storage. The internal code is decoded by the printer control unit into the appropriate alpha-numeric symbols. Printing speeds range to over 1,200 lines per minute with line widths of up to 132 characters. The printer control allows single and double spacing and skipping to the top of the next page to be printed. The paper itself is available in essentially continuous lengths with perforations at the top and bottom of each sheet to allow it to be separated from the others. Much of the data which is processed by a computer need not be read by human eyes. Consider an insurance policy file which is processed monthly. A printed record needs to be created only for that fraction of the policies which require some exceptional treatment, expired policies, delinquent policies and so forth. As a result, the file can be maintained on magnetic tape. This allows reading or writing speeds of up to several hundred thousand characters per second. Magnetic tape and other magnetic recording media will be discussed in Chap. 9. In the interim, we will rely on the conceptual picture of a continuous medium which is processed sequentially in much the same fashion as the tape in an ordinary tape recorder except that recording is not continuous. Between individual records, which may be any convenient length as opposed to the fixed record size of punched cards, there is a blank section of tape, usually less than 1/2 inch in length. When tape is read or written, an entire record is either read into the computer's internal storage or written onto tape.

The computer's working storage is made up of doughnut-shaped magnetic cores which may be as small as .01" in diameter. Typical storage sizes range from about 100,000 cores to over 10,000,000. These cores may be magnetized in either a clockwise or a counter-clockwise direction. If we arbitrarily assign the value "1" to the clockwise direction and the value "0" to the other, we then have the basis for developing an internal code. We may regard the cores with value 1 as being in an "on" condition and those with value 0 in an "off" condition. The computer is equipped with circuitry which allows cores to be read, that is to determine what values they represent. The cores may also be changed, or written, by the computer. Reading and writing magnetic core storage can be done in times of the order of one-millionth of a second which is also termed a microsecond (.000001 sec).

Now a single core does not give us very much representational capability since the largest digit it can handle is 1. However, if we combine a number of cores and let the group represent a single number, or character, we have for practical purposes an essentially unlimited code capacity. As an illustration, suppose that the entire working storage, or core storage, is allocated in units of four cores each. Also, let the topmost core in the stack of four have the value 8, the next one 4, the next 2 and the bottom one the value 1. Referring to each group of four as a core position, the digit value of a core position is determined by adding the values of each core which is "on". As an example, the digits 0 through 4 are shown below

Core value					
8	0	0	0	0	0
4	0	0	0	0	1
2	0	0	1	1	0
1	0	1	0	1	0
Digit value =	0	1	2	3	4

Exercise 1-1   Extend the table above through digit value 9.

If we wanted to, each core position could represent numbers as high as 15. However, in this code scheme, which is known as Binary Coded Decimal (BCD), each core position is used for a single digit from 0 to 9. In addition to the core position we have discussed above, it is convenient to group core positions into larger units called words. For the moment, we will assume that 10 core positions make up a word. A word will be used to store a single number as opposed to the purpose of a core position which can hold only a single digit. Recall our analogy between the clerk's worksheet and the computer's core storage. Very often, the clerk's worksheet is divided into lines and columns which are numbered for convenience. The clerk's instructions can then be stated in terms of manipulating various sections of his worksheet such as: "Add line 9, column a, to line 2, column a, and write the sum in line 3, column b." The computer's instructions also are given in terms of operations on the contents of its storage. To allow each location in storage to be referenced, a unique numerical address is assigned to every word. Our analogy between the clerk and the computer cannot be drawn any further. While we may depend on the clerk to remember some of his instructions, and so avoid having to write down everything for him, this is not the case with the computer. Each and every step of the calculation must be provided for by computer instructions which are also placed in core storage along with the data of the problem. We may assume that each instruction will occupy a single word. The computer's instruction processing unit, or control unit, will fetch each instruction from storage and execute it. An instruction has a fixed format. The type of operation to be performed may be indicated by the first two digits of the word and the next eight digits may represent storage addresses, four digits per address. (Incidentally, this implies an addressing limitation of 9999 and therefore a maximum storage size of 9999 words.) If we wished to add the contents of word 1019 to the contents of word 1020, assuming an operation code of 21 for addition, the correct instruction is 2110201019. The instruction processing unit will fetch this instruction and decipher the first two digits to determine that an addition is to be performed. It will next fetch the contents of word 1019 and add 18 to the contents of word 1020 and store the 10-digit result in word 1020.

Note that we are not asking the computer to compute 1019 + 1020! After this instruction has been executed, the instruction in the next higher address word is fetched and executed and so on. If the add instruction were located in 512, then the next instruction will be taken from location 513. It is interesting to note that the time required to execute an instruction can range from a few hundred microseconds on slower computers to less than a microsecond on the fastest ones.

Since most people are inclined to impute an oracular-like prowess to computer generated output, the premises on which this rests should be examined. As we have pointed out, the computer supplies no insights of its own. If by some human error, an employee's hourly pay rate is coded as $25, 000. 00 rather than $2. 50, then the computer, or more accurately, the payroll program will create a check for $1, 000, 000 for 40 hours' work. The only way to avoid the consequences of such input errors are better validation of source documents, over-all accounting controls and self-consistency checks on each calculation. As an example, we may know that no hourly employee can ever be paid more than some maximum amount based on the present wage scale and the number of hours in a week. Before each check is written, the check amount can be compared with this upper limit. If a given check exceeds this maximum, the computer will not print the check but will instead print an appropriate error message. There is an apt expression in common use in data-processing: "GIGO, Garbage In, Garbage Out", which summarizes the consequences of bad data. Let us now consider the effect of random computer errors. These can be caused by physical and chemical changes in its electronic components or by mechanical failure in their interconnections. When the effect is permanent, it is easily diagnosed and fixed. However, there is the possibility of temporary or intermittent changes which may cause some component to malfunction for only a short interval. There are two issues here. First, we must detect the error. The validity checks and accounting controls we discussed previously can be used for this purpose. As an illustration, we may be running an inventory update program. For each item we calculate the amount in inventory according to

$$\text{New Amount} = \text{Old Amount} - \text{Issues} + \text{Receipts}$$

As a check, we can sum each of the four quantities above for every inventory item. At the end of the program, the sum of all New Amounts should equal the total of Old Amounts minus the total of Issues plus the total of Receipts. If it does not, then either an error was made in calculating one of the New Amounts or an error was made in calculating one of the control totals. In either event, the job will have to be rerun. The second issue is devising a means for the computer to detect failures. It turns out that this is fairly easy - we will describe it subsequently. At this point, you may question the need for accounting controls. If the computer can detect its errors, why bother to check its results? The reason that we must is simply that if a component of the computer can fail, we may also have a simultaneous failure in the error detection circuitry. While this is unlikely, it is still wise to take precautions. As an illustration of how unlikely an undetected error is, assume that a given component will fail once per million operations. (Actual component reliabilities are much higher. The figure used here is strictly for illustrative purposes.) Assuming that the corresponding component in the error detection circuit has the same failure rate, the chance that they will both fail at the same time and give an undetected error is 1/1,000,000,000,000. Assuming this component is used 100,000 times a second, 24 hours a day, this means that one undetected failure will occur once every four months, on the average. If the programmer has built adequate control checks into his program, the undetected failure will be caught. With this as background, we will describe the technique used for error detection. The essence of this technique is redundancy -- that is, providing extra coding which is related to the data in core storage. A parity bit is added to each group of bits comprising the basic storage element. The parity bit is then either set on or off to make the total number of on-bits in the storage element an odd number or an even numbet according to the choice of the system designer. For System/360, an odd-parity code is used. Since the basic data element for the 360 is eight bits, a ninth bit is added as a parity bit. In all data manipulations, parity checking circuitry verifies that if the parity bit is on, an even number of data bits are on; if the parity bit is off, there should be an odd number of data bits turned on. Any exception to this, either because a data bit has been dropped or because the parity bit has been dropped, will cause an automatic interruption to occur. The same coding technique is used for data stored on magnetic disks and tapes.

At this point, the reader should not concern himself with the details of the storage organization of the 360 -- it will be covered in Chapters 2 and 3. Here, only the basic concepts of core storage data representation and parity checking are important.

1.3 Programming

The flexibility and power of the electronic computer is derived from essentially one aspect of its architecture -- its stored program. In essence, the instructions for a calculation are stored in the computer as coded digits in the same way as the numerical data of the problem itself. This gives the computer the capability of operating on its instructions as if they were data. Before the development of the stored program computer, the instructions for a computation were set up on wired plug boards. Their ability to make simple tests at intermediate points of the calculation, and to take alternate courses depending on the outcome, was derived from testing the status of switches, or relays. This type of logic is inherently limited by the number of relays in the computer and by the availability of sockets on the plug boards.

The stored-program computer is not faced with these difficulties. Since its instructions are numbers, the computer can be directed to modify its own program and take an almost endlessly varied series of actions.

Before we describe how programs are written, it will be helpful to understand how the computer manages its storage. We will work with the storage model introduced in Sec. 1. 2. While the actual storage implementation on the 360 differs from our conceptual model, the differences are not important at this point. We will assume that the magnetic core storage of our hypothetical computer comprises 10,000 words, each with the capacity to store 10 digits. So that the computer can identify individual storage locations, each word will have an address, a four-digit number from 0 to 9999. Computer instructions reference a location via its address so that the contents of that location may be operated on. As an example, in a payroll program, the storage location which will contain gross pay may be in the word whose address is 4056 and its contents may be 0000012398. It is important to distinguish between a given storage location, the address used to reference that location and the contents of the location. As an analogy, we may have in mind a particular location on Manhattan Island which contains the tallest building in the world. We may use a number of addresses to identify this location such as 34th Street between Fifth and Sixth Avenues, or Longitude W 73° 58'53", Latitude N40° 43'30". The contents of this location is of course, the Empire State Building. Note that its address is not unique -- there is more than one way to to identify its location. We have a similar situation with 360 storage addresses which are somewhat more complex than the hypothetical computer we are describing. System/360 storage addresses are made up of a base and a displacement whose values are added together to produce an address. As examples, a given location, say word 10000 in storage, may have an address whose base is 7500 and displacement is 2500, or the base may be 9500 and the displacement 500. We may then use several addresses to identify a given location.

Each storage word of our hypothetical computer may contain 10 decimal digits or it may contain an instruction. We will assume that instructions have a two digit Operation Code followed by two four-digit addresses, which we will refer to as operand addresses. The first operand will be called Operand 1, or Op 1, and the second, Op 2. We will assume that the results of an operation will be placed in the location of Op 1. As an example, if the operation code for addition is 21, and we wish to add the contents of locations 1020 and 1021 and place the result in 1020, the proper instruction is

```
21   1020   1021
```

If 1020 contained 0000000045 prior to the execution of the instruction above, and 1021 contained 0000000015, the contents of 1021 would be unchanged after the operation was performed but the contents of 1020 would become 0000000060. So that we can describe some simple examples, we will assume that our hypothetical computer has the instruction set given in Fig. 1-6; we will discuss each of these as they are used. We have omitted input-output instructions as these are complicated even on hypothetical computers. As an illustration of programming with this instruction set, suppose that we wish to calculate an employee's net pay from the formula

NET PAY = HRS * RATE - DEDUCTION

where the symbol * denotes multiplication. In addition, it is known from this particular payroll that no employee earns more than $300.00, a test against this upper limit can be made. In this way, we avoid catastrophic errors caused by human errors in the input data. The complete logic of a typical payroll program would involve reading an employee work record, say a time card, calculating his pay, taxes, deductions and so forth, creating a check, and then repeating the process for the next employee. In addition, a variety

Instruction	Op Code	M nemonic	Explanation
Add	21	A	OP1 OP1+OP2
Subtract	22	S	OP1 OP1-OP2
Multiply	23	MPY	OP1 OP1*OP2
Compare	31	C	Compare OP1 with OP2
Move	32	MV	OP1 OP2
Branch High	41	BH	If OP1 OP2, branch
Branch Equal	42	BE	If OP1= OP2, branch
Branch Low	43	BL	If OP1 OP2, branch
Branch	44	B	Branch
Halt	51	H	Halt

Fig. 1-6 Instruction Set for Hypothetical Computer

of payroll records would be updated such as the employee year-to-date journal. We will ignore these complications and assume that each time card contains the employee's rate and the single deduction which is to be made. We will not discuss input-output except to indicate where they will take place. The first step is to lay out a storage map, that is, a table showing the storage locations of the various data items: NET PAY, HRS, RATE, DEDUCTION. In addition to these, there are several other items which we would have to reserve storage for. They include such information as employee name, man number, department number and so on, which will be needed when the check is to be printed. For the sake of simplicity, we will ignore these and concentrate on the four items above. We will reserve one word for each starting with NET PAY on location 1000, HRS in 1001, RATE in 1002, DEDUCTION in 1003, and the constant value, 300, in location 1004. The flow chart for the program is shown in Fig. 1-7.

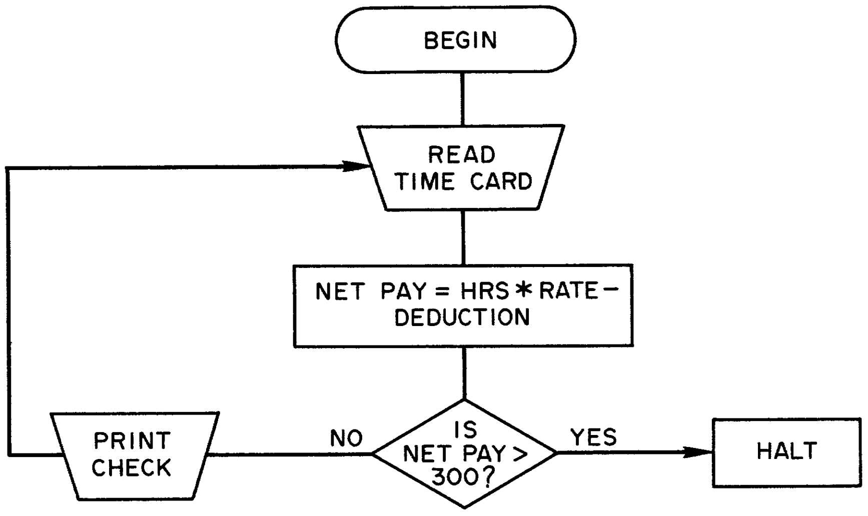


Fig. 1-7 Payroll Program

If the comparison of NET PAY with the upper limit of $300. 00 fails, the program will halt. In practice, this would not be done since the ideal is non-stop operation of the computer. Invalid records would be written on a separate tape and then a listing of this tape would be created for subsequent correction after the program has processed all other payroll records. The program is given in Fig. 1-8. To the right of each statement, explanatory comments are given. We are assuming that the first arithmetic instruction begins in location 1500 and that reading and writing data each take 20 instructions.

Location	Instruction/Data	Comments
1480	-- ---- ----	Instructions for reading time
----	-- ---- ----	Cards begin at location 1480
1500	23 2001 2002	Multiply HRS*RATE
1501	32 2002 2001	Move HRS*RATE to NET PAY
1502	22 2002 2003	Subtract DED. from HRS*RATE
1503	31 2000 2004	Compare NET with 300
1504	41 1526 0000	Branch to 1526 if NET > 300
1505	-- ---- ----	Instructions for check writing
----	-- ---- ----	Begin at 1505 and go to 1524
1525	44 1480 0000	Branch to 1480 for next employee
1526	51 0000 0000	Input error has occured

2000	XXXXXXXXXX	NET PAY is stored here
2001	XXXXXXXXXX	HRS worked is stored here
2002	XXXXXXXXXX	RATE of pay is stored here
2003	XXXXXXXXXX	DEDUCTION is stored here
2004	00 0003 0000	Upper limit is $300. 00

Fig. 1-8 Payroll Program in Machine Language

At this stage, the starting location, or origin, of the program is arbitrary, We chose to begin this one at location 1480. The first arithmetic operation occurs after the instructions (20) for card reading. It causes HRS in location 2001 to be multiplied by RATE in 2002. The product will be placed in 2001. The next instruction, MOVE, moves the product from 2001 to 2000, the location for what will eventually be NET PAY. At this point, DEDUCTION in 2003 will be subtracted from the product giving NET PAY. We could just as well have omitted the MOVE and done the subtraction directly in 2001 provided we kept in mind that location 2001, which used to contain HRS now contains NET PAY. The next instruction at location 1503 will compare the contents of 2000 (NET PAY) with the contents of 2004 ($300. 00). This comparison will set what we can think of as a three position switch in the computer, one setting if the first operand (NET) is high, another if it is low, and a third if both are equal. These settings can be tested subsequently and will remain unchanged by testing until another arithmetic instruction is executed. The next instruction at 1504 is interesting in several respects. First, up to this point, the program has been executing instructions sequentially, that is, the next instruction to be executed is taken from the storage location immediately following the previous instruction. Here, we have an opportunity to continue with this sequence or to take alternate action if NET is greater than $300. 00. This instruction will test the three-position compare switch and if it finds it set too high, it will branch to the instruction whose location is given by the address of OP1, 1526. That is, the next instruction to be executed, if NET is greater than $300. 00, will be at location 1526. If this conditional branch is not taken, program execution will continue in sequence through the 20 instructions for check writing after which an unconditional branch will be met at 1525. This is an unconditional branch in the sense that it will always be taken. Its effect is to cause the entire sequence of operations to be repeated for the next employee time-card. After this program had been written and checked for errors, it would be punched into cards, say five instructions per card with the starting address of each group of five also on the card. Another program, called a loader, would be used to load the program into the proper locations in storage. Lest the reader be concerned that we need another program to load the loader into storage, and so on ad infinitum, most computers have in effect a "load button" which causes a card to be read from the card reader into a specific storage area and the first instruction in that area to be then executed. The card in question is the first card of the loader deck and the first instruction on that card is one which starts the loading process. The loader thereby brings itself into storage by its own bootstraps.
In addition to loading the instructions, any constant data such as the 30000 in location 2004 would also have to be put into the load deck with appropriate address information so they can be properly stored. The loader does not have to take any action with respect to locations 2000 through 2003 which have been reserved for data, except that it should not load anything into those locations. The input routine would be arranged to store the relevant information in those four locations after having read a data card. Once the program has been loaded, the data deck would be placed in the card reader, the proper paper

would be inserted in the printer and the computer is then ready to produce results. With even this brief perspective on computer programming, there are several issues which may occur to the reader. First, there is the "atomic" level at which computer instructions operate. A single instruction will handle only a small segment of the problem and every step of the calculation must be provided for explicitly -- there is no room for implications here. Second, as an outgrowth of our first observation it is clear that a respectable number of instructions will usually be required for even fairly simple problems. As a result, the coding exercise we have just gone through could become very tedious and error prone for a longer program. The heart of this problem is that the programmer is burdened with much extraneous detail. He has to remember machine operation codes as well as the addresses of data and significant instruction locations in the program so that branches can be programmed. In our short example, there is nothing particularly provocative about location 2002 so that we would associate it with RATE; the bare fact of location 1526 does not tell us that this is the point where a branch is to be taken when a check is written for larger than the maximum amount. What is needed is a shorthand way of writing programs which is easier to remember than the series of digits we have been working with. Suppose we could write the program as

```
            MPY     HRS, RATE
            MV      NET, HRS
            S       NET, DEDUCT
            C       NET, C300
            BM      ERROR
            - -
            B       START
ERROR       M
```

Here we substitute a short <u>mnemonic</u> for the operation code and the <u>symbols</u> for the data addresses. Also, we <u>label</u> those portions of the program to which reference will have to be made, such as ERROR or START which indicate the error handling part and the beginning of the program. We can also extend this shorthand to the part of storage which is reserved for data and constants. As an illustration

```
NET         DS      1
HRS         DS      1
RATE        DS      1
DEDUCT      DS      1
C300        DC      '30000'
```

The abbreviation DS means that we are defining a symbol at that point. That is we are defining the address of that symbol and indicating that one storage word should be reserved for it. The DC statement indicates that we are defining a constant with the value 30000 ($300.00) which will be referred to by the name C300. This is a much easier way to write a program. Whenever we want to write an instruction involving net pay, we may write NET and not be concerned about the fact that NET is found at location 2000. When we have finished the program we can then concentrate on translating our shorthand notation into machine language. We can develop a systematic method for doing this. First we define the program origin, or load point, which is the initial location in storage which the program will occupy. For compatibility, with the program in Fig. 1-8 we select location 1480 as the load point. The first step in converting to machine language is to create a symbol table or dictionary showing each symbol and its address. This can be done by scanning the label column and noting, for instance, that ERROR is the label for the 46th word in the program. As a result, it has address 1480 + 46 = 1526. The other labels can be handled in the same way. After we have built up the symbol table as shown in Fig. 1-9 we can translate the operand portions of each instruction by simply looking up each symbol in the dictionary and replacing it by its address value from the dictionary or symbol table. The operation codes can also be translated using the table of mnemonics in Fig. 1-6. At the conclusion of this translation, we will have assembled the shorthand version of our program into machine language.

Symbol Table

Symbol	Value (Address)
START	1480
ERROR	1526
NET	2000
HRS	2001
RATE	2002
DEDUCT	2003
C300	2004

Fig. 1-9 Symbol Table for Payroll Program

Since the assembly operation is so straight forward, we can relegate it to the computer. A program which performs this function is called an Assembly Program or Assembler, for short. The abbreviated notation we have been using is termed, appropriately enough, Assembly Language and it has some very definite, although simple, grammatical rules and a well defined vocabulary which we will present throughout the remainder of this book. At this point, we will introduce the coding form in Fig. 2-6 on which 360 Assembly Language programs are to be written. One statement is written per line with each position on the form punched into a card column. The label or name begins in column 1 and may be from one to eight characters long. The operation code begins in column 10 and the operands begin in columns 16 through 71 with commas separating the operands. If explanatory remarks are required, they may be added after the operands provided they are separated by at least one blank from the operands. If more than one line is required for a statement, a continuation character, which may be any non-blank character, is written in column 72. The continued part of the statement may be resumed starting in column 16 of the next line. If more space is required for comments, an entire line can be used by placing an * in column one. The entire line will then be ignored by the assembler.

A number of useful programming features can be added to an Assembler. One of these is address arithmetic. Operands can be written as arithmetic expressions such as ERROR+2. This has the effect of increasing the value of the symbol by two. If this operand appeared in an instruction such as

```
B      ERROR+2
```

it would be assembled as a branch to ERROR+2, or 1526 + 2 = 1528. Very often, this type of operand is more convenient than defining another label two words from ERROR. The assembler's location counter can also be used by the programmer. The location counter is initialized to the program's origin and then after each instruction is assembled, it is advanced by 1 so that it always contains the location of the instruction being assembled. The location counter is referenced by the symbol *. To branch to a point three locations away from the present instruction, the programmer would write

```
B      *+3
```

If this instruction was at location 2000, the branch would be to location 2003.

After the program has been written, it is punched into cards. This deck of cards is known as the assembly language source deck or source program. The next step is to load the assembler into the computer. The assembler reads the source deck as input and performs a translation from assembly language to machine language which is punched into cards. The machine language deck is known as the object deck. After it has been punched, the programmer is then able to execute his program. Notice that the assembler does not do this, it only translates the program. The program is executed by loading the object deck into the computer. The input to the computer at this stage is the problem data. It will be helpful to the reader to keep the details of this process in mind. Figure 1-10 summarizes the operation.
Each of these steps can be handled by an operator who loads the required programs into the computer in the proper sequence. However, in the interest of using the computer more efficiently, most of the operators functions can be handled by a program which is appropriately termed an Operating System. It resides continually in core storage. Each job to be run is preceded by a control card which indicates what actions the Operating System should take for that particular job. We will comment on several functions of the operating system in the following chapter and will discuss the subject in greater detail in Chaps. 11 and 12.

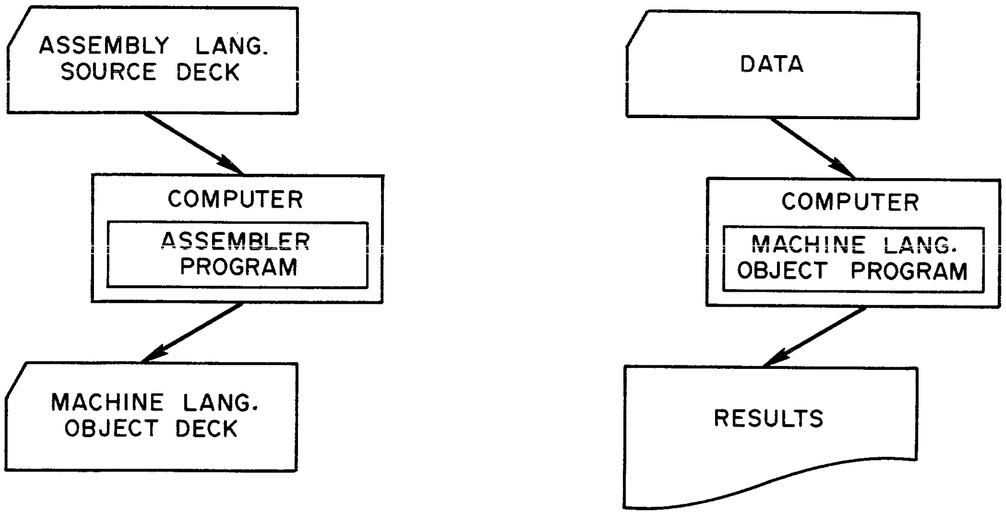

ASSEMBLY LANG. SOURCE DECK
COMPUTER
ASSEMBLER PROGRAM
MACHINE LANG. OBJECT DECK
DATA
COMPUTER
MACHINE LANG. OBJECT PROGRAM
RESULTS

Fig. 1-10 Executing an Assembly Program

Problems

1-1 As an exercise in flow charting, prepare a flow chart for some everyday activity such as starting a car and driving it out of its garage. Consider what should be done if some expected action does not occur such as a failure of the engine to start. Also, be sure to include so-called "obvious" details such as verifying that the garage door is open before backing out.

1-2 Consider a hypothetical "black-box" which faithfully transcribes spoken English language syllables into computer decipherable codes. Suppose that the code for the syllables in "spoken" was *$, for instance. A program could be written which would look up *$ in a table and retrieve the word, "spoken." In this way, we have at least the rudiments of an electronic stenographer. Neglecting problems of individual pronounciation differences and assuming that the "black box" will faithfully transcribe into code whatever it hears, what additional difficulties must be overcome for it to be completely effective?

Chapter 2

DECIMAL PROGRAMMING

2.1 Data Representation and Addressing

Let us expand on the discussion of core storage given in Sec. 1.2. We will start with the requirement that each unit of core storage will be able to store letters, special characters such as $, =, +, as well as digits. Let us now count the number of different code combinations required: For upper and lower case letters, we require 52 and the digits 0 through 9 bring this to 62; there are a number of special characters such as #, %, $ extending through 23 other relatively high frequency symbols for a total of 26 special characters, or 88 total characters, so far; in addition, certain functions of input-output units can be activated and controlled by simple codes which can be considered as additional "characters"; this adds 24 more to the total bringing us to 112 characters in all.

In Chap. 1, we tentatively introduced the notion that each word, or position, in core storage would be made up of 10 four-bit magnetic cores, or bits, with each four-bit unit containing a single digit. With four bits, we have 16 possible different codes ranging from 0000 to 1111. This can be verified directly by counting them or by noting that for each bit position we have two possible codes, 1 or 0. With two bits, the number of different codes is 2 x 2 = 4, namely 00, 10, 01, 11. That is, for each possibility in the first bit, there are two different possible conditions for the second bit. Since there are two possibilities for the first bit, the total is four. By extension to four bits, the number of possible codes is 2 x 2 x 2 x 2 = 16. As a result, it is clear that we cannot handle even alphabetic characters within a single group of four bits. If we expand to seven bits, the total number of codes is 2 x 2 x 2 x 2 x 2 x 2 x 2, or 128. This will handle the requirements we have developed so far with some room for expansion of the character set. However, by going to eight bits per unit of storage we gain two important advantages. First, we have the possibility of expressing any one of up to 256 (2 x 128) different characters in a single storage unit. This allows significant growth for additional character sets and new input-output devices. In addition, we can store two decimal digits of four-bits each in a single unit. For these reasons, eight bits per storage unit was adopted as the standard for the 360. Since we will have occasion to refer to this basic storage unit of eight bits frequently, we will define it as a byte.

The next point to be considered is defining which bit pattern, of the total of 256 different ones available, will be used to represent each character. Apart from the conventions used for numbers, the precise bit definitions of the other characters is not particularly important to the programmer. When information is being read from, or written to, disks, drums or magnetic tape, the data is treated as a string of bits and merely stored or retrieved in units of eight consecutive bits. The bit definitions are important only when information is being transmitted to or from character sensitive devices such as card reader/punches, printers and graphical displays. For these devices, the translation between the external and internal representation is handled by the device control unit. As an example, if a punched card containing an N in column 12 (11-5 punch) is read into an area in storage, the 12th byte of that area will contain 11010101 which is the preferred code for N. The translation from the 11-5 punch to 11010101 is done by the card reader control unit. Similarly, if that area of storage is then printed, the 12th character in the printed line will be an N. The printer control unit will decipher the bit code, determine that it represents an N and then activate the print mechanism to print an N in the 12th position.

While the central processing unit of a 360 can use any eight-bit character code, there are certain restrictions in decimal arithmetic and editing operations which we will discuss in Secs. 2.4 and 2.7. Character sensitive devices require either the extended binary-coded-decimal interchange code (EBCDIC) or the American Standard Code for Information Interchange for use in an eight-bit environment (ASCII-8). The choice between the two can be set by a mode switch as discussed in Chap. 9. The EBCDI code is the more common one for 360 operation and so our discussion of character codes will concentrate on it. Figures 2-1 and 2-2 show the conventions for each of these codes and Fig. 2-3 shows the translation between Hollerith coding in a punched card and EBCDIC. Note that the numbering conventions for the bit positions within a character are different for EBCDIC and ASCII-8. The conventions are

EBCDIC	01234567
ASCII-8	76X54321

If we visualize the cores within a single byte as being laid out left to right, the leftmost bit in EBCDIC convention is referred to as bit 0 and the rightmost one as bit 7. In ASCII-8, the convention is reversed.

The coding for decimal digits is of primary interest here. In EBCDIC, the digits 0 through 9 are represented by the codes 11110000 through 11111001. The only difference between these and the convention we established in Chap. 1 is that bit positions 0 through 3, called the zone bits or high-order bits, are all 1. The remaining four bits, the digit or low-order bits, follow the convention of Chap. 1. When an arithmetic sign is associated with a number, the sign is usually punched over the units position of the number. A + sign is represented by a 12-punch and a - sign by an

Bit Positions 01 →	00				01				10				11			
23 → / 4567	00	01	10	11	00	01	10	11	00	01	10	11	00	01	10	11
0000	NULL				ƀ blank	&	-						>	<	‡	0
0001							/		a	j			A	J		1
0010									b	k	s		B	K	S	2
0011									c	l	t		C	L	T	3
0100	PF	RES	BYP	PN					d	m	u		D	M	U	4
0101	HT	NL	LF	RS					e	n	v		E	N	V	5
0110	LC	BS	EOB	UC					f	o	w		F	O	W	6
0111	DEL	IDL	PRE	EOT					g	p	x		G	P	X	7
1000									h	q	y		H	Q	Y	8
1001					•		,	"	i	r	z		I	R	Z	9
1010					?	!		:								
1011					.	\$	,	#								
1100					←	*	%	@								
1101					(	)	⌒	'								
1110					+	;	_	=								
1111					‡	¢	±	✓								

Fig. 2-1 Extended Binary-Coded-Decimal Interchange Code (EBCDIC)

Bit Positions 76 →	00				01				10				11			
X5 → / 4321	00	01	10	11	00	01	10	11	00	01	10	11	00	01	10	11
0000	NULL	DC$_0$			ƀ blank	0					@	P				P
0001	SOM	DC$_1$			!	1					A	Q			a	q
0010	EOA	DC$_2$			"	2					B	R			b	r
0011	EOM	DC$_3$			#	3					C	S			c	s
0100	EOT	DC$_4$ STOP			\$	4					D	T			d	t
0101	WRU	ERR			%	5					E	U			e	u
0110	RU	SYNC			&	6					F	V			f	v
0111	BELL	LEM			'	7					G	W			g	w
1000	BKSP	S$_0$			(	8					H	X			h	x
1001	HT	S$_1$			)	9					I	Y			i	y
1010	LF	S$_2$			*	:					J	Z			j	z
1011	VT	S$_3$			+	;					K	[			k	
1100	FF	S$_4$			,	<					L	\			l	
1101	CR	S$_5$			-	=					M	]			m	
1110	SO	S$_6$			.	>					N	↑			n	ESC
1111	SI	S$_7$			/	?					O	←			o	DEL

Fig. 2-2 American Standard Code for Information Exchange For Use in Eight-Bit Environment (ASCII-8)

Card Code	Printer Graphics	Internal Representation	Card Code	Printer Graphics	Internal Representation
	blank	01000000	11,1	J	11010001
12,8,3	. (period)	01001011	11,2	K	11010010
12,8,4	<	01001100	11,3	L	11010011
12,8,5	(	01001101	11,4	M	11010100
12,8,6	+	01001110	11,5	N	11010101
12	&	01010000	11,6	O	11010110
11,8,3	$	01011011	11,7	P	11010111
11,8,4	*	01011100	11,8	Q	11011000
11,8,5	)	01011101	11,9	R	11011001
11	-	01100000	0,2	S	11100010
0,1	/	01100001	0,3	T	11100011
0,8,3	,	01101011	0,4	U	11100100
0,8,4	%	01101100	0,5	V	11100101
8,3	#	01111011	0,6	W	11100110
8,4	@	01111100	0,7	X	11100111
8,5	' (quote)	01111101	0,8	Y	11101000
8,6	=	01111110	0,9	Z	11101001
12,1	A	11000001	0	0	11110000
12,2	B	11000010	1	1	11110001
12,3	C	11000011	2	2	11110010
12,4	D	11000100	3	3	11110011
12,5	E	11000101	4	4	11110100
12,6	F	11000110	5	5	11110101
12,7	G	11000111	6	6	11110110
12,8	H	11001000	7	7	11110111
12,9	I	11001001	8	8	11111000
			9	9	11111001

Fig. 2-3 Hollerith Card Codes and EBCDIC

11-punch. (As a practical matter, the + sign is almost never used in this manner. The universal custom is to consider unsigned fields positive.) The zone codes for + and - are, respectively, 1100 and 1101, with the zone code for unsigned numbers, 1111, being acceptable as a positive sign. When decimal digits are involved in arithmetic operations the signs generated in the EBCDIC mode are 1100 and 1101 for + and - respectively, whereas in the ASCII-8 mode, they are 1010 and 1011 respectively. In addition, the zone code 1110 will be treated as positive. Any other zone codes are invalid. Fig. 2-4 summarizes the digit and sign codes. These details are points the reader should be aware of, not necessarily intimately acquainted with.

Exercise 1-1 The number -1 would be punched as an 11-1 which is also the code for a J. Referring to Fig. 2-1, compare the zone code for J with the zone code for -.

The format in which each digit appears with its zone code and requires one byte per digit is called the zoned decimal format. When the zones are stripped away and the decimal digits are stored two to a byte, we have the packed decimal format. The packed decimal format is important because it saves storage and because the decimal arithmetic instructions require it. Fig. 2-5 gives a schematic of these formats. As examples, 123 would appear in storage in the zoned format as

11110001 11110010 11110011

-123 would be punched in a card as $12\overline{3}$ and would appear in storage as

	Digit Codes	Sign (Zone)		Codes
0	0000	1100	+	EBCDIC
1	0001	1101	-	EBCDIC
2	0010	1010	+	ASCII-8
3	0011	1011	-	ASCII-8
4	0100	1111	*	EBCDIC
5	0101	1110	+	
6	0110			
7	0111			
8	1000			
9	1001			

* implied + sign
for unsigned fields

Fig. 2-4 Digit and Sign Codes

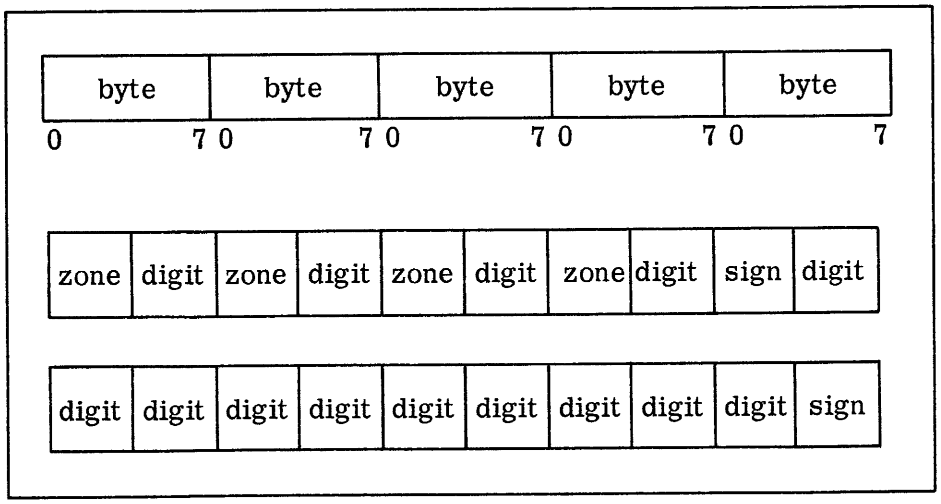

byte
byte
byte
byte
byte
0
7 0
7 0
7 0
7 0
7
zone digit zone digit zone digit zone digit sign digit
digit digit digit digit digit digit digit digit digit sign

Fig. 2-5 Storage Organization: Bytes; Zoned Digits;
Packed Digits

11110001	11110010	11010011

In the packed format, we have

123	0001	0010	0011	1111
-123	0001	0010	0011	1101

Where the low-order four bits give the sign.

Core storage in the 360 is divided into discrete units of bytes. Each byte in storage has a unique address ranging from 0 to whatever is the maximum storage available on a particular machine. Typical storage sizes are 4096, 8192, 16384, 32768, 65536, 131072, 262144, 524288, and 1048576 bytes. Larger sizes are also available. Because the addressing circuitry of the 360 works on binary numbers for addresses rather than decimal numbers, the size of storage modules are multiples of two, rather than multiples of 10. To make it more convenient to refer to these module sizes, they are abbreviated 4k, 8k, 16k, 64k, 128k, 256k, 512k, 1 million, where 1k = 1024. In conversational usage, one speaks of a 32k-machine, or a 32k-byte machine, for instance. In Chap. 1. we tentatively introduced the concept of a core storage addressable in units of words where each word contained 10 digits with the advice that this picture is an oversimplification. In some applications, it is an advantage to deal with a machine whose storage is organized in words. These applications are categorized by data with limited precision, usually well under ten digits so that a word size which will accommodate 10 digits or so, is satisfactory. Engineering and mathematical calculations are in this category. If an engineer is designing a structural member, it may not make much difference if one of his results turns out to be, say 320916.51, as opposed to 320917.32. Chances are, he looks upon the result as "around 321000." There is a great advantage to a fixed-word size machine since its data paths and arithemetic circuits can be designed around the fixed length. As an example, a single core access on a word machine fetches an entire word. In effect, the construction of the machine is optimized around the word size with considerable advantages in performance. However, there are applications which cannot tolerate being restricted to fixed size numbers. These applications are characterized by requiring exact results. To a comptroller or a banker, the difference between 320916.51 and 320917.32 is .81 and the difference is important. One solution would be to build a computer with a word size large enough to hold the reasonably largest number which could occur. However this has the disadvantage that much smaller numbers would also require a full word, thereby wasting a good deal of storage. A better solution is to recognize the variable-length requirements of these applications and design the computer accordingly. In effect, we want to be able to address to within a single digit as closely as possible. Since we also want to address single characters, a compromise is in order--the 360 is designed to address each byte or pair of digits. This allows us to define data fields of whatever length is appropriate for each application. Within this architecture, the requirements for fixed-length data can be satisfied by designing a set of instructions to handle data organized into consecutive groups of four or eight bytes--four bytes being a word and eight bytes, a double-word. Each data field, whether it falls into the fixed-length category, or is variable-length is addressed by its leftmost byte. As an example, if we have two eight-byte fixed length fields, followed by an 11-byte field, a 13-byte field and a nine-byte field, starting at byte location 16000, following are the addresses of each field:

```
16000 --  8 bytes
16008 --  8 bytes
16016 -- 11 bytes
16027 -- 13 bytes
16040 --  9 bytes
```

To address the 11-byte field, we would use the address 16016. In addition to being the location of the high-order byte of the field, 16016 is also the address of the entire field and can be used to fetch it. We will show how this is done in the next two sections.

Let us consider how boundaries are established between variable-length data. Suppose we have in storage the following consecutive fields of packed decimal digits and characters:

```
+0012345   -0017   ABC   0367
```

With two packed decimals per byte, these fields would require 13 consecutive bytes of storages which we assume begin at byte address 600.

```
00   12   34   5+   00   01   7-   A   B   C   00   36   7+
 ▲                   ▲                          ▲         ▲
600                 604                        610       612
```

On paper, it would seem that we can easily distinguish field boundaries by looking for + or - signs or characters. However, it is not as simple as that. In core storage, we have only a string of 1 and 0 bits within each byte, and as a result, field boundaries cannot be identified. As an illustration, how can you tell the difference between -1 and J? Their codes are identical, it is only their usage by the programmer which distinguishes between them. One way out of this dilemma would be to define a special character as a field separator. This would waste one byte per field in addition to requiring circuitry to identify the boundary character. The approach which has been selected for the 360 is to specify the length of each field as well as its starting address whenever arithmetic operations are required. As an example, if we wished to add the - 17 field above to the + 367 field, we note that the former begins at location 604 and is three bytes long and the latter begins at location 610 and is also three bytes long. The add instruction will tell the computer, in essence, to add the three-byte field which begins at 610 to the three-byte field which begins at 604. This is done by including the length of each field, as well as their addresses, in the add instruction.

2.2 Storage Allocation

In Chap. 1, we mentioned that one of the functions of an assembler is to allocate storage for constants, data, and work areas. In this section, we will discuss how this is done for decimal and character data. In Sec. 3.4, storage allocation for binary data will be presented. Here, we will be concerned with decimal data in the packed and zoned formats. The packed constants are used in arithmetic operations and the zoned constants are used where numerical information is to be printed. Let us first consider how to define constants. Recall that in the zoned format, each digit occupies one byte with the rightmost byte containing the number's sign as well as the low order digit, whereas in the packed format, each byte contains two digits with the rightmost byte again containing the sign and low order digit. As a result, the length of a zoned constant is one byte for each digit and for a packed constant, half the number of digits including sign rounded up, if necessary, to the nearest whole byte. That is, in the packed format, the constant +217 requires two bytes but the constant +21 also requires two bytes. The general form of the assembler instruction to define zoned (Z) or packed (P) constants is

```
label        DC      P or Z 'constant'
```

The label is the name of the constant and is used to make reference to it throughout the program. The constant is written between single quotation marks. The letter P or Z denotes packed or zoned, respectively, and together with the constant constitute the operand portion of this assembler instruction. The largest packed constant which may be specified is 31 digits and the largest zoned constant is 16 digits.

As examples,

```
CONA      DC       P'-1'
CONB      DC       P'315'
CONC      DC       Z'216.26'
```

Decimal points may be written to clarify the usage of a constant but they are ignored by the assembler. Also, no blank spaces may appear in the operand. The constants above require one, two, and five bytes, respectively. We stress that the DC is an assembler instruction since it causes the assembler to set up the constant in the proper form and include it in the object deck of the program which is being assembled. In this sense, it can be considered to be executed at assembly time. When properly placed in the program, it does not have any effect on instruction processing at execute time. Since the computer cannot distinguish between instructions and data, it is the responsibility of the programmer to make sure that constant data is not located within the instruction stream of the program. We will illustrate this point in several examples in this chapter.

Variations on the definition above are permitted. To get repetitions of a constant, we write, for example

```
CONST     DC     5P'2'
```

which will give five bytes, in storage, each containing a +2. In addition, multiple constants may be defined in a single DC instruction such as

```
RATETBL        DC       P'1.0475', P'1.0575', P'1.0675'
```

The individual constants are separated from each other by commas. However, the label RATETBL can only be used to retrieve the first constant in the list, 10475. Since each constant in the list is three bytes in length (sign plus five digits), the addresses of the second and third constants are RATETBL +3 and RATETBL +6. The assembler will interpret these expressions correctly by adding three and six to the address assigned to RATEBL.

In the examples we have seen so far, the length of the constant is given implicitly by the size of the constant. In some cases, it is useful to specify the length explicitly. This is done as follows

```
CONST     DC     PL6'25'
```

which will give a constant of +25, six bytes in length which is padded to the left with zeros. As another example, to define 10 zoned fields each of length 12 bytes with each field being equal to zero, we write

```
CZERO     DC     10ZL12'0'
```

This shows the utility of the zero padding feature which occurs when the explicit length is greater than the implicit length. This saves us from having to repeat all 12 zeros. Constants should not be written whose implicit length is greater than their explicit length. Note that the length indicator is in bytes for both packed and zoned data.

Character constants are useful for page and column headings and for explanatory remarks in a print-out. Character constants up to 256 bytes in length may be defined in a similar manner to zoned constants. As examples,

```
TBLHD1    DC     C'NET PAY'
BLANKS    DC     CL132' '
BASELN    DC     12CL10'.'
```

The only difference occurs when the explicit length is greater than the implicit length and the constant is then padded to the right. Thus, the constant BASELN consists of a period followed by nine blanks. Again, note that when multiple repeated constants are defined, the label references only the first constant in the list. To retrieve all 12 constants a simple technique can be used which we will illustrate in several of the worked examples to follow. Two other precautions must be observed when writing character constants. Multiple constants (as opposed to a repeated constant) cannot be defined within one DC since the separating commas would be treated like ordinary character data. Also, when the single quote mark character, ', is required within a character field, it must be preceded by another quote mark. The added quote mark has no effect on the length of the character. As examples, to define the character field 'X', we write C'''X''', which results in three bytes; again, to define a single quote mark, we write C''''. The ampersand character (&) must be treated similarly by preceding it with a single ampersand when it appears in a character constant field.

In addition to defined constants, a program requires storage for input data, working storage for intermediate results and storage for output. This storage may be reserved using the Define Storage instruction. Its operands are similar to the ones we have seen so far in this section except that, of course, the constant in quotes is omitted. As examples,

```
IOWORK   DS      CL132
GRPAY    DS      PL6
RECPTS   DS      ZL6
RATETBL  DS      100PL3
BLOCK    DS      CL300
```

Note that when a series of DS or DC statements is written, the assembler assigns space continuously. The statements above will reserve 744 bytes of storage. The first 132 are reserved for IOWORK, the next six for GRPAY, the next six for RECPTS and so forth.

The use of C, P. or Z is immaterial since the length indicator for all three types is in bytes. If no length indicator is given, a length of 1 will be assumed. However, the reader is advised to use whichever type modifier (C, P, or Z) best indicates how the particular field will be used. Note that the DS will only reserve storage, it does <u>not</u> set it to zero. When a program is loaded, the contents of a given location depends on what was put there by the previous program. As a result, if the programmer requires an area to be clear, he must arrange for this himself; we will return to this point again. Another way of looking at the DS is to note that it does not cause any data to be put into the object program, it does cause open spaces to be set aside in core storage.

The conventions used in writing DC and DS statements are summarized in Table 2-1. Before leaving the subject of storage reservation, we will review what the assembler does when it encounters a label, irrespective of whether the label is the name of a constant, a work area or an instruction in the program. As we mentioned in Chap. 1, the assembler creates a label table and makes entires in it whenever a label is encountered. The entries comprise the label itself, as well as the storage address, or location, of that label. When a name is encountered as an operand in an instruction, the assembler than searches for that label in the label table and comes up with its core storage address which is then substituted in the instruction in place of the symbolic name. In addition, to the storage address, an additional entry must be made for each label -- the length, in bytes, of the data field the label refers to. This is necessary, because, as we discussed in Sec. 2. 1, the 360 has the ability to process variable length data. Since there are no explicit boundaries in storage between adjacent fields, each instruction which references variable length data must contain the length of the particular data field as well as its address. The length is a number from one to 16 for numerical data and one to 256 for alphanumeric data and is taken from the explicit length modifier in the DS or DC if one is present, otherwise the implicit length of the field is used. The length entry is often termed the length attribute and the address entry, the address value or attribute. As an example, the label RATEBL which is defined as 100PL3, has a length attribute of three. For instructions, even though they are not usually processed as variable length data, the length of the labeled instruction is stored in the label table: Two, four or six bytes depending on the particular instruction. The reader need not concern himself about instruction lengths at this point; they will be treated in Sec. 3. 7.

Type	Duplication Factor	Type Modifier	Length Modifier	Max. length in bytes	Padding
DC and DS	optional	P	optional	16(31 digits)	to left
	optional	Z	optional	16	to left
	optional	C	optional	256	to right

```
Examples:         CONA     DC   Z'-5'
                  CONB     DC   12P'100'
                  CONC     DC   PL16'0'
                  WORKA    DS   10PL5
```

Table 2-1   DC and DS statement conventions

Exercise 2-2 Write out constant definitions for the following numbers as zoned digits, packed digits and characters: -101. 5, 167, 2. 14159, -3, +2. How many bytes does each require. Write out constant definitions for the following character fields: X =, 'NET', NET & GROSS (The , is used to separate fields, all other characters must appear in the constant definitions).

2.3 Data Movement

The most direct way to bring data into the computer is by reading a punched card. The instructions for operating the card reader, which will be covered in Chap. 9, direct the card reader, in effect, to store the information contained in an 80-column card in 80 consecutive bytes of storage. While this is being done, the card reader control unit translates the card punches into the proper internal code as illustrated in Fig. 2-3. The programmer reserves an 80-byte area for this data by using a DS statement. The next step is usually moving the data from the input area to various processing areas. This allows another card to be read into the input area. The instruction used is the Move Character which has the format

```
label     MVC     A, B
```

The use of a label or name is optional; B is the address of the source field which will be moved to the destination field at address A. As we have mentioned before, all data in the 360 is addressed at its leftmost point. As an example, if the string of zoned numbers 00017 00513 starts at location 1000 and comprises two five byte fields, the address of the second field would be 1005. That is the storage layout would be

```
       ZO  ZO  ZO  Z1  S7  ZO  ZO  Z5  Z1  S3
  1000 ↑               1005↑
```

where the zone bits are indicated by Z and the signs by S, each zone (or sign) and digit occupying a byte. The number of characters moved, that is, the length of the field moved, is determined by the length attribute of the first operand. As an example, suppose A is the field located at 1000 and B is at 1005, in the illustration above. The instructions

```
           --
           MVC     B, A
           ---
           --
           --
  A        DS      ZL5
  B        DS      ZL5
```

would give

```
       ZO  ZO  ZO  Z1  S7  ZO  ZO  ZO  Z1  S7
  1000 ↑               1005↑
```

(The dashes above indicate the unspecified remainder of the program) Since the length of the A field is 5, the assembler will incorporate this fact into the MVC instruction so that when it is executed, only five characters will be moved. Also, the movement of characters within a field is from left to right. That is the character at 1000 is moved first, then the character at 1001 is moved, and so forth. The source field is unchanged by the move. The fields may overlap as we illustrate below.

Consider an eight-byte field at location AMT, which contains

```
ZO  ZO  ZO  Z1  Z2  Z3  Z4  Z5
```

Suppose that AMT must be shifted left two bytes, perhaps to align it correctly with other numbers in a tabulation. What we have to do is move that portion of the field beginning at AMT+3 over to AMT+1 as we have noted before. The assembler does allow this type of expression to be used as an address. The instruction to do the job would then seem to be

```
MVC     AMT+1, AMT+3
```

However, we must also indicate that only five characters are involved, otherwise the assembler will use the length attribute of the first operand, AMT, which is eight. This is accomplished by enclosing the desired length in parentheses as follows

```
MVC     AMT+1(5), AMT+3
```

This overrides the length attribute stored in the assembler's symbol table. After execution of this instruction, we will have in storage

```
ZO  Z1  Z2  Z3  Z4  Z5  Z4  Z5
```

The last twobytes can be set to zero by moving a two-byte zoned zero field into those locations. However, if we attempted to use this approach in shifting AMT one position to the right using

```
MVC     AMT+4(4), AMT+3
```

the Z1 byte would be propagated through the last four bytes of the field.

<u>Exercis 2-3</u> Describe the effect of the following coding

```
        MVC     AREA(1), ZERO
        MVC     AREA+1(255), AREA
        --
        --
AREA    DS      CL256
ZERO    DC      B'00000000'
```

(The B operand defines a binary constant of eight zero bits; it will be discussed further in Sec. 3.5)

Once the numerical data of a program is available in storage, it will have to be rearranged into packed format, if it is to be used in arithmetic operations. The Pack instruction is provided for this purpose its format* is

```
PACK    P, Z
```

where Z is the address of a zoned field which will be packed and stored in P. The length attributes of P and Z will be used unless they are overridden by the programmer. The first operand should contain enough bytes so that it has room to receive all the digits plus sign in the second operand. If the packed field is not completely filled, the left-over high order positions are set to zero. Processing begins at the rightmost byte of each field and continues to the left. All zones are ignored except the zone over the low-order digit which is assumed to represent a sign. The sign and low order four bits are interchanged and stored in the low order byte of the destination field. The sign and digits are <u>not</u> checked for validity. In particular, converting a blank field from zoned to packed format will generate an invalid sign which will cause processing to terminate when arithmetic operations are performed on the field - more on this point later. Overlapping fields may occur and are handled by storing each packed byte only after the necessary operand bytes are fetched. Following are examples of packing. The comments portion show the result of packing after the instruction is executed.

```
              PACK PACK1, ZONE1          01 5-
              PACK PACK2, ZONE2       00 12 3+
              --
              --
              --
      ZONE1   DC     ZL3'-15'         ZO Z1 -5
      PACK1   DS     PL2
      ZONE2   DC     ZL3'123'         Z1 Z2 Z3
      PACK2   DS     PL3
```

The Unpack instruction is provided to perform the reverse operation. Its format is

```
UNPK        Z, P
```

where P is a packed field which will be zone and stored in Z. The digits and

*We will omit an explicit reference to labels in instruction definitions from this point forward with the understanding that all instructions may be labeled.

signs of the packed operand are stored unchanged in the zoned operand location. The zone 1111 is added to all digits except the low-order one if the system is operating in the EBCDIC mode, the zone 0101 is supplied for ASCII-8 mode; the low order byte receives the sign of the packed operand. Again, the sign and digits are not checked for valid codes. UNPK and PACK are alike in their handling of overlapping fields and zero fill of the first operand.

Exercise 2-4 a) If the contents of location A are Z1 Z2 Z3 Z4 Z5 +6, show the contents of the destination field after each of the following instructions are executed; consider them independently.

```
PACK    B(4), A
PACK    B(2), A
PACK    B(6), A
PACK    A(3), A
PACK    A+2(3), A
```

b) If A contains 00 01 23 45 6+, show the contents of the destination field after each of the following, independent instructions.

```
UNPK    B(8), A
UNPK    B(3), A
UNPK    A(4), A+3(2)
UNPK    A, A+2(2)
```

Two instructions are provided for moving the zones of each position of the second operand into the zone positions of the first operand and for moving the numeric portions of each position of operand to the second operand. The formats for Move Numeric and Move Zones are

```
MVN     A, B

MVZ     A, B
```

Processing occurs from the second operand to the first operand left to right*. No validity checks are made and the overlap situation is similar MVC. These instructions both use the length attribute of the first operand. They are often

*As an aid to the reader, MVC, MVN and MVZ are the only decimal instructions which operate on data in left to right sequence; all other instructions including the arithmetic ones operate in a right to left sequence as we should expect by analogy with hand calculation.

useful in moving single zones or numeric portions as we shall see in successive examples.

When moving a field into a larger field, we often have to first set the larger field to zeros. This is particularly important when a total will be accumulated in the larger field. The Zero and Add Packed allows us to do this. Its format is

```
ZAP     A,B
```

The B field is moved to the A field, one byte at a time, from right to left; any unfilled high-order positions in A will be set to zero. The B field must not be larger than the A field, otherwise, an error will occur and processing will be interrupted. Only the B field is checked for valid sign and digit codes. The A and B field may overlap. When the rightmost byte of the A field is coincident with, or to the right of the rightmost byte of the B field. As illustrations, the following instructions show the destination field after instruction execution.

```
         ZAP     A,B                          00  00  00  43  2+
         MVN     C+2(1),C+4                   00  01  2+  45  6+
         ZAP     C,C+2(3)                     00  00  00  01  2+
*        ORIGINAL LAYOUT OF C                 00  01  23  45  6+
*        C HAS BEEN SHIFTED RIGHT FOUR PLACES
*        MVN USED TO CREATE VALID SIGN FOR 2ND OPERAND OF ZAP
         --
         --
A        DC      PL5'-123456789'
B        DC      PL2'432'
C        DC      PL5'123456'
```

An instruction which is particularly useful in shifting packed decimal data is the Move with Offset. Its format is

```
MVO     A, B
```

The entire second operand is offset to the left four bits and then placed adjacent to the <u>low order</u> four bits of the first operand. Processing occurs from right to left; overlapping may occur as we have described previously. As examples, with A = 00 12 3+ and B = 45 67 89 0+, the comments fields of the following instructions show the first operand after instruction execution. Each instruction should be considered independently of the others.

```
MVO     B,A(2)      00  00  01  2+
MVO     A,B(3)          56  78  9+
MVO     A,A(2)          00  01  2+
```

Worked Examples

2-1 Shifting

There are many occasions when the programmer must shift decimal data. A field may have to be shifted to align it with other numbers which will be added to it, or a dividend may have to be shifted to introduce enough additional places to develop a quotient of the proper size, or the quotient may have to be shifted after rounding, and so forth.

While there is no explicit shift instruction which will take care of all decimal shift requirements, we can perform shifts using the move instructions in this section. It will help the reader to keep in mind that MVC instructions cannot be used for right shifts since it processes from left to right; the MVO instruction cannot be used for left shifts since it processes from right to left. It is also important to keep in mind that all data is addressed at the leftmost byte. In summary:

MVC	MVO
left shifts	right shifts

There are four possible shift operations, right and left shifts, each an even or odd number of places. We will consider each.

Right Shift, Odd   To shift a field, A=01 23 45 6S, three places to the right, we write

```
MVO     A,A(2)          00  00  12  3S
```

If we need to preserve the original field, then A must be moved to another field B which is four bytes long and filled with arbitrary digits, say all 9's, to show the effect of the MVO.

```
MVO     B,A(2)          00  00  12  39
MVN     B+3(1),A+3      00  00  12  3S
```

The MVN is required to move the sign of A which is in the numeric portion of byte A+3. Note that each operand in MVO carries a length attribute. Since we want only the first two bytes of A to participate in the move, we override its length attribute of four as shown. For the MVN, the length attribute of the first operand determines the number of numerics to be moved. Since only one byte is involved, we again override the length attribute.

Right Shift, Even   To shift the A field, above, two places to the right

```
         MVN      A+2(1),A+3        01  23  4S  6S
         ZAP      A,A(3)            00  01  23  4S
```

The ZAP, which operates from right to left, fetches the first byte of A(3) which is 4S and stores it into the first byte of A, 6S; the second byte of A(3) to be operated on is 23 which is stored in the second byte from the right of A wiping out its previous contents of 4S, and so forth. The MVN is necessary to supply the second operand with a valid sign code which is required by the ZAP. When the original A field must be preserved, the shifted version of A will be developed in another four-byte field, B, by

```
         ZAP      B,A               01  23  45  6S
         MVN      B+2(1),B+3        01  23  4S  6S
         ZAP      B,B(3)            00  01  23  4S
```

Left Shift, Even   To shift a field, C = 00 00 01 23 45 6S four places to the left, we write the instructions

```
         MVC      C(4),C+2          01  23  45  6S  45  6S
         MVC      C+4(2),ZEROS      01  23  45  6S  00  00
         MVN      C+5(1),C+3        01  23  45  6S  00  0S
         MVN      C+3(1),ZEROS      01  23  45  60  00  0S
         --
         --
ZEROS    DC       BL2'0'
```

If the original C field must be preserved, the same logic can be used to develop the shifted version of C in another sixbyte field, D, by first moving C into D using MVC D(4), C+2. The constant, ZEROS, has two bytes of zero bits.

Left Shift, Odd   The most direct method here is to shift one extra position to the left so that we have an even number of places--the method above can be used for this--and then shift back one position to the right using the MVO instruction. We will leave it to the reader to work out the details for a left shift of three on the C field of the previous example.

Worked Example

2-2 As an illustration of some of the considerations which go into setting up input and work areas, we will discuss a hypothetical payroll program whose input data is arranged, one card per employee, in the following card column format

cc 1-8	Man Number
cc 9-12	Department Number
cc 13-32	Last Name
cc 33-34	Initials
cc 35-36	Day
cc 37-38	Month
cc 39-40	Year
cc 41-70	Unused
cc 71-75	Hours
cc 76-79	Rate
cc 80	Code

A card with an 11-punch in cc80 will be used to signal that the last card has been processed. The card with the 11-punch will not be processed but will initiate an end of job routine which may print out department totals, number of checks written and other control information. The overall logic will be to signal the control program to read a card into an 80-character area, INPUT. An immediate check will be made to see if cc80 contains an 11-punch, if it does, branch to the end of job routine, otherwise, move the contents of INPUT to another 80-character area, WORK, where payroll processing will be done. We adopt the approach of clearing out the input area as soon as possible to allow the control program to read another record while the previous record is being processed. At this point, we will limit the discussion to the storage reservation statements which follow

```
INPUT    DS    OCL80
         DS    CL79
CC80     DS    CL1
WORK     DS    OCL80
MANNO    DS    CL8
DEPNO    DS    CL4
NAME     DS    CL20
INIT     DS    CL2
DATE     DS    OCL6
DAY      DS    CL2
MONTH    DS    CL2
YEAR     DS    CL2
         DS    CL30
HRS      DS    CL5
RATE     DS    CL4
         DS    CL1
```

The first statement gives INPUT a length attribute of 80 but does not reserve any storage because of the duplication factor of zero. The next two statements reserve the next 80 characters in storage, which can be referenced by INPUT. In particular, the third DS allows the 80th input character to be referenced by the name CC80 with length attribute 1. An alternate approach is

```
INPUT    DS     CL80
CC80     EQU    INPUT+79
```

The EQU statement is interpreted by the assembler only, has no effect on the sequencing of data or instructions in the object program, but is used to make two symbols equivalent to each other, both address value and length attribute. Whenever CC80 is referenced, its address will be the 80th character in INPUT but it will also have a length attribute of 80. To avoid this problem, the symbol CC80(1) would be used signifying explicitly that the length is 1, overriding the implied length of 80. The reason for defining DATE with a six character length attribute but reserving no space is that we may want to handle all six characters as a unit, for example, to transmit them to an output area for printing. Having a name which references them collectively facilitates this. Or, we may want to process individual components--we can do this using the individual names which address the right characters and possess the proper length attributes. An alternative approach is to equivalence DAY, MONTH, and YEAR to DATE, DATE+2 and DATE+4 respectively with DATE defined as CL6. However, the length attributes of these symbols will not be correct and explicit lengths will have to be used--DAY(2), for instance. To avoid this added complication, we recommend the approach illustrated above. The DS after YEAR skips over the 30 unused columns and the final DS is necessary to fill up 80 positions. If we omitted this and followed RATE with another field, the first byte of that field would be wiped out when 80 bytes are transferred into WORK which would then be only 79 bytes long.

2.4 Decimal Arithmetic Instructions

The decimal arithmetic instruction set is an optional feature in some models of the System/360. However, because of the great utility of decimal arithmetic, many installations will have this feature and so, we will discuss its operation here. The decimal arithmetic instructions operate on data in the packed decimal form in a right to left sequence. Since all digits and signs are checked for validity, care must be taken to ensure that operand fields either do not overlap at all, or that they have coincident rightmost bytes. The latter occurs when a number is subtracted from itself to give a zero field, or added to itself to give an effective multiplication by two.

The format for the Add Packed Decimal instruction is

```
AP      A, B
```

The operand at B is added to the operand at A. The sum replaces the previous contents of A. The addition is algebraic taking into account the signs and all digits of both operands. An overflow occurs when result digits are lost due to either a carry out of the high order position of A, or if the B field is larger than the A field causing result digits to be lost. The length attributes of A and

B determine the number of bytes to be added. When a zero sum results, it will always have a + sign. As examples (the results are shown as comments):

```
A    DC     PL3'-185'
B    DC     PL3'+185'
C    DC     PL6'495'
D    DC     PL6'99999999999'
X    DS     CL3
Y    DS     CL6
     --
     --
     --
     ZAP    Y,B        Y = 00  00  00  00  18  5+
     AP     Y,C        Y = 00  00  00  00  68  0+
     AP     Y,A        Y = 00  00  00  00  49  5+
     MVC    Y,D        Y = 99  99  99  99  99  9+
     AP     Y,C        Y = 00  00  00  00  49  4+
*    OVERFLOW WILL OCCUR ABOVE
     MVC    X,A        X = 00  18  5-
     AP     X,B        X = 00  00  0+
```

When an overflow occurs, the programmer can detect it by testing the overflow indicator after every addition or subtraction or by arranging for the computer to detect overflows automatically. These options will be covered at greater length in Sec. 2.5.

The format for the Subtract Packed Decimal instruction is

```
SP      A,B
```

The field at B is subtracted from the field at A and the difference replaces the contents of A. Subtraction is identical to addition except that the sign of the B operand is changed after fetching it from storage and before the subtraction begins. The overflow situation is the same here as for addition.

As examples,

```
A   DC     PL2'150'
B   DC     PL2'-150'
X   DS     PL2
    --
    --
    ZAP    X,A      X = 15   0+
    SP     X,B      X = 30   0+
    SP     X,X      X = 00   0+
    AP     X,B      X = 15   0-
    SP     X,A      X = 30   0-
```

The format for the Multiply Packed Decimal instruction is

```
                                 Multiplicant      MCAN
MP   MCAN, MPLY          x       Multiplier        MPLY
                                 ------------
                                   Product
```

The product of MCAN and MPLY replaces the contents of MCAN. MPLY is limited to 15 digits plus sign and must be smaller in length than MCAN. Since the number of digits in the product equals the number of digits in the multiplier plus the number of digits in the multiplicant, the multiplicand field must have enough high order zeros for a field equal in length to the multiplier field size. If this condition is not met, processing will terminate. The maximum product size is 31 digits; its sign is determined by the rules of algebra from the signs of both factors. If both factors have the same sign, either + or -, the sign of the product will be +, if the factors have different signs, the sign of the product will be -.

Before getting into examples of the MP instruction, let us examine the operation of multiplication itself and extract the rules determining decimal point placement and product size. Consider the following calculations

```
  85          13.25
x 42        x  1.03
----        -------
 170          39 75
340         1325 0
----        -------
3570        13.6475
```

First, note that the number of digits in the product is not larger than the sum of the number of digits in both factors and may be one less--this will be important when we discuss division. Second, the number of decimal places in the product--digits to the right of the decimal point--equals the sum of the number of places in each factor. We will now discuss the instructions for both of these calculations above with the requirement of rounding off the second one to two decimal places. This will be done by adding a 5 to the third decimal place and then discarding or truncating the last two places; the result will be 13.65. The rounded quantity is then to be stored in AMT, a four byte field.

```
         ZAP     PROD,A                  00  00  08  5+
         MP      PROD,B                  00  03  57  0+
         ZAP     WORK,PRINC              00  03  01  32  5+
         MP      WORK,RATE               00  01  36  47  5+
         AP      WORK,ROUND              00  01  36  52  5+
         MVN     WORK+3(1),WORK+4        00  01  36  5+  5+
         MVC     AMT,WORK                00  01  36  5+
         --
         --
A        DC      PL2'85'
B        DC      PL2'42'
PROD     DS      PL4
PRINC    DC      PL3'13.25'
RATE     DC      PL2'1.03'
WORK     DS      PL5
AMT      DS      PL4
ROUND    DC      PL2'50'
```

The first ZAP clears the PROD area where the product of A and B will be formed by the next instruction. Note that PROD, although equal to the sum of the field sizes of A and B, is one byte too large in this particular example. However, A and B being two bytes each, could be as large as three digits each giving a six digit product. In that case a four byte product field would be necessary. While it may occasionally require an extra byte, the safest approach is to define the product field equal in size to the sum of the lengths of each factor field. In the second multiplication, the rounding is of interest. We need to perform the sum

$$\begin{array}{r} 136475 \\ \underline{+\quad 5} \\ 136525 \end{array}$$

However, this cannot be done by defining ROUND as a 5 because this will not give the correct alignment. Note also that, while decimal points may appear in a packed decimal constant, they are ignored by the assembler. Their only function is to improve readability for the programmer. The MVC instruction will move only the required first four bytes from WORK because the length of the field to be moved is controlled by the length attribute of the first operand, AMT, which is four bytes.

The operation of division is somewhat more complicated than the preceding instructions. Before we introduce the divide instruction, we will review some of the basics of division. Consider the following division where it is required to develop three decimal places in the quotient 12.56/11.3:

$$\begin{array}{r} 1.1115 \\ 11.3\overline{)12.5600} \\ \underline{11\ 3} \\ 1\ 26 \\ \underline{1\ 13} \\ 130 \\ \underline{113} \\ 170 \\ \underline{113} \\ 570 \\ \underline{565} \\ 5 \end{array} \quad + \quad \frac{.00005}{11.3} \qquad \begin{array}{r} \text{quotient} \\ \text{divisor}\overline{)\text{dividend}} \end{array} + \frac{\text{remainder}}{\text{quotient}}$$

To develop a three decimal place quotient, four places must be computed and then rounded to three giving 1.112. This is the usual form in which division problems are stated for a computer: Given the number of decimal places in the divisor and the desired number of decimal places in the quotient, find the number of zeros which must be shifted into the dividend. Since the dividend equals the quotient times the divisor, we can use the rule we developed for decimal points in multiplication:

$$\boxed{\text{Dividend places} = \text{Quotient places} + \text{Divisor places}}$$

This indicates for example, that two places (zeros) must be added to the dividend to get a three place quotient since the dividend above (12.56) has two places and the divisor (11.3) has one.

Again, referring to the rule that the number of digits in a product is equal to, or one less than, the sum of the number of digits in both factors, we can devise a rule for determining the maximum number of digits, including decimal places, in a quotient. For a maximum size quotient we have

$$\text{or,}\ \boxed{\begin{array}{lclcl} \text{Dividend digits} & = & \text{Quotient digits} & + & \text{Divisor digits } -1 \\ \text{Quotient digits} & = & \text{Dividend digits} & - & \text{Divisor digits } +1 \end{array}}$$

The remainder is that portion of the dividend which is left over after the required number of quotient places have been calculated. The remainder's decimal point is located in the same position as the dividend's point as can be seen from the sample calculation. However, since the remainder will always be smaller than the dividend, so that a field equal in size to the remainder field will hold it. The remainder will have the same sign as the dividend. While the sign of the quotient is determined algebraically from the signs of the divisor and dividend. If their signs are alike, either + or -, the sign of the quotient will be +; if their signs are different, the sign of the quotient will be -. We suggest that the reader consider a few sample divisions, such as 231/11, to reinforce his understanding of the rules above.

With these preliminaries behind us, the Divide Packed Decimal instruction format is

```
DP DVDND, DVSOR
```

The dividend (DVDND) is divided by the divisor (DVSOR) and the quotient and remainder replaces the dividend. The remainder is placed rightmost in the DVDND field and has a size equal to the divisor size. The quotient is placed leftmost in the DVDND field and together with the remainder occupies the entire DVDND field. An interruption will occur if the divisor field is larger than 8 bytes or if the quotient is too large for its field. When this happens, the operation is suppressed and the dividend remains unchanged. Since the minimum divisor is one digit and sign, and the maximum decimal field is 31 digits and sign, this leaves 29 digits and sign as the maximum quotient. To aid the reader in programming division, we will devise a simple rule which will ensure correct divisions. It will simplify matters to use a simple notation, rather than writing everything out. Let D be the number of digits (not counting high order zeros but including low order zeros) in the dividend, S, in the divisor and Q in the quotient; we will use s to indicate a single sign place. Our first rule is

$$Q = D - S + 1$$

which was developed above. We note that Q+s+S+s, which replaces the dividend field after division, can be at maximum 32 digit places since this is the largest allowed decimal field. From our formula above, Q+S = D+1 so the maximum D is given by

$$Q+s+S+s = 32$$

or

$$D+1+s+s = 32$$

Since the signs take one digit place each,

$$D = 29 \text{ (maximum)}$$

we will next develop a rule which will show that a successful division will occur if two more high-digit places are added to the dividend. From the argument above, we know that we will have room for them even in the case of the maximum dividend. Let us ask the question, "How many digit places must be added to the dividend plus sign to accommodate the quotient and remainder plus signs?" with x as the unknown number of places, we are asking what value of x will solve the equation

$$D+s+x = Q+s+R+s$$

Since, $Q = D-S+1$ and the remainder, R, equals S, we have

$$D+s+x = D-S+1+s+S+s$$

or

$$x = s+1$$
$$= 2$$

For the reader who doesn't possess a taste for this line of algebraic reasoning, our rule nets out to: "Always add two high order zeros to the dividend field before dividing." Now for some examples.

Worked Examples

2-3 Find the quotient of 12.56/11.3 to three decimal places. Note that for a three decimal place quotient, we must calculate four places and then round off to three. We must therefore plan for a four decimal place quotient. Since the quotient decimal places (4) plus the divisor decimal places (1) must equal the dividend decimal places, three more decimal places must be added to the dividend. In addition, according to the division rule above, two high order zeros must be added to the dividend field. In all, the dividend field will be five bytes including the three low order places which have to be added. The quotient will be five digits, at most, since $Q=D-S+1 = 7 - 3 + 5$, so the quotient field must be three bytes. Following is the coding. The remarks section of each instruction shows the destination field after instruction completion.

```
                 ZAP  DVDND,C1256            00 00 01 25 6+
                 MVC  DVDND(3),DVDND+2       01 25 6+ 25 6+
                 MVO  DVDND,DVDND(4)         00 12 56 +2 5+
                 MVC  DVDND+3(1),ZERO        00 12 56 00 5+
                 MVZ  DVDND+4(1),ZERO        00 12 56 00 0+
*LEFT SHIFT OF THREE PLACES HAS BEEN COMPLETED
                 DP   DVDND,C113             11 11 5+ 00 5+
                 AP   DVDND,ROUND            11 12 0+ 00 5+
                 MVO  QUOT,DVDND(2)          01 11 2+
                 --
                 --
                 --
```

```
C1256   DC   P'12.56
C113    DC   PL2'11.3'
DVDND   DS   PL5                00 XX .XX 00 0S
ZERO    DC   BL'0'              GIVES ONE BYTE OF 00000000
QUOT    DS   PL3
ROUND   DC   PL1'5'
```

The first five instructions are a slightly modified version of the technique developed in Example 2-2 for left shifting. In this division, the quotient reached the maximum size, five digits. The remainder, occupying a field equal to the divisor in length, is +5. The constant C113 is shown with an explicit length of two bytes as a reminder that the remainder will occupy two bytes in DVDND. Of course, one does not write a program to compute the quotient of 12.56/11.3. We use a specific example to show the results of each step of the calculation.

2-4 Given a dividend of the form XXXXXXX.XX where X is any digit and a divisor which can range from XXX.XX to X.XX, it is desired to compute the quotient to three decimal places and round off to two. All numbers will be positive.

For a three place quotient and a two place divisor, five places are required in the dividend so that the dividend must be shifted left an extra three places giving D = 9+3 = 12. We have an additional complication here, the range in size of the divisor, which is typical of many real life division problems for a computer. The problem might be to compute output produced per man-hour in a number of production departments where it is known from experience that the number of man hours worked can be as low as 1.00 but not higher than 999.99. To plan the correct size dividend, we must take both sizes into account. The remainder field can be as large as XXXXXS and the maximum quotient field is D-S+1 where we use the minimum S value. Therefore Q maximum = 12-3+1 = 10, or including sign, 11 digits. Adding the remainder which is six digits we have 17 digits or a nine byte dividend field.

In this case, merely adding two high order zeros to the dividend would not be enough but only because we are dealing with a range of sizes of the divisor. The coding is

```
*SHIFT LEFT 4, RIGHT 1 TO GET LEFT SHIFT OF 3
 ZAP   DVDND(7), D             00  00  XX  XX  XX  XX  XS  XX  XX
 MVC   DVDND+7(2), ZEROS       00  00  XX  XX  XX  XX  XS  00  00
 MVN   DVDND+8(1), DVDND+6     00  00  XX  XX  XX  XX  XS  00  0S
 MVN   DVDND+6(1), ZEROS       00  00  XX  XX  XX  XX  X0  00  0S
 MVO   DVDND, DVDND(8)         00  00  0X  XX  XX  XX  XX  00  0S
 DP    DVDND, DVSOR            0X  XX  XX  XX  XX  XS  XX  XX  XS
 AP    DVDND(6), ROUND         0X  XX  XX  XX  XX  XS  XX  XX  XS
 MVO   DVDND(6), DVDND(5)      00  XX  XX  XX  XX  XS
 MVC   AVG, DVDND+1            XX  XX  XX  XX  XS


          --
          --
          --
 DVDND  DS     PL9
 ZEROS  DC     BL2'0'          GIVES TWO BYTES OF ZEROS
 D      DS     PL5             XXXXXXX. XX
 DVSOR  DS     PL3             XXX. XX  TO  X. XX
 ROUND  DC     PL1'5'
 AVG    DS     PL5             XXXXXXX. XX
```

The first five instructions set up D in DVDND properly shifted. The ZAP instruction applies only to the first seven bytes of DVDND, therefore, the next MVC is used to clear the low order two bytes. The DP creates a three decimal place quotient which is rounded by adding five into the third decimal place. The following MVO truncates the now unwanted third digit by shifting right one place. After the truncation, the maximum quotient size, including sign, is 10 digits, or five bytes, the field AVG is set up to handle this size and therefore, the final MVC references DVDND+1.

Our simple rule of "Add two zeros to the dividend" can be extended to handle the case where the length of the devisor (S) may vary from a maximum value (S_{max}) to a minimum value (S_{min}). Without going through the details, which we leave to the reader as Prob. 2-16 at the end of this chapter, the amount to be added to the dividend (which is assumed to contain space for its own sign) is

$$\boxed{2 + S_{max} - S_{min}}$$

As an example, the maximum divisor in the example above has five digits and the minimum has three digits. The dividend has 12 places plus one more for the sign; since 2 + 5 - 3 = 4 places have to be added, we have a total of 17 places, or nine bytes.

2.5 Branching and Testing

Stored program computers derive their versatility from their ability to test what is going on in a calculation, and depending on the outcome of that test, to take an almost endless variety of alternative actions. At the heart of this ability are the branching and testing instructions. These instructions allow comparisons to be made between data, whether the data is interpreted as numbers, characters or as a string of 1's and 0's. In addition, they allow alternative actions to be taken depending on the outcome of arithmetic operations: Positive, negative, zero or overflow. As we shall see in the examples which follow throughout this book, these capabilities and their extensions are very powerful programming tools. Fundamental to these instructions is a register in the 360 which is called the condition code indicator. This register can be regarded as a switch with four settings. This switch is set after every add or subtract instruction as well as after compare instructions. The setting of this switch may be determined by the branching and testing instructions. The meanings of the switch settings for arithmetic operations are: Less than zero result, zero result, greater than zero result and overflow. For compare instructions, only three of the settings are used, low comparison, equal comparison, and high comparison. These three correspond exactly with the first three settings discussed above for arithmetic operations, only their interpretations are different. Testing the condition code will not alter its setting. In general, the condition code is changed only by an add/subtract instruction, a compare instruction or an edit instruction. There are a few exceptions to this statement which we will discuss as they come up. Let us first examine the Compare Packed Decimal instruction. Its format is

```
CP    A,B
```

A right to left algebraic comparison is made between both operands taking into account signs and all digits.

If one operand is shorter than the other, it will be effectively padded out with zeros for the purpose of comparison. However, neither operand is changed by this instruction. The condition code is set according to whether the first operand compares equal to, higher, or lower than the second operand. These possibilities can be tested by inserting a conditional branch instruction after the compare. There are several varieties of this instruction depending on the condition being tested. Their formats are

```
BH    LABEL
BL    LABEL
BE    LABEL
```

These are, respectively, branch to LABEL only if first operand is high (BH), low (BL), or equal (BE). If the given condition is not met, instruction processing continues sequentially with the next instruction immediately following the branch. As an example, with A = 25 and B = 10, the instructions

```
      CP   A,B
      BL   TEST
ENTRY AP   C,D
```

will cause the instruction at ENTRY to be executed, rather than the one at TEST because A is greater than B and, therefore, the BL condition cannot be met. If A were less than B, then the branch would be taken and the next instruction to be executed would be the one at TEST rather than the one at ENTRY. In addition to the three possibilities above, additional combinations of results can be tested for by the following instructions.

```
BNH   LABEL
BNL   LABEL
BNE   LABEL
```

The instruction mnemonics mean branch to LABEL if the first operand is <u>not</u> high (BNH) (the branch will be taken if the first operand is equal to or lower than the second operand), not low (BNL) or not equal (BNE).

The results of arithmetic operations can be tested by the following conditional branch instructions

```
BP   LABEL
BM   LABEL
BZ   LABEL
BO   LABEL
```

The branch to LABEL will be taken if the result of an addition (including ZAP) or subtraction is positive (BP), minus (BM) or zero (BZ). A zero result, irrespective of its algebraic sign, which will normally be +, will set the condition code indicator to zero, only. The condition code will also be set for overflow results (BO). Using the BO instruction, the programmer can determine if an overflow has occurred. However, as we have stated previously, it is possible to put the computer into an automatic overflow detection mode in which it will take automatic action whenever an overflow occurs. The action usually consists of aborting the program after printing out an indication of where and how the offense took place. By a suitable modification to the control program, the user can substitute for the abort procedure a routine of his own. This will be discussed further in Sec. 3.8. At this point, assume that the computer is in the automatic overflow detection mode and that the BO instruction will not be necessary. The programmer should also take particular caution to set up adequately sized fields so that overflows will not occur. To do this, it is essential that he know what size numbers his program will have to deal with.

In addition to the conditional branches, which will be taken only when the condition they are testing is met, there is also an unconditional branch which is always taken whenever it is executed. Its format is

```
B    LABEL
```

As an example of an instance where this branch would be used, consider a payroll program which reads in a time card, processes it and then returns to start the operation over again. If the initial point is labeled FIRST, the return can be accomplished by a B FIRST instruction.

Exercise 2-5 With A = 20, B = 15, C = 5, D = -20, indicate the sequence in which the following instructions will be executed

a)
```
START  CP   B,C
       BNE  ADD
       BE   OUT
ADD    AP   C,A
       BP   START
OUT    --
       --
```

b)
```
TEST   CP   C,D
       BH   ADD
       SP   C,B
       B    OUT
ADD    AP   D,B
       B    TEST
OUT    --
       --
```

Also, what are the final values of all variables, A, B, C and D. Consider a) and b) separately.

Worked Examples

2-5 Self-Checking Number

When certain numbers are frequently transcribed manually and it is essential that they be transcribed correctly, savings account numbers for instance, a common practice is to include additional coding in the number which is related by some formula to the values of the other digits. If adjacent digits are transposed, a frequent clerical error, the coding scheme can be designed to detect this. There is also protection against fraudulent use of made-up numbers in that it is statistically unlikely that a made-up number will match the coding supplied by the formula. As an example of a typical coding scheme, consider a six digit number with a seventh self-checking digit appended to the right. Let the seventh digit be computed by taking the second fourth and sixth numbers, adding them and then multiplying the sum by two, and finally, adding the product to the sum of the other three digits. The low order digit of the sum is the self checking digit. As an example, the check digit for 201635 is 8 since

$$2 + 1 + 3 + 2\ (0+6+5) = 28$$

So that the entire account number is 2016358. If the third and fourth digits were transposed as 201635, we would calculate a check digit of 3 since

$$2 + 6 + 4 + 2\ (0+1+5) = 23$$

which would not match. Let a seven digit number be in location ACNO in zoned format. The seventh digit is the check digit; write a routine which will verify that the number agrees with its check digit using the logic above for calculating the check digit.

Our approach will be to sum the odd-place digits in ODD, the even place digits in EVEN, then compute the check digit and compare it with the one supplied as input. If it is not valid, go to ERROR, otherwise go to CONTINUE.

```
PACK      HOLD, ACNO(1)
AP        ODD, HOLD
PACK      HOLD, ACNO+(1)
AP        EVEN, HOLD
PACK      HOLD, ACNO+2(1)
AP        ODD, HOLD
PACK      HOLD, ACNO+3(1)
AP        EVEN, HOLD
PACK      HOLD, ACNO+4(1)
AP        ODD, HOLD
PACK      HOLD, ACNO+5(1)
AP        EVEN, HOLD
MP        EVEN, TWO
```

```
         AP     EVEN,ODD
         PACK   HOLD,ACNO+6(1)      INPUT CHECK DIGIT
         CP     HOLD,EVEN+2(1)
         BE     CONTINUE
         B      ERROR
 ACNO    DS     ZL7
 HOLD    DS     PL1
 EVEN    DS     PL3
 ODD     DS     PL2
 TWO     DC     PL1'2'
CONTINUE --
         --
         --
```

The EVEN field is three bytes because it must have room for the multiplier and multiplicand which requires one byte plus two digits and sign. The computed check digit with sign will be in the low-order byte of EVEN which has address EVEN+2. Because EVEN has a length attribute of three, an explicit length of one is stated in the CP instruction since the first operand, HOLD, has a length attribute of one.

If the computed and the input check digits are equal, the BE instruction when executed will cause a transfer to the instruction at CONTINUE at which point execution will continue sequentially until another branch is taken. If the two are not equal the branch to CONTINUE will not be taken and execution will proceed sequentially to the next instruction which is an unconditional branch to ERROR where appropriate action can be taken such as printing the invalid account number. When several branches are coded in sequence, the best programming strategy is to put at the top of the list the branch which is most likely to occur, then the next most likely one and so forth. This will speed up program execution since fewer unsuccessful conditional branches will have to be tested. One other point should be noted--we have included working storage, ACNO, . . . , TWO, within a sequence of instructions. In this case, no harm will come since the data area is preceded by an unconditional branch. However, if we did not take this precaution, the computer would attempt to execute the contents of ACNO, as if it were an instruction, with unpredictable but undoubtedly terminal results for the program. Specifically, if the first eight bits of ACNO did not constitute a valid operation code, an interrupt would occur.

2-6 Round-off

So far, we have been rounding-off by adding a round-off constant as we have been dealing with positive quantities only. However, if the number which is being rounded-off is negative, the round-off constant should be subtracted. As an illustration -117.647 when rounded-off to two digits should be -117.65.

To get this result, .005 should be subtracted, not added. As an example of how this can be coded, assume the number to be rounded-off is in AMOUNT and has three decimal places and may be either positive or negative. The following coding will round-off AMOUNT to two decimal places and store the result in FINAL.

```
            ZAP    AMOUNT, AMOUNT          SET CONDITION CODE
            BM     MINUS
            AP     AMOUNT, ROUND
            B      MOVE
MINUS       SP     AMOUNT, ROUND
MOVE        MVN    FINAL+3(1), AMOUNT+4    MOVE SIGN
            MVO    FINAL, AMOUNT(4)
            --
            --
            --
AMOUNT      DS     PL5                     OX  XX  XX.  XX  XS
FINAL       DS     PL4                     XX  XX  X.X  XS
ROUND       DS     PL1'5'
```

To test the sign of AMOUNT, the first instruction will ZAP AMOUNT onto itself. This does not change AMOUNT but does set the condition code which is then tested by BP. If this branch is not taken, AMOUNT is then either positive or zero and will be rounded-off in the plus direction. If AMOUNT is zero, the round off constant will be truncated by the MVO and the quantity stored in FINAL will be zero.

2-7 The "Indian" Problem

An important programming technique is looping, or executing a particular sequence of instructions a given number of times. As an illustration of this technique consider the "Indian" Problem, a classic programming exercise. We will develop a routine to compute the 1967 value of a $24 deposit made in 1627 by the Indians who sold Manhattan Island. Assume that the bank rate in 1624 -- when money was not as tight in the land -- was 3%, to be compounded annually. Three percent interest can be added by multiplying the current principal by 1. 03. This has to be done 1967-1627 = 340 times. We will set up 340 in a counter and decrement once for each time interest is taken. When the counter reaches zero, we have finished. The other point of interest is how large will the final number be. Reference to compound interest tables indicates that the amount will be around $500, 000. 00 which will require five bytes. Since the multiplier, 1. 03, requires two bytes, the product field should be seven bytes long. After each multiplication, the product will be rounded from four decimal places to two. The final value will be set up in zoned format in PRINC. The coding is

```
        MVC   COUNT,C340
        MVC   PROD,DEPOSIT
LOOP    MP    PROD,RATE          OO OX XX XX X.X XX XS
        AP    PROD,ROUND
        MVN   PROD+5(1),PROD+6   OO OX XX XX X.X XS XS
        ZAP   PROD,PROD(6)       OO OO OX XX XX X.X XS
        SP    COUNT,ONE          DECREMENT COUNTER
        BP    LOOP               LOOP NOT FINISHED
        UNPK  PRINC,PROD+2       ZX ZX ZX ZX ZX ZX.ZX SX
        --
        --
        --
COUNT   DS    PL2
C340    DC    PL2'340'
PROD    DS    PL7
DEPOSIT DC    PL3'24.00'
RATE    DC    PL2'1.03'
ROUND   DC    PL2'50'            .0050 SINCE PROD HAS 4 PLACES
ONE     DC    PL1'1'
PRINC   DS    ZL8
```

The number of years, C340, is moved to a separate location, COUNT, where it will be decremented rather than in C340 itself which would destroy the constant. This is good programming practice since a given constant may be required again in a program, or it may be used in a different section at which point the programmer may not remember that he had previously changed its value. In light of these possibilities, it is sound programming strategy not to do arithmetic in constant locations. The multiplication and rounding with subsequent right shift of two places illustrates nothing new. However, it is important to note that since the computer hardware doesn't know anything about our intentions in regard to decimal points -- it deals with whole numbers only -- we must arrange to take care of decimal point placement. We note that PROD and RATE are understood to have two decimal places; as a result their product will have four places and, since we wish to round it to two places, 50 is added. In terms of our assumptions about decimal point placement, this round-off constant is .0050. The SP COUNT,ONE instruction decrements the loop count. As long as this quantity is greater than zero, that is, one or more, the loop has not been completed and so we follow with a BP LOOP. On the final pass through the loop, COUNT=1; when it is decremented, its value will be zero and the conditional branch will not be taken. Execution will proceed sequentially with the UNPK instruction. The second operand, PROD+2, of UNPK could be written PROD. In that case, since PRINC is not large enough to contain all 13 zoned digits which would result when PROD is unpacked, the extra high order digits (in this case, zeros) would be ignored. While the result is the same for either operand address, we recommend the usage in the coding above as this makes clearer the intent of the instruction.

The ability to make comparisons between characters is an important part of a computer's instruction repertoire. The format for the Compare Logical* Character instruction is

```
CLC      A, B
```

The length of fields to be compared is governed by the length attribute of field A. If desired, an explicit length may be given with A. The comparison proceeds left to right, comparing a character at a time, and terminates as soon as an inequality is found. The condition code is set to distinguish between equality (BE) and inequality (BNE) as well as high, low and the other combinations discussed with the CP instruction. It is important to be able to distinguish between high and low alphabetic (and zoned numberical) comparisons. One of the most important applications of computers is the sorting of data files which combine alphabetic and numberical information. Catalog part "numbers" are a good example since they often have mixed alphabetic and numberical formats. If you refer to Fig. 2-1, you will notice that the codes assigned to the letters and numbers are not randomly assigned. There is a definite progression from low to high in the codes assigned A through Z and O through 9. In fact, the codes can be considered as binary numbers with A having the value 11000001, B, 11000010 which is greater than A, down through the digits which have higher code values than the letters. In this way, through the CLC instruction, we are able to arrange alphabetic data in the proper sequence. As an additional note, when the character set is sequenced by code number from low to high, we have what is called the <u>collating sequence</u>. It is an important consideration when identification codes are being established for data files.

As an example of the use of CLC, if field A contains ABCDEFG and B contains ABCDEF*, the instruction

```
CLC      A(6), B
```

will set the condition code to equal, whereas the instruction

```
CLC      A, B
```

will result in an unequal setting because of the mismatch between G in field A and * in field B.

* "Logical" is used to distinguish CLC from algebraic comparisons which take into account signs. This will be explored at length in Chap. 5; the distinction is not important at this point.

Worked Example

2-8 One of the potential pitfalls for the unwary will occur when a field of blanks is packed and then arithmetic performed on it. A blank appears in storage as 0100 0000 in EBCDIC. When it is packed, the result is a digit zero with a sign code of 0100 which is invalid. This will be detected as soon as arithmetic is attempted on the field and an interruption will occur.

We will describe here a routine to check for the presence of a blank in the low order position of a field and substitute a zero. In another version, the entire field will be set to zero if a blank occurs in the low order position. The zoned field in question is 10 bytes long at location FIELD

```
         CLC    FIELD+9(1), BLANK
         BNE    PACK
         MVC    FIELD+9(1), ZERO     REPLACE BLANK WITH ZERO
PACK     PACK   NUMBER, FIELD
         --
         --
         --
FIELD    DS     ZL10
NUMBER   DS     PL6
BLANK    DC     CL1' '
ZERO     DC     ZL1'0'
```

To set the entire field to zero, we write

```
         CLC    FIELD+9(1), BLANK
         BNE    PACK
         MVC    FIELD(1), ZERO       THESE TWO INSTRUCTIONS
         MVC    FIELD+1(9), FIELD    PUT ZEROS THROUGH FIELD
PACK     PACK   NUMBER, FIELD
```

Since the length attribute of the first operand (10 bytes) of CLC controls the length of the compare, and since BLANK is only one byte in length, an explicit length of one must be stated. The technique of setting the entire field to zero has been discussed in Exercise 2-3.

2.6 Introduction to Input-Output

In this section we will present a few simple "prescriptions" for input-output, so that the reader may write programs which permit the computer to communicate with the outside world. A more detailed explanation of the subject will be given in Chaps. 9 and 10, at this point, we will limit the background discussion to a brief sketch.

The Operating System allows the programmer to define the data files he is working with. A data file may be a deck of cards containing related information, or it may be a series of printed pages containing the results of a calculation. The file definitions consist of describing the size of each record in the file, which I/O device the file will reside on, and which area in storage will be used to service the device, just to mention a few of the more important file parameters. As the reader will see in Chap. 10, more complex file definitions are possible, particularly those relating to disk files and tape. Here, we will consider only card input and printer output.

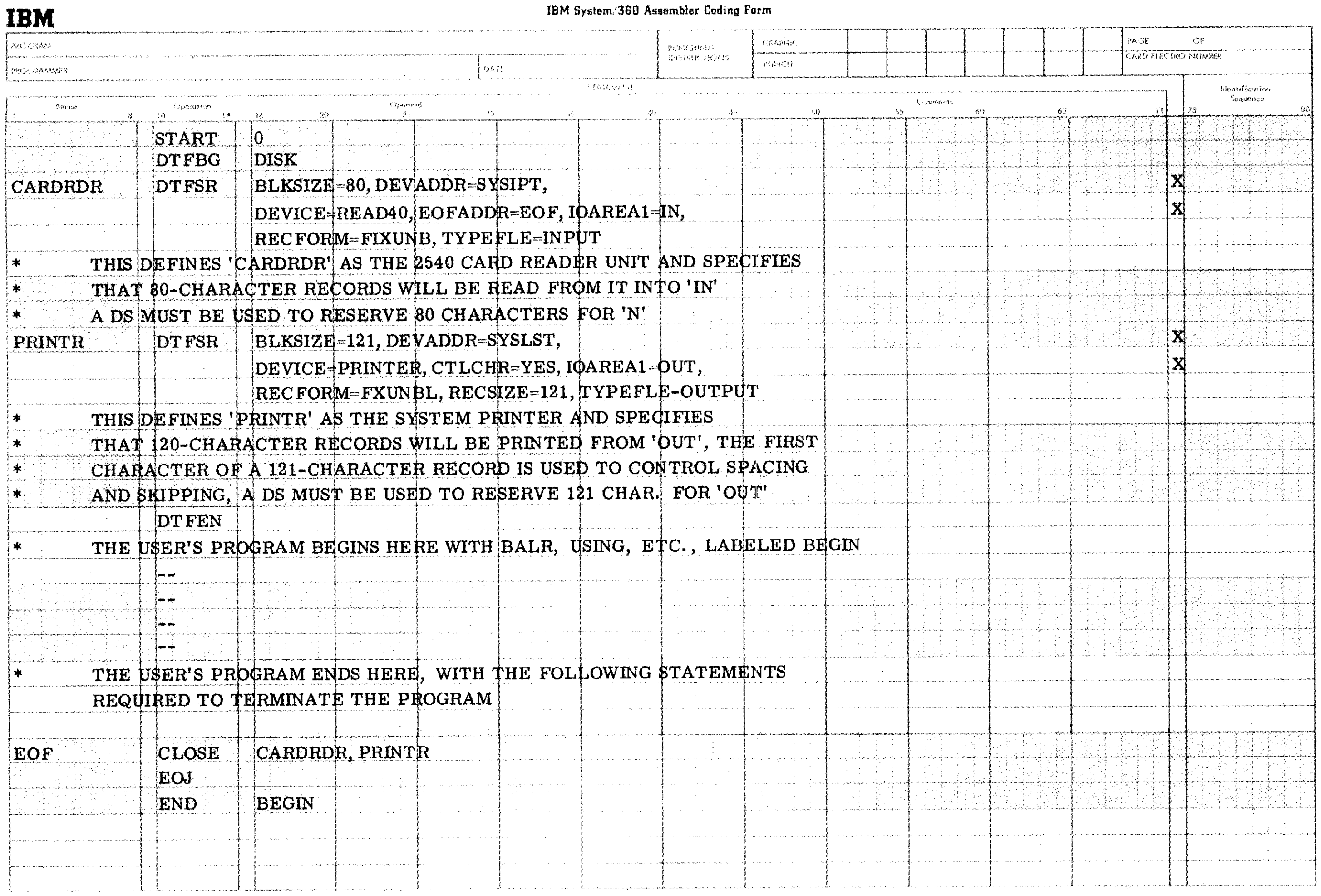
Figure 2-6 shows the file definition statements to allow input from the card reader (CARDRDR) and output on the printer (PRINTR). The areas which service them are IN, for the card reader, and OUT for the printer. This means that whenever a card is read, all 80 columns of information will be found in the 80-byte area, IN. Similarly, when a line of information is to be printed, it will first be assembled in OUT. The comment cards in Fig. 2-6 remind the reader that DS statements must be written reserving 80 bytes for IN and 121 for OUT. The first byte of OUT will be used by the Operating System to control printer spacing and will not be printed. See Fig. 2-7 for a list of printer control characters. The following 120 bytes will be printed in print positions 1-120. The statements for card reading or printing are very simple. They are

```
GET     CARDRDR

PUT     PRINTR
```

These statements may be labeled, if required. When GET is executed, a card is read and its contents placed in area IN destroying whatever was there previously. When PUT is executed, bytes 2 through 121 of area OUT are printed and the printer is spaced according to byte 1. This does not change the contents of OUT. These two statements are quite different in their effect on the assembler than the statements we have encountered previously. As the reader might suspect, GET or PUT is assembled, not into just one machine instruction, but into a number of instructions. (For this reason they are called <u>macro</u> instructions) While the file definition statements and the Operating System functions they represent in Fig. 2-6 may seem at first complicated, they spare the programmer from much coding which would otherwise be required to deal with input-output. Before GET or PUT may be used, the devices they reference must be readied for use with the statement

```
OPEN        CARDRDR, PRINTR
```



IBM System/360 Assembler Coding Form

```
         START    0
         DTFBG    DISK
CARDRDR  DTFSR    BLKSIZE=80,DEVADDR=SYSIPT,                              X
                  DEVICE=READ40,EOFADDR=EOF,IOAREA1=IN,                   X
                  RECFORM=FIXUNB,TYPEFLE=INPUT
*     THIS DEFINES 'CARDRDR' AS THE 2540 CARD READER UNIT AND SPECIFIES
*     THAT 80-CHARACTER RECORDS WILL BE READ FROM IT INTO 'IN'
*     A DS MUST BE USED TO RESERVE 80 CHARACTERS FOR 'N'
PRINTR   DTFSR    BLKSIZE=121,DEVADDR=SYSLST,                             X
                  DEVICE=PRINTER,CTLCHR=YES,IOAREA1=OUT,                  X
                  RECFORM=FXUNBL,RECSIZE=121,TYPEFLE-OUTPUT
*     THIS DEFINES 'PRINTR' AS THE SYSTEM PRINTER AND SPECIFIES
*     THAT 120-CHARACTER RECORDS WILL BE PRINTED FROM 'OUT', THE FIRST
*     CHARACTER OF A 121-CHARACTER RECORD IS USED TO CONTROL SPACING
*     AND SKIPPING, A DS MUST BE USED TO RESERVE 121 CHAR. FOR 'OUT'
         DTFEN
*     THE USER'S PROGRAM BEGINS HERE WITH BALR, USING, ETC., LABELED BEGIN
         --
         --
         --
         --
*     THE USER'S PROGRAM ENDS HERE, WITH THE FOLLOWING STATEMENTS
      REQUIRED TO TERMINATE THE PROGRAM
EOF      CLOSE    CARDRDR,PRINTR
         EOJ
         END      BEGIN
```

Fig. 2-6 File definition statement for printer and card reader. User program statements are inserted where indicated.

Character	Function
X'09'	Single space after printing
X'11'	Double space after printing
X'89'	Skip to top of page after printing

Fig. 2-7 Printer Control Characters

The Statements shown in Fig. 2-6 allow a user's program to be sandwiched between the necessary file definition statements. The user's program is inserted in the space indicated by dashes. The complete deck of cards, file definition statements plus user statements, can then be assembled and executed. Control cards must be added to the file definition statements to guide the Operating System. In Chap. 12, these are discussed. However, the contents of the control cards depend on the configuration of equipment being used. We suggest the details be discussed with your local installation management prior to submitting programs.
First, some remarks are in order about the closing statements in Fig. 2-6. Consider the statement labeled EOF. This is referenced in the file definition statement for CARDRDR as an end of file address (EOFADDR). An end of file condition occurs when all the data in the input file has been read. It can be indicated in one of two ways. First, the card reader's hopper can run out of cards. When this happens, the Operating System will signal the computer operator that an error condition exists and will wait until additional cards are placed in the hopper before continuing. This is recognized as an error condition because a trailer card, punched /* in cc 1-2, is required to terminate a file. When the Operating System encounters this signal, it knows that it has reached the end of a given file and usually the end of that job. In this way, the system will not attempt to read the Job Control cards of the next job as data cards for the previous job. The second, and correct way to indicate an end of file condition is, therefore, a trailer card which follows the last data card. When the GET statement reads a trailer card, a branch will be taken to EOF which "closes" or deactivates the reader and printer. The EOJ statement causes the job to be terminated and control transferred to the Operating System which will then process the next job on the input tape or card reader.

With these preliminaries behind us, we will next consider a complete, if rather simple program.

Worked Example

2-9 A card file has two numbers, A and B, punched in cc 1-5 and cc 6-10 respectively. For each card, compute their sum and print it together with the two numbers. The output should have column headings A, B, and A+B. The code is

```
START     BALR   11,0                  THESE STATEMENTS WILL BE
          USING  *,11                  COVERED IN CHAP. 3
          OPEN   CARDRDR,PRINTR
          MVC    OUT+1(1),BLANK
          MVC    OUT+2(119),OUT+1      CLEAR PRINT AREA
          MVC    OUT(1),CTOP
          PUT    PRINTR                ADVANCE TO TOP OF PAGE
          MVC    OUT+3(1),CA           SET
          MVC    OUT+12(1),CB                 UP
          MVC    OUT+22(3),CAB                       HEADINGS
          MVC    OUT(1),DOUBLESP       PRINT CONTROL FOR DOUBLE SPACE
          PUT    PRINTR                PRINT HEADINGS AT TOP OF PAGE
READ      GET    CARDRDR               READ A CARD INTO 'IN'
          PACK   A,IN(5)               PACK CC 1-5 = A
          PACK   B,IN+5(5)             PACK CC 6-10 = B
          AP     A,B                   A CONTAINS A+B
          UNPK   SUM,A                 SUM = A+B ZONED
          MVC    OUT+1(1),BLANK
          MVC    OUT+2(119),OUT+1      CLEAR PRINT AREA
          MVC    OUT+1(5),IN           MOVE A TO PRINT AREA
          MVC    OUT+10(5),IN+5        MOVE B TO PRINT AREA
          MVC    OUT+20(6),SUM         MOVE A+B TO PRINT AREA
          PUT    PRINTR                PRINT A LINE FROM 'OUR'
          B      READ                  GET NEXT CARD
BLANK     DC     C' '
DOUBLESP  DC     X'11'                 2 SPACES AFTER PRINTING
A         DS     PL4                   4 BYTES FOR A+B
B         DS     PL3
SUM       DS     ZL6
IN        DS     80C                   INPUT CARD
OUT       DS     121C                  OUTPUT PRINT LINE
CA        DC     C'A'
CB        DC     C'B'
CAB       DC     C'A+B'
CTOP      DC     X'89'
```

The first two statements, BALR and USING will be explained in Sec. 3-4. They must be included as the first two statements in every program. Let us now consider the layout of the print line and its headings. It is good practice to leave some space between columns of data so the A-column will be printed in print positions (pp) 1-5, the B column in pp 10-14 and the sum column in pp 20-25. (Since A and B are each five digits, we have allowed six digits for their sum.) The column headings will be centered if the letter A is printed in pp 3, B in pp 12 and A+B in pp 22-24. Our first step, after opening CARDRDR and PRINTR, is to clear the print area, OUT. This is done by propagating blanks (not zeros!) throughout the area. Since the printer controls operate after printing a line, it is necessary to "print" a line of blanks to advance the paper to the top of a page. This is done by the first PUT statement. Next, the column headings are put into place, the control character which causes the printer to double space after printing is inserted into the first position of OUT and finally, the line is printed. The next statement, at READ, reads a card into IN. The data is packed, summed, and set up to be printed. Note that the print area is cleared prior to data being moved in. Strictly speaking, this would not have to be done since the first line of data would overlay what was in the print area from the previous PUT, the column headings, and successive lines would have their data in the same positions. However, this may not always be the case and so, it is good practice to clear the print area between line format changes. When these statements are inserted at the proper place into the statement in Fig. 2-6, the assembly program is complete. To execute this program, the assembly statement cards and the data cards should be combined with the proper job control cards as indicated in Fig. 2-8. We do add one note of caution,or perhaps qualification. The statements in Figs. 2-6 and 2-8 are intended for a fairly standard 360 hardware and programming systems configuration. However, it is possible that differences will exist at a given installation and slight modifications may be required. We suggest the reader consult on these points with operating personnel at this installation.

When the program is run, the output is surprising. Input data such as 11111 and 22222 when summed will be printed as 3333C! The reason for this is not hard tc find. The original input will have 1111 as zone bits which is interpreted as a + sign. However, the arithmetic sum of the input quantities will have the preferred + sign, 1100, as the zone bits of the low order digit. From Fig. 2-1, the zoned digit 1100 0011, which we understand as a +3 in the context of this problem will be printed as a C. There is a way around this difficulty --it will be discussed in Sec. 2-7. When the last input card is read, the trailer card punched /* in cc 1-2, the GET routine will transfer to EOF which will terminate the job.

The requirement for control checks provides an interesting extension to this problem. To insure against the undetected loss of data cards, very often the number of input items will be counted after they have been keypunched. (This can be done easily using a mechanical sorter equipped with a card counter.) The card count and perhaps the total of some key item from each card, will be punched into a card which follow the data file. This card may be identified with a punch which is unique for that data file, say a - in cc 80. The program is then modified to perform a card count and key item total while it is carrying out its primary operation. At the conclusion, the program count and total are compared with the data on the last card. If a mismatch occurs, the data file can be audited to find the missing or extra items. This will be left to the reader in Prob. 2-21.

2.7 Editing

Most output data requires editing which may involve converting a result such as 00123456 into 1,234.56 or $1,234.56 or $**1,234.56, to mention a few possibilities. This can be done by setting up what is termed an edit mask in that part of the output area from which the edited result will be printed. The edit mask is made up of a variety of characters which define the format of the edited result; the EDIT command matches the data or source field with the edit mask. The edited result replaces the edit mask and printing can then take place. Since the edit mask is destroyed by this operation, a new copy will have to be moved to the output area for the next print line. The format of the edit instruction is

```
ED      MASK, DATA
```

The length indicator pertains to the first operand which gives the address of the edit mask; the second operand is the address of the data to be edited. The data field, which must be in packed format, is processed one character at a time from left to right. The mask characters and the data digits determine if a data digit will be stored in the print area or not. The condition code is also set to plus, minus or zero according to the sign of the source field.

The first character in the edit mask is termed the fill character, which is always left unchanged in the edit mask after the data field has been edited. There are a number of possibilities for the successive mask characters. We will start by considering the simplest one, the digit selector. The basic idea here is that for each digit selector character in the edit mask, one digit (four bits) will be taken from the data field. If the digit is non-zero, it will be expanded to zoned format and stored in the edit mask replacing the digit select character. If the source digit is zero, and no non-zero character has been encountered so far, the fill character will be stored. If non-zero source digits precede a zero, the zero will be stored. This feature permits suppression of high order zeros by effectively propagating the fill character throughout the edit mask area until a non-zero or significant digit is found. Using B for the blank mask character, * for the asterisk character, and D for the digit select character, consider the following examples. The addresses of each field are shown in parentheses. In the interests of readibility of the illustrations, b will be used to indicate a blank print position.

Data Field (11000)	Edit Mask (12000)	Edited Result (12000)
00 23 45 6+	*DDDDDDD	***23456
01 23 05 6-	BDDDDDDD	bb123056
00 00 00 0+	BDDDDDDD	bbbbbbbb

Note that the edit masks are at least one character longer than the maximum number of digits in the data field and that data digits start one byte to the right of the first character in the edit mask. Note also that the signs are not printed. In general, for positive data, it is not necessary to print a + sign. However, negative data is usually required to be identified, often by printing either a - sign or the symbol, CR for credit. The edit instruction allows for characters to be included to the right of the data digits. The algebraic sign of the data field is examined, if it is +, the fill character replaces whatever characters are to the right of the data digits. If the sign is -, the characters to the right are printed. Commas and decimal points may also be printed by placing them in the desired places in the edit mask. As examples, consider the following.

Data Field (11000)	Edit Mask (12000)	Edited Result (12000)
00 12 34 5+	BDD, DDD. DD	bbbb123. 45
00 12 34 5+	*DD, DDD. DDBCR	****123. 45***
12 34 56 7+	BDD, DDD. DDB -	b12, 345. 67bb
01 23 45 6-	BDD, DDD. DDBCR	bb1, 234. 56bCR
00 00 00 0+	BDD, DDD. DD*	bbbbbbbbbb
00 00 00 1-	BDD, DDD. DD*	bbbbbbbbb1*

The output in the last two examples above is often not satisfactory. Better practice would be to print them as . 00 and . 01*, respectively. To accomplish this, the significance start character which we will indicate as (, is used. When it is encountered in an edit mask and a significant digit has been encountered previously, the source field digit replaces it. When only zeros have occurred previously, and a zero digit also occurs in the significance start position, the fill character is used but on subsequent data digits, the instruction behaves as though a significant digit had been found in the (position. As examples:

Data Field (110000)	Edit Mask (12000)	Edited Result (12000)
00 00 1+	BDD(. DD*	. 01
01 23 4-	BDD(. DD*	12. 34*
00 00 0+	BDD(. DD*	. 00

In the event that we wanted to blank out entirely zero fields, a BZ instruction could be placed after the edit instruction to move blanks into the last three positions of the edited field. One other edit character is of interest, the field separation character which we will indicate as). As its name indicates, it is used to separate fields when multiple fields are to be edited by a single edit instruction. When this is done, the setting of the condition code refers only to the sign of the last field. The field separator is replaced by the fill character and causes each field to be edited separately. As an example:

Data Fields (10000)	Edit Field (11000)	Edited Result (11000)
01 23 4+ 56 78 9-	BDD(.DDB*)DD(.DDB*	bb12.34bbb567.89b*

As a final note on editing, the reader is cautioned that the characters we have used to identify edit masks -- BD,.()CR*- -- are a symbolic notation, only. In an assembly program, the actual bit configurations must be given in a DC statement with an X, for hexadecimal, operand. Hexadecimal constants will be explained in Chap. 3. At this point, Table 2-2 can be used to develop the edit masks in recipie fashion. As an example, the edit mask for BDD,DD(.DDB*)DDD is

```
MASK      DC      X'4020206B2020214B2020405C22202020'
```

B	40	)	22
D	20	C	C3
.	4B	R	D9
,	6B	*	5C
(	21	--	60

Table 2-2 Hexadecimal equivalents of edit characters

Worked Example

2-10 Two columns of data are to be printed, single spaced, as

bbbbbxx.x - b$bxx,xxx.xxbCR

Where - and CR are to be printed only when the corresponding fields are negative. If the first field is zero, blanks are to be printed. The output area is labeled OUT the two data fields are A and B, respectively. The code follows.

```
START   BALR   11,0
        USING  *,11
        OPEN   PRINTER                          READY PRINTER
        MVC    OUT(1), BLANK                    CLEAR
        MVC    OUT+1(120), OUT                         'OUT' AREA
        MVI    OUT,X'09'                        SET UP SPACE CONTROL
        MVI    OUT+12,C'$'                      SET UP $
LOOP    --
        --
        --
        MVC    OUT+5, MASKA                     MOVE A-MASK
        ED     MASKA, A                         EDIT A=FIELD
        BNZ    NEXT
        MVC    OUT+8(2), BLANK                  A=O, SET TO BLANKS
NEXT    MVC    OUT+13, MASKB                    MOVE B-MASK
        ED     MASKB, B                         EDIT B-FIELD
        PUT    PRINTER                          PRINT LINE
        B      LOOP                             DO NEXT LINE
        --
        --
        --
MASKA   DC     X'4020214B2060'                  BD(. D-
MASKB   DC     X'4020206B2020214B202040C3D9'    BDD, DD(. XXBCR
BLANK   DC     C' '
A       DC     PL2                              XX. X
B       DC     PL4                              XXXXX. XX
OUT     DS     CL121
        END    START
```

The remainder of this section may be skimmed over in a first reading -- its full understanding requiress Chap. 3. The instruction we will next discuss facilitates the insertion of floating currency symbols as a check protection devise. As an illustration, this would allow the printing of results such as

```
$1, 015, 617. 35
           $2. 67
       $1, 117. 40
          $ . 01
```

In addition, if it is desired to identify negative results by placing a minus sign immediately before the number, this instruction -- the Edit and Mark -- can also be used. Its format is

```
EDMK    MASK, DATA
```

Its operation is identical to Edit except that the address of the first significant result digit will be stored in GPR1. A BCT 1, 0 can be used to subtract one from GPR1. The result is the address where the $ or - will be stored. When multiple fields are edited by a single EDMK, GPR1 will contain only the address of the first significant digit in the last field. The address will not be stored if significance is forced by the significance start character. Therefore, the address of the character following the significance start should be stored in GPR1 prior to issuing the EDMK. As an example of floating $ insertion, AMT contains a seven digit number which is to be edited in the form XX, XXX. XX with $ immediately to the left of the first significant digit. The number is to be printed starting in pp 45. The coding is

```
         LA      1,OUT+51                     LOAD ( ADDRESS PLUS 1
         MVC     OUT+44(10),MSK               MOVE EDIT MASK
         EDMK    OUT+44(10),AMT
         BCT     1,0                          SUBTRACT 1 FROM GPR1
         MVI     0(1),C'$'                    MOVE $ TO PRINT ADDRESS
         PUT     PRINTER                      PRINT LINE
         --
         --
MSK      DC      X'4020206B202021682020'      BDD,DD(.DD
AMT      DS      PL4
OUT      DS      CL121
         --
         --
```

Since AMT is to be printed starting in pp 45, the edit mask must start in pp 44. Since (is the seventh character in MSK, its address is OUT+50. The first instruction will put the address of the <u>next</u> character in GPR1. The MVI specifies a base of 1 and a displacement of 0. Since the EDMK stores the address of the first significant digit and the BCT subtracts one, GPR1 then contains the address where $ is to be put and the MVI accomplishes that.

<u>Answers to Exercises</u>

<u>2-1</u> In EBCDIC, they are equal, namely 1101.

<u>2-2</u> PL3'-101.5'; PL2'167'; PL4'2.14159'; PL1'-3';PL1'2'. The lengths are shown as the minimum field size which will accommodate the constant. It is not necessary to show the explicit lengths, however, since the assembler will determine them at assembly time. We will do this with the character fields and indicate the implied size in parenthesis C'X ='(3); C'''NET'''(5); C 'NET && GROSS' (11).

2-3 This coding will propagate the first byte of AREA, namely zero, through the remaining 255 bytes. Normally this kind of overlap would produce unwanted results. In this case however, it is a useful technique to clear an area of storage.

2-4

```
a)       A = Z1   Z2   Z3   Z4   Z5   +6
      PACK  B(4), A         01   23   45   6+
      PACK  B(2), A         45   6+
      PACK  B(6), A         00   00   01   23   45   6+
      PACK  A(3), A         5+   45   6+   Z4   Z5   +6
      PACK  A+2(3), A       Z1   Z2   23   4+   6+   +6

b)       A = 00   01   23   45   6+
      UNPK  B(8), A         Z0   Z0   Z1   Z2   Z3   Z4   Z5   +6
      UNPK  B(3), A         Z4   Z5   +6
      UNPK  A(4), A+3(2)    Z0   Z+   Z6   +6   6+
      UNPK  A, A+2(3)       ZZ   Z4   Z4   Z5   +6
```

2-5

a) The sequence in which the instructions will be executed is

```
      CP    B, C         B GREATER THAN C
      BNE   ADD          BRANCH TAKEN
      AP    C, A         C = + 15
      BP    START        CONDITION CODE + BRANCH TAKEN
      CP    B, C         B = C
      BNE   ADD          BRANCH NOT TAKEN
      BE    OUT          CONDITION CODE SET TO =, BRANCH TAKEN
*     NEXT INSTRUCTION TAKEN FROM OUT
*     FINAL VALUES           A=20, B=15, C=15, D= -20
```

b) Note that in an algebraic comparison -5 is greater than -20. The sequence in which the instructions will be executed is

```
      CP    C, D         C GREATER THAN D
      BH    ADD          BRANCH TAKEN
      AP    D, B         D = -5
      CP    C, D         C = D
      BH    ADD          BR. NOT TAKEN
      SP    C, B         C = -20
      B     OUT          NEXT INSTRUCTION EXECUTED AT OUT
*     FINAL VALUES          A=20, B=15, C= -20, D= -5
```

Problems

2-1 What is the difference in storage requirement between the constants Z'-216.26' and Z'21626'; between the constants Z'215' and P'215'; Z'5'and P'5'?

2-2 How many bytes will the following constants require?

```
5P'-1'          C'ABC'          6CL4'A'
5PL3'215'       CL6'ABC'        ZL6'5'
```

What will the storage layout be for each constant?

2-3 What are the storage requirements in bytes for the following areas

```
DS  PL50        DS  CL50        DS  ZL50
```

What determines the choice between each form?

2-4 What are the length attributes of the following symbols:

```
A     DS      5PL3
B     DS      0ZL15
C     DS      256CL1
D     EQU     A+611
```

2-5 Referring to Prob. 2-4, if A is located at 10000, what are the addresses of B, C, and D?

2-6 With A containing ABCDEFG and B containing VWXYZ show the contents of the destination field after each of the following instructions are executed. Consider each instruction independently.

```
MVC   A(5),B        MVC  A+2(3), B+1

MVC   A(2),B        MVC  B,A
```

2-7 Show the contents of the destination fields after the following instruction sequence is executed

```
          MVC     A,B
          MVC     B+4(4), B+3
          MVC     A,B+3
          --
          --
A         DC      CL6'ABCDEF'
B         DC      CL4'KLMN'
C         DC      CL5'*****'
```

2-8 If A contains 01 23 45 67 8+, show the effect of the following instructions which should be considered independently

```
MVO     A,A(4)              MVO     A,A(3)

MVO     A+1,A+3(2)          MVO     A+1, A+2(2)
```

2-9 Complete the instructions, begun in Ex. 2-1, to shift left an odd number of digits.

2-10 Referring to Ex. 2-2, write an instruction to move a four-byte field into DATE; write an instruction to move the low order character in work to a one-byte field, CHAR.

2-11 Which of the move instructions use the explicit or implicit length of the first operand, to determine the length of the field to be operated on. The remaining move instructions allow an explicit length to be given for the second operand as well as the first; name them also.

2-12 Write the program steps to add the packed decimal numbers A=XXX. XXXXS, B=XX. XXS and C=X. XXXS.

2-13

a) Write the program steps to compute average monthly net sales from (Yearly Gross Sales - Total Cancellations)/12. Gross sales has the form XXXXXXXXX. XX and cancellations, XXXXXXX. XX.

b) Round the average to the nearest 1000 and show the result in the form XXXXX000.

2-14 Write the program steps to compute the following percentage, rounded to three decimal places

$$\frac{\text{1967 INDEX - 1966 INDEX}}{\text{1966 INDEX}} \times 100\%$$

where the indices have the form XX. XXXX. Note that the differences between indices may be negative.

2-15 Consider the following program steps. Can you find any instructions which will cause interrupts when executed? Why?

```
ZAP     TOTAL, SUBT(5)
SP      TOTAL, CANC
BM      EDIT
```

TOTAL, SUBT and CANC are each defined by DS PL6 and contain respectively 972016755S, 201653216S, and 31542S.

2-16 Verify the rule given in Ex. 2-4 for handling division problems with varying size divisors. Hint: Remember that the maximum divisor size must be used to reserve space for the remainder but the minimum divisor must be used to find the maximum dividend. From this point, essentially the same approach can be taken as was done for the simple "Add two" rule.

2-17 Given three six-byte packed decimal numbers, A, B, and C, write the program steps to find the algebraically largest and smallest numbers of the three and store them in BIG and SMALL, respectively.

2-18 For the same three numbers as in Prob. 2-17, sort them and put the largest in location A, the next largest in location B, and the smallest in location C.

2-19 For the same three numbers as in Prob. 2-17, test for equality among the three and branch to a routine labeled EQUAL3 if all three are equal, EQUAL2 if two are equal and EQUAL0 if none are equal.

2-20 Modify the code of Example 2-9 so that the print area is cleared only once after the column headings are printed.

2-21 Refer to the extension mentioned in the last paragraph of Example 2-9. Modify that program to count the data cards, total all the A-items and verify these against the control amounts punched in cc 1-10 and 10-30, respectively, of a card with a - punch in cc 80 which follows the last data card.

2-22 Referring to Example 2-8, what will be the effect on the program if the definition of ZERO is changed to DC PL1'0'?

2-23 A deck of cards contains a 10-digit number in cc 1-10. The last card contains only an 11-punch in cc 80. Write a program to add these 10-digit numbers and print the total, together with a card count, when the last card has been read. This can be done by using the ZAP to put the first number into the total field and adding the following numbers to the total field. However, the logic is more straightforward if the total field is cleared first, or initialized to zero by ZAP TOTAL,TOTAL or moving a field of zeros into it. If this is done, each number can then be added to TOTAL without having separate logic for the first card. This approach to initialization is extremely important since the programmer cannot assume that an area of storage reserved by a DS will contain zeros at execute time.

Write the program and test it using your own sample test data. Allow for a 20-digit total.

2-24 The end-of-job cards we have been using, with an 11-punch in cc 80, are really not necessary to initiate end-of-job operations. We could place card count information, and so forth, at the head of an input deck and use the trailer card, punched /* in cc 1-2, to initiate the necessary action. This could be done at location EOF (See Fig. 2-6) by placing the appropriate instructions before the CLOSE. Incorporate this approach into your program for Prob. 2-23 and test it.

2-25 The input to a payroll program is a card with the following layout

Department Number	cc 1-3	
Employee Number	cc 4-10	
Employee Last Name	cc 11-30	
Employee Initials	cc 31-32	
Hourly Rate	cc 33-35	X. XX
Hours Worked	cc 36-39	XX. XX

The input deck will be sorted by employee name within Department Number. For each employee card, print the following information, single spaced:

Employee Number	pp 6-12	
Employee Name	pp 14-33	
Employee Initials	pp 35-36	
Total Pay	pp 40-44	XXX. XX

Total pay is to be computed to the nearest cent by paying time-and-a-half (1 1/2 times hourly rate) for all time over 40 hours but under 70 hours and by paying double time for all hours over 70. Each department number should be printed once, at the top of a page in pp 2-4. When all cards in a department have been processed print the total hours, total pay, and average pay per hour (total pay ÷ total hours) for that department in the format.

```
DEPARTMENT  XXX
TOTAL HOURS        =     XX,XXX
TOTAL PAY          =  $  XXX,XXX.XX
AVERAGE PAY/HOUR=     $  X.XXX
```

(The fact that all cards in a department have been processed can be detected when the Department Number of the next card is <u>high</u> when compared with the previous one. As a limit, define PREDPNO as the previous department number and start it off with a negative value. In this way, the first card read will force a high compare and the department number can then be printed. However, on the first Department number break, the TOTAL HOURS, PAY, etc. should not be printed. As an additional note, the programmer must initialize to zero the areas where department totals will be developed by setting them to zero.) The last card contains a card count in cc 1-10, and an 11-punch in cc 80. When it is read, verify the card count with the number of cards processed and also print out at the top of a new page the following information:

```
PLANT TOTALS

HOURS              =     XXX,XXX
PAY                =  $  XX,XXX,XXX.XX
AVERAGE PAY/HOUR=     $  X.XXX
```

These totals can be developed by adding the department totals to plant totals at every Department Number change. The average pay/hour is still computed by dividing total plant pay by total plant hours worked.

Write a flow chart to do the operations outlined above, develop a program, devise sample test data and run your program on the test data.

<u>2-26</u> Redo the "Indian Problem", Example 2-7, including complete input-output statements and allow for a number of cases to be run with the number of years, principal and interest rates as input items with each case on a single card.

<u>2-27</u> What is the effect on the condition code setting of the following instruction:

```
CLC = C'AB%',=C'AB*'
```

Chapter 3

BINARY PROGRAMMING

3.1 Number Systems

In Chapter 2, we discussed the 360 instructions and data formats for decimal arithmetic operations -- that is, for operations on numbers in the decimal system. We are so familiar with these numbers that we almost never think to apply the qualification decimal to them. It is almost as if decimal numbers were the numbers -- the ten fingers of man have indeed left a very deep imprint on his method of counting. But in fact, other number systems have come into use in the past. The Mayans and Aztecs used a numbering system based on 20 while some primitive tribes in Australia use a system based on the number two, the binary system. The system of numbers used by the ancient Romans is strongly influenced by the number five, while much of the English monetary system and English weights and measures depend on the number 12 and its multiples. Well established conventions, and to a lesser degree, convenience, are the main influences that determine a society's enumeration system. Now it is true that computers in their direct communication with the outside world must deal in symbols and numbers that are directly intelligible to people and so the importance of convention in determining the choice of a computer numbering system cannot be overlooked. However, computers also do a great many operations on numbers and symbols which do not relate immediately to the job of communicating with their users. These are the so called "machinehood" operations: Rearrangement of data into formats more suitable for computer processing, moving data from place to place in storage, calculations with storage addresses, testing the status of various registers and condition indicators, and so forth. For this class of operations, efficiency is the sole criterion determining which number base is to be used. Considerations of efficiency also enter when number bases are being considered for arithmetic operations. As we shall see, it happens that computer arithmetic on binary numbers is more efficient than on decimal numbers. This efficiency is usually reflected in speed -- the computer does binary arithmetic faster than decimal arithmetic for a given cost of circuitry. However, if a computer is to operate on binary numbers, since the computer's input and output are decimal quantities, translations between the two systems are necessary. The time required to do this must be balanced against the greater speed of binary arithmetic.

As we discussed in Chap. 1, computer storage is made up of small washer-like cores which may be magnetized in two directions, clockwise or counterclockwise. As a result, the most natural coding scheme is binary, that is, one which is based on two symbols, one symbol for clockwise magnetized cores and another symbol for counterclockwise magnetized cores.

The restriction to two basic symbols is not as limiting as it may seem at first sight. The status of machine indicators which are either "on" or "off" and logic tests which essentially reduce to a "yes" or "no" result fit naturally into such a scheme and it also turns out that for a given number of cores, significantly more digits can be represented by the binary number system than by the (binary coded) decimal numbers discussed in Chapter 2. Because of the importance in computing of number systems, we will develop the more important concepts here.

The most convenient starting point is the system most familiar to us -- the decimal system. Consider what is meant by a number such as 2132. This notation represents, of course, the number two thousand one hundred and thirty two. The basic idea here is that the position of a digit indicates its value. For example, the first digit 2 in the number has the value 2000, the last one the value two. We can indicate this property by writing the number

as $$2000 + 100 + 30 + 2$$

or, $$2 \times 10^3 + 1 \times 10^2 + 3 \times 10^1 + 2 \times 10^0$$

since $10^1 = 10$ and $10^0 = 1$. This is the mathematical representation of the positional notation concept. In this system the base, or radix, is ten and there are ten digits, 0, 1, 2, 3, 4, 5, 6, 7, 8, and 9. In the binary system, the base is two and there are two digits, 0 and 1. The concept of number representation is the same. That is, the binary number 1101 represents

$$1 \times 2^3 + 1 \times 2^2 + 0 \times 2 + 1$$

Since, $2^3 = 8$ and $2^2 = 4$, 1101 is the binary equivalent of

$$1 \times 8 + 1 \times 4 + 0 + 1$$

or 13. A table of binary integers is built up in much the same fashion as a table of decimal integers. Since the techniques of decimal counting are so ingrained in our thinking that we count almost without conscious control, we will pause to state these techniques. Consider how we would develop a table of integers, that is, whole numbers. We start with 0 and build up the table by adding 1 to each entry. It will be convenient to depict the process of building the table in positional notation. For simplicity, we will assume that the table will not go beyond 9999. The first entry, zero, is then

$$0 \quad 0 \times 10^3 + 0 \times 10^2 + 0 \times 10 + 0$$

As we proceed by adding 1's, we note that at every multiple of 10, that is, at every multiple of the base, an "overflow" occurs from the units position into the tens position. That is, as we proceed from 9 to 10, we have

9	$0 \times 10^3 + 0 \times 10^2 + 0 \times 10 + 9$	$+ 1$
10	$0 \times 10^3 + 0 \times 10^2 + 1 \times 10 + 0$	

where the overflow is indicated by the arrow. As another illustration, as we go from 10 to 20, we have

19	$0 \times 10^3 + 0 \times 10^2 + 1 \times 10 + 9$	$+ 1$
20	$0 \times 10^3 + 0 \times 10^2 + 2 \times 10 + 0$	

the overflow from the units to the tens position can propagate other overflows. For example, from 99 to 100, we have

99	$0 \times 10^3 + 0 \times 10^2 + 9 \times 10 + 9$	$+ 1$
100	$0 \times 10^3 + 1 \times 10^2 + 0 \times 10 + 0$	

To summarize, for every multiple of the base which is counted in the units position, there is an overflow of one into the next higher order position (tens position for decimal numbers) and for every multiple of the base counted in the tens position there is an overflow of one into the next higher order position (hundreds position for decimal numbers), and so forth. To apply these rules to the binary number system, we note that corresponding to the units, tens, hundreds, thousands, etc., of the decimal system, we have for binary numbers, the units, twos, fours, eights, and so on. As we had overflows in going from 9 to 10, for instance, in decimal system, we will have an overflow in counting from 1 to 2. That is, $1 + 1 = 10$ (= 2 in binary). Building up a table of binary numbers is a straightforward process since there are only two rules of addition in that system: $1 + 0 = 1$ and $1 + 1 = 10$ (of course, $0 + 0 = 0$). The first nine binary numbers are then

Decimal Number	Binary Equivalent
0	0
1	1
2	10
3	11
4	100
5	101
6	110
7	111
8	1000

Exercise 3 - 1 Write the next eight entries in the table above. Answers are given at the end of this chapter.

An important question with any number system is how large a number can be represented in a given number of positions. With decimal numbers, in two positions, the largest number representable is 99, for three positions, the largest is 999. These numbers are respectively, $10^2 - 1$ and $10^3 - 1$ so that for an arbitrary number of positions, n, the largest number which can be represented is $10^n - 1$. For binary numbers, this is $2^n - 1$. As an example, the largest binary number which can be represented by 3 binary digits is $2^3 - 1$ or 7 which in binary is 111. Since the term binary digit occurs so frequently, we will use from now on a shorter, equivalent term, bit. In 12 bits, the largest binary number is $2^{12} - 1$ or 4095. Or, if we count zero, the total of all numbers given by 12 bits is, 4096 which explains why the displacement field (12 bits) in a 360 instruction (See Sec. 3.4) can be used to address a span of 4096 bytes of storage. Let us now compare the storage efficiencies of the binary coded decimal numbers introduced in Chapter 1. Recall that each decimal digit required four bits in the binary coded decimal format. This allows any digit from 0 to 9 to be represented. Note that if we were dealing with binary numbers, in four bits any number from 0 to 15 could be represented so that binary numbers are fundamentally more efficiently represented in magnetic core storage than decimal numbers. In fact, for 12 bits, the largest decimal number is three digits long (12/4 = 3) whereas the largest binary number is 4095. In binary, using 12 bits, we have 3 digits and a part of the fourth digit -- we can think of this as approximately 3.4 digits in a sense. As a result for 12 bits binary numbers are approximately 13% more efficient than decimal numbers (. 4/3 x 100% = 13%). For larger bit fields (20 bits and up), the binary representation will give approximately 20% more digits than the BCD notation.

In addition to binary and decimal numbers, one other number system is important for the 360 -- the hexadecimal or base 16 system. There are two reasons why the hexadecimal system is important. The 360 allows calculations on so called "floating point" numbers which are important in engineering and scientific calculations. Floating point numbers are represented in hexadecimal notation by the 360 -- we will study them in detail in Chapter 6. The second reason hexadecimal notation is important is because it gives a convenient shorthand for binary notation. For instance, the fields in 360 instructions are either four bits long, or multiples of four bits in length. In the hexadecimal notation since the base is 16, there are 16 digits. That is, one digit (symbol) for every possible combination of 1's and 0's in each four bits of storage. As a result, hexadecimal symbols can represent the contents of the 360's storage in an assembly language listing more compactly than its binary equivalent. The hexadecimal digits -- or hex digits, for short -- with their decimal and binary equivalents are shown in Table 3 - 1 below. Note that, since 16 symbols are needed -- one for each hex digit, six additional symbols are needed to supplement the digits 0 through 9. Any six symbols could be used such as @, $, *, !, &, ?, for example. However, the first six letters of the alphabet are used because their sequence is easier to remember than six arbitrary symbols. As an illustration of using hex notation to represent binary, consider the instruction BALR 13, 12. Since the

Decimal	Hexadecimal	Binary
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
10	A	1010
11	B	1011
12	C	1100
13	D	1101
14	E	1110
15	F	1111

Table 3 - 1 Hexadecimal numbers with decimal and binary equivalents

operation code for BALR is 00000101 and 13 and 12 are, in binary, respectively, 1101 and 1100. The instruction would appear in storage as 0000010111011100. If this instruction appeared in an assembly listing, the string of 1's and 0's above would be tedious to read and so the hex equivalent is shown. This is 05D~~E~~C. As a final point on number systems, we will use a subscript notation to distinguish between the various number systems; Base ten numbers will be shown unsubscripted unless the base is not obvious from context. As illustrations,

$$101_{10} = 1 \times 10^2 + 0 \times 10 + 1 \quad (= 101)$$

$$101_2 = 1 \times 2^2 + 0 \times 2 + 1 \quad (= 5)$$

$$101_{16} = 1 \times 16^2 + 0 \times 16 + 1 \quad (= 257)$$

3. 2 Number Conversion.

While almost all of the conversions between the various number systems which may be required in a program can be performed by special circuitry in the computer, or else by programmed routines supplied by the computer manufacturer, it will aid the readers understanding of the limitations of the various number systems if he has a grasp of the conversion methods which will be presented in this section.

In our discussion of the positional notation concept, we have given the essence of converting from a given number base to the decimal base. As an example, consider the binary number 110101. That is,

$$1 \times 2^5 + 1 \times 2^4 + 0 \times 2^3 + 1 \times 2^2 + 0 \times 2 + 1$$

or

$$1 \times 32 + 1 \times 16 + 0 + 1 \times 4 + 0 + 1$$

which is the decimal number 53. As another example, consider the hex number A5F which is

$$A \times 16^2 + 5 \times 16 + F$$

or, since $A = 10$, $F = 15$, and $16^2 = 256$, we have

$$10 \times 256 + 5 \times 16 + 15$$

which is the decimal number 2655. On the strength of these examples, we can concoct a rule for converting from other bases to decimal:

<u>Express the binary or hex number in its positional notation equivalent. For hex numbers, replace the alphabetic symbols (A through F) by their decimal equivalents (10 through 15). Carrying out the indicated multiplications and additions gives the decimal result.</u>

We can distill this idea further into a simple computing recipe which will be shown by example below for the binary number 110101 and the hex number A5F.

```
   A5F                110101

   10 (=A)                1
  x 16                  x 2
  ----                  ---
  160                     2
 +  5(=5)               + 1
  ----                  ---
  165                     3
  x 16                  x 2
  ----                  ---
 2640                     6
 + 15(=F)               + 0
 -----                  ---
 2655  =  A5F             6
                        x 2
                        ---
                         12
                        + 1
                        ---
                         13
                        x 2
                        ---
                         26
                        + 0
                        ---
                         26
                        x 2
                        ---
                         52
                        + 1
                        ---
                         53 = 110101
```

Note how relatively "long winded" the process of converting from binary to decimal is. The operation would be a lot shorter, if we could first convert the binary number to a hex number. Fortunately, this can be done almost by inspection by grouping each four bits of the binary number, from right to left, and then replacing each group by its hex equivalent from Table 3-1:

1 1	0 1 0 1	binary
3	5	hex

The conversion is then

```
    3
 x 16
 ----
   48
 +  5
 ----
   53
```

Exercise 3-2 For practice in number conversion, calculate the decimal equivalents of the following hex numbers: ABC, 5E2F, 10101 and AOA; find the decimal equivalents of the following binary numbers: 111111, 10101 and 101010111100.

For the mathematically inclined reader, the justification for the conversion recipie given above lies in factoring the positional representation into a nested polynomial. As an example, A5F which is

$$A \times 16^2 + 5 \times 16 + F$$

can be represented as the nested polynomial

$$(A \times 16 + 5)\ 16 + F$$

from which the recipie above follows.

The reverse operation, of converting from decimal to hex or binary, is accomplished by successively dividing by 16 or 2 and using the remainder to develop the digits of the converted numbers. Again, rather than describe the process in words, we will use the examples cited above: converting 2655 into hex and 53 into binary. The remainder will be shown following the quotient by a slash mark (/).

```
      165 / 15 (= F)             26 / 1
  16/2655                    2 / 53
      16
     105                         13 / 0
      96                     2  /26
       95
       80                         6 / 1
       15                    2  / 13

       10 / 5 (=5)                3 / 0
  16 /165                    2  / 6

         / 10 (=A)                1 / 1
  16/10                      2  / 3

                                   / 1
                             2  / 1
```

Or $2655_{10} = A5F_{16}$ and $53_{10} = 110101_2$. Again, the conversion from decimal to binary takes longer than a decimal to hex conversion. As a result, the usual practice is to convert from decimal to hex and then go to binary by using Table 3-1. As an example

```
       3 / 5
  16 / 53

         / 3
  16 / 3
```

Or, $53_{10} = 35_{16} = 0011\ 0101_2$

Exercise 3-3 As a check on your answers to Exercise 3 - 2, convert your results back to binary or hex as required using the successive division method above.

So far, we have only considered conversion of whole numbers. The process of converting numbers with fractional parts is somewhat more involved. In the interest of brevity, we will illustrate the technique for decimal/binary conversions. Decimal/hex conversions are similar.

Consider a decimal fraction such as 123.456. In positional notation this represents

$$100 + 20 + 3 + .4 + .05 + .006$$

or

$$1 \times 10^2 + 2 \times 10 + 3 + 4/10 + 5/10^2 + 6/10^3.$$

Therefore, the binary number 110.111 represents in positional notation

$$1 \times 2^2 + 1 \times 2 + 0 + 1/2 + 1/2^2 + 1/2^3$$

Or, since *

$$1/2 = .5,\quad 1/2^2 = 1/4 = .25 \text{ and } 1/2^3 = 1/8 = .125,$$

we have

$$110.111_2 = 4 + 2 + 0 + .5 + .25 + .125 = 6.875_{10}.$$

To calculate the binary equivalent of a decimal number such as 37.613, we first determine the binary equivalent of 37. Using the successive division technique, this is 100101. To convert the fractional part, we proceed as follows:

$$\begin{array}{lr}
 & .613 \\
 & \underline{\times\ 2} \\
1 & .226 \\
 & \underline{\times\ 2} \\
0 & .452 \\
 & \underline{\times\ 2} \\
0 & .904 \\
 & \underline{\times\ 2} \\
1 & .808 \\
 & \underline{\times\ 2} \\
1 & .616 \\
 & \underline{\times\ 2} \\
1 & .232 \\
\text{etc.} &
\end{array}$$

* The Appendix contains a table of powers of two and their decimal equivalents.

So 37.613_{10} = 100101. 100111 +, the plus sign indicative that the binary fraction is non terminating. That is, we could continue adding 1's and 0's indefinitely and still not be able to represent . 613 exactly as a binary fraction. If we terminate the binary fraction at six places and then reconvert to decimal, we have

$$\begin{aligned} .100111 &= 1/2 + 1/2^4 + 1/2^5 + 1/2^6 \\ &= .5 + .0625 + .03125 + .015625 \\ &= .609375 \end{aligned}$$

Carrying the result to 10 binary places give a fraction of . 10011100111 = . 61279296875. This problem of inexact representation between decimal and binary or hex can cause severe problems if it is ignored. Fortunately, there are techniques for coping with these difficulties which will be presented in Example 3 - 12.

3. 3 Binary and Hexadecimal Arithmetic.

In this section we will discuss the basic rules for calculating with binary and hex numbers. The 360 performs arithmetic on binary numbers in 32 bit registers called General Purpose registers. There are 16 of these registers in all System/360 models except for the 360/20 which has 8 GP registers. The GP registers can be considered as analogous to the accumulator(s) in an electric calculating machine. They contain the numbers the computer is currently working on and can be used to develop sums, differences, quotients and products. A GP register is shown below in Fig. 3-1.

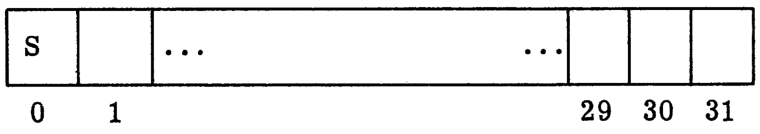


Fig 3-1. 32 bit General Purpose Register (GPR)

Bit 0 is used for the sign position, a 1 bit indicating a negative number and a 0 bit, a positive number. To add binary numbers, it is only necessary to apply the basic rules of binary addition (0 + 0 = 0, 1 + 0 = 1, 1 + 1 = 10) consistently and correctly (and tirelessly). As examples

$$\begin{array}{rrrrrrrr} 6 & 110 & 5 & 101 & 3 & 011 & 11 & 1011 \\ +1 & 001 & +1 & 001 & +1 & 001 & 13 & 1101 \\ \hline 7 & 111 & 6 & 110 & 4 & 100 & 24 & 11000 \end{array}$$

Exercise 3 - 4 Do the following additions in binary: 6 + 2, 6 + 4, 7 + 5, 7 + 7, $101101_2 + 100101_2$

However, the methods used by the 360 for operating on negative numbers are sufficiently different that they should be understood by the programmer so that results may be interpreted correctly. We will present this method by way of examples. In the interest of brevity, we will assume a 4-bit accumulator, the 4th bit being used for the sign position. The principles we will deduce apply directly to the 32-bit case. Positive numbers will be represented by their binary equivalents with bit 4 zero. Negative numbers will be represented in their two's complement form. The two's complement representation of a given number is obtained by reversing all bits (1's become 0's and vice versa) and then adding 1. As examples,

```
 5 = 0 101
-5 = 1 010  +  0 001
   = 1 011     (two's complement)

 1 = 0 001
-1 = 1 110  +  0 001
   = 1 111     (two's complement)

 0 = 0 000
-0 = 1 111  +  0 001
   = 0 000     (two's complement)

 7 = 0 111
-7 = 1 000  +  0 001
   = 1 001     (two's complement)
```

Exercise 3-5 Assuming a 4-bit register, write out in two's complement form all negative numbers which will fit into that register. The answers are given below.

There are several interesting points about two's complement numbers. First, zero has a unique representation. That is, there is no -0. Second, the range of negative numbers is one greater than the range of positive numbers. That is, for our 4-bit example, the largest positive number is 7(0111) whereas the largest negative number is -8(1000). Following are all binary numbers which can be accommodated by a 4-bit register.

7	0 111	-8	1 000
6	0 110	-7	1 001
5	0 101	-6	1 010
4	0 100	-5	1 011
3	0 011	-4	1 100
2	0 010	-3	1 101
1	0 001	-2	1 110
0	0 000	-1	1 111

Whenever numbers are generated which lie outside the range of a GP register (here 7 to -8) an overflow condition exists. The overflow indicator in the 360 is turned on and the programmer can test this indicator after those arithmetic operations which are likely to overflow. If the indicator is on (an overflow has occurred) the programmer can write a branch to a routine which takes appropriate action*. As an example, the sum 5 + 6 exceeds the capacity of a 4-bit register:

$$\begin{array}{r} 5 \\ +\ 6 \\ \hline 11 \end{array} \qquad\qquad \begin{array}{r} 0\ 101 \\ 1\ 110 \\ \hline 1\ 011 \end{array}$$

overflow

Notice that the sum in this example is represented as a negative number. If the overflow went undetected, the sum of 5 + 6 would be carried as -5.

Consider the addition of a positive and a negative number.

$$\begin{array}{r} 6 \\ +(-5) \\ \hline 1 \end{array} \qquad\qquad \begin{array}{r} 0\ 110 \\ 1\ 011 \\ \hline 0\ 001 \end{array}$$

no overflow

Notice that a carry into the sign position and a carry out of the sign position does not give an overflow condition. A carry out of the sign position, without a corresponding carry into the sign position, does give an overflow. As an example

$$\begin{array}{r} -5 \\ +(-6) \\ \hline -11 \end{array} \qquad\qquad \begin{array}{r} 1\ 011 \\ 1\ 010 \\ \hline 0\ 101 \end{array}$$

overflow

Again, an overflow has resulted in an incorrect sign as well as an incorrect sum.

---

* This is an oversimplification. A more complete description of overflow operation will be postponed to Sec. 3.8.

Subtraction is essentially the same as addition except that the subtrahend is complemented first, and then added to the minuend. As examples

```
    6      0 110  (minuend)               6    0 110
   -3  -   0 011  (subtrahend)        +(-3)    1 101
   ---                                -----------------
    3                                     3    0 011
                                            no overflow

    4      0 100                          4    0 100
-(-3)  -   1 101                         +3    0 011
 ----                                   ----  ------
    7                                     7    0 111
```

Notice that the complement of a negative number is a positive number. The discussion on overflows in addition applies equally to subtraction.

Multiplication of binary numbers is straightforward. The basic rules are: 0 x 0 = 0, 0 x 1 = 0, 1 x 1 = 1. As examples

```
   5     101           7        111
 x 3    x 11         x 5      x 101
 ---    ----         ---      -----
  15     101          35        111
        101                    1110
        ----                 ------
        1111                 100011
```

When either or both factors are negative, that is, when they are in complement form, the 360's operation can be understood as multiplying two positive numbers, and then complementing the product if the signs of the factors are unlike. If one factor is positive and the other negative, their product is negative and will be complemented. If both signs are alike, either both positive or both negative, the product is positive and will remain uncomplemented. From the programmer's point of view, multiplication has an effect which becomes significant much more rapidly than with addition -- the size of the result. If two 10-bit numbers are added, the result is at most an 11-bit number. However, if the same numbers are multiplied, the result can be a 20-bit number. The rule is: The number of significant bits in the product is at most the sum of the number of significant bits in each factor, it may be one less. (Leading zeros do not count as significant digits). As an illustration, 00110 and 01011 have three and four significant bits, respectively. Their product will have seven bits, 1000010. As another example, the product of 101 and 1000 has six bits, 101000.

Division is essentially the reverse of multiplication and the same logic that is used in decimal long division can be employed here except, of course, that we must use the binary subtraction rules: 10 - 1 = 1, 1 - 0 = 1, 0 - 1 = 1 with a carry of 1 into the next position of the subtrahend. As an example

```
       1010 / 11                Quotient/Remainder
101 / 110101            divisor / Dividend
      101
      ---
       110
       101
       ---
        11
```

The meaning of the remainder term is that

$$Q = \frac{D}{S} + \frac{R}{S}$$

In our example above, we have $\overset{10/3}{5\,\overline{)\,53\quad}}$, or 10 + 3/5 = 10.6. The usual practice in division is to discard the remainder. As a result, if the quotient is to be computed to an acceptable accuracy, the dividend must contain enough places. In this example, the dividend would be 10 and fractional places will be lost. The binary arithmetic hardware on the 360 handles whole numbers only, so that any "understanding" about where the decimal (or binary) point goes is between the programmer and his program. As an example, to get three fractional places in the division above, we add three zeros to the dividend (i. e. shift it left three places) and then proceed

```
          1010.100/100
101 /  110101.000
       101
       ---
        110
        101
        ---
         110
         101
         ---
          100
```

Here the quotient is 10.5 plus a remainder term of 100/101000 in binary. (Since the dividend has been extended left by three zeros, the same must be done to the divisor.) In decimal this is 1/10 so that the quotient is low by .1. This is not necessarily a disaster; in some engineering and statistical work, answers are usually required to some preset number of places. In fact, this is troublesome only in programs which involve financial calculations. Even here, there are definite lower limits placed on accuracy, such as to the nearest 1/2 cent. For such calculations, there are straightforward methods for computing results to the desired accuracy. These will be presented in Example 3-12.

Once the reader has an understanding of binary arithmetic, hex arithmetic can be readily grasped. We will not go into the details here, save for addition and subtraction, since these are usually the only operations the programmer will do directly. The typical need arises when the programmer is reading an assembly language listing, and wishes to compute the storage space required for a section of code starting at some location and ending at another specified location; assume starting and ending locations AFIA and C625. We could convert both of these numbers to decimal by either using the recipie of Sec. 3.2 or looking them up in a table. Unfortunately, the table may not contain the numbers we want. Given a little practice, it is easier to compute the difference directly in hex and then convert the result to decimal. To do this, we recall that A = 10, B = 11, C = 12, D = 13, E = 14, F - 15 and $10_{16} = 16_{10}$. Then, using the familiar borrowing technique 5 - A is 15_{16} - A = $21_{10} - 10_{10} = 11_{10}$ = B with a carry of one into the next subtrahend position, and so forth. We then have

$$\begin{array}{c} \text{C 6 2 5} \\ \underline{\text{A F I A}} \\ \text{1 7 0 B} \end{array} \qquad\qquad \begin{array}{rcl} 1 & = & 1 \\ \underline{16} & & \\ 16 & & \\ \underline{+\ \ 7} & = & 7 \\ 23 & & \\ \underline{\text{x } 16} & & \\ 368 & & \\ \underline{+\ \ 0} & = & 0 \\ 368 & & \\ \underline{\text{x } 16} & & \\ 5888 & & \\ \underline{+\ 11} & = & \text{B} \\ 5899 & & \end{array}$$

This section of code therefore requires 5899 positions of storage.

3.4 Storage Addressing

Before we consider the binary instruction set, let us examine in greater detail the decimal arithmetic instructions of Chap. 2. In particular, we will see how the 360 uses the information supplied by the assembler to address its storage. For an instruction such as

```
AP          SUM, AMOUNT
```

The assembler substitutes into its operand portions the storage addresses of SUM and AMOUNT. To better understand what is involved here, we shall dissect this instruction. Since the decimal add instruction requires eight bytes of storage, 48 binary bits of information must be supplied by the assembler. The format of the AP instruction is

11111010	L_1	L_2	B_1	D_1	B_2	D_2
0 7	8 11	12 15	16 19	20 31	32 35	36 47

where the numbers below the figure are the bit locations of the different segments of the instruction. These segments will be referred to as fields. For example, the first field (bits 0 through 7) is used for the AP operation code which happens to be the string of binary bits, 11111010. It is for the sake of the example only that the actual bits of this operation code are shown, the precise detail is of no particular consequence and need not be remembered -- we shall continue to refer to instructions by their mnemonic codes. The letters L, B and D refer to the length, base and displacement fields of operands one (SUM) and two (AMOUNT) which are indicated by subscripts 1 and 2. The length fields contain number of bytes of each operand to the right of the operand address, or one less than the length of the operand. The assembler takes care of this difference by subtracting one from all lengths prior to assembling a decimal instruction.

The base and displacement fields, which together constitute the operand address, are used to generate the location in storage of the operand. This happens in the following way: The base field contains a (binary) number from 0 to 15 which references one of the 16 General Purpose Registers. The GPR are part of the addressable storage of the 360. The programmer may place information in these registers, manipulate it and retrieve it in much the same way as the magnetic core storage we have been discussing so far. In fact, in some models of the 360, the GPR actually reside in core storage, in other models, they are implemented in high speed solid state circuits. For operations on binary numbers, the GPR also serve as registers for doing arithmetic. Figure 3-2 shows schematically the 16 GPR and core storage.

System/360 storage addressing works this way: The contents of the register specified in an instruction base field are added to the (binary) number

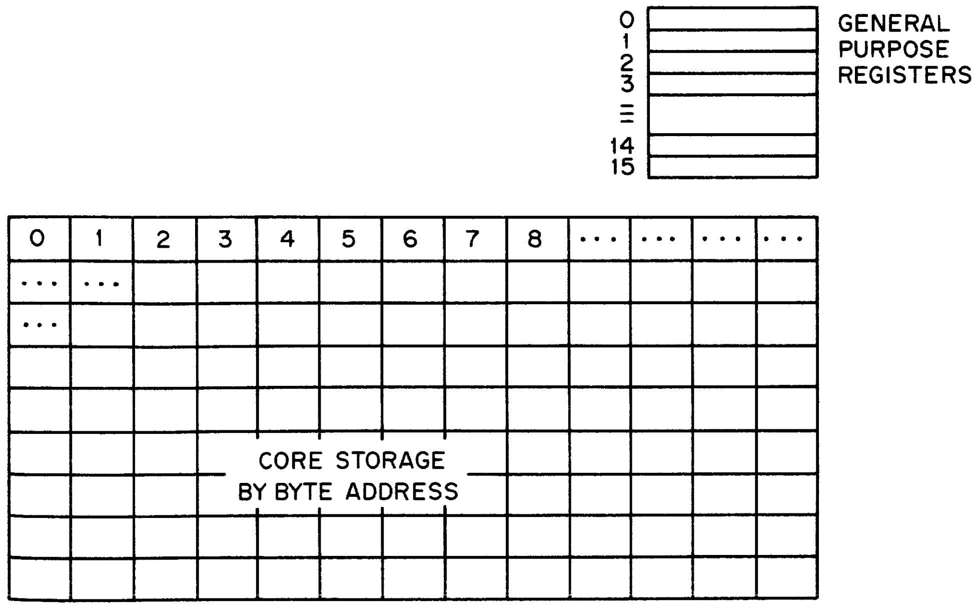


Fig. 3-2 Schematic of general purpose registers and core storage

in the displacement field, the sum gives the location in storage of the operand. The addition is done by the computer's addressing circuitry for each operand in the instruction. As an example, suppose that the locations of SUM and AMOUNT are in a data pool which contains all the data and working storage for the program. Assume that the data pool begins in location 12400 and that SUM is in location 12460 and AMOUNT is in location 12512. In order that our program may access these locations one of the general purpose registers, say, register number 10, would contain the location of the beginning of the data pool, the number 12400. Since the locations of SUM and AMOUNT are 60 and 112 storage positions, respectively, from the beginning of the data pool, the displacements D_1 and D_2 will contain these numbers. The same instruction is now

	L_1	L_2	B_1	D_1	B_2	D_2
AP	XX	YY	10	60	10	112

The actual bases and displacements are binary numbers but for simplicity their decimal equivalents are shown. When this instruction is executed by the computer, the address of the first operand, SUM, is

$$C(10) + 60 = 12400 + 60 = 12460$$

and the address of the second operand, AMOUNT, is

$$C(10) + 112 = 12400 + 112 = 12512$$

Here C(10) refers to the contents of general purpose register 10. Figure 3-3 shows the process schematically for one of the operands, AMOUNT.

Let us review System/360 storage addressing. The address of a given location in storage consists of a base field and a displacement field. The base field contains a number from 0 to 15 referencing one of the 16 GPR in the System/360. (The System/360 Model 20 contains 8 GPR, numbered 0 through 7. Programs written for the Model 20 should not contain references to registers higher than 7.) Also, on the other Models of 360, registers 0, 1 and 12 through 15 are used for various purposes by the operating systems. To avoid error, it is suggested that the programmer restrict himself to registers 2 through 11. However, if additional registers are required, it is possible to use the other registers provided the programmer is aware of the restrictions on their usage. (These will be covered in Chaps. 7, 9, and 10). Let us call the general purpose register which is referenced by the base field in an address, a base register. The base register can be thought of as containing a pointer to a location in storage, the base location, which is near the location we wish to access. The displacement field contains the distance in bytes between the base location and the desired location. Figure 3-4 gives a picture of this concept. There is an important limitation on the size

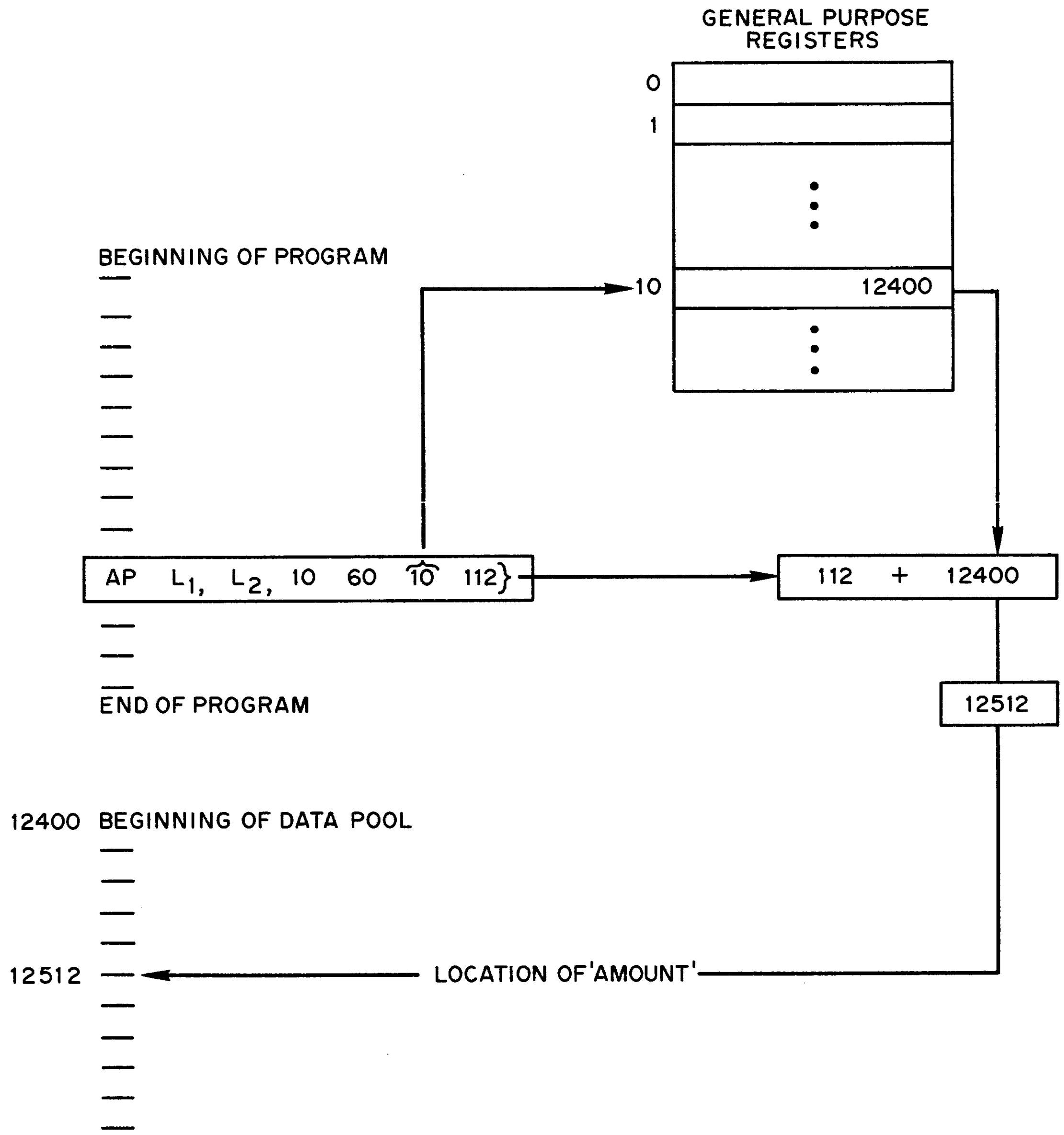


Fig 3-2 Illustration of the storage addressing technique of System/360

of the displacement. The displacement field is 12 bits wide. As we have discussed in Sec. 3.1, the largest number that can be represented in 12 bits is +4095. This means that no location can be more than 4095 storage positions from its base location. Or, to put the matter in a different light, each base register spans 4096 locations, the base position plus the next 4095 higher address locations. Note that since the displacement field must contain a positive quantity, displacements can only be counted from the base location to higher addressed storage. One other point should be noted: When GPR 0 is specified as a base, the addition of base and displacement does not take place and the contents of the displacement field alone gives the address. This technique can be used to address the lowest 4096 positions of storage without using up a base register.

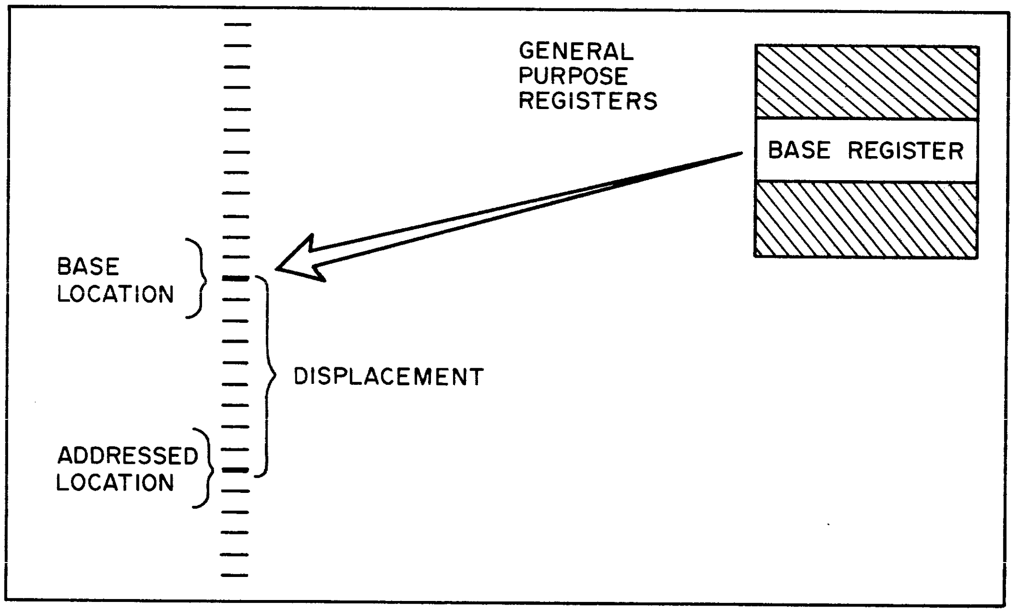


Fig. 3-4 Base addressing

Worked Example

3 - 1 Assume that registers 5, 6 and 7 contain the values 8000, 9000 and 13096 respectively. Write base and displacement addresses for the following locations: 8000, 9000, 9100, 13200 and 17191.

Solution

Base Register	Contents	Location	Base/Displacement
5	8000	8000	5/0000
6	9000	9000	6/000 or 5/1000
7	13096	9100	6/0100 or 5/1100
		13200	7/0104
		17191	7/4095

Note that when the coverage of two base registers overlaps, such as registers 5 and 6 here, a given location can be addressed in more than one way as is illustrated for locations 9000 and 9100.

Worked Example

3 - 2 If registers 0, 13, 14, 15 contain 4096, 10000, 0 and 15000 respectively, the following base and displacement fields address the indicated storage locations:

Base Register/Contents		Base/Displacement Fields	Location
0	4096	0/3000	3000
13	10000	14/3000	3000
14	0	13/0500	10500
15	15000	0/0000	0
		15/4095	19045

Note that the use of register 14 to address locations below 4096 is unnecessary and wasteful of a base register. These locations can be addressed by specifying 0 as the base register. Note also that when GPR 0 is used for this purpose, its contents are not used. Can location 4,096 be addressed by the base registers above with their indicated contents?

We have now progressed far enough in our study of base register addressing to look more deeply into the recipie presented in Chapter 2 for starting 360 programs. In Chapter 2, each program was prefaced by the instructions

```
BALR   11,0
USING  *, 11
```

BALR is the mnemonic for the Branch and Link Register instruction which has the following format:

$$\boxed{\text{BALR}\quad \text{R1, R2}}$$

This will cause the address of the next location in storage after the BALR to be placed in register R1 and a branch to the location specified by the contents of R2. The BALR instruction is one of the RR class of instructions, so called because its execution involves Register to Register operations. RR instructions occupy two bytes of storage. As an example of BALR usage, if the instruction

```
BALR   10,11
```

begins in position 10,000, and register 11 contains the value 19756 then its execution will cause the value 10000 + 2 = 10002 to be stored in register 10, replacing whatever was there previously. The next instruction to be executed will be the one which starts at location 19756. If the R2 operand is 0, then no branch is attempted and program execution proceeds sequentially after the BALR. As an illustration, if the previous example were BALR 11, 0, then its execution would cause the value 10002 to be stored in register 11. The next instruction executed will be the one starting at 10002, that is, the instruction immediately after the BALR.

The general format of the USING instruction is

```
USING      A, R1, R2, R3, R4, R5
```

where A is an address (an address symbol or an absolute address) and the R's specify base registers. Their values must be between 0 and 15 and from one to five base registers may be specified in a single USING instruction. The base register R1 is assumed by the assembler to contain at execute time the base address represented by A, R2 through R5 are assumed to contain A + 4096, A + 8192, A + 12288, A + 16384, respectively. There is a major and important difference between the BALR instruction and the USING instruction. The BALR can be executed by the computer, the USING supplies information only to the assembler; it does not appear in the object code produced by the assembler. In a sense, it is "executed" by the assembler only. This is another example where it is important to distinguish between what happens at assembly time and what happens at execute time. In the light of this distinction, it may be helpful to the reader to think of the USING as a pseudo instruction, or as an assembler instruction.

Let us return to the question of the contents of the registers R1 through R5. Note that the Assembler assumes that these registers will contain certain addresses at execute time, it will not introduce instructions into the program to load these values into the base registers, it is up to the programmer to do this. For the moment, we will discuss a program whose entire length can be spanned by a single base register. Figre 3-5 shows such a program. We are assuming that the address of AMOUNT is less than 4096 bytes from BEGIN

```
             START      4096
BEGIN        BALR       11,0
             USING      * , 11
             --
             --
ADD          AP         SUM, AMOUNT
             --
             --
SUM          DS         P5
AMOUNT       DS         P5
             END        BEGIN
```

Fig. 3-5 Sample program which can be spanned by a single base register. AMOUNT is less than 4096 bytes from BEGIN.

The START instruction tells the assembler to locate the first instruction, BALR, at location 4096. This also initializes the assembler's location counter to 4096. At execute time, the BALR instruction will load register 11 with the location of the instruction immediately following, that is, 4098. After the assembler has processed this instruction, its location counter will read 4098. The USING instruction, which is not assembled into the object program, tells the assembler to assume that register 11 will contain the current value (*) of the location counter, 4098, at execute time. Note that it does not in any way cause the assembler to load 4096 into register 11. That will be done by the BALR at execute time. Assume that SUM begins at 8000 and AMOUNT at 8005. When the assembler encounters these names in the program, it will compute their distances (displacements) from the next lower base position, here 4098. These displacements are SUM (3002) and AMOUNT (3007). When the instruction at ADD is assembled, for instance, the assembler will use GPR 11 as the base register for each operand and displacements of 3002 and 3007 for SUM and AMOUNT, respectively.

Let us now consider a program, such as the one in Fig. 3-6 which requires more than one base register to span its length. The determination of how many base registers will be required to span a program is up to the programmer. He does this on the basis of an estimate of the total storage requirement of his program (instructions plus data). If he estimates that between 9000 and 12000 bytes will be needed for a particular program, then he must allocate three base registers, for example. One might suspect that this places a heavy burden on the programmer for large programs or even that the 360 is incapable of addressing more than 15 x 4096 positions. This is not so because, first, large programs are written as a number of small, more or less independent, modules and second, if for some reason a program cannot be segmented conveniently, the contents of base registers may be stored and reloaded with new or the previous values as needed.* To handle the multiple base register program, we will need an additional instruction and a new type of defined constant. The new instruction is the Load Multiple. Its format is

```
LM      R1, R2, ADD
```

This will cause all the general purpose registers between R1 and R2, inclusive, to be loaded starting from storage location ADD and continuing for as many locations as needed to load the specified registers. If R2 is less than R1, the registers are loaded from R1 to 15 and then from 0 to R2. Since each register is 32 bits long, they will be loaded from ADD in multiples of four bytes. We will also need to define address constants. This is done using

*As additional perspective, note that the coding required to fill the span of a single base register, approximately 1000 instructions, fills an entire pad of assembly coding paper.

```
        START   4096
BEGIN   BALR    *,10          GPR10=4098 WHEN THIS INSTRUCTION IS EXECUTED
        USING   *,10,11,12    THIS STATEMENT INFORMS THE ASSEMBLER
*       THAT GPR10 WILL CONTAIN AT EXECUTIVE TIME * (=4098), AND THAT
*       GPR11 AND GPR12 WILL CONTAIN * +4096 AND * +8192--THE
*       ASSEMBLER WILL NOT LOAD ANY OF THESE VALUES INTO THE GPRS
*       IT IS UP TO THE PROGRAMMER TO PLACE INSTRUCTIONS IN THE
*       PROGRAM TO LOAD THESE VALUES
FIRST   ZAP     SUM, CZERO THE COVERAGE OF GPR10 EXTENDS
*       FROM LOCATION FIRST TO LOCATION FIRST+4095
        B       GO            BRANCH AROUND ADDRESS DATA
ADD     DC      A(FIRST +4096), (FIRST+8192)
*       INSTRUCTION IS EXECUTED--THE PRECISE PLACEMENT
*       OF THIS INSTRUCTION IS ARBITRARY, HOWEVER IT MUST BE
*       EXECUTED BEFORE ANY OPERAND BEYOND THE COVERAGE
*       OF GPR10 IS REFERENCED
        --
        --
SUM     DS      PL5
CZERO   DC      PL5'0'
*       (END OF COVERAGE OF GPR10--THIS LOCATION NEED
*       NOT BE NOTED BY THE PROGRAMMER PROVIDED GPR11 AND
*       GPR12 HAVE BEEN LOADED BEFORE ANY OPERANDS BEYOND
*       THIS POINT ARE REFERENCED)
        --
        --
AREA 1  DS      PL5
        --
        --
*       END OF COVERAGE OF GPR11
        --
        --
AREA 2  DS      PL5
        --
        --
        AP      SUM, AREA1    GPR11 IS BASED FOR AREA 1, GPR10 IS
        AP      SUM, AREA2    BASE FOR SUM, GPR12 IS BASE FOR AREA 2
*       THE ASSIGNMENT OF BASE REGISTERS ABOVE WILL BE DONE
*       AUTOMATICALLY BY THE ASSEMBLER WITH THE INFORMATION
*       SUPPLIED BY THE USING STATEMENT
        --
*       THIS POINT IS LESS THAN 3x4096 LOCATIONS FROM FIRST
        END     BEGIN
```

Fig. 3-6 Program requiring three base registers.

the A operand for the DC instruction. This is necessary because we wish the assembler to store in the program, at assembly time, those addresses which will be loaded at execute time into general purpose registers R1 through R2. As an example

```
ADD     DC    A  (FIRST + 4096)

        DC    A  (FIRST + 8192)
```

will cause the address of FIRST + 4096 to be stored in the four storage locations (32 bits) starting at location ADD and the address of LOCA + 8192 to be stored at ADD + 4 through ADD+7.

The major point of interest in the example program in Fig. 3-6 is the method of loading multiple registers with their base locations. The first register can be loaded by using a BALR instruction just prior to the origin of that base register's coverage, location FIRST. However, for the additional base registers (we are assuming a total of three), instructions must be included to load them with addresses equal to FIRST + 4096 and FIRST + 8192. These addresses can be stored as constants by using DC instructions with A type operands, A (FIRST + 4096) and A (FIRST + 8192). The two words containing these addresses will be located from ADD through ADD+7 in Fig. 3-6. The Load Multiple instruction at location FIRST will load these addresses into registers 10 and 11. The branch to GO instruction is included to allow the program to branch around the address data. This data need not be stored in such close proximity to the USING statement. However, it must be included somewhere within the span of the first base register since, when the LM instruction which references it is executed, only the first base register contains a base location. For the sake of convenience and clarity, we have located the additional base address data close to the beginning of the program and we recommend this approach to the reader for the same reasons.

During the assembly of the program in Fig. 3-6, the assembler will indicate next to each symbol in its symbol table which base register covers that symbol as well as calculating the displacement of the symbols' location from the origin point. As examples, AREA 1 would be covered by base register 11 and AREA 2 by base register 12. As a result, when a symbol appears as an operand, the assembler has sufficient information about the symbol to generate its address.

Exercise 3-6 In the following program consider how the B BEGIN instruction could be handled by the assembler. Specifically, consider the base and displacement assignment

```
BEGIN   START   256
        BALR    11,0
        USING   *, 11
        --
        --
        --
        B       BEGIN
        --
        --
        --
        END     BEGIN
```

3.5 Storage Allocation

In Chap. 2 we discussed storage allocation and constant definition for decimal and character fields. The techniques are similar for binary fields but with the added requirement that binary fields must be aligned on their proper storage boundaries. In Chapter 2, the 360 was represented as a computer whose storage is organized by byte. That is, grouped into individually addressable units of eight bits (and a parity bit) each. In addition to the byte which is the basic building block of information, the 360 can operate on data formats grouped in consecutive two-byte units (half-words), consecutive four-byte units (words) and consecutive eight-byte units (double-words). Fig. 3-7 shows these data formats schematically. Byte fields are used for character and hexadecimal information, half and full-words for binary numbers and double-words for control information on input-output operations. Both full and double-words can also be used for floating-point numbers which are required by many mathematical computations. The subject of floating-point will be covered in Chapter 6.

The location of each of these units is given by the address of its leftmost byte. Each must be located on an integral boundary for that unit of information. An integral boundary is defined as an address which is an integral multiple of the unit's length. As examples, the addresses 0, 8, 16, ... are valid locations for double-words (8 bytes), 0, 4, 8, 12, 16, ... are valid for full-words (4 bytes), whereas 0, 2, 4, 6, 8, 10, 12, 14, 16, ... are valid for half-words (2 bytes). Byte oriented information, such as a field of alphabetic characters, may of course be located at any address, even or odd.

The necessity for alignment on integral boundaries is imposed by the hardware requirements of the higher performance models of the 360. As an illustration, consider binary arithmetic which is performed on fixed length quantities, usually full-words. The 360 Model 50 accesses its core storage 32 bits at a time, from integral full-word boundaries. As a result, in one storage cycle, a full-word can be brought to the arithmetic unit. If binary data were not located on integral boundaries, two storage cycles would be

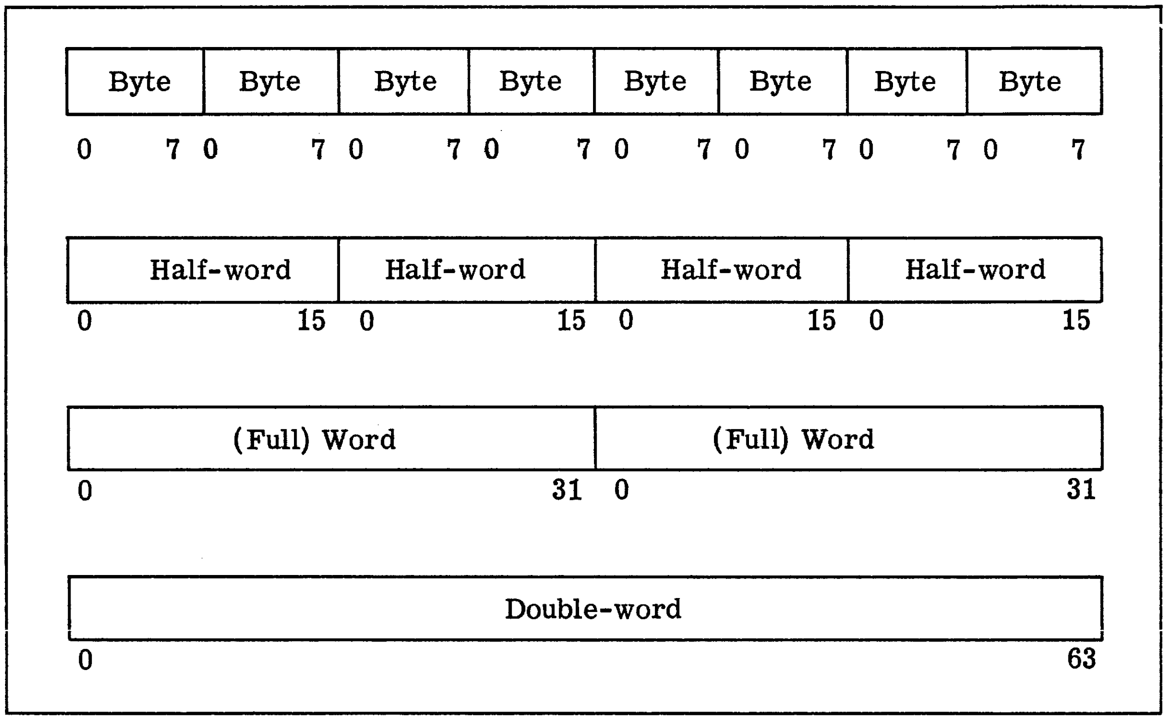


Fig. 3 - 7 360 Data Formats. Bit numbers are shown below each format. Parity bits, one for each group of eight bits, are not shown.

required thereby slowing down the operation by up to a factor of two. As a result of requiring alignment, compatibility between the various models of the 360 is preserved as well as giving the user the full performance inherent in a given model. Further, no model of the 360 will execute an instruction which references improperly aligned data.

We will next discuss how to reserve storage and define constants for the various information formats. To reserve storage in units of half-words, words or double-words, the DS statement is used with operands of H, F and D. As examples, the statements

```
TABL1   DS   13D
TABL2   DS   15F
TABL3   DS   200H
```

will reserve 13 double-words with TABL1 as the address of the first double word, 15 words starting at TABL2 and 200 half-words starting at TABL3.

The assembler will automatically align half-word, full-word and double-word fields on their proper boundaries.

Exercise 3-7 In the above example, what would be the total storage requirements if the DS statements were written in the order TABL2, TABL1, TABL3?

In addition to the DS statement, storage may be reserved, or skipped over by the assembler by resetting the location counter with the ORG instruction. Its format is

```
ORG     r
```

where r is a relocatable expression whose value is assumed by the location counter. The ORG instruction should not be labeled. As an example, the statements

```
DS   50D
DS   200H
```

can be replaced by

```
DS   0D
ORG  *+800
```

since the two storage areas occupy 800 bytes and their defining statements cause the location counter to be advanced 800 positions after it is moved to a double word boundary, if necessary. The zero operand in the DS reserves no storage but does move the assembler's location counter to a double-word boundary if it is not at a location whose address is a multiple of eight. If it is required to label the storage reserved by the ORG, the DS 0D statement can be labeled.

Binary constants are written as follows

```
CONA   DC   H-157
CONB   DC   F 209572
```

The decimal number following the DC is converted into a binary half-word (H) or full-word(F). If the number is unsigned or has a plus sign, it will be treated as a positive number. Numbers preceded by a minus sign will be converted to their two's complement representation. The range of numbers which can be accommodated by a half-word is 32,767 to -32,768 whereas for a full word, the range is 2,147,483,647 to -2,147,483,648. All constants are stored right-justified and padded out with zeros to the left. Multiple constants can be defined in a single statement such as

```
CONTBL   DC   F -3,612,-711,4051
```

which will give four constants, each of four bytes, with the symbol CONTBL as the address of the leftmost byte of the constant -3. Since commas are used as separators, a constant may not be written with commas. Fractional binary data introduces a new problem. As we discussed in Sec. 3.2, a decimal fraction often cannot be represented in a finite number of bits. As an example, 37.613 becomes in binary, 100101.100111000111..., the sequence 000111 repeating indefinitely. The more places to the right of the decimal point which are retained, the more accurate is the decimal to binary conversion. However, compromises have to be made since the total storage of the computer poses a limitation on conversion accuracy, if not the more immediate restriction of a 32-bit word size. Since the binary arithmetic instruction set deals with whole number data only, fractions must be converted to whole numbers by shifting them to the left. The number of places shifted left equals the number of fractional positions we desire to retain. As an example, to retain 12 binary places of the fraction 37.613, we write FS12 37.613 where S12 is the scale factor. This will generate the constant 100101100111000111 which will be right justified and padded to the left with zeros. It is understood that the decimal point lies between the 12th and the 13th bit positions. If scaling is not specified for fractional constants, the fractional part will be truncated and the remainder of the constant treated as a whole number. If bits are lost because of scaling, rounding will occur in the leftmost bit of the lost position where a 1 will be added. The carry, if any, will be added to the rightmost bit of the saved portion. As an example, FS11'37.613' would give 100101.10011100011 plus a carry of 1 into the low order position since the 12th bit, which is lost, is 1. The result is then 100101.10011100100. A duplication factor may be used to repeat a given constant the number of times specified. As an example

```
TBLZER    DC    100 F '0'
```

will generate a 100-word field of zeros starting at word TBLZER.

Hexadecimal constants are written as follows

```
HEXCON    DC    X'123FA4C'
```

The constant may comprise up to 32 hex-digits chosen from the hex-digits, 0 1 2 3 4 5 6 7 8 9 A B C D E F. Since each hex-digit requires four bits, every pair of hex-digits requires one byte. If the number of digits are odd, the field will be padded with a zero in the leftmost byte. The constant above would give 0 1 2 3 F A 4 C, for example. Boundary alignment is not performed but may be forced, if required, by the following device. The statements

```
          DS    0F

BITPAT    DC    X'00FF00FF'
```

will cause BITPAT to be aligned at a full-word boundary and the second and fourth bytes of the word at BITPAT set to ones. The alignment is forced by the storage definition statement which calls for zero duplication of a full word. The assembler interprets this by advancing the location counter from zero to up to three bytes, as required, to the next full-word boundary. Half-word or double-word alignment may also be forced in this manner. A duplication factor and explicit length may be used for hexadecimal constants as in the following example

```
CONHEX   DC   5XL3'FFF'
```

The length code specifies explicitly the length of the field in bytes. When the explicit length is greater than the implicit length specified by the constant itself, the constant is padded to the left with zeros. As a result, CONHEX will consist of five repetitions of the three-byte field, 000FFF. It is not mandatory that the duplication factor and explicit length accompany each other in a hexadecimal constant definition.

In addition to hexadecimal constants, there is another way of specifying bit oriented data by using the bit constant, B. As examples

```
BITCON   DC   B'10101010'
B1       DC   BL1'101'
B2       DC   BL2'101'
```

Here, BITCON will occupy one byte of storage in the bit configuration shown; B1 will also occupy one byte and will be padded to the left with five zero bits; B2 will occupy two bytes, the first byte will contain eight zero bits and the second, the bit pattern 00000101.

Worked Example

3-4 Given a section of a program, labeled PROG1 which consists of instructions and data areas, reserve an equal amount of storage immediately adjacent and label it PROG2.

Solution The obvious (and hardest) way is to count the number of instructions, taking into account their varying lengths, and add their storage requirements to the storage reserved for data and then write a DS reserving this many bytes. An easier way is

```
PROG1   --
        --
        --
        --
        --
PROG2   EQU *
        ORG * + PROG2 - PROG 1
```

To veryify this, assume PROG1 has the address value 10000 and requires 1016 bytes. Therefore, after the last statement in the PROG1 section has been processed, the location counter (*) will read 11016 which also becomes the address value of PROG2. The ORG instruction will then advance the location counter another 1016 positions thereby reserving that amount of storage for PROG2.

In Sec. 3.4, we introduced address constants, or more specifically, the A-type address constant. Its general format is

```
s       DC       A(r)
```

Where s is the symbolic name of the constant r is the relocatable expression whose address value is stored in the four bytes beginning at s. The Y-type address constant is similar to the A-type except that only two bytes are used. If it is known that the address constants used in a program are always less than 65535, then the Y-type address constant can be used to save storage. In addition to these two address constant types, there is also the S-type. It is similar in every respect to the Y-type except that its two bytes are used to store an address in base and displacement form: the first four bits are used for the base and the remaining 12 for the displacement. As examples, if LOCA has the address value 16252 and is spanned by base register 10 with a base point of 16002, the statement

```
STYPE    DC    S(LOCA)
```

will create a two-byte constant at location STYPE the first four bits of which are 1010 (= 10) and the last 12, 000011111010 (=250).

3.6 Address Symbols

In Chapter 2, when the START statement was introduced, there was the tacit assumption that its operand gave the precise location in storage where the program would be loaded into. That is,

```
PROGA    START    4096
```

would seem to indicate that the program named PROGA will be assembled relative to 4096, and loaded into storage beginning at location 4096. That is, an instruction 50 bytes from the beginning of the program would be assigned address 4096 + 50 = 4146 by the assembler, and will be loaded into storage starting at 4146. This is an over-simplification. It is a rare circumstance when a programmer wants to control the load point of his program. This is so because he rarely has the entire core storage to himself. As we discussed in Sec. 1.3, a number of functions which are required frequently by the programmer are provided by the computer's operating system. These include routines to control the operation of input-output devices, transmit messages to the computer operator, load the program and process error conditions such as arithmetic overflows. The programs which handle these functions usually reside in the lower positions of core storage. As a result, problem programs can be loaded only above these system functions. We still do not have a definite fixed point at which to load programs, however. The boundary between system programs and problem programs is not rigidly fixed. As systems functions are added or deleted, the boundary will change. As an example, the operator's console may be changed from a typewriter to a graphic display device which will require a larger service program than the typewriter. The boundary would then move up and all programs which were assembled relative to the old boundary would have to be modified in some fashion to take this into account. This could be done by reassembling all user programs relative to the new origin which may take a respectable amount of computer time. This is not a good solution, however, because the boundary can and does undergo slight but frequent fluctuations as errors and inefficiencies in the systems programs are fixed and patched. The alternative is to arrange the load program so that it can locate, or relocate, a program at an arbitrary point in core storage; this load program is termed a relocatable loader. Any relocatable loader is essentially a program which does simple arithmetic on all instruction addresses in the programs it loads. Figure 3-8 shows a branch instruction in a program assembled relative to location 4096. If the program is loaded starting at location 4896, 800 positions from the starting location assumed by the assembler,

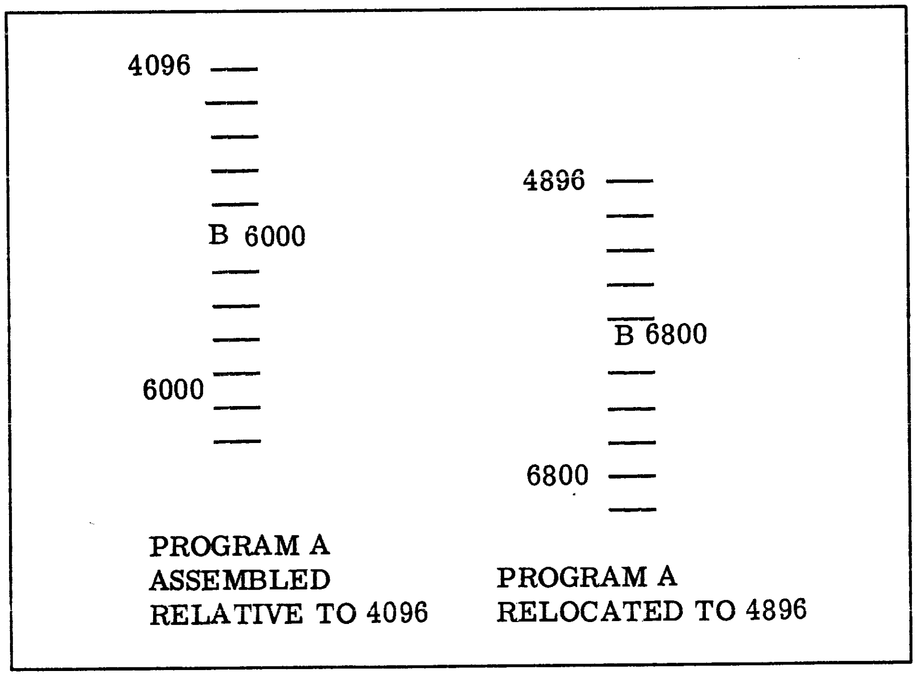


Fig. 3-8 Program Relocation

every location in the program including the one referenced by the illustrated branch will also be 800 positions removed from its original location. As a result, to ensure correct operation of the relocated program, all operand addresses must be increased by 800. Specifically, the B 6000 instruction mustbe changed by the relocatable loader to B 6800. Given the architecture of the 360, these operand relocations are easily accomplished. In the previous example, if GPR 11 contained 4096, then address 6000 would be generated as base register 11 with a displacement of 6000-4096, or 1904. The displacement gives the difference between the location of an operand and the nearest base point below the operand. This difference is the same for any choice of starting location since moving a contiguous program around in core storage does not change the distance between any two locations in the program. As a result, the displacement does not have to be modified. Further, if the program requires less than 4096 contiguous bytes for its instructions and data, and if GPR 11 is loaded by an instruction such as BALR 11, 0, no change at all needs to be made to the program to relocate it. In the more likely case of programs larger than 4096 bytes, which then require more than one base register, only the address constants defining the base points for those registers defined in a USING statement need to be modified. These relocatable addresses or relative addresses are identified by special coding punches in the binary object deck for the particular program. The modification is nothing more than addition of a relocation constant to each relocatable address; the relocation constant is defined as the difference between the starting location specified by a program's START statement and the actual load point. In the previous example, the relocation constant is 4896 -4096 = 800.

Not all addresses in a program require relocation. The ones which do not are termed absolute addresses. An absolute address is one which does not change when the program is relocated. The operand of the instruction B 7004 is an absolute address, for instance. It is also possible to have absolute symbols. Consider the following coding

```
                  B     CKPT
                  —
                  —
         CKPT     EQU   7004
```

Here CKPT is an absolute symbol because it has been assigned an absolute value in the EQU statement. Symbols which are not absolute are termed relative symbols. Their address values will change with program relocation. In Chapter 2, we discussed relative addressing involving address symbols such as ENTRY + 8, LOCA - 2. These symbols can be referred to an expressions, since they comprise symbols (relative and absolute) and operators, +, - , and * respectively. The rules for constructing address expressions are: No more than three symbols may appear in an expression and each symbol must be separated by only one operator. The expression, ENTRY + 2 - FIELD + TEST is invalid (more than three symbols) and the expression ** 2+A is also invalid (two operators in sequence, **).

Because relocation removes the direct correspondence between the address assigned to a symbol and its location in core storage at execute time, it is more appropriate to speak of the address value of a symbol, rather than its address. The address value will be defined as the relative address assigned to a symbol by the assembler.

In addition to the address symbols we have been using so far, hexadecimal and character self-defining values may be used. A hexadecimal self-defining value comprises from one to six hex digits enclosed in single quotation marks and preceded by an X. Following are examples: X 'A15', X '123456', X 'ABCDEF'. A character self-defining value is a single character, enclosed in single quotation marks and preceded by the letter C; as examples: C 'A', C '+'. In addition, the decimal numbers we have been using as absolute symbols can be considered as decimal self-defining values. The self-defining values in an expression are first converted to their binary equivalents and the indicated operations are then carried out. As an example, LOCA + 196, LOCA + X 'C4' and LOCA + C 'D' are all equivalent, since 196 is, in binary, 11000100, X 'C 4' is 11000100 (C = 1100 and 4 = 0100), and C 'D' has the bit pattern 11000100. The use of hexadecimal and character self-defining values in address arithmetic is rather bizarre since the same result can be had by using decimal values. However, they are useful with the immediate instructions which will be discussed in Chapter 5.

Just as symbols are either absolute or relocatable, expressions are absolute if they contain only absolute symbols or self-defining values or if they contain the difference of relative symbols. All other expressions are relocatable. As examples, with LOCA an absolute symbol and TEST, BETA and AREA relative, following are absolute and relocatable expressions:

ABSOLUTE	RELOCATABLE
X 'C4A' + 5 + C '+'	TEST-BETA + AREA
TEST-AREA + 5	BETA + LOCA
LOCA + 4096	TEST + 4000

Literals In Sec. 3.5, we discussed how data constants can be defined and referenced by use of the DC statement. Constants may also be defined and referenced by literally using the constant itself as an operand. Literal operands are prefaced by an = sign and are defined in the same way as DC operands except that S-type address constants may not be specified. As an example, the following code

```
                   AP     SUM, = P '635'
```
gives the same result as
```
                   AP     SUM, CON635
                   ___
                   ___
                   ___
         CON635    DC     P '635'
```

When the assembler encounters a literal, it sets up the equivalent constant in an area of storage called the literal pool which it will add to the problem program at the end of assembly. The address of the constant in the literal pool is then used as an operand. Multiple usage of the small literal in a program will not cause more than one entry for the constant in the literal pool. Following are additional examples of literals

```
         MVC     OUTPUT(18),=3CL6'TOTAL"
         AH      REG1,=HS8'11.62'
         A       10,=F'69785'
         LH      2,=X'4F'
```

3.7 Instruction Formats

The 360's instructions are classified into five formats

1. RR Format: Register-to-register
2. RS Format: Register-to-storage
3. RX Format: Register-to-storage, indexed
4. SS Format: Storage-to-storage
5. SI Format: Storage and immediate-operand

The format codes indicate the operation to be performed. RR denotes register to register operations such as comparing the contents of two GPR; RS, a register to storage, or storage to register operation such as Load Multiple; RX, register-to-storage, indexed such as adding four bytes in storage to a GPR where the indexing feature (Chapter 4) allows easy modification of the address of the data word; SS, storage-to-storage operations such as the decimal instructions of Chapter 2; and SI, storage and immediate-operand operations in which the data is contained within the instruction itself (im (immediate-operand). Figure 3-9 illustrates the bit organization of these instruction types. Note that RR instructions require two bytes, SS instructions require six bytes, and all others require four bytes. A knowledge of instruction lengths is important and the reader should keep the above facts in mind as additional instructions are introduced in this book.

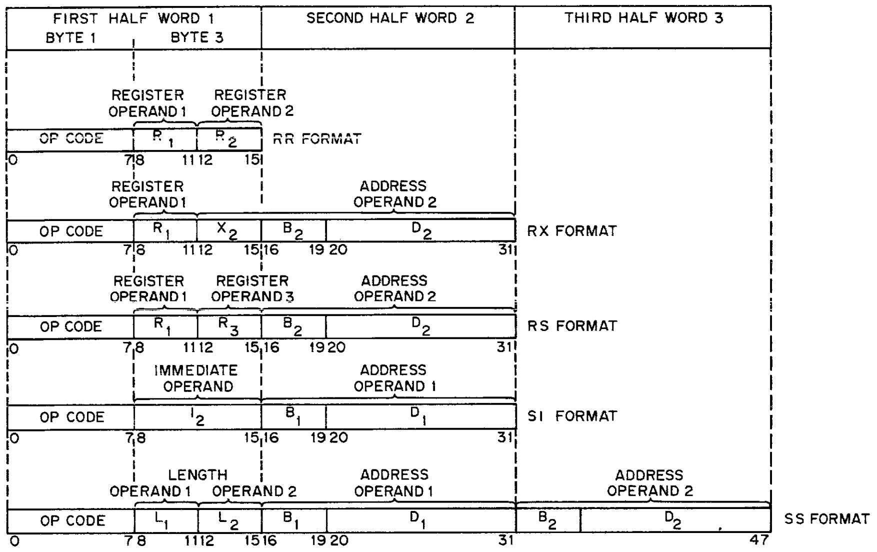


Fig. 3-9 Bit organization of the five instruction formats of the 360

Only occasionally does the programmer need to concern himself with the details of the bit organization of an instruction. Of much greater significance are the options allowed by the assembler in writing instructions. To describe these options, we will use the following designations:

R1, R2, R3, X2, B1, B2 -- An absolute symbol or number denoting one of the 16 GPR being used as a binary register (R), a base register (B) or an index register (X).

S1, S2 -- A relocatable symbol or expression whose base register and displacement are implied by virtue of a USING statement.

D1, D2 -- An absolute symbol, number, or expression denoting an operand's displacement field.

L, L1, L2 -- An absolute symbol or number denoting the explicit length of an operand.

I2 -- Immediate data, an absolute symbol or number giving eight bits of data to be used in an immediate instruction.

These designations can be used to relate the assembler formats to the actual bit configuration of an instruction, if required by referring to Fig. 3-9. These symbols will be used when instruction definitions are introduced subsequently. Following are the assembly language formats with examples for the indicated class of instruction (OP indicates the operation mnemonic).

An asterisk indicates the most frequently used option:

Register-to-register

```
            *OP    R1, R2                  BALR 5, 11
```

Register-to-storage

```
            *OP    R1, R3, S2              LM   1, 5, AREA
             OP    R1, R3, D2(B2)          LM   2, 12, 3094(14)
```

Register-to-storage, indexed

```
no indexing  *OP   R1, S2                  A    15, WORD
   indexing  *OP   R1,  S2(X2)             A    15, WORD(4)
   indexing   OP   R1, D2(X2, B2)          A    15, 2048(4, 12)
no indexing   OP   R1, D2 (0, B2)          A    15, 2048(0, 12)
   indexing   OP   R1, D2(X2)              A    15, 2048(4)
```

Notice that if an explicit base is specified with no indexing, the index register must be written as a zero.

Storage-to-storage

```
            *OP    S1, S2                  AP   TOTAL, AMT
             OP    S1(L1), S2(L2)          AP   T(10), A(7)
             OP    D1(L1, B1), D2(L2, B2)  AP   2012(10, 4), 1046(7, 4)
             OP    D1(L, B1), D2(B2)       MVC  A(14), AREA
```

Storage and immediate-operand

```
            *OP    S, I2                   MVI  CHAR, C'*'
             OP    D2(B2), I2              MVI  402(12), ASTRK
```

While there are some minor departures from the above illustrations (e. g., shift instructions, which are RS, do not use an R3 operand), they serve to convey the various assembler options for the majority of 360 instructions.

The EQU statement which was introduced in Chapter 2 has wider possibilities which we illustrate below:

```
ASTRK   EQU   C'*'
REG1    EQU   1
ADDR    EQU   4036
XREG    EQU   13
REG10   EQU   10
```

these absolute symbols may appear in instructions such as:

```
MVI     CHAR, ASTRK
A       REG1, ARRAY(XREG)
LM      REG1, REG10, ADDR
```

Note that the information supplied by the EQU operands is used by the assembler only and does not generate any data into the object program. The usage of EQU to denote the character * by the symbol ASTRK has quite a different effect from the statement

```
ASTRK1      DC  CL1'*'
```

Whenever the symbol ASTRK1 appears in an instruction, the assembler will substitute the address of the character * whereas when ASTRK appears, the actual character itself will be substituted into the instruction. The latter is desired result in the Move Immediate instruction which moves the 8-bit character stored in its I-field to the address specified by its first operand.

3.8 Branching

In the discussion on conditional branch instructions in Chapter 2, several different mnemonic codes were introduced: BH, BL, BE, BZ, and so forth. These mnemonics are conveniences for the programmer: In fact, there are only two conditional branch instructions in the 360; the various possible results of the comparison or arithmetic operation are accommodated by different bit configurations in a given field within these two branch instructions. Their assembly language formats[1] are

```
BC     M1, D2(X2, B2)     (RX)

BCR    M1, R2             (RR)
```

For these two instructions, M1 is a four-bit field in what would normally be termed the R1 field. M1 is compared, as will be explained shortly, with the

[1]From this point forward, when instructions are defined, we will show the full operand so that the instruction can be completely specified. This is done with the tacit assumption that the programmer will usually prefer to let the assembler assign bases and displacements. The instruction class will also be shown in parentheses to the left of the instruction.

condition code. If a match is made, a branch is made to either the second operand (RX) or to the contents of the second operand (RR). Let us first discuss the condition code. It is set by compare instructions, certain register load and shift instructions, addition or subtraction type instructions and input-output operations and remains set until an instruction is executed which changes it. Testing the condition code does not change it. The condition code comprises bits 34-35 of a very important 64-bit register called the Program Status Word. This register has a great many uses which will be described in successive chapters. Since the condition code has two bits, there are four possible settings 00, 01, 10 and 11 or 0, 1, 2, and 3, respectively. These settings are matched with the four bits of M1 (bits 8-11 of either instruction) as follows

CC Setting		M Bit
Binary	Decimal	
00	0	8
01	1	9
10	2	10
11	3	11

As an illustration, if the M field was 0100 (bit 9 = 1) and the condition code was 01, a match would be made and the branch taken. In fact, M could be 0111 and the branch would still be taken since this implies branch on condition code setting 1 or 2 or 3. As a specific illustration of how this applied, refer to Fig. 3-10 which illustrates the various settings of the condition code for the Compare instruction.

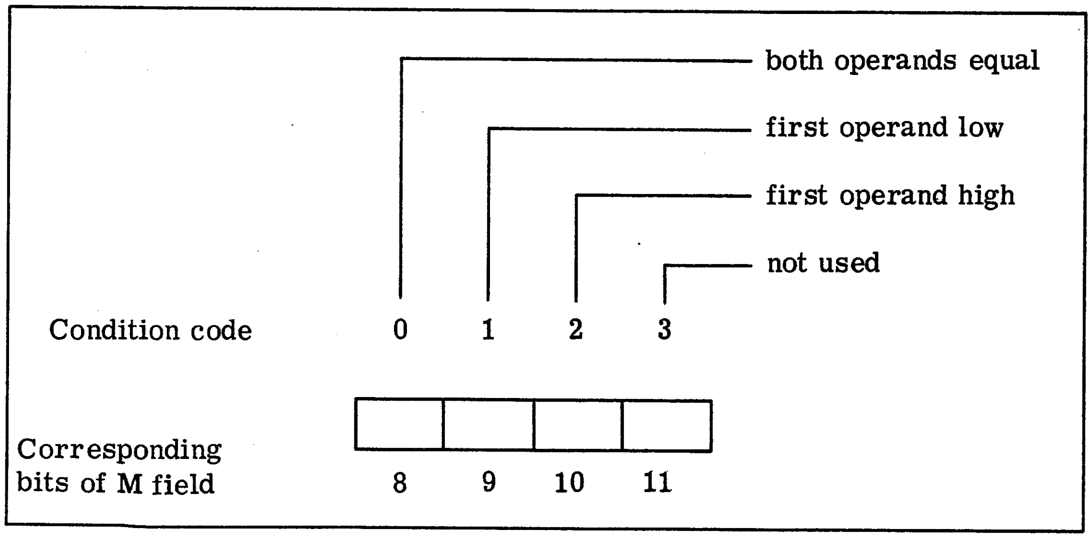


Fig. 3-10 Condition Code for Compare Instruction

To branch to GO on first operand high, we would write

```
BC 2, GO
```

since the match we are seeking requires a 1 in bit position 10, that is, M = 0010 or 2. To spare the programmer the inconvenience of remembering the different M values, the assembler will decode the instruction

```
BH    GO
```

into its equivalent above. Since BH is not a part of the 360 instruction set, it is termed an extended mnemonic code. Figure 3-11 shows a complete table of extended mnemonic codes. As a further illustration, if we wished to detect the condition not equal, M would be 0111, or 7. The instruction for this is

```
BC    7, NOTEQL
```

or in its extended mnemonic equivalent

```
BNE    NOTEQL
```

EXTENDED CODE	OPERAND	MEANING	MACHINE INSTRUCTION
B	D2 (X2, B2)	Branch Unconditional	BC 15, D2 (X2, B2)
BR	R2	Branch Unconditional (RR Format)	BCR 15, R2
NOP	D2 (X2, B2)	No Operation	BC 0, D2 (X2, B2)
NOPR	R2	No Operation (RR Format)	BCR 0, R2
		USED AFTER COMPARE INSTRUCTIONS	
BH	D2 (X2, B2)	Branch on High	BC 2, D2 (X2, B2)
BL	D2 (X2, B2)	Branch on Low	BC 4, D2 (X2, B2)
BE	D2 (X2, B2)	Branch on Equal	BC 8, D2 (X2, B2)
BNH	D2 (X2, B2)	Branch on Not High	BC 13, D2 (X2, B2)
BNL	D2 (X2, B2)	Branch on Not Low	BC 11, D2 (X2, B2)
BNE	D2 (X2, B2)	Branch on Not Equal	BC 7, D2 (X2, B2)
		USED AFTER ARITHMETIC INSTRUCTIONS	
BO	D2 (X2, B2)	Branch on Overflow	BC 1, D2 (X2, B2)
BP	D2 (X2, B2)	Branch on Plus	BC 2, D2 (X2, B2)
BM	D2 (X2, B2)	Branch on Minus	BC 4, D2 (X2, B2)
BZ	D2 (X2, B2)	Branch on Zero	BC 8, D2 (X2, B2)
		USED AFTER TEST UNDER MASK INSTRUCTION	
BO	D2 (X2, B2)	Branch if Ones	BC 1, D2 (X2, B2)
BM	D2 (X2, B2)	Branch if Mixed	BC 4, D2 (X2, B2)
BZ	D2 (X2, B2)	Branch if Zeros	BC 8, D2 (X2, B2)

Fig. 3-11 Extended Mnemonic Codes

The Branch Conditional instruction can be used to generate an unconditional branch, that is, a branch which will always be taken by using an M of 1111 so that a match will be made for any setting of the condition code. The unconditional branch instruction we have been using in the form B UNCON is actually the extended nmemonic code for BC 15, UNCON. In a similar fashion, a No Operation can be generated by using an M of 0000. Execution of this instruction will have no effect on the sequential execution of the program; when it is encountered, the branch cannot be taken and the computer proceeds to the following instruction. There are occasions when it is required to leave two or four byte units of space in the instruction stream; the extended instructions

NOP	D2(B2,X2)	(RSX)
NOPR	R2	(RR)

accomplish this. They are equivalent to BC 0, D2(B2,X2) and BCR 0, R2, respectively. Before leaving the subject of branch instructions, it is of interest to note how the 360 actually performs branches and normal sequential processing. One of the functions of the Program Status Word (PSW) is to indicate the location of the current instruction as well as its length (i. e. 2, 4, or 6 bytes). When the current instruction is completed, if it is not a branch, its length is added to the instruction location to give the location of the next instruction in sequence. If the current instruction is a branch, and the branch condition is met, the branch address will be entered into the PSW. In this way, the next instruction to be processed will be taken from the storage location specified by the branch address.

In the discussion of decimal arithmetic, we mentioned overflows which occur when a sum or difference is too large for the storage space allotted to it. Overflow will set the condition code and can be detected by inserting the extended instruction BO after each arithmetic instruction which may cause an overflow. However, since arithmetic instructions occur so frequently, this technique will require a significant number of additional instructions in the program. The situation could almost be handled automatically since similar action would be taken on every overflow. The 360 does provide for automatic handling of overflows through its interrupt system. There are a number of conditions which can give rise to an interrupt but at this point, we will restrict ourselves to the class of interrupts called program interruptions. These include incorrect operation codes, incorrectly aligned operands as well as overflows. When an interrupt occurs, the program status word (PSW) is automatically stored in a specific location in lower storage. For a program interruption, the location is bytes 48 through 55. Immediately after the current PSW is stored in which is called the old PSW location another double word is automatically loaded into the PSW register from a location which depends on the interrupt class. For program interruptions, this location is bytes 104 through 111. Figure 3-12 gives a schematic of the process. The PSW contains an Interruption code in bits 16-31 which identifies the interruption cause.

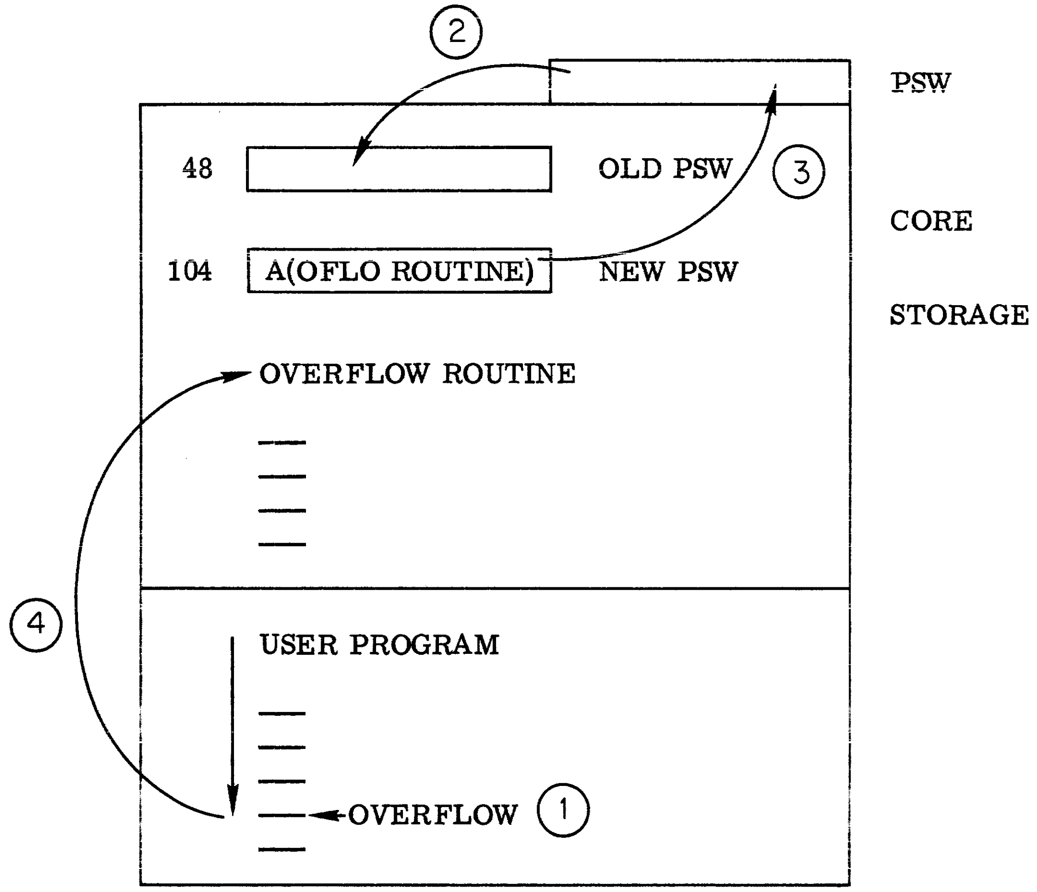


Fig. 3-12 Overflow Interruption. (The encircled numbers indicate the sequence of events.)

Note that since the PSW contains the address of the current instruction, if the new PSW location contains the address of the interrupt handling routine, when it loaded the effect will be a branch to the interrupt handling routine. Usually three possible ways of handling interrupts are provided to the user of one of the 360 operating systems. These are: Abort -- the job is terminated and the operator is informed of the reason via console type-out; Dump and Abort -- The contents of all GPR, the old PSW, various tables used by the operating system, and the entire program are printed and the job is then terminated; User Option -- The user may supply the address of a subroutine which will be used whenever program interruptions occur. The choice between the three is determined by a control card which is interrogated at load time -- more about this in Chap 12.

3.9 Binary Arithmetic Instructions

As we have mentioned previously, the 360 has instructions for converting numbers from binary to decimal and from decimal to binary. The formats of these instructions are

CVB	R1, D2(X2, B2)	(RX)
CVD	R1, D2(X2, B2)	(RX)

for converting to binary (CVB) and for converting to decimal (CVD), respectively. The CVB operation converts to binary a packed decimal number occupying a double-word in storage. The result is placed in register R1. Notice that, whereas a double-word used for packed decimal data allows 15 decimal digits plus a sign, a 32-bit register can accommodate only numbers in the range 2,147,483,647 to -2,147,483,648 so that only nine plus digits can be converted to binary by a single CVB instruction. Negative decimal numbers are converted to their two's complement binary equivalents. The CVD operation performs the reverse operation by converting the (binary) number in R2 to a decimal number and storing the results in a double word in storage. These two instructions give us the capability of converting decimal input into binary values for computer processing and then converting the results back to decimal for subsequent printing.

As we have mentioned previously, one of the major advantages of binary computation is the time saved over decimal computation. Of course, this must be balanced against the binary/decimal conversions required. Table 3-2 shows comparative times for the 360 models 30, 40, and 50 in microseconds (millionths of a second abbreviated as us.)

Operation	30	40	50
CVB	250	60	28
CVD	220	64	30
AP (9 digits)	100	61	35
A (binary word)	40	12	4

Table 3-2 Approximate, Average Operation Times for 360 Models 30, 40, 50 in Microseconds (us.)

As an illustration, suppose a particular calculation on the 360 Model 30 involves ten additions followed by printing of two numbers. If the data file is in binary, the additions will take 10 x 40 = 400 us plus two conversions to decimal, or 400 + 2 X 220 = 840us. If the file is in decimal format, no conversions are required and so the time is 10 x 100 = 1000 us, so that in this instance, binary arithmetic is faster by 160 us = .00016 sec. Naturally, it will take a good many repetitions of this calculation for the time saving to be noticeable; but then, this is what computers are for.

Once the data is converted to binary, arithmetic operations will be performed in the 16 GPR. A variety of instructions are provided for loading these registers. The basic ones are Load (L), Load Register (LR) and Load Half-word (LH). Their formats are

L	R1, D2(X2, B2)	(RX)
LR	R1, R2	(RR)
LH	R1, D2(X2, B2)	(RX)

The Load instructions loads the full word at its second operand address into R1, Load Half-word loads a half-word into the low order 16 bits of R1 and propagates the sign through the high order 16 bits. Load Register loads R1 with the contents of R2. As examples

```
          L     R,A                 32-BIT WORD
          L     10, =A(CON+4096)    32-BIT ADDRESS CONSTANT
          LH    2,B                 16-BIT HALF-WORD
          LR    11, R2              32-BIT WORD
          —
          —
          —
A         DS    1F
B         DS    1H
R         EQU   8
R2        EQU   2
```

Additional load instructions are provided for testing and changing the signs of data in registers. Load Complement Register (LCR, Load Positive Register (LPR) Load Negative Register (LNR), and Load and Test Register (LTR). Their formats are

LCR	R1, R2	(RR)
LPR	R1, R2	(RR)
LNR	R1, R2	(RR)
LTR	R1, R2	(RR)

LCR moves into R1 the complement of the number is in R2, that is, effectively changes the sign; LPR takes the number in R2, makes it positive and stores the result in R1; LTR moves the number in R2 into R1 and sets the condition code according to the sign (plus, zero, negative) of the number. In all of these instructions, R1 and R2 may reference the same GPR.

Worked Example

3-4

```
         L    2,=F'10'     R2   CONTAINS + 10
         LH   4, MFIVE     R4   CONTAINS -5
         LCR  3,2          R3   CONTAINS -10
         LPR  5,4          R5   CONTAINS +5
         LNR  6,5          R6   CONTAINS -5
         LNR  7,6          R7   CONTAINS -5
TEST     LTR  7,7          CONDITION CODE SET NEGATIVE
         BP   TEST1
         BM   TEST2        THIS BRANCH WILL BE TAKEN
         BZ   TEST3
MFIVE    DC   H'-5'        COULD BE DEFINED AS LITERAL
```

Note that the 16 high order bits of GPR4 will be filled with 1's since MFIVE is negative. Note also the use of LTR at TEST to set the condition code based on the contents of GPR7.

Storing the contents of registers in core storage is the reverse operation of loading. The following store instructions are provided:

```
ST      R1, D2(X2, B2)     (RX)

STH     R1, D2(X2, B2)     (RX)
```

The Store instruction (ST) stores all 32 bits of register R1 at the word specified by the second operand; the Store Half-word instruction stores the low order 16 bits of R1 into the half-word at the second operand location. Both of these operations leave the contents of R1 unchanged.

For completeness, we will review the Load Multiple (LM) instruction introduced in Sec. 3.4 and describe its reverse operation, Store Multiple (STM).

```
LM      R1, R2, D2(B2)     (RS)

STM     R1, R2, D2(B2)     (RS)
```

The set of registers started with R1 and ending with R2 are either loaded from (LM) or Stored at (STM) the location specified by the second operand. The registers are operated on in ascending order, starting with R1 and ending with R2, wrapping around 0 if R2 is less than R1. As examples:

```
            LM     8,11,ACON
            STM    12,1,SAVER
            —
            —
            —
   ACON     DC     A(GO+4096),A(60+8192),A(GO+12288),A(GO+16384)
   SAVER    DS     6F
```

The STM will store GPR 12, 13, 14, 15, 0, 1, in that order, at locations SAVER through SAVER +23.

The operations of addition and subtraction are similar enough to treat them together. In addition, the second operand is added to the first and the sum replaces the first operand. Half-word operands are effectively expanded to full-words prior to being added to a register. Subtraction operations subtract the second operand from the first by complementing the first operand; the difference, replaces the first addition.

The condition code is set to distinguish positive, negative, zero or an overflow result. Positive overflows give a negative result and negative overflows give a positive result as we discussed in the section on binary arithmetic. For both classes of operations, the second operand is unchanged. The various instructions follow with half-word operations indicated by an H in the program code mnemonic, A indicates addition, S, subtraction and R, a register-to-register operation.

A	R1, D2(X2, B2)	S	R1, D2(X2, B2)	(RX)
AR	R1, R2	SR	R1, R2	(RR)
RH	R1, D2(X2, B2)	SH	R1, D2(X2, B2)	(RX)

Worked Example

3-5

```
 TEN     DC    F'10'
 FIVE    DC    H'5'
 START   L     2,TEN      R2 = 10
         LH    3,FIVE     R3 = 5
         LNR   3,3        R3 =  -5
         AR    2,3        R2 = 10-5 = 5,  R3 =  -5
         LR    4,2        R4 = 10, R2 = 10
         SR    4,3        R4 = 10-(-5) = 15,  R3 =  -5
         AH    4,FIVE     R4 = 15 + 5 = 20
         S     4,TEN      R4 = 20 - 10 = 10
         BP    PLUS       THIS BRANCH WILL BE TAKEN
         BM    MINUS
         BZ    ZERO
         BO    OFLO
```

Multiplication and division are more complicated than addition or subtraction and so, they will be treated separately. Both require a pair of adjacent registers, termed an even-off pair which comprises an even-numbered register and the adjacent higher odd-numbered register such as 4-5, or 8-9.

The instruction formats for full-word operand multiplication are

```
 M     R1,D2(X2,B2)     (RX)

 MR    R1,R2            (RR)
```

The multiplier is the second operand; the multiplicand is in the odd register of the even-odd pair specified by R1 which must always be even; the initial contents of the even register are ignored, unless it contains the multiplier. The double word product replaces the multiplicand in the even-odd register pair. The sign of the product is determined by the signs of both operands: If both operands are of the same sign, the product will be positive; if the operands have different signs, the product will be negative. The significant part of the product will usually occupy 62 or fewer bits and only when both operands are the maximum negative number will 63 bits be required. The number of significant bits in the product cannot be larger than the sum of the number of significant bits in the multiplier and the multiplicand. For instance, two 15-bit numbers when multiplied together will give a product which is not larger than 30 bits. When the product is negative, the sign bit will be extended right until the first significant product digit is encountered. The condition code is unchanged and overflow cannot occur.

Worked Example

3-6 A quantity, HRS, is multiplied by RATE to determine GRPAY. It is assumed that GRPAY can be contained in a full-word.

```
               L      5,HRS       MULTIPLICAND
               M      4,RATE      MULTIPLIER
               ST     5,GRPAY     PRODUCT
               —
               —
      HRS      DS     1F
      RATE     DS     1F
      GRPAY    DS     1F
```

If GRPAY were greater than 31 bits, we would have to store both GPR4 and 5. A 32-bit product implies that bit position 0 (the sign position) of GPR 5 is a 1 and hence the product will show up as negative even though both operands are positive. For an illustration, see Example 3-7, below. This is one of the pitfalls of working with complement numbers. To overcome these potential difficulties, the programmer must be well informed about the ranges of his data and wherever necessary, make validity tests.

For half-word multiplications, we have

```
MH     R1,D2(X2,B2)     (RX)
```

Here, the multiplier is the half-word second operand. The multiplicant is in R1 and the low order 32 bits of the product replaces R1 which may be even or odd. If the product is greater than 32 bits, the high order bits will be lost. The condition code is unchanged.

Worked Example

3-7 The following example gives an illustration of how a 32-bit or larger product can give an incorrect sign. The sequence of instructions:

```
LR     5,=F'87654'
M      4,=F'87654'
ST     5, PROD
```

will give a result of 5858830849 which is too large to fit into GPR5. When GPR 5 is stored into PROD, PROD will contain -6710876. The reason for this strange state of affairs can be found if we examine the contents of GPR 4 and 5 in hexadecimal. This is 1C9F4B0A4, nine hex digits. As a result the leading digit in GPR5 is C, or 1100. Therefore, GPR5 is negative since its sign bit is one. The solution to this difficulty is to know in advance that PROD will require, in general, two registers and then store them with the instructions

```
STM        4,5,PROD
```

where PROD will be defined as a double-word. In Worked Example 3-14 we will present a technique for converting a double word to decimal. This example should make clear the importance of the programmer's knowing the size of the numbers that his program will have to handle.

Before we discuss division, we will present the Shift instructions. These allow the contents of registers to be shifted right or left as desired and are particularly important when dealing with fractional numbers. The Shift instruction formats follow. The letter A is appended to the mnemonics to distinguish these arithmetic shifts from logical shifts which will be discussed in Sec. 5.2.

SLA	R1, D2(B2)	(RS)
SLDA	R1, D2(B2)	(RS)
SRA	R1, D2(B2)	(RS)
SRDA	R1, D2(B2)	(RS)

The low-order six bits of the second operand address, D2 plus the contents of GPR B2, gives the amount of right (R) or left (L) shift for R1. The double register shifts, SLDA and SRDA reference even-odd register pairs and treat them as 64-bit words with one sign in position 0 of the even register; position 0 of the odd register is treated as an ordinary data bit. Any bits right-shifted out of the register or register pair are lost; the sign bit is propagated through the vacated positions at the high order end on right shifts. On left shifts, when a bit different from the sign bit is shifted our of bit position 1 of the even register an overflow occurs, the sign bit, itself, is not shifted. All shift instructions set the condition code; this property can be used to test the sign of a double-word result by specifying a zero length shift. Also, it will be useful for the reader to keep in mind that left shifting is equivalent to multiplying the contents of a register by two for each position shifted. Similarly, right shifting is equivalent to dividing by two for each position shifted. As a final point on shifting, the specification of a base register allows the amount of shift to be varied in a program without modifying the shift instruction itself. The amount by which the shift is to be changed is simply added to the base register specified in the shift instruction. If no base register is specified, the displacement field gives the shift amount directly.

<u>Worked Example</u>

3-8

```
         LR     4,=3
LOADB    LR     6,=B'1010101'           GPR6 = 1010101
         SRA    6,2(4)                  GPRG = 10
         SLA    6,5                     GPR5 = 1000000
         LM     8,9,CON
         SRDA   8,0                     SET CONDITION CODE
         BP     PLUSCON
         BZ     ZEROCON
         BM     MINCON                  THIS BRANCH WILL BE TAKEN
CON      DC     D'-816543216798502'
```

The instruction at LOADB loads 1010101 into GPR6. When this register is shifted right as indicated by the address 2(4), which is 2 plus the contents of GPR4, or 5 places, the result if 10; the bits 10101 are lost.

The instruction formats for division are

```
DR     R1,R2             (RR)
D      R1,D2(X2,B2)      (RX)
```

The dividend is a 64-bit signed integer in an even-odd register pair specified by R1 which must be even. The divisor is a 32-bit signed integer specified by the second operand. The remainder and quotient, both 32-bit signed integers, replace the dividend in the even and odd registers, respectively. The remainder and dividend have the same signs and the sign of the quotient will be positive if both dividend and divisor have the same sign, otherwise the sign of the quotient is negative. Note that the even register of the dividend pair is not ignored during division as it is for multiplication. As a result, if the dividend can be contained by the 32 bits of the odd register, the even register must be set to zero or to ones depending on the sign of the dividend. The condition code is unchanged by division. However, if the divisor and dividend are of such a relative magnitude that the quotient cannot be contained in 31 bits plus sign, a divide exception or error occurs which gives an interruption; the division will not be attempted. In the following paragraphs, we will establish a rule which will allow the programmer to determine when divide exceptions can happen.

Let us now establish rules for scaling, that is, for treating decimal points in binary operations. The important thing to keep in mind is that binary operations are performed by the 360 hardware on integer numbers. Therefore any understanding about decimal points or more properly, binary points, must be a "behind the scenes" one. It is up to the programmer to arrange his data to provide for the proper number of binary points and to interpret the results in the correct way. The hardware will know nothing of his plans; it is capable of handling integers, only. In the discussion which follows, it will be helpful to set up a simple notation for fractional numbers. We will use capital letters to indicate the number of binary places to the left of the binary point in a given quantity and small letters for the number of places to the right; the letters will be separated by a period. Multiplication is the key operation to consider. If the operands in a multiply operation are A. a and B. b, then the number of bits to the left and right of the binary point in the product is given by the <u>Multiplication Scaling Rule</u>.

$$(A.\,a) \times (B.\,b) = (A + B).\ (a + b)$$

This is just a restating in our adopted notation of the familiar technique for placing the decimal point in multiplication of fractional quantities. Consider the following examples (the decimal quantities in parentheses are not equivalent to the binary numbers; they are examples of the above notation).

```
   11.01   (2.2)            11.01   (2.2)
   10.01   (2.2)            11.01   (2.2)
   -----                    -----
   1101                     1101
110100                     11010
-------                   1101
1110101    (3.4)          --------
                          10101001  (4.4)
```

The first example brings out an exception to the rule. That is, it is possible to have one less position to the left of the decimal point than the sum of both operand places to the left. However, it is safer to assume that the number of places to the left will be given by our multiplication scaling rule above. In particular, we will make this assumption in the discussion below on the number of positions to be allowed for the quotient.

For division, we have a slightly different situation in the matter of scaling. We are given a dividend (D. d) and a divisor (S. s) and we require a certain number of fractional places in the quotient (Q. q). This will usually require that the dividend be shifted. The rule for determining the amount of shift is

$$\boxed{\text{SHIFT} = q + s - d}$$

Since the dividend equals divisor multiplied by quotient, the number of places after the dividend point is q + s which must equal d plus any amount the dividend is shifted. This rule applies irrespective of the relative magnitudes of the dividend and divisor. If SHIFT is positive, we shift the dividend this number of places to the <u>left</u>; if SHIFT is negative, the direction of shift is to the right. As an example, suppose we wish to do the calculation, AVG = UNITS/PRICE. If UNITS has the binary place form (10. 10), PRICE, (8. 10) and we require 12 places after the binary point in AVG, then

$$\begin{aligned} \text{SHIFT} &= q + s - d \\ &= 12 + 10 - 10 \\ &= 12 \text{ places to the left} \end{aligned}$$

The following instructions will carry out this calculation:

```
LR      4, UNITS      DIVIDEND(10. 10)
SRDA    4, 32         SHIFT + PROPAGATE SIGN
SLDA    4, 12         SHIFT LEFT 12 PLACES
D       4, PRICE      DIVISOR (8. 10)
ST      5, AVG        QUOTIENT (2. 12) or (3. 12)
```

Both shifts above are not always necessary. A double right shift of 20 would suffice provided the remaining bits in GPR5 were set to zero. If this could not be guaranteed, our method would then be safest. Note that the SRDA propagates the sign bit, if any, through GPR4. This is a necessary precaution for negative dividends. If the divisor is less than the dividend, the number of places to the left of the binary point in the quotient is D-S or D -S +1, at most. The total number of bits required by the quotient is then

$$\text{Quotient bits} = D - S + 1 + q$$

If the number of bits required by the quotient is greater than 31, a divide exception will occur. In the above example, the quotient will occupy 10 - 8 + 1 + 12, or 15 bits. The binary point will be between the 12th and 13th positions from the low order end of the full word AVG. If a quantity BASE in the binary place form (5. 6) were to be added to AVG to give TOTAL, it would have to be shifted left six places prior to being added to align its binary point with that of AVG. The coding for this is:

```
LR      7, TOTAL     BINARY PLACE FORM (5. 6)
SLA     7, 6         ALIGN POINT TO (5. 12)
AR      7, 4         ADD TO (3. 12)
ST      7, TOTAL
```

Table 3-3 below summarizes the scaling rules.

Exercise 3-8 Compute the amount and direction of shift for each of the following and also indicate if a divide exception will occur. Answers are given at the end of this chapter.

Multiplication Scaling Rule

$(A.a) \times (B.b) = (A + B).(a + b)$

Division Scaling Rules

$$S.s \overset{Q.q}{\overline{\big)\, D.d}} \quad + \quad \frac{R.r}{S.s}$$

The binary form of the remainder is the same as the divisor.

1. Quotient Scaling Rule (Divisor(S) less than dividend(D))

   SHIFT $= q + s - d$,

   Places occupied by quotient $= D - S + 1 + q$

   Binary form of quotient $(D - S + 1 \,.\, q)$

2. Quotient Scaling Rule (Divisor greater than dividend)

   SHIFT $= q + s - d$

   Places occupied by quotient $= q$

   Binary form of quotient $(0.q)$

   Note that the first S-D places of q will be zero.

3. Division Exception Rule

   A division exception and interrupt will occur if the number of bits required for the quotient is greater than 31.

Table 3-3 Scaling Rules

Dividend form	Divisor form	Quotient Binary Places (to left of binary point)
(12.15)	(8.6)	13
(30.12)	(2.5)	8
(2.15)	(3.5)	4

Worked Example

3-9 As an example of integer division consider the following coding

```
                SR    4,4          ZERO GPR4
                LR    5,DIVIDND    DIVIDND = 1052
                D     4,DVSOR      DVSOR =  -17
                ST    5,QUOT       QUOT = -61
                ST    4,REM        REM = 15
                —
                —
DIVDND          DC    F'1052'
DVSOR           DC    F' -17'
```

Worked Example

3-10 This example will introduce the important topic of rounding. Suppose the dividend in a particular calculation has the binary place form (10.20), the divisor, (9.4) and 12 places are required after the point in the quotient. The shift amount is $q + s - d = 12 + 4 - 20 = -4$, or four places to the right. This means that four places of the dividend will be lost. Rather than lose this information completely, we will round the dividend. If the dividend was a decimal number, say 10716345, with the last four digits to be dropped, it would be rounded to 1072. In effect, what we have done was add 5000; since the amount to be truncated is 5 or greater (actually 6) in the fourth place, the addition carries a one into what will be the low-order position of the remaining number. We can generalize a rounding rule from this: For decimal numbers, add a 5 into the high order position of the part to be truncated; for binary numbers, add a 1. If the number to be rounded is negative, then the roundoff quantity should be subtracted. As an example of binary rounding, suppose the dividend is 11001101 and we wish to round and truncate the last four bits. We add 1000 to the dividend and then shift right four places. The result is

```
11001101
    1000
--------
11010101
```

Rather than round and shift before division, we will leave the dividend unshifted and will operate instead on the quotient after the division. In this way, we are guaranteed maximum accuracy. Instead of 12 binary places in the quotient, we will have 16. The coding for this is

```
         LR     4,DIVDND      BINARY PLACE FORM (10.20)
         SRDA   4,32          CLEAR GPR4 + PROPAGATE SIGN
         D      4,DVSOR       BINARY PLACE FORM (9.4)
         SLDA   4,0           SET CONDITION CODE FOR QUOTIENT
         BM     SUBTR
         AH     5,ROUNDB
         B      SHIFT
ROUNDB   DC     BL2'1000'
SUBTR    SH     5,ROUNDB      FOR NEGATIVE QUOTIENTS
SHIFT    SRA    5,4           TRUNCATE QUOTIENT
         ST     5,QUOT        BINARY PLACE FORM (2.12)
```

As a check, note that the total number of places required for the quotient, D - S + 1 + q = 10 - 9 + 1 + 16 = 18 and therefore, no divide exception will occur. Halfword operations are used for rounding because they allow a two-byte constant to be used, thereby saving two bytes over a full word constant. The constant ROUNDB could be defined as H'8' or XL2'8'; the binary constant seemed more natural in this example, however.

Worked Example

<u>3-11</u> Assume that the divisor in a calculation has the binary place form (14.16), the dividend, (10.15) and 12 significant bits are required in the quotient. To get 12 places in the quotient, the dividend must be shifted right 12 + 16 - 15 = 13 places. However the first 14 - 10 = 4 places of the quotient will be zero so that only eight significant bits will result for a shift of 13 places. To get four more significant bits, the dividend must be shifted right another four places. The quotient will then have 12 + 4 places, the first four being zero.

Worked Example

<u>3-12</u> This example will present a technique for performing operations on fractional numbers expressed as binary integers. The calculation, compounding interest at 4%, is AMT=PRINCxRATE rounded to two decimal points. As a specific example, PRINC = $11.38 and RATE = 1.04. We could also multiply PRINC by 4% and then add the product to PRINC. However, multiplying by 1.04 saves an addition. The coding is

```
                LR      5, PRINC
                M       4, RATE
                AH      5, =H'50'
                D       4, =F'100'
                ST      5, AMT
                —
                —
PRINC           DC      F'1138'
RATE            DC      F'104'
```

The product of PRINC and RATE is 118352, adding 50 brings this to 119402. Note that the multiplication clears the even register of the pair. The next step after the rounding is truncating the low-order two digits. However, we cannot just shift right two places or any number of places, for that matter. The problem with shifting is that two decimal places is not equal to a whole number of binary places. The only remaining approach is to divide by 100. This will give AMT, rounded to the nearest cent. This technique of using integer quantities where exact amounts are required, such as payroll or interest calculations, is strongly recommended. It has the disadvantage of an added division operation to obtain the equivalent of a truncation but is much safer than the approach to be described in the following example. Incidentally in a real program, PRINC, and probably RATE also, would be supplied as input items and not as constants. We took the present approach for the sake of illustration only.

Worked Example

3-13 We will present the previous example using a binary fraction for RATE carried out to 14 binary places. We have

```
RATE            DC      FS14'1.04'
PRINC           DC      F 1138
HALF            DC      FS14'.5'
                —
                —
                LR      5, PRINC
                M       4, RATE
                A       5, HALF
                SPA     5, 14
                ST      5, AMT
```

RATE is defined as a binary constant with 10 places after the binary point. It will be multiplied by PRINC expressed in cents so that the rounding quantity must be .5 cents expressed in the same binary scaling as RATE. A right shift of 14 places after the rounding should give AMT to the nearest cent. But it does not! The result is 1183, not 1184. How has this happened? The difficulty can be traced to the fact that 1.04 cannot be expressed exactly in any number of binary places. In fact, to 14 bit accuracy 1.04 is in binary approximately 1.0399783 so that multiplication by 1134 gives 11.83494949. We could get this calculation to work properly by extending RATE to more binary digits but it would fail for sufficiently large PRINC amounts. The moral of the story is when amounts must be calculated exactly, use integer forms for the numbers and then divide at the end.

Again, we do not mean to suggest that binary operations are so complex and error prone as to be not worth bothing about. Binary operations have limitations as suggested above and in the discussion on complementing in Example 3-7. The programmer who is aware of these limitations can best make the judgment on when to use the decimal instruction set and when to take advantage of the time saving inherent in binary operations.

Worked Example

3-14 In this example we present a technique for converting double-word binary numbers to decimal using the CVD operation. Let us start by considering some of the properties of double-word numbers. First, they use only one sign bit, namely bit position 0 of the high-order word. Second, the low-order word can always be considered as positive and added to the first. To see that this is true, we will examine 8-bit "double"-words. Consider the double-word 1111/1011. The low-order word, considered as positive is equal to 11. When the high-order word is complemented, we have -1. However, since the high-order word is left-shifted four places from the low-order word, it is 16 times greater (2^4) so that the double word has the value -1x16+11, or -5. As a check, if we do a double-"word" complement of 5, 00000101, we have 11111011 which is identical with what we began with above. Since the CVD will handle only 31 binary bits of data we will break the 63 bit-number into two 31-bit numbers and a 1-bit number in bit position 63. The high-order part, HO, will be converted and multiplied by 2^{32}, the low-order part, LO, by 2 and finally, the last bit, B, with value of 1 or 0 will be added to the sum of the other two parts. The formula for this is then

$$2^{32} \times HO + 2 \times LO + B$$

LO and B will be treated as positive (unsigned) numbers and HO will be treated as a signed number. In this way, we are assured of having the correct sign for the result. Following is the coding. The double word to be converted is in the GPR 2 and 3.

```
         CVD   2,DEC+8           CONVERT HO,STORE IN DEC+8
         SR    2,2               CLEAR GPR2 TO AVOID OVERFLOW
         SLDA  2,31              LO TO GPR2, B TO SIGN BIT OF GPR3
         CVD   2,TEMP            CONVERT LO, STORE IN TEMP
         MP    TEMP,TWO
         LTR   3,3               SET CONDITION CODE FOR B
         BP    MULT              IS B=O, IF YES GO TO MULT
         AP    TEMP,ONE          B=1, ADD ONE TO TEMP
MULT     MVC   DEC(8),ZERO       CLEAR HI-ORDER HALF OF DEC
         MP    DEC,TWO32
         AP    DEC,TEMP
         ——
         ——
TWO32    DC    PL6'4294967296'
TWO      DC    PL1'2'
ONE      DC    PL1'1'
TEMP     DS    1D
DEC      DS    2D
ZERO     DC    8x'00'
```

As starting point, note that the maximum size for a 63-bit integer is 20 decimal digits so that DEC could have been defined as an 11 byte field. However, the CVD requires a double-word address for its second operand and defining DEC as two double-words is an easy way to ensure that DEC will contain a double word address within its field. When HO is converted, it is stored in DEC+8 (through DEC+15) to put it in the low order half of DEC. This is necessary for proper alignment with LO and B after HO has been multiplied by 2^{32}. The logic in separating LO and B is to shift LO into bits 1-31 of GPR2. However, GPR2 must be cleared prior to the shift, otherwise bit of GPR2 would still remain in the sign position after a double register left shift of 31. (In addition, if a bit different from the sign bit of GPR2 was shifted out, an overflow would occur.) This double shift will store B in the sign position of GPR3. As a result, if B=1, GPR3 will be negative. The LTR 3,3 sets the condition code prior to being tested by the BM MULT instruction. The high order part of DEC is cleared by the MVC. This is a necessary precaution prior to the multiplication because the DS only reserves storage for DEC, it does not set it to zero. Notice that the implicit length of DEC, 16 bytes, must be overridden if we are to avoid wiping out HO which is stored in the eight low order bytes of DEC. This is done by specifying an explicit length of 8 bytes; since the symbol DEC refers to the leftmost byte, this allows only the leftmost eight bytes to participate in the move.

3-15 Fraction Conversion. Conversions between binary and decimal fractional numbers present interesting problems. Since the conversion hardware can handle integers, only, additional logic must be added to conversion routines to handle fractional numbers. Let us first consider conversion of a packed decimal number to a binary number. The integer portion of the decimal number can be converted directly using the CVB instruction. We will concentrate on the remaining N fractional digits. For the moment we will assume that N is nine or less, the number of digits which can be converted to binary in one application of the CVB instruction. We may consider these N fractional digits as N integer digits divided by 10^N. The integer digits can then be converted to binary and the result is then divided by the binary equivalent of 10^N. As a specific example, consider five fractional digits in the double word FRAC. They will be converted to a 31-bit binary fraction by the coding below. Note that 10^5 requires 16 bits and that any five digits will convert to not more than 16 bits which will be less than 10^5. To get a 31-digit quotient, the 16-bit dividend must be shifted to include 31 additional low order positions. To verify that a divide check will not occur, we note that the number of places in the quotient $= Q+q = D-S+q = 31$. $(q = d - s = 31 - 0 = 31)$. We do not use the formula $Q = D - S + 1$ since the extra bit will occur only when the first $S + s$ digits (16 here) of the dividend are larger than the divisor. This is not the case here. The coding is

```
            CVB      2, FRAC       16 BITS + SIGN IN GPRS
            SRDA     2, 1          DIVIDEND NOW CONTAINS 31
*                                  BINARY PLACES (IN GPR3)
            D        2, TEN5
            ST       3, BFRAC
            -
            -
TEN5        DC       F'10000'
BFRAC       DS       F
FRAC        DS       D
```

To convert more than nine digits, the first nine can be handled as above but dividing by 10^9. The remaining digits (N-9) are converted and then divided by 10^N.

To convert a binary number containing a fraction to decimal, the integer portion can be handled with a straightforward CVD. The binary portion can then be converted by deciding first, how many decimal digits are required. A good rule of thumb is one decimal digit for every 3. 3 binary bits. With N as the number of decimal digits, the fraction is then aligned in an odd GPR so that its point is at the extreme left. When it is multiplied by 10^N, the even register, left shifted by one, contains the N fractional digits in binary form. When these are converted to decimal, we have the desired N decimal fractional digits.

As an example, suppose we have the binary fraction $.11100001 = .875$. This is worth approximately 2 decimal digits. Assuming a hypothetical computer with eight data bits plus sign per word, multiplication by $10^2 = 1100100_2$

gives the following result, stored in an even-odd register pair:

0	0 0 1 0 1 0 1 1	1	1 0 0 0 0 0 0 0
s	Even r	s	Odd r

Since the binary point of the multiplicand was originally immediately after the sign bit of the Odd register, the product will contain eight places after the point. A double left shift of one place will put the integer part of the product entirely in the Even register which becomes

0	0 1 0 1 0 1 1 1	1	0 0 0 0 0 0 0 0
s	Even r	s	Odd r

This is the binary equivalent of 87, the desired result. If rounding is required, we should examine the high order bit of the Odd register. (In a double word, this is a data bit, not a sign bit.) If this bit is one, add or subtract a one into the even register according to its sign. The coding below will convert a 23-bit binary fraction left justified in GPR3 into a seven digit decimal fraction. We will ignore rounding.

```
M         2,TENT
CVD       2,DFRAC
-
-
TENT      DC  F'1000000'
DFRAC     DS  D
```

The low order five digits of DFRAC contain the fraction with the decimal point assumed to the left of the first digit.

Answer to Exercise

<u>3-1</u>

9	1001	13	1101
10	1010	14	1110
11	1011	15	1111
12	1100	16	10000

Problems

3-1 What is the largest decimal number which can be represented in 16 bits and in 24 bits?

3-2 If the symbol X has the address value, 12462, and is covered by base register 11 with a base point of 9000, how much storage will be required for TABL defined as

```
          TABL          DC          A(X)
                        DC          Y(X)
                        DC          S(X)
```

3-3 What are the values of the constants defined in Prob. 3-2?

3-4 A table consists of seven 1-word entries. It is required to push down all entries in the table and move the seventh one to the first entry location. That is, the second entry will be in words three, and so forth. Write code to accomplish this.

3-5 Write code to evaluate the expression

$$A + (A - B)/2$$

where A and B are half-word numbers. The fractional result from the division should be discarded.

3-6 Modify the code for Prob. 3-5 so that the difference between A and B will always be treated as a positive number.

3-7 Modify the code for Prob. 3-5 so that the result is rounded to the nearest integer according to the rule that if A - B is odd, the result would be rounded up, if it is even, no rounding should be done.

3-8 Devise the technique to determine how many high-order and low-order zero bits a given binary attains. Write code to perform the appropriate counts on full-word and test your program.

3-9 A dividend has a binany place form (7.21), its divisor (5.6), and the quotiant is required to be rounded to 11 binary points. Write coding to do this.

3-10 Write a program to convert an 18-digit packed decimal number to a 64-bit signed binary number.

Chapter 4

INDEXING

4.1 Looping

One of the most important techniques in programming is looping, or iteration as it is sometimes called, which is the repetition of a given sequence of instructions a specified number of times. We have encountered this type of logic previously in the "Indian Problem" in Chap. 2. The problem consists of calculating the present value of a $24 deposit 340 years ago, at 3% interest, compounded annually. The basic calculation is multiplying the principal by 1.03 to compute the year end principal and then, repeating the process 339 more times. (Multiplication by 1.03 is the same as multiplying the principal by .03 to compute the interest, and then adding to the principal. Our approach saves a multiplication.) As an illustration of looping using registers, we will code the Indian Problem using binary fixed-word size numbers in contrast to the decimal approach in Chap. 2. We will keep count of the number of iterations through the loop by loading 340 into a register and then subtracting one for each iteration. At the bottom of the loop, this register will be tested for zero. If it is not zero, the program will branch to the top of the loop for an additional iteration and test; if it is zero, the loop is finished, or satisfied and the remainder of the problem can then be started. The principal will always be stored as a two decimal place number which is rounded after each multiplication. The coding follows.

```
         LM    2,3,YEARS
         L     5,PRINC
AGAIN    M     4,RATE            PRODUCT HAS 4 DEC. PLACES
         A     5,=F'50'          ROUND TO 2 DEC. PLACES
         D     4,=F'100'         "SHIFT" RIGHT 2 DEC. PLACES
         SR    2,3               SUBTRACT 1 FROM GPR 2
TEST     BP    AGAIN             IF GPR2 POSITIVE, GO TO AGAIN
STORE    ST    5,PRINC
         --
         --
YEARS    DC    F'340'
         DC    F'1'
PRINC    DC    F'2400'           24.00
RATE     DC    F'103'            1.03
```

The coding will cause the instructions between TEST and AGAIN inclusive, to be executed 340 times in succession. On the last iteration through the loop, GPR2 will be one, the instruction SR 2, 3 will reduce it to zero. At this point, the Branch on Positive at TEST will not occur. Sequential instruction execution will then proceed starting from STORE. The subtract-test sequence for this type of problem occurs so frequently that a single instruction is provided to deal with it, Branch on Count. Its formats are

```
BCT     R1, D2(X2, B2)    (RX)

BCTR    R1, R2            (RR)
```

The contents of the first operand are algebraically reduced by one. When the result is not zero, a branch is taken to the second operand address (RX) or to the contents of the second operand (RR). The branch address is set up prior to the subtraction; when GPR2 is zero, counting is performed but no branch is taken. Also, this instruction cannot cause an overflow nor does it change the condition code. The loop coding for the Indian Problem can now be written

```
AGAIN    M      4, RATE         START OF LOOP
         A      5, =F'50'
         D      4, =F'100'
TEST     BCT    2, AGAIN        END OF LOOP
```

The LM 2, 3, YEARS can be replaced by L 2, YEARS. Are any other changes in order?

As an aid to the discussion of looping throughout the remainder of this chapter, we will define the basic terms here. The index of a loop is the quantity which either counts the number of iterations directly, or is directly related to the count; the increment is the amount added to the index after each iteration; the initial value is the value of the index before or during the first iteration; the final value is the index value during the last iteration. The initial value, increment, and final value comprise the loop parameters. When the last iteration is completed, the loop is said to be satisfied. In the Indian Problem above, the loop index can be found in GPR2; the initial value is 340; the increment is -1 and the final value is 1. The general programming technique for handling loops is straightforward. To repeat a section of code N times starting with an instruction labeled GO, for instance, load a register with N immediately before GO, and place a BCT REG, GO immediately after the last instruction in the sequence.

Exercise 4-1 Referring to the following program segment, how many times will the code between GO and LASTA be executed? How many times will the instruction at LASTB be executed? When the instruction at FIRST is executed, what are the contents of GPR1 and 2?

```
                L       1,=F'3'
GOGO            L       2,=F'2'
GO              --
                --
                --
                --
LASTA           BCT     2, GO
LASTB           BCT     1,GOGO
FIRST           --
```

The code above illustrates a loop within a loop; we will have several occasions to use this type of logic again.

4.2 Indexing

Up to this point, we have considered operand addresses as remaining unchanged during the execution of a program. However, there are many instances where a program can be written more compactly and easily if some of the operand addresses can be changed while the program is being run. As an example, suppose an employee's weekly payroll record consists of eight words of data; the first word contains the gross pay and the other seven have various deductions. To calculate the net pay, each of the seven deductions must be subtracted from the gross pay. This could be done by the brute force approach of writing a subtract instruction for each of the seven deductions. Rather than write seven subtract instructions such as

```
                L       2,GRPAY
                S       2,DED
                S       2,DED+4
                S       2,DED+8
                S       2,DED+12
                S       2,DED+16
                S       2,DED+20
                S       2,DED+24
                ST      2,NETPAY
                --
                --
                --
GRPAY           DS      F
DED             DS      7F
NETPAY          DS      F
```

the coding could be shortened if we could write one subtract instruction and in some fashion loop through it seven times, each time adding four bytes to its operand. The 360 does indeed have this capability and it is obtained through the indexing feature of the RX class of instructions. To understand how this feature works, recall that in Sec. 3.4, we described storage addresses as being generated by adding the displacement field of an instruction to the contents of the GPR specified in its base field. When indexing is used, an extra addition comes into play: The contents of the GPR specified in the index field are added to the sum of base plus displacement. As an illustration, consider a general RX-instruction in the form

```
OP      R1,D2(X2,B2)
```

where X2 specifies the GPR to be used as an index register. The operand address is D2 + C(X2) + C(B2) where C( ) refers to the contents of a particular register. Suppose D2 = 4000, X2 = 2 and B2 = 3 where GPR2 contains 16 and GPR3 contains 8096. The instruction would have the form

```
OP      R1,4000(2,3)
```

with an operand address of 4000 + 16 + 8096 = 12112. Figure 4-1 gives a schematic of the process. To modify this address, say to increase it to 12116, we would add four to the index register, GPR2. We could obtain the same result by adding four to GPR3 but this would change the base value for all other instructions which used GPR3 as a base register. Or, we could arrange to add four to the displacement field but this would be going about it the hard way; the easiest approach is to do the addition in the index register. In the case of the payroll example above, the subtract instruction would be written

```
S       2,DED(XREG)
```

where XREG would be any one of the GPR excluding 0 and any other register used as a base. The use of GPR0 implies no indexing; similarly, when the programmer specifies no index register, the assembler will incorporate GPR0 into the instruction. (To write the subtract instruction without indexing, we have S 2, DED.) With this background, we can write the following code to handle the seven subtractions of the payroll program:

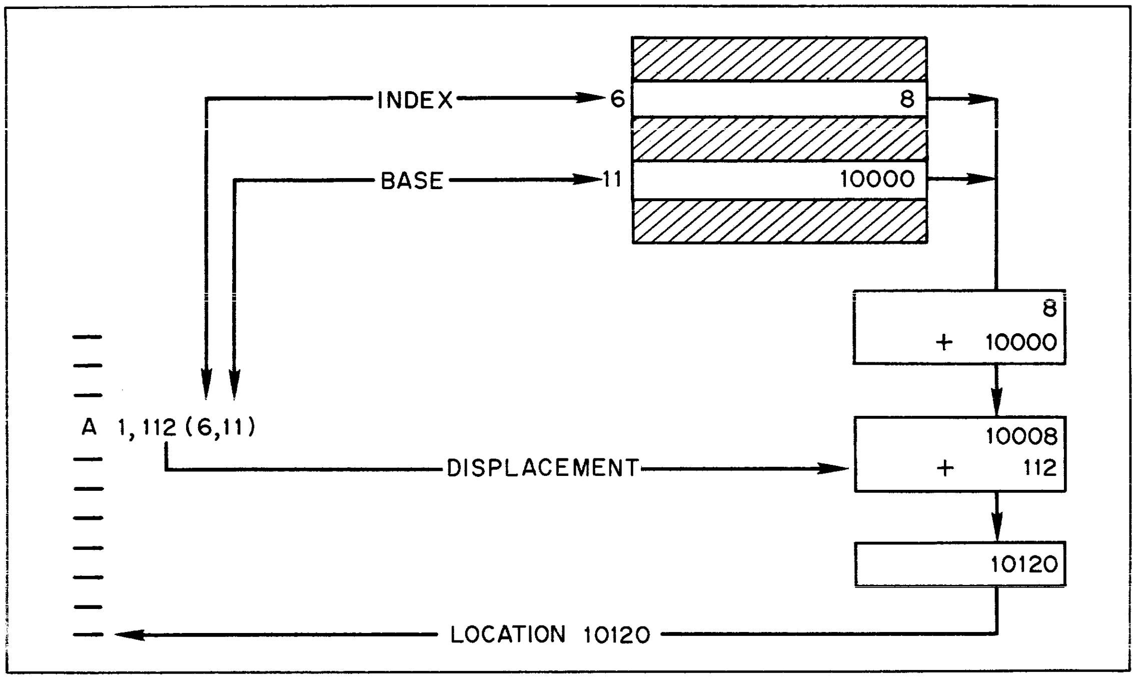


Fig. 4-1 Indexing -- The word at 10120 will be stored in GPR1

```
        SR      3,3         GPR3 IS XREG, CLEAR IT
        L       4,=F'7'     INITIALIZE GPR4 AS COUNTER
        L       2,GRPAY
SUBTR   S       2,DED(3)
        A       3,=F'4'     ADD 4 TO XREG
        BCT     4,SUBTR     TEST FOR 7 ITERATIONS
        ST      2,NETPAY
        --
        --
        --
GRPAY   DS      F
DED     DS      7F
NETPAY  DS      F
```

On the first pass through the loop, GPR3 = 0 so the first entry in DED will be subtracted; on the second iteration, GPR3=4 and we subtract the contents of DED +4; on the last iteration, GPR3=24 and the last entry, DED+24, is subtracted. While GPR3 is being incremented from 0 through 24, GPR2 is being stepped down from 7 to 0. The indexing logic here is different from the Indian problem in that we are counting by fours from 0 to 24. Since the BCT will only count down by 1's, an additional add statement is necessary. The programmer will encounter this type of problem frequently where a count is required to terminate on some value other than zero and is incremented or decremented, by a value other than one. The 360 has two instructions which do this, Branch on Index High and Branch on Index less than or Equal. Their formats are

BXH	R1, R3, D2(B2)	(RS)
BXLE	R1, R3, D2(B2)	(RS)

where R3 will usually reference an even-odd pair of registers. R3 contains the increment and the next higher odd register contains the final value, or comparand. R1 contains the current value of the index. Prior to starting the loop, R1 contains the starting value. When either instruction is executed, the contents of R3 are added algebraically to the contents of the odd register. The branch is taken if the index is higher than the odd register (BXH), or it is taken if the index is less than or equal to the odd register (BXLE). If the condition is not met, instruction processing proceeds sequentially after the BXH or BXLE. If the R3 register is odd, its contents will be used for the increment and the compare value. Overflow cannot occur with either of these instructions, the condition code is unchanged, and the branch address is determined prior to the incrementing and comparing. Following are some examples which show the successive values of the index, given the initial register contents shown below. As you can see from the successive index values, BXH is used for counting down and BXLE for counting up.

			Initial Contents				
BXH	1, 2, AGAIN	GPR1:	12	9	6	3	0
		GPR2:	-3				
		GPR3:	-3				
BXH	1, 2, AGAIN	GPR1:	11	8	5	2	-1
		GPR2:	-3				
		GPR3:	-3				
BXLE	1, 2, FIRST	GPR1:	0	2	4	6	8
		GPR2:	2				
		GPR3:	8				
BXLE	1, 2, FIRST	GPR1:	-2	0	2	4	6
		GPR2:	2				
		GPR3:	7				

In the first example GPR1 has the initial value 12 and will be stepped down to 0 in increments of -3. Note that if we wanted to go down to -3, GPR3 would have to contain a-4, -5 or -6, since the branch is taken only when the index is higher than the contents of 6PR3. A loop ending with the first instruction would be repeated five times. During the last cycle, the index would be zero and when the BXH is executed for the last time after the last iteration it would go to -3, register an equal compare with GPR3 and the branch would not be taken. Note that the contents of GPR1 would be -3 after the loop was satisfied. In the other three examples, GPR1 will contain -4, 10 and 8, respectively, after each loop is satisfied.

Note that in the first two examples, a register could be saved since GPR2 = GPR3. The instruction would be written BXH 1, 3, AGAIN. The programmer should always look for ways to save registers as it is possible to run out of registers in problems which require extensive indexing. When this happens, it is not necessarily disastrous as a single register can be put to multiple uses by loading it with the proper value for each usage and then storing this value prior to the next usage. These extra load and store instructions will increase the running time of a program, perhaps significantly, and should be avoided where possible. We will now apply the BXLE and BXH to the payroll problem. Using BXLE,

```
         LM      3,5,NDXCON
         L       2,GRPAY
SUBT     S       2,DED(3)
         BXLE    3,4,SUBT
         ST      2,NETPAY
         --
         --
         --
NDXCON   DC      F'0'         STARTING VALUE
         DC      F'4'         INCREMENT
         DC      F'24'        FINAL OR COMPARAND VALUE
GRPAY    DS      F
DED      DS      7F
NETPAY   DS      F
```

On the first loop iteration, C(GPR3) = 0 so DED+0 will be subtracted; on the second iteration, C(GPR3) = 4 so DED+4 will be subtracted; on the final iteration, C(GPR3)=24 so DED+24 will be subtracted. At that point, the BXLE will increase GPR3 to 28, the index is no longer less than or equal to the comparand and the branch will not be taken. Instruction execution will continue sequentially below the BXLE. Note that the indexing approach will require about twice as much computer time as simply coding seven successive subtractions. The extra time is taken by the BXLE and the use of indexing in the subtraction. However, the latter is much the smaller of the two. This approach does save storage. This little example portrays the programmer's dilemma: Striking a balance between "tight" code -- code which economizes on storage but takes additional run time or fast code which often requires a good deal of additional storage. It is an unusual occasion when a programming strategem which reduces storage requirements significantly will also allow a significant reduction of running time. Improvements in one of the two parameters -- storage space or running time -- is usually at the expense of the other. Alternate approaches to this problem are possible. A BXH could be used although, in this example, it is not as natural as BXLE. To use BXH, NDXCON would have to be changed to 24, 4, -1, respectively. The -1 is required for the last value, 0, to be processed. This must be done since the branch will only be taken when the index is higher than the comparand.

We can develop a simple notation for the loop or indexing parameters, first value, increment and final or comparand value. In the context of the example above, we can write the parameters as 0(4)24 where the quantity in parentheses is the increment and the initial and final values precede and follow it, respectively. For BXLE, these parameters can then be set up in that order by a DC and can be loaded by an LM instruction into three adjacent registers beginning with an odd one. For BXH, the final value would have to be reduced by one.

Exercise 4-2 a) For a BXLE terminating a loop with parameters 3(7)25 how many times will the loop be executed and what will be the successive index values? b) The same conditions as a) except the parameters are -4(3)11. c) For a BXH, with parameters 100(4)0, how many times will the loop be executed? What are the first two and last two values of the index? What will be the value of the index after the loop has been satisfied. d) Can you devise a simple formula to calculate how many times a loop will be repeated? Consider BXH and BXLE loops.

Worked Examples

4-1 A block of 10,000 words has been reserved for a data file with one entry per word. The number of items which are read in at any one time may be any number from 0 to 10,000. A word of all 1's marks the end of the data file and is placed after the last word. The entries may be positive, negative, or zero, except that no entry will be -1 which is a full word of 1's in two's complement form. The following instructions count the number of entries and stores it in COUNT.

```
                LM       1,3,NDXCON
                L        0,ONES          LOAD WORD OF 1's
TEST            C        0,BLOCK(1)
                BE       STORE
                BXLE     1,2,TEST
STORE           SRA      1,2             'DIVIDE' BY 4
                ST       1,COUNT
                --
                --
NDXCON          DC       F'0'            INITIAL VALUE
                DC       F'4'            INCREMENT
                DC       F'39996'        FINAL VALUE
ONES            DC       X'FFFFFFFF'
BLOCK           DS       F10000
                DS       F
COUNT           DS       F
```

Let us first discuss NDXCON. The first word is at BLOCK, the 10,000th word is at BLOCK + 39996; since we are incrementing by four, the loop parameters are then 0(4)39996. The initial value of GPR1, the index, is 0 so that if the file contained no entries, the word at BLOCK would contain all 1's and the BE would cause a branch out of the loop. The number of entries would then be found in GPR1. If the second word contained all 1's, then the number of entries would be one. However GPR1 would contain four at that point since it would have been incremented once by four. The right shift of two places has the effect of dividing by four. The reader should extend this line of reasoning for a few more iterations to verify that GPR1, after having been shifted, will contain the number of entries in the file. Note that if the file contains all 10000 entries, the last interation of the loop will be testing the 10,000th word with an unequal result. The BXLE will then increment GPR1 from 39,996 to 40,000. Since the index will then be higher than the comparand, the branch will not be taken and the next instruction to be executed will be at STORE. After the effective division by four, GPR1 will show a count of 10,000, the correct value. Note that an extra word has been reserved at the end of the 10,000 words of BLOCK. This allows for the possibility that a data file may contain all 10,000 words with a word of 1's in the 10001st word. Note that the program segment we have written makes the assumption that if a match with ONES is not made in the first 10,000 words, then it will be made in word 10,001 but no explicit check is made. This is not the best programming practice since records which contained more than 10,000 entries, and therefore are in error, would go undetected. In cases such as this, the programmer should proceed cautiously keeping in mind the First Law of Programming: "If something can possibly go wrong, it probably will, eventually". We leave it to the reader in Prob. 4-7 to make the necessary additions to our code to guard against records of more than 10,000 words.

While loops are fairly easy to set up, it is also easy for programming errors to be made particularly in reference to the starting and final index values. To guard against these errors, which can be difficult to catch when the program is running in a production environment, a thorough desk check is a must. The programmer should trace through the logic for the first and last points as well as several intermediate points. When the program is tested, the test cases should include tests of the end point conditions.

4-2 Re-code the payroll loop using only registers 3, 4, and 5. Our problem is to select one of these registers to replace GPR2 which was used to compute NETPAY. GPR3 cannot be used because it is needed for the index, so our choice can be either of GPR4 or 5 -- we will select GPR5. Our strategy here is to alternate the usage of GPR5 as a loop control register and a register for calculating NETPAY. For each type of usage, the proper data will be loaded into GPR5 after storing its previous contents. The coding for the loop is

```
SUBT     ST       5,SAVEREG     SAVE LOOP DATA
         L        5,NETPAY      LOAD PAYROLL DATA
         S        5,DED(3)
         ST       5,NETPAY      STORE PAYROLL DATA
         L        5,SAVEREG     LOAD LOOP DATA
         BXLE     3,4,SUBT
         --
         --
```

The ST 2,NETPAY which followed the loop in the previous code, will not be necessary here since the data will be stored in NETPAY in the loop. We seem to have saved an instruction outside the loop at the cost of four additional ones inside the loop, or a net gain of three instructions. However, the picture is much worse than this. The loop will be repeated seven times so that the extra four instructions will give 28 additional instruction executions, or a net gain 27 instruction executions which will just about triple the running time of this segment. The moral of this tale should be clear -- loops should be programmed as efficiently as possible and register allocation should be done economically to avoid reallocation within a loop.

The programmer will have to deal with blocks of data very frequently. We will next describe a simple notation for describing operations on the elements of a data block which is often termed an array. That is, an ordered table of data. In mathematics, the individual elements of an array are indicated by a subscript notation, B_i, for the ith entry in the B array. Since the 360 character set has no subscripts, we will write it as B(I). To fetch the Ith element of an array, we note that if B gives the starting address, then the address of the Ith entry is B - 1 + I. As a check, if I = 1, the address of B(1) = B - 1 + 1 = B. The address of the 10th element is B - 1 + 10 = B + 9, or nine locations away from the initial location. In addition, the index, I, must be multiplied by a storage factor, f, depending on whether the entries in B are bytes (f = 1), half-words (f=2), words (f = 4), double-words (f = 8), or some other size. Our addressing formula is then

$$\boxed{\text{address of } B(I) = B + f*(I-1)}$$

where B is the starting address (label address) for the block. A more natural way for a programmer to look at this formula is

$$\boxed{(B - f) + f*I}$$

where B-f appears in the operand and f*I is the index. This has the advantage that for a loop with parameters 1(1)100, for instance, the loop registers can be set up with f(f)100f which is more straightforward than 0(f) 99f.

Using this notation, we will flowchart Ex. 4-1 and discuss its translation into assembly language.

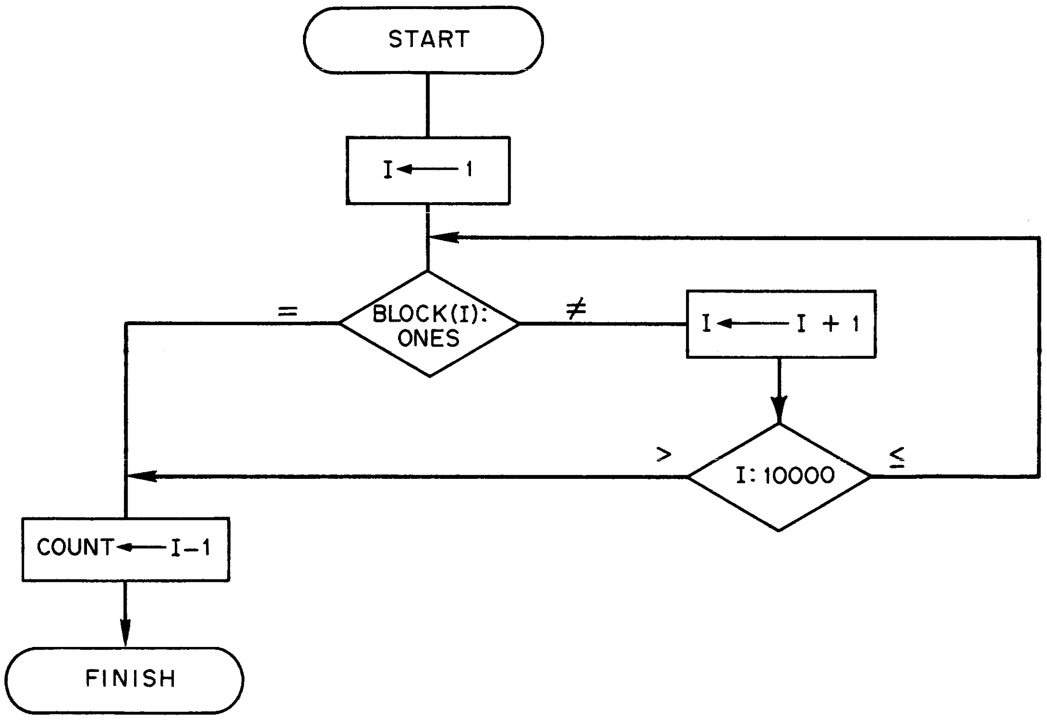


Fig. 4-2   Flow Chart for Worked Example 4-1

As can be seen from Fig. 4-2, the major change to be made when this version is coded is the starting point for the index. If the entries in BLOCK are single words, then the addressing formula is

$$(BLOCK-4) + 4I$$

and since the loop parameters are 1(1)10000, NDXCON will contain 4(4)40000. The code follows

```
        LM      I,3,NDXCON      PARAMETERS OF I ARE 4(4)40000
        L       0,ONES
TEST    C       0,BLOCK-4(I)    COMPARE B(I)WITH ONES
        BE      STORE
        BXLE    I,2,TEST
STORE   SRA     I,2             DIVIDE BY 4
        BCTR    I,0             SUBTRACT 1 FROM I
        ST      I,COUNT
        --
        --
NDXCON  DC      F'4'
        DC      F'4'
        DC      F'40000'
ONES    DC      4X'FF'
I       EQU     1
COUNT   DS      F
BLOCK   DS      10000F
        DS      F
```

The address of BLOCK-4 will be computed only once, at assembly time, when four will be subtracted from the displacement of BLOCK from its base point. With this correction, when I = 1, the first word in BLOCK will be fetched, when I = 2, the second and so forth. The subtraction of one from I is accomplished using a BCTR with zero for the branch register which nullifies branching and results in one being subtracted from I. (What would happen if the subtraction occurred at STORE, before the division by four?) While this logic is not significantly different from our previous approach in Example 4-1, we recommend this technique because it corresponds more closely with the way an indexing problem would be described on a flow chart or as a problem statement for a higher-level language such as PL/I, FORTRAN or COBOL. Note that subtracting one from the index is not a necessary feature of this approach. It just so happens that when a loop exist is made, we have the index of the end of record character which is one higher than the last word in the record.

An interesting point to consider is the number of base registers required for this program. To simplify matters, assume that the program, up to and including the label, BLOCK, requires less than 4096 bytes. Even though BLOCK is larger than 4096 bytes, extra base registers are not required. The point is that if BLOCK is within 4095 bytes of the start of a base register's span, the displacement field of any operand which references BLOCK will be able address it. The fact that indexing subsequently increases the address to beyond the span of a base register is not relevant since the indexing is carried out in 32-bit registers where the restriction of displacement size to 4095 does not apply. However, if we were careless enough to place either constants or individual data words (NDXCON, COUNT) beyond BLOCK, another base register would be required to access them. Our program would then read

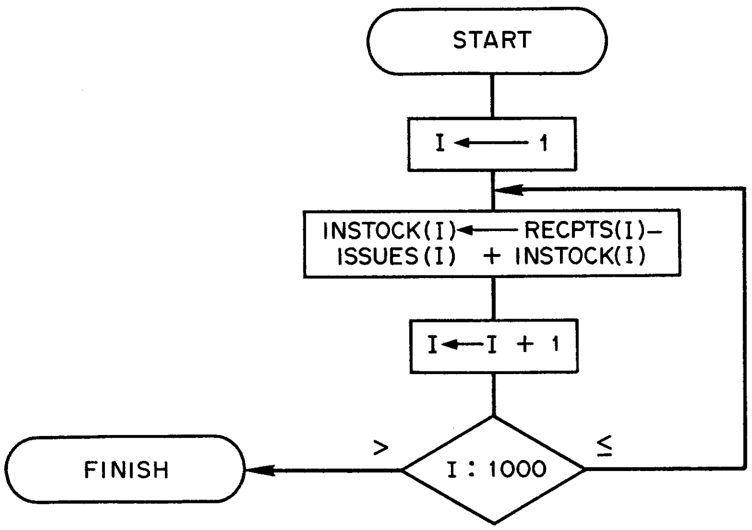
```
                 START      0
        FIRST    BALR       10, 0
                 USING      *, 10
                 USING      COUNT, 11
                 L          11, ACOUNT
                 B          GO
        ACOUNT   DC         A(COUNT)
        GO       --
                 --
                 --
        BLOCK    DS         10000F
        COUNT    DS         F
        ONES     DC         4X'FF'
                 END        FIRST
```

This wastes what the programmer should regard as a valuable commodity, one base register. To avoid this it is recommended that data storage be assigned in order of increasing length, with the shortest fields defined first and the largest field defined last. Note too, that if ACOUNT were placed after BLOCK, the displacement field in L 11,ACOUNT would not be able to address ACOUNT since it is more than 4095 bytes away from the base point of the only base register loaded at that point, GPR10. The only way out of this bind is to place ACOUNT somewhere within the span of GPR10 since it is the only one loaded at that point.

This example illustrates one of the few occasions when the programmer must keep the addressing characteristics of the computer very clearly in mind. Example 4-5 will show another instance of this.

Worked Example

4-3 Assume that a particular inventory consists of 1000 items, numbered 1 to 1000. There are three 1000-word arrays in storage, INSTOCK, RECPTS, ISSUES which give the in-stock position as of the last reporting period and the receipts and issues since the last reporting period, respectively. Write code which will update the in-stock table by adding receipts and subtracting issues. Assume each entry requires two bytes and is a binary number. The flow chart is

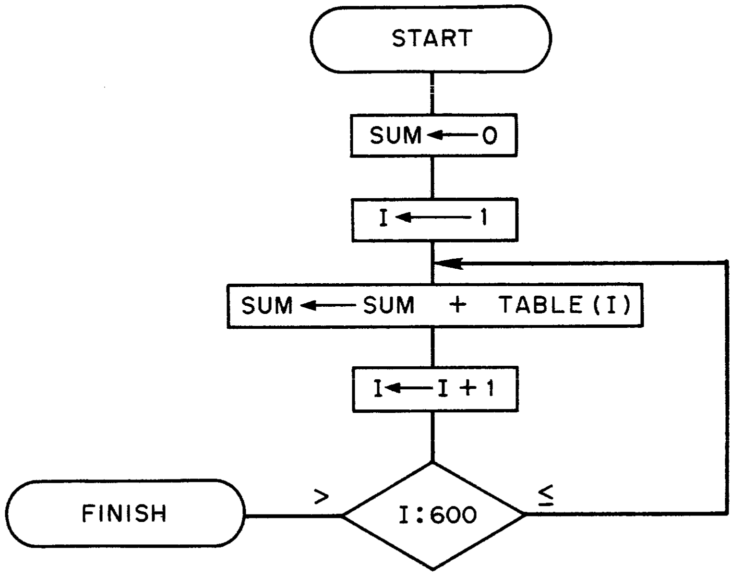


The coding is

```
          --
          LM      I,3,NDXCON
LOAD      LH      0,INSTOCK-2(I)
          AH      0,RECPTS-2(I)
          SH      0,ISSUES-2(I)
          STH     0,INSTOCK-2(I)
          BXLE    I,2,LOAD
          --
          --
          --
NDXCON    DC      F'2'
          DC      F'2'
          DC      F'2000'
INSTOCK   DS      1000H
RECPTS    DS      1000H
ISSUES    DS      1000H
I         EQU     1
```

Since all entries are half-words (which can accommodate a data range of -36768 to 36767), half-word arithmetic operations are used and the operand corrections are -2, not -4 as in the previous example. Also since the loop parameters are 1(1)1000, NDXCON is 2(2)2000. Note that this program will require more than one base register because of the large data fields of more than 95 bytes of instructions and constants precede INSTOCK.

4-4 A 600-word array, TABLE is to be summed and stored in SUM. The flow chart is



The coding is

```
ADD       LM     I, 3, NDXCON       I EQU 1
          SR     0, 0               CLEAR GPR0
          A      0, TABLE-4(I)
          BXLE   I, 2, ADD
          ST     0, SUM
          --
          --
SUM       DS     F
NDXCON    DC     F'4'
          DC     F'4'
          DC     F'2400'
I         EQU    1
TABLE     DS     600F
```

Since the loop parameters are 1(1)600 and we are dealing with full-word entries, NDXCON is 4(4)2400. Note that when a sum is being accumulated in a register, and the first item will be added rather than loaded, the register must be cleared before use. This is essential for correct operation -- it is analogous to clearing an adding machine before using it.

4-5 The indexing techniques we have seen in the last few examples seem very useful for binary data. The question naturally arises, "How can decimal-data fields be indexed?" As a specific example, suppose the data of Ex. 4-3 is in the packed decimal format, three bytes per entry, with 100 entires per array. (The original length of 1000 each would cause additional base register complications which we want to avoid for the moment.) Since SS instructions do not have index-register fields, a direct approach is ruled out. However, SS address are in base register plus displacement form so that the base register can be used as an index register provided we find a way to do this without invalidating other operand addresses. The approach we will take is to specify one register (here GPR8) as a base register for data only and to arrange that it will be used only in the section of the program which does the indexing. As an aid to the reader in tracing through the following code, note that the USING statement is nothing more than a statement of intent by the programmer. He is advising the assembler every time he writes a statement such as USING *, 11 or USING INSTOCK, 7 that he will arrange to load GPR11 and 7 with the addresses, * and INSTOCK. The assembler then arranges to assign every label which falls within the span of coverage of GPR11 or GPR7 to one or the other of these base registers. Usually, base register coverage is arranged so that the span of each register is 4096 bytes and each one takes up where the previous one terminates. In the case where the spans of two base registers overlap, the base register which gives the smallest displacement is assigned. The coding follows; IOIN and IOOUT are the assumed input and output sections. We are also assuming that the entire program requires less than 4096 bytes.

```
         START    0
GO       BALR     11,0
         USING    *,11
IOIN     --
         --
         --
         USING    INSTOCK,7
         LM       7,9,BASECON
ADD      AP       INSTOCK,RECPTS
         SP       INSTOCK,ISSUES
         BXLE     7,8,ADD
         DROP     7
         B        IOOUT
BASECON  DC       A(INSTOCK)         GPR7
         DC       F'3'               GPR8
         DC       A(INSTOCK+297)     GPR9
INSTOCK  DS       100PL3
RECPTS   DS       100PL3
ISSUES   DS       100PL3
IOOUT    --
         --
         --
         END      GO
```

The second USING instruction informs the assembler that it may assume that GPR7 will be loaded at execute time with A(INSTOCK) and that every label within INSTOCK and INSTOCK+4095 can be assigned GPR7 as base register. This will overlap with the span of GPR11 but GPR7 will be assigned to those labels at INSTOCK and below (including IOOUT) since it gives a smaller displacement. The loop parameters are 1(1)100 but since we are dealing with three-byte fields and since the AP and SP instructions reference the starting points of INSTOCK, ISSUES, RECEPTS, the parameters become 3(3)297. Because the base register we will be modifying, GPR7, contains A(INSTOCK), the loop parameters are A(INSTOCK) (3) A(INSTOCK+297). The BXLE instruction will cause the base register to be incremented by three at every iteration until all 300 items have been processed. The "indexing" for all three arrays can be done by one base register since the displacement fields of ISSUES and RECPTS take care of the spacing between them. That is, on the first iteration, the three operand addresses are INSTOCK, INSTOCK +300, and INSTOCK+600; on the second iteration, they become INSTOCK+3, INSTOCK+303, and INSTOCK+603, or RECPTS+3 and ISSUES+3, respectively. (What objection could be made to writing INSTOCK-3, RECEPTS-3 and ISSUES-3 in the AP and SP instructions as opposed to the way they are written in the sample program? What displacements will be assigned in either case?)

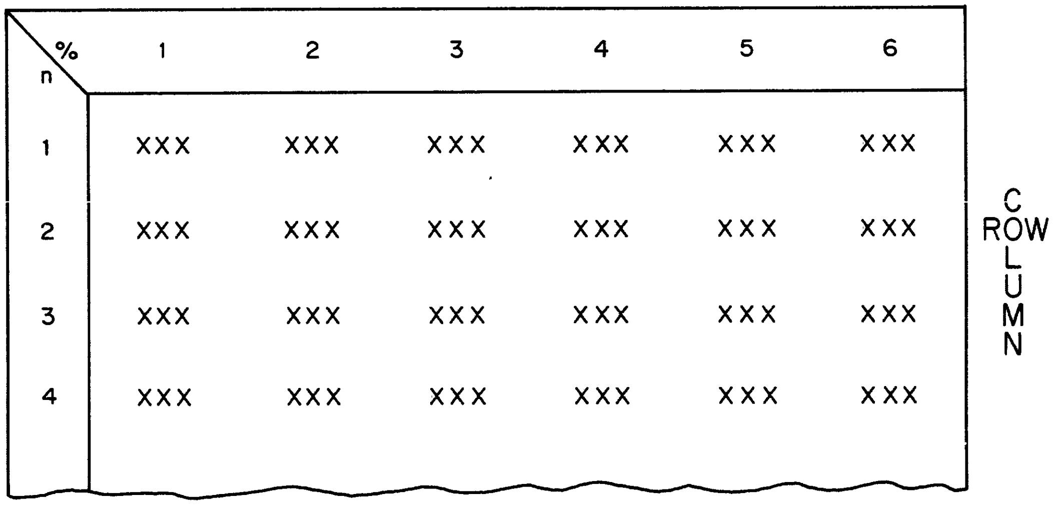
Note the DROP 7 assembler instruction immediately after the BXLE. This informs the assembler that GPR7 is no longer available as a base from that point on. As a result, only those operand addresses between the USING INSTOCK, 7 and the DROP7, which refer to operands between INSTOCK and INSTOCK+4095, will be assigned GPR7 as a base register. If we did not take this precaution, the B IOOUT would be assembled as

```
BC     15, 600 (0, 7)
```

which would be correct at assembly time but when executed, GPR7 would contain A(INSTOCK+300) and so the actual branch address would be 300 bytes further down the program than intended. With the DROP statement included before the B IOOUT, GPR11 will be used, instead of GPR7. Again, this is one of the infrequent occasions when the programmer must take into account the addressing characteristics of the machine.

4.3 Tabular Data

Previously, we have considered only simple arrays consisting of a sequence of entries -- the first entry is in word one, the second in word two and so forth. We can visualize such an array as a column of data. For this reason, these arrays are often called one-dimensional arrays. We can also talk about a two-dimensional array and, in fact, two-dimensional arrays have great utility in programming. As an example of a two-dimensional array, consider a compound interest table. Going down the left hand side of the page, we have the number of years of investment, say one to 50. Across the top, we have the interest rates from, say 1% to 6% in steps of 1%. To find the value in 30 years of $1.00 invested today at 4%, we read across the 30 year row until we come to the 4% column, the entry there gives us the answer, $3.2434. Since programs very often use tablular data in a two-dimensional form, we will develop a simple technique for addressing these arrays. First, some notation. Figure 4-3 shows part of the compound interest table and the acrostic to the right indicates that the rows are read from left to right and the columns from top to bottom. We will use the same modified subscript



n \ %	1	2	3	4	5	6
1	XXX	XXX	XXX	XXX	XXX	XXX
2	XXX	XXX	XXX	XXX	XXX	XXX
3	XXX	XXX	XXX	XXX	XXX	XXX
4	XXX	XXX	XXX	XXX	XXX	XXX

Fig. 4-3 Part of a Compound Interest Table

notation here which we introduced previously for one-dimensional arrays. To refer to an item in the Rth row and Cth column of a two-dimensional array, A, we write A(R, C) indicating that the first subscript gives the row number and the second, the column number. (We offer the reader the mnemonic arc as an aid to remembering this convention.) The interest table in Fig. 4-3 has 50 rows and six columns. While we can depict it on paper as a two-dimensional array, in core storage, it must be arranged in what amounts to a one-dimensional layout since core addresses are sequential. We will form the array by columns. The first column will start at TABL, the second TABL + 50(words), the third at TABL+100, and so forth. Figure 4-4 gives a schematic.

We must now develop a technique for addressing an arbitrary element in the table, TABL(I, J), that is, the Ith entry (row) in the Jth column.

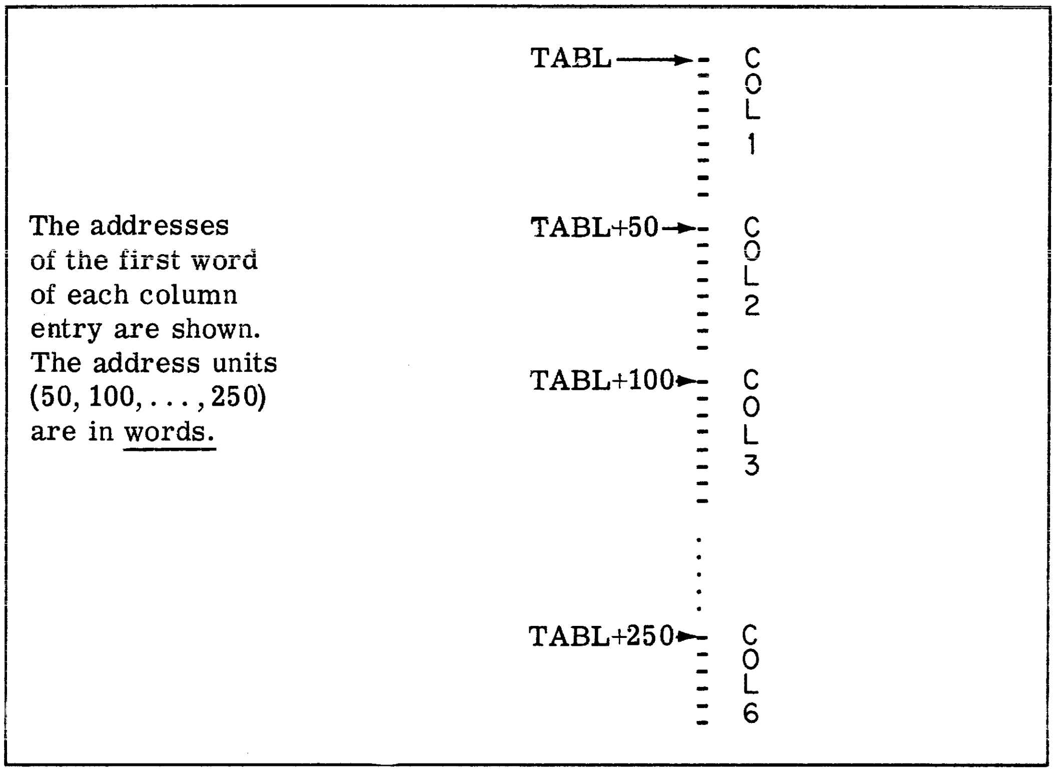

The addresses of the first word of each column entry are shown. The address units (50, 100, . . . , 250) are in words.
TABL
TABL+50
TABL+100
TABL+250
COL 1
COL 2
COL 3
COL 6

Fig. 4-4 Two dimensional Array in Core Storage

Looking at Fig. 4-4, we note that each column has 50 entries and since there are J-1 columns before the Jth column, the number of entries preceding the Jth column are 50 (J-1). That is, this number of entries brings us to the last entry in the J-1st column. To reach the Ith entry, or row, of the Jth column, we add I to 50 (J-1). With TABL as the label, or beginning address of the array, and f the number of bytes per entry, the address of TABL (I, J) is then TABL + f (50 (J-1)+I-1). Since TABL is the address of the first entry in the table, TABL (1, 1), we must subtract 1 from I. To verify this formula, substitute sample values of I and J including I=J=1 and compare your result with a direct count. For convenience the addressing formulas for one and two-dimensional arrays are given below.

Address of TABL(I, J)=TABL+f(N(J-1) + I-1)*
Address of A(I) = A + f(I-1)
* N is the number of rows in the array.

In a typical application of a two-dimensional table, the array indices are supplied as input items. As an example, in an interest problem a particular case may require the value of an investment at 6% interest in 18 years. The column number is then 6 and the row number is 18. Assuming full-word entries (f=4), the desired address is

TABL + 4 (50(6-1) + 18-1)

or TABL + 4*267 = TABL + 1068. We will show sample coding which will calculate the address of an element in a 50 row array whose row number (I) is in location ROW and whose column number (J) is in location COL. The result will be stored in ANS.

```
--
--
L      3, COL           J
BCTR   3, 0             SUBTRACT 1, GPR3 = J-1
M      2,=F'50'         GPR3 = 50(J-1)
A      3, ROW           GPR3 = 50(J-1) +I
BCTR   3, 0             BPR3 = 50(J-1) +I-1
SLA    3,2              MULTIPLY BY 4
L      2, TABL(3)       TABL+4 (50(J-1) + I-1)
ST     2, ANS
--
--
```

The use of GPR3 as an index register performs the address modification. We could define the address of TABL and add it to GPR3, then use a L 2,0(3)instruction which would give the same result. Our method saves the addition because the indexing hardware will do it for us anyway.

In some cases, the indices will not be given directly but must be calculated. As an illustration, suppose we have a separate column for each interest rate from 1% to6% in steps of 1/2%. The column indices are related to the rates by

Column	1	2	3	4	5	6	...
Rate	1%	1.5%	2%	2.5%	3%	3.5%...	

The programmer would then have to develop a formula to connect the two. If he is not mathematically minded, and wants to make up for it with a little extra computer time, he could arrange a rate table in storage and program a routine which searches the table to find the right column for each rate which occurs. We will discuss table look-up subsequently. However, a formula can be developed easily and we will pause to do so since this example is representative of a general technique: Note that if the rate is doubled and then one subtracted the result is the column number. The reader is urged to try this a few times to verify it. If the rates are defined as F'10', F'15' for 1%, 1.5% and so on, the coding for our formula is

```
L       3, RATE        RATE=X.X
SLA     3, 1           DOUBLE RATE
S       3,=F'10'       SUB. ONE:X.X-1.0
SR      2,2            CLEAR GPR2
D       2,=F'10'       'DECIMAL SHIFT' RIGHT 1

* GPR3 NOW CONTAINS COLUMN NUMBER
```

The tabular arrangements we have discussed so far have been relatively clean. To retrieve a particular value, we had its indices available as input, or lacking this, we had something from which an index could be readily computed. However, not all tabular data can be arranged in this fashion. As an example, consider the symbol table constructed by the assembler. When the assembler processes an operand, to find its address, a search must be made through the symbol table comparing the given operand name with every entry in the table. When a match is found, its address is then available, usually in some adjacent locations. This table is typical of the argument-value table as opposed to an indexed table which we have discussed previously. The argument can be considered the table input and the value, the table output. The reader may envision an argument-value table as comprising two separate, adjacent columns, one containing arguments, the other, the corresponding values. A dictionary is an excellent example of an argument-value table. The word to be looked up is the argument and the definition is the value of the table corresponding to that argument. Note that with this type of table, an access will involve some form of search, or look-up. If we wished to find the dictionary definition of "mnemonic", for instance, we would first turn to the M's, then skim back and forth until we found a page which bracketed "mnemonic", such as "mitered-mode", and finally after searching that page, find "mnemonic". If the dictionary was organized instead as an indexed table, we would go immediately to the 19th entry on page 506, for instance. Even this would only be possible by a search through another list, one which gives page and entry numbers for each word. At the heart of our difficulty with a dictionary or any true argument-value table is this: While we can arrange the arguments in some order, alphabetic order in the case of a dictionary, we have no way of making an immediate correspondence between the argument and an integer, or integers, which would comprise its index.

From our study of the interest table, we can immediately assign to the argument "3.5% 15 years", the integers "15, 6" for row 15, column 6. However, we do not have a formula for assigning page 506, entry 19 to "mnemonic", we can only do so by a search.

Worked Example

4-6 As an example of a table search program, suppose we have an employee file on disk arranged in alphabetical order and we have a table in core showing the range of disk addresses for a particular range of names. Assume we have divided the employee name file into 20 equally sized groups. The core table, NAME, gives the names which constitute the upper boundaries for each name group. These names are 15-character fields, left justified and padded to the right with blanks. To retrieve the data from disk corresponding to a particular name, DANIELS for instance, a search is made in the NAME table and it is found that this name is in the fourth group. Using this as an index to the address table, ADDRS, we find that the fourth group of names begin at disk address 207516. A search can be made through all names in the group starting at this point, or a dictionary for the fourth group can be stored on disk at this address and the process repeated until we are close enough to DANIELS for a name by name search to be made. The final dictionary might show all the names within a given range of the alphabet and the disk address of each one. Figure 4-5 shows the NAME table. The coding follows with the search name left justified in a 15-character field, INPUT. The program finds the group index which is a number from 1 to 20 and stores it in a full word, NDX. The coding follows with the flow chart in Fig. 4-6.

```
         --
         --
         LH     I,=H'1'         I EQU GPR0
         LM     1,3,ADCON       GPR1-3 FOR LOOP CONTROL
         USING  NAME, 1
LOOP     CLC    INPUT, NAME
         BNH    STORE           INPUT LESS THAN OR EQUAL
         AH     I,=H'1'         INPUT GREATER, ADD 1 to I,
         BXLE   1,2, LOOP       INCREASE BASE REG BY 15
         DROP   1
STORE    ST     I,NDX
         --
INPUT    DS     CL15
I        EQU    0
ADCON    DC     A(NAME)         LOAD INTO BASE REG1
```

```
         DC    F'15'
         DC    A(NAME+19*15)
NAME     DC    CL15'BEVINS'       EMPLOYEE NAME TABLE
         DC    CL15'BURNS'
         DC    CL15'COLUMBO'
         --    ---
         --    ---
```

NAME	GROUP
BEVINS	1
BURNS	2
COLUMBO	3
DEERING	4
EWELL	5
GAYNOR	6
--	
--	
--	
--	
SPRAGUE	17
TILTON	18
WATKINS	19
ZZZZZZZZZZZZZZZ	20

Fig. 4-5 NAME Table for Example 4-6
Each name is the upper boundary for
1/20 of the employee name file

Our strategy is to compare the INPUT name with the group boundaries in ascending sequence. Since the collating sequence allows a correspondence to be made between alphabetic data and binary numbers, the compare circuitry can detect when a given alphabetic field is less than or equal to another field (note that blanks are lower in the collating sequence than any letters). As soon as a less than or equal condition is detected, the group in which NAME belongs is the group whose upper boundary has just been compared.

The index of this group is the table argument in this foreshortened version--actually, the index would be used to retrieve the appropriate disk address from another one-dimensional table; it is this disk address which is the ultimate table value. The logic of indexing in an SS-instruction such as CLC has been discussed in Ex. 4-5 so we will not comment further here. Note that it is possible that an INPUT will be higher than

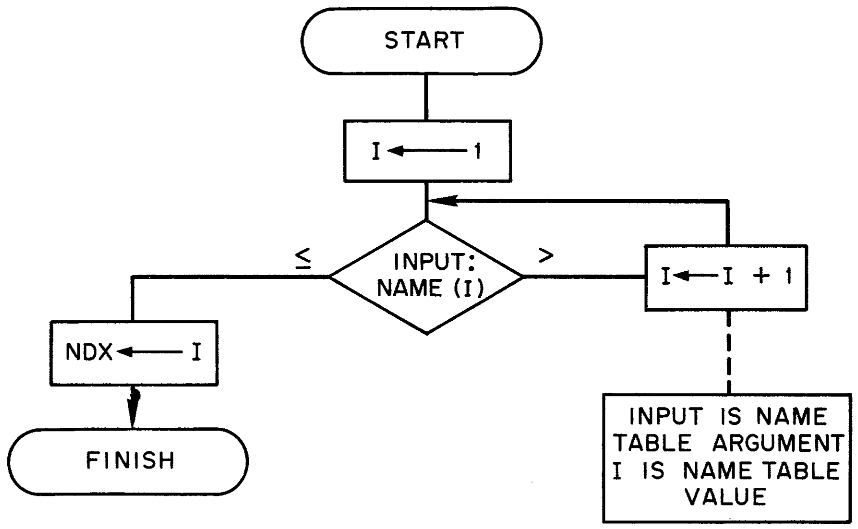


Fig. 4-6 Flow Chart for Example 4-6

all entries in name including ZZZ...ZZZ. This will happen if a digit is inadvertently substituted for the high-order character, 4ANIELS, for example. The data should be validity checked earlier in the program to preclude this possibility or else, a branch to an error routine should be inserted after the BXLE. If the loop is satisfied, no match has been found and INPUT then contains an invalid high order character.

Worked Example

4-7 Binary Search The previous example shows how a sequential search can be made. A sequential search, of course, is one which begins at the first entry in the table and scans through it, entry by entry, in sequence until the desired entry is found. It has the advantage of being easy to program but many comparisons may be required. In a table of length 2,000, on the average 1000 comparisons are required to find the desired entry. There are search techniques which reduce significantly the number of comparisons. One of them, the binary search, will be discussed here. It is necessary for this technique that the arguments be arranged in some order, say low to high. The binary search is based on dividing the table in half, determining which half contains the argument, dividing that part in half and so on until a match is made. The first division must be made to the smallest power of two greater than half the table size. This is necessary to ensure that any entry in the table can be reached. As an example, a 2000 entry table would be examined in position 1024 since this is the smallest power of two greater than 1000. Assume the arguments are four-character part numbers coded as alphabetic fields, sorted from lowest to high in a 2000-word table. We have another 2000 word price table whose entries correspond in sequence with the part number table. We are given PARTNO and desire the corresponding PRICE. The two tables are PARTBL and PRCETBL. As an example of a typical search, assume the desired value is actually located in entry 1997; at each step, PARTNO will be compared with the table entry until entry 1997 is reached. After each compare, the remainder of the table will be divided in half and the next compare made at that point. If we exceed the table size, we will drop back to the next lower half-way point. Our first try is 1024 (PARTNO compares high); the second, 1024 + 512 = 1536 (high); the third, 1536 + 256 = 1792 (high); the fourth, 1792 + 128 = 1920 (high); the fifth, 1920 + 64 = 1984 (high); the sixth, 1984 + 32 = 2016 (invalid, PARTNO compares low); the seventh, 2016 - 16 = 2000 (low); the eight, 2000 - 8 = 1992 (high); the ninth, 1992 + 4 = 1996 (high); the 10th, 1996 + 2 = 1998 (low); the 11th, 1998 - 1 = 1997 (match). The maximum number of seeks for this size table is 11. The minimum could be as low as one with the average around 10. This is 100 times less than a sequential search. The extra programming effort will be worth it if we can code the binary search so that it can be completed in less than 100 times as many instruction executions as a sequential search.

The flowchart appears in Fig. 4-7. As our example above indicates, the binary search generates a series of test tries by adding or subtracting a series of successively lower powers of two to a starting value, 1024. We add, if PARTNO compares higher than the test entry, subtract if it is lower. The index of the current test entry is in a register, NDX; the current power of two is in a register, HALF, which is divided by two on each iteration. Note that a check must be made to ensure that NDX does not go above 2000. If it does, we wish to prevent the actual comparation, since the table does not extend that high, and treat it as though PARTNO compared low, (that is, our next attempt will have to be lower in the table). Note that we have not taken precautions against a PARTNO which might be lower or higher than the ends of the table. If this happened, the program would go into an endless loop because it would be trying for a comparison at entry 1 or 2000 which could not be met, HALF would be shifted to zero and a compare attempted at the same point as the last compare. To avoid this, PARTNO should be given a range check immediately prior to the binary search. The coding follows.

```
            --
            --
            --
            L      PARTR, PARTNO               PARTR EQU GPR1
            L      NDX, =F'4096'               NDX = 4*1024 IN GPR2
            L      HALF, =F'2048'              HALF = 4*512 IN GPR3
            L      MAX,  = F'8000'             MAX = 4*2000 IN GPR0
BSEARCH     CR     NDX, MAX                    COMPARE NDX WITH 2000
            BNH    TEST                        NDX EQ OR LT 2000
            B      LOW                         NDX GT THAN 2000
TEST        C      PARTR,  PARTBL-4(NDX) PARTNO:PARTBL(NDX)
            BH     HIGH
            BE     OUT
LOW         SR     NDX, HALF                   NDX = NDX - HALF
            B      SHIFT
HIGH        AR     NDX, HALF                   NDX = NDX+HALF
SHIFT       SRA    HALF, 1                     HALF = HALF/2
            B      BSEARCH
OUT         A      NDX,  = F'8000'             ADDRESS OF PRCETBL
            L      0, PARTBL(NDX)              LOAD PRCETBL(NDX)
            ST     0, PRICE
            --
PARTNO      DS     F
MAX         EQU    0
PARTR       EQU    1
NDX         EQU    2
HALF        EQU    3
PRICE       DS     F
PARTBL      DS     2000F
PRCETBL     DS     2000F
```

Note also that the branches after the compare instructions have been sequenced to minimize execution time. The branch condition which will be taken most often is placed immediately after the compare. For example, at BSEARCH we should expect that probably 95% of the time, NDX will be less than 2000. As a result, for every 100 times the comparison is made, the BNH will be executed successfully 95 times. The remaining five times, it will merely pass control to the next branch which will then be executed. The total number of instructions executed is 105. If the BNH and B were interchanged, and the B replaced by BH, the BH would be taken five times every 100 tries, the remaining 95 times, instruction processing will proceed to the next branch which will be executed successfully 95 times. The total instruction executions this time amounts to 195, almost a 100% difference. (For a first approximation, the execution times of a branch instruction is the same, whether it is taken or not.) The comparison between PARTNO and PARTBL involves an interesting question: Can it be done using the RX compare instruction? The only advantage to the RX Compare is that it is faster than the SS compare. However,

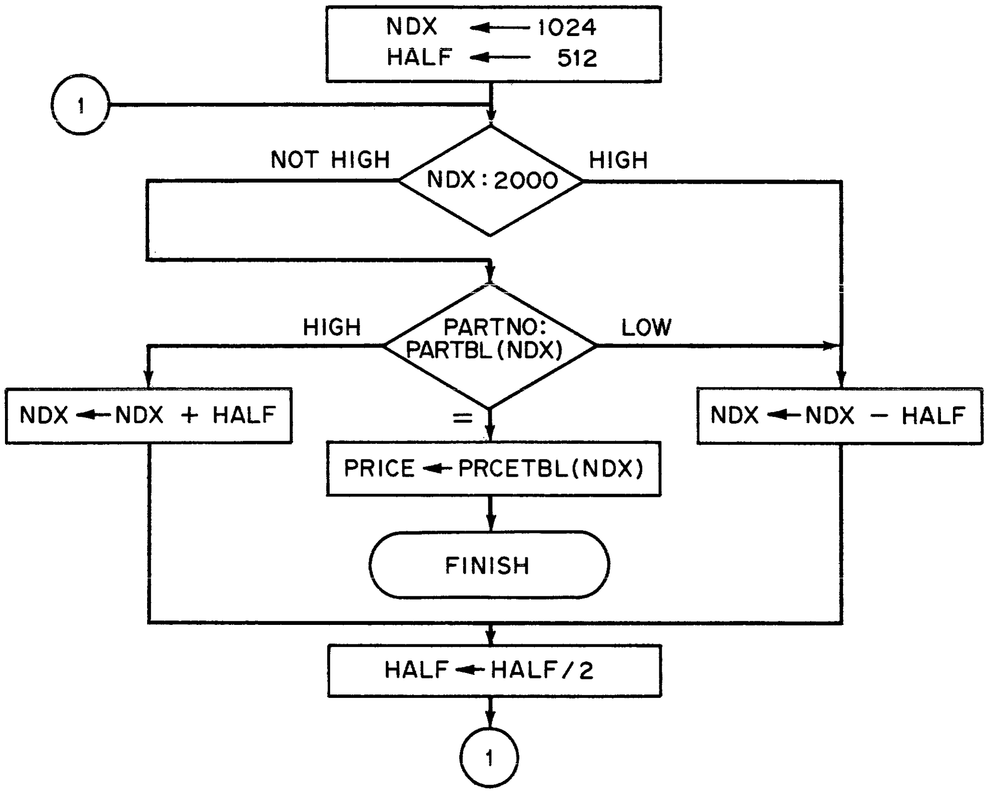


Fig. 4-7 Flow Chart for Example 4-7

the RX compare is algebraic, that is, the presence of a sign bit in one of the comparands will cause it to be interpreted as a two's complement number. We must ensure that the collating sequence is unchanged in this representation. An A is represented internally as 11000001 and Z is 1101001. When these are interpreted as two's complement numbers, we have A = -00111111 and Z = -00010111, or A = -63 and Z = -23 so that Z will still compare higher than A. The collating sequence is therefore unchanged. If there was a shortage of registers, the CLC could be used. However, this would require slightly different treatment of the contents of the registers, NDX and MAX. We leave this for the reader to work out as a problem at the end of this chapter. Base register usage offers a number of interesting considerations also. Assuming that the label PARTBL is 4095 bytes, or less, from the beginning of the program so that one base register will suffice to address all labels up to that point, then it would seem that an additional base register is needed for PRCETBL since it is 8000 bytes removed from PARTBL. However, the price table is only referred to once in the program at OUT+4. Rather than use an extra base register for it, we can reference the desired price as PARTBL+8000(NDX). However, this operand is not valid because 8000 exceeds the size of the displacement field. We obtain the same effect by adding 8000 to NDX. However, we are not out of the woods yet. The literals, =F'4096', =F'2048' and =F'8000' will be assigned to locations at the end of the program, that is, after PRCETBL. As a result, with only one register, they cannot be addressed. The assembler will not rectify this since base register allocation and loading are the responsibility of the programmer. The way out is to have the assembler place all literals at a more convenient point such as before PARTNO. This can be done with the LTORG statement whose format is

```
label     LTORG
```

This is a mnemonic for Literal Pool Origin, it may be labeled but should have no operands. It causes the assembler to place all literals, which have occurred prior to the LTORG, starting at the first double-word boundary after the LTORG.

Worked Example

4-8 Array Interleaving An important technique for conserving on base registers is array interleaving. Rather than set aside separate contiguous blocks for each array, we interleave them in a sequence such as A(1), B(1), C(1), A(2), B(2), ... A(N), B(N), C(N). As an illustration, assume we have to add together two 2000-words arrays, A and B, element by element, and store the results in C. The operation is A(I) + B(I) = C(I), I = 1(1)2000. Using conventional storage allocation techniques, we would write the code as follows

```
            START     0
GO          BALR      11,0
            USING     *,11
            USING     B-4,10
            USING     C-4,9
            LM        9,10,BSCON
            B         INPUT
BSCON       DC        A(B),A(C)
INPUT       --
            --
            LM        1,3,NDXCON
LOOP        L         0,A-4(I)
            A         0,B-4(I)
            ST        0,C-4(I)
            BXLE      1,2,LOOP
OUTPUT      --
            --
            --
NDXCON      DC        F'4'
            DC        F'4'
            DC        F'8000'
```

```
A           DS          2000F
B           DS          2000F
C           DS          2000F
            END         GO
```

We are assuming GPR11 will suffice to address the program up to and including A. If the arrays are interleaved by storing them in the order: A(1), B(1), C(1), A(2), B(2), C(2), . . . , GPR9 and 10 will no longer be needed. The storage area for the arrays will be defined as

```
A           DS          6000F
B           EQU         A+4
C           EQU         A+8
```

The loop instructions should be changed as though the program was being written for 12-byte fields. That is,

```
L           0, A-12(I)
A           0, B-12(I)
A           0, C-12(I)
```

Provided the index constants are changed appropriately, this will work since only four bytes will be taken from storage each time and the displacements will separate the arrays from each other. The assembler will assign a displacement to A which is 8 bytes less than the previous one, B's displacement will be four more than A's and C's, eight more. Let us now consider the index constants. If we consider A for the moment, each element is 12 bytes removed from the previous one, and since there are 2000 entries in the array, the last entry is 24000 bytes from the first. The first index constant must be changed to 12 to allow the first word of each array to be addressed correctly. We then have

```
NDXCON      DC          F'12'
            DC          F'12'
            DC          F'24000'
```

Figure 4-8 shows the addresses generated by this technique on the first, second and last iteration. Array interleaving is an important technique for conserving base registers, and where possible, it should be used. One of the requirements for its efficient usage is that the interleaved arrays be of approximately equal length, otherwise significant amounts of storage will be wasted. The possibilities for array interleaving are greatest in mathematical work. Where arrays are either generated in storage, or they are read in and relatively large amounts of calculations are done (significantly larger than the input or output time) so that time can be afforded to manipulate the individual arrays into an interleaved set of arrays. In commercial data processing the relative proportion of calculation is usually lower than in

	I = 12	I = 24	I = 24, 000
A-12(I)	A = A(1)	A=12 = A(2)	A+23, 988 = A(2000)
B-12(I)	A+4 = B(1)	A+16 = B(2)	A+23, 992 = B(2000)
C-12(I)	A+8 = C(1)	A+20 = C(2)	A+23, 996 = C(2000

Fig. 4-8 Array Interleaving

mathematical work so that the arrays should be in interleaved form on the input medium, if possible.

4-9 Sorting

Sorting is perhaps the single most significant application, in terms of time consumed, of data processing equipment. While it is beyond the scope of this book to describe the various techniques for manipulating I/O devices to produce an efficient sort, we will discuss several techniques for performing so called internal sorts. That is, a sort on records which are contained in core storage. An internal sort, of course, is an essential part of a sorting program. While it is unusual for the average programmer to write his own sort program -- this is usually supplied by the computer manufacturer -- it is important to understand the basic principles.

While all sorts have the ultimate goal of sorting a file of complete records, the internal sort phases usually do not deal with the entire record. Each record has a key, or identifier, usually much smaller than the full record. For instance, an insurance policy number may be only eight digits long whereas the entire record for that policy may contain 100 to 200 or more characters. The key contains the information which determines a record's position in the file, it may be an account number, policy number, part number, or an alphabetic name. Along with each key, there will be a storage address which gives the location of the full record. We will sort only the keys with their accompanying storage addresses. Assume that the keys are 32-bit binary numbers and the storage addresses are three-byte fields. The input routine brings in N records into contiguous storage locations and constructs an array of N seven byte entries, the first four give the record key, the next three, its address. We will develop a program which will sort this array, from low to high keys in ascending storage locations. One of the simplest methods of doing this is the exchange sort which scans through the array to find the lowest key which is then stored in the first entry, the process is repeated with the next lowest key which is stored in the second entry and so on, for a total of N-1 scans. (On the N-1st scan, the next to largest item will be stored in the N-1st entry; the largest one will then be automatically in the last entry.) The flow chart appears in Fig. 4-9, the coding follows.

```
          L       1, SEVEN
          ST      1, PN             INITIALIZE  PN TO 7
          M       0, N              UPPER LIMIT FOR
          ST      1, SEVEN+4            A-ARRAY = 7*N
          S       1, SEVEN
          ST      1, SEVNM7         PN UPPER LIM=7*N-7
          USING   A, I              I EQU 1
          LM      2, 3, SEVEN       LOAD LOOP PARAMETERS
          A       3, =A(A-7)        GPR3 = A-7+7*N
NEWPASS   L       I, =A(A-7)
          A       I, PN             I=PN
          MVC     LOW(7), A         LOW = A(PN)
          A       I, SEVEN          I=PN+7
COMPARE   CLC     LOW(4), A         LOW:  A(I)
          BNH     INDEX
          MVC     TEMP(7), A        STORE A(I)IN TEMP
          MVC     A(7), LOW         STORE LOW IN A(I)
          MVC     LOW(7), TEMP      NEW LOW = A(I)
INDEX     BXLE    1, 2, COMPARE     I=I+7; I:7*N
          S       I, SEVEN+4        I=A-7+7N+7 - 7N
          S       I, SEVEN          I=A-7+7-7
          A       I, PN             I=A-7+PN
          MVC     A(7), LOW         STORE LOW IN A(PN)
          L       0, PN
          A       0, SEVEN
          ST      0, PN             PN=PN+7
          C       0, SEVNM7         PN:7*N-7
          BNH     NEWPASS
          DROP    I
          B       OUTPUT            SORT  FINISHED
          --
          --
          --
          LTORG
PN        DS      F
SEVEN     DC      F'7'
          DS      F                 =7*N IN EXECUTION
LOW       DS      PL7
TEMP  .   DS      PL7
SEVNM7    DS      F                 =7*N-7 IN EXECUTION
N         DS      F
*    FIRST FOUR BYTES CONTAIN KEY,  LAST THREE,  ADDRESS -- SORT
*    ON KEY ALL LOOP PARAMETERS IN FIG. 4-7 MUST BE MULTIPLIED
*    BY 7 TO ACCOUNT FOR 7-BYTE LENGTH OF EACH ENTRY IN A
I         EQU     1
A         DS      1000PL7           MAX LENGTH 1000 ENTRIES
```

The flowchart describes the logic in a machine independent form. Indices are incremented by one rather than by seven, as an example. We suggest that this approach to flowcharting is to be preferred because it allows the programmer to focus on the essential logic of the problem first, rather than enmesh himself what thus-and-such base register should contain, or whether the BLXE should have an upper limit of 7N, 7N-7, and so forth.

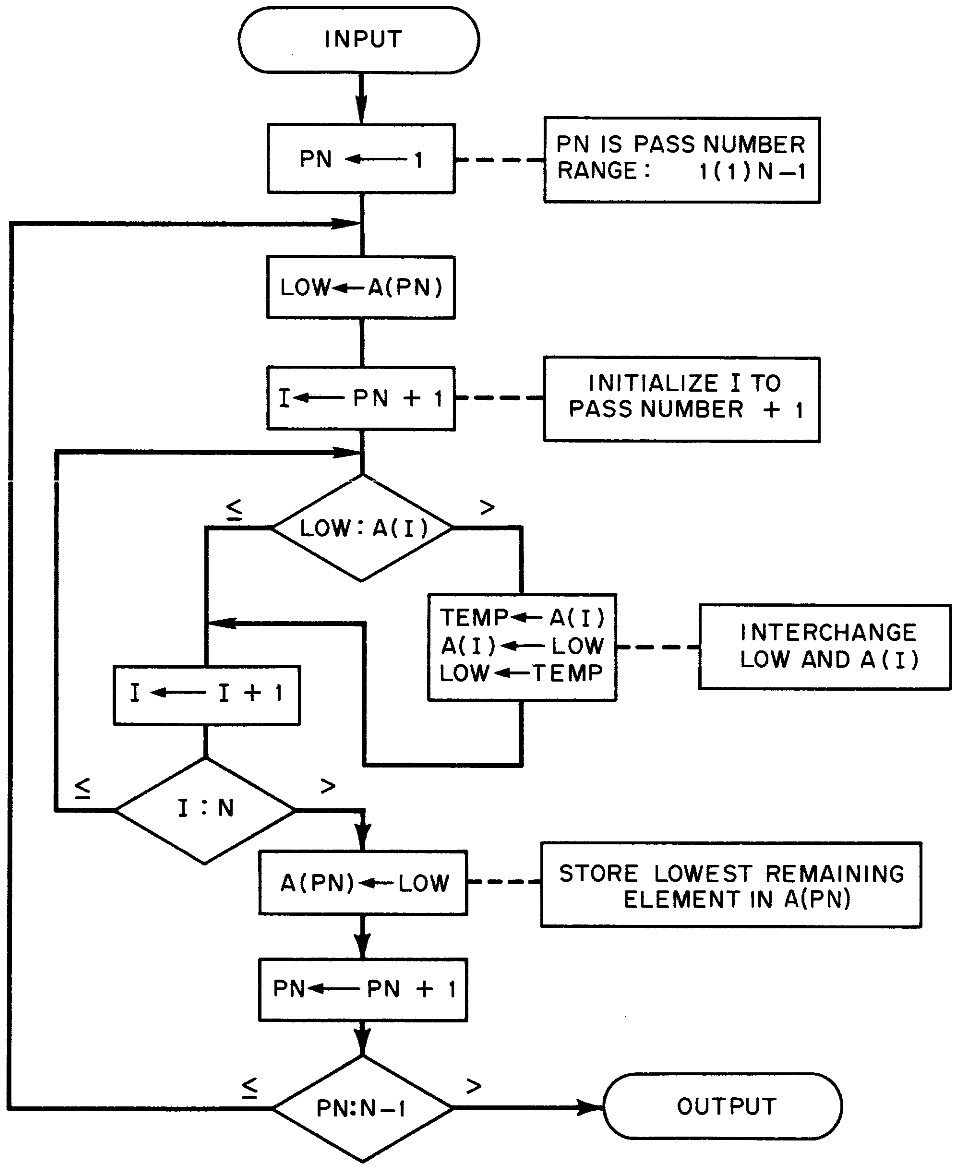


Fig. 4-9 Flow Chart for Example 4-7, SORTING

These considerations, while vital, are nonetheless secondary to the primary one of first coming up with a solution. After this has been done, and its logic captured in a flowchart, the programmer then can begin coding. The comments portion of each assembly statement should be used liberally to forge a connection between the program steps and the flow chart. There is also the advantage to a machine independent flow-chart that if some machine oriented item should be changed, such as changing from a seven-byte entry to a four-byte entry, the flowchart is still good so that the programmer has less rework to do.

Answers to Exercises

4-1 The code between GO and LASTA will be executed six times. The BCT at LASTB will be executed three times. When the instruction at FIRST is executed, GPR1 = 0 and GPR2 = 0.

4-2 a) The index values are 3, 10, 17, 24, and 31. The loop will be executed four times. After the last execution, the index will be incremented to 31 at which point the BXLE will not be taken.

b) -4, -1, 2, 5, 8, 11, 14; six iterations.

c) 25 times; 100, 96, 8, 4. The index value after the loop has been satisfied is 0.

d) for BXLE with parameters I(J)K, the number of iterations is (K-I)/J+1 where means "integer part of"; 3.9 = 3, for example. For BXH the formula is the same (exchange K and I to get a positive result) except for the +1 being omitted only when the division gives no remainder. As examples, 12(4)0 gives three iterations whereas 12(4)-1 gives four iterations

Problems

4-1 Show how to use the BCT to subtract one from a register and continue sequentially.

4-2 Suppose a section of code is to be repeated until some test is satisfied and it is desired to count the number of iterations (and show the result as a positive number). Show the result as a positive number. Show how this can be done using BCT.

4-3 Suppose GPR2 is loaded with 100 prior to the start of a loop and a BCT2 is placed at the end of the loop with an appropriate branch address. If the only operation performed on GPR2 during the loop is A 2,=F'1', can you determine how many times the loop will be executed? What if the instruction is A 2, = F'-1?

4-4 Determine the number of iterations, the first and last values of the index and its value after the loop has been satisfied for the following BXLE loop parameters: 1(1) 10, 0(2)512, -3(3) 27, 17(3) 191; BXH loop parameters 27 (3) 0, 400 (4) -3, 512(2)-2.

4-5 Referring to the payroll examples in Sec. 4.1, if the operand at SUBT were changed to DED-4(3), what changes would be necessary for NDXCON?

4-6 Referring to Example 4-1, if the operand at TEST were changed to BLOCK-4(1), what changes would be required for NDXCON? Are any other changes in order?

4-7 Referring to Example 4-1, add the necessary coding to check the presence of a word of 1's in word 10,001 after the loop has been satisfied. If this condition is not met, branch to ERROR.

4-8 Modify the coding of Example 4-1 so that the total number of positive, zero and negative entries will be tabulated and stored in PLUS, ZERO and MINUS, respectively.

4-9 Write a program to read into storage a deck of N+1 cards. The first card contains the number N in cc 1-2 (N=99). The next N cards each contain a part number which ranges from 1 to N in cc 1-2 and the inventory position for that part number in cc 3-6. Set up an inventory array large enough to contain the maximum number of items. Assuming that the input is not sorted by part number, the program should use the part number punched in the card as an index to generate the address in the inventory table where the inventory position will be stored as a binary number. When all cards have been processed, terminate the job. Also, if a part number is greater than N, print the message, THE FOLLOWING ERROR DATA HAS BEEN READ: followed by cc 1-6 of the error card, and then continue with the input. Should the entries in the inventory array be half-words or full-words?

4-10 Refer to the technique for computing a self-checking digit in Example 2-5. Using indexing, apply this technique to a 13 digit number, whose first 12 digits are the account number, with the 13th being the check digit.

4-11 Assume that we have a deck of cards containing census data for each of the inhabitants of a town. If cc. 20-21 contain the age at last birthday of a given inhabitant (00 indicates a child less than one year old, 99 indicates an age of 99 or greater), write a program which will read the census data and develop an age tabulation in a one-dimensional array in storage. The first entry should contain a count of all cards with 00 age; the second, 01 age, and so on up to the fifth entry which tabulates 04 age; the sixth tabulates ages

05-09; the seventh tabulates ages 10-14 and so on with the last entry tabulating ages 95-99. The last card contains a - in cc 80 and a card count in cc 70-79 which should be cross checked with the count developed by the program. When the last card is read, the tabulation should be printed in a column with the age or age range shown next to each entry in the column.

If the card count does not cross check, a message should be printed on the top of the page following the tabulation listing which indicates the number of extra or missing cards. Note: The DS statement which reserves storage for the age tabulation array will not set its locations to zero. This must be arranged for by the programmer prior to reading in the first card. Binary or decimal arithmetic should be used wherever either one is most convenient. Construct a small deck of census cards to test your program. Each of the error possibilities should be tested.

4-12 Assume that the census deck of Prob. 4-11 contains annual income rounded to the nearest dollar in cc 40-45 and number of years of schooling in cc 38-39. Write a program which will use the census deck to create a cross tabulation of years of schooling vs. annual income. The rows of the table should be years of school from 0 to 25. (row 25 indicates 25 or more years of schooling); the columns should be annual income in increments of 2000. That is, column one indicates annual incomes from $0 to $1, 999; column two, from $2, 000 to $3, 999, and so on with the last column indicating annual incomes from $98, 000 and higher.

4-13 Assume N variable length records are stored beginning at location DATA. Their lengths, which may range from four to 256 bytes, are indicated in binary by the first byte of each record. The next three bytes are the record key in binary. Write a sequential search routine to locate a given key. Note that if the length of the first record is known, the address of the second record can be found by adding the length of the first record to the location of the first record. This can be extended to address any field. The search should be programmed to search within N records and print an error message if the desired record is not found within the N records in storage.

4-14 The search in Prob. 4-13 can also be done by binary search. However, an argument- value table is required for a binary search of variable length records. The argument column would contain the keys for all records, sorted in ascending sequence. The value column would contain the storage locations corresponding to the record keys in the argument column. The binary search would be made in the argument column; when a match is made, the record address can then be taken directly from the value column. The desired record can then be fetched and processed. Write coding to create this argument-value table assuming a), that the records are already in ascending sequence, and b), they are not sorted.

Chapter 5

LOGICAL OPERATIONS

5.1 Bit Manipulation

Up to this point, we have been concerned with operating on data in units usually larger than a single byte. In the programs we have discussed so far, the data was organized either as individual words, or if variable length, as a group of consecutive bytes. However, it is often useful to employ single bits to contain discrete items of information. This is feasible when each piece of information has only two possible values. For one of these values, we may use a bit code of 0, for the other, a bit code of 1. Consider some of the items that would be part of an automobile insurance policy record. The record would have to show if options such as collision or theft are covered, or if the driver is under 25 years of age or if multiple autos are covered. Since these may be coded as yes or no responses, a single bit may be used to indicate the status of each. The programmer should always be on the alert for such situations which allow bit coding for they permit substantial saving of core storage and tape or disk storage. Since bit coding results in shorter physical records, the reading and writing times for these records will also be shorter so that processing time savings can also result from bit coding techniques. However, their principal justification lies in storage savings and in fact, bit coding may require additional programming beyond what would be required if full bytes were used instead.

So far, we have mainly been concerned with arithmetic operations. On bit-oriented data, additional operations are useful. They are the And, Or and Exclusive Or operations. The subject of bit operations is often referred to as logical operations because of the antecedents in symbolic logic of these three instructions. Another name for this topic is Boolean Operations after George Boole, the originator of the algebra of symbolic logic. We will introduce the use of these operations with an example. Suppose we have a tape file of personnel records. Each record contains an experience summary indicating whether the individual in question has had experience in finance (F), manufacturing (M), research (R) or sales (S). The experience summary is a Join-bit field which contains a 1 or 0 in each position in the order FMRS as a yes or no indicator. This file can be used for personnel searches. Typical searches may require individuals with backgrounds in finance or manufacturing (either or both backgrounds qualify); a search may be for people with backgrounds in manufacturing and sales (both are required to qualify); a search for a training program may require a background in finance or sales, but not both (this is the Exclusive Or). To determine if a particular experience summary -- let us call it a test field -- matched the desired profile, it would be operated on by the three logical instructions in conjunction with a mask whose bit pattern is related to the desired profile. Figure 5-1 shows the results of these operations on individual bits. It will be helpful in reading this figure, if the reader thinks of a 1 bit as a "yes and a 0 bit as a "no" (the designations True or False are equally satisfactory).

AND	0	1
0	0	0
1	0	1

OR	0	1
0	0	1
1	1	1

EXOR	0	1
0	0	1
1	1	0

Fig. 5-1 Bit definitions of And, Or, and Exclusive Or. For fields larger than one bit, these rules are applied to each bit position in sequence.

As an example, Fig. 5-1 shows that the AND function of a 0-bit and a 1-bit is 0. In the context of our example, if the search criterion is manufacturing and sales backgrounds, an individual with M = 0, S = 1 will not pass. As another example, the Exclusive Or of a 1-bit and a 1-bit is 0. That is, for a search of sales or finance but not both (Exclusive Or), a candidate with F = 1 and S = 1 would be rejected.

The reader should read through the balance of Fig. 5-1 making up similar examples as he proceeds.

The general approach to such problems as the personnel search is to set up a function which expresses the search criterion. Various masks will then be set up to determine if each test field matches the search criterion. To simplify the discussion, we will use the symbol + for Or, $\cdot$ for And, and x for Exclusive Or. To indicate a negative condition, we will place an overscore above the letter for that condition. For instance, to indicate a search criterion of no financial background,we write $\bar{F}$. To indicate a financial or a manufacturing background, the function is F + M; for financial and sales, we have F$\cdot$ S; manufacturing or sales but not both, MxS; either manufacturing or sales and finance is M + S$\cdot$ F but finance and either sales or manufacturing is F$\cdot$ (S + M). As an illustration, if M = 1, S = 0, F = 1, R = 0 for a given test field, the indicated functions have the following values:

Function	Value
F + M	1
F$\cdot$ S	0
MxS	1
M + S$\cdot$ F	1
F$\cdot$ (S + M)	1
Mx(S + R)	1

Test Field Format: F M R S

F + S (1xx0, 1xx1, or 0xx1, only, will satisfy the search criterion)

Operation	Mask	Result	Comment
And	1001	x00x	x's may be 1 or 0
Test for non-zero			Non-zero indicates match

FxS (1xx0 or 0xx1, only)

Operation	Mask	Result	Comment
And	1001	x00x	x's may be 1 or 0
Ex-Or	1000	000x	Match if zero, if not, go to next operation
Ex-Or	0001	0000	Match if zero

F· R (1x0x only)

Operation	Mask	Result	Comment
And	1010	x0x0	x's may be 1 or 0
Ex-Or	1000	0000	Match only if zero

F· (R + M) (110x or 101x or 111x)

Operation	Mask	Result	Comment
And	1110	xxx0	x's may be 1 or 0
Ex-Or	1100	0000	Match if zero, if not go to next operation
Ex-Or	1010	0000	(ditto)
Ex-Or	1110	0000	(ditto)

A better approach is to test for F = 1 using a mask of 1000 and if a match is made, test for R + M with a mask of 0110 as in the first example above.

Fig. 5-2 Search examples. Note that each Ex-Or is applied to the result of the first And, not to the result of the previous Ex-Or.

Functions containing parenthesized expressions should be evaluated starting with the expression in parentheses. As an example, consider the function Mx(S + R), S + R = 0 + 0 = 0, Mx0 = 1x0 = 1.

Exercise 5-1 a) Write the functions corresponding to manufacturing or finance or sales, manufacturing and either research or sales, finance or sales but not both, either sales or manufacturing and finance but not both. b) With S = F = 0 and M = R = 1, evaluate the following functions: M + F + S, M· S + F, MxF + SxR, Fx(S + M), M· S + FxR. c) Express in words the meaning of the functions in b).

We will next discuss how masks are developed and used. Suppose we are searching for candidates with financial and sales backgrounds; the function is then F· S. Since we are not concerned about the other two possibilities, research and manufacturing we can limit the test to the first and fourth bits in the test field which is in the form F M R S. We can isolate these bits by Anding the test field with a mask of 1001. This gives a result of x00x where x indicates either a 0 or a 1. Since the only result we are interested in is, 1001, if we Ex-Or the result with the same mask, a zero result will be obtained only for a test field with F - S = 1. As an example, if F = 1 and S = 0, that is 1000, an Ex-Or with a mask of 1001 gives a result of 0001. The zero or non-zero state of the final result can be determined by placing a Branch on zero instruction after the Ex-Or.

As further examples, Fig. 5-2 shows the masks and operation sequence for the functions, or search criteria, F + S, FxS, F + R, and F·(R + M). The Ex-Or operations in Fig. 5-2 are applied to the result of the first And operation, not to the result of the previous Ex-Or.

5-2 Logical Operations

Figure 5-3 describes the And, Or and Exclusive-Or instructions. The result of each operation is placed in the first operand location. The operation is performed bit by bit with each operand treated as an unsigned logical quantity. As examples, the comments field of the following statements shows the results of each operation. The first operand is 11011101 before each operation. Each instruction is considered to be independent of the others.

```
NI      OP1,B'11110000'          OP1 = 11010000
OC      OP1(1),=B'00000010'      OP1 = 11011111
XI      OP1,B'10101010'          OP1 = 01110111
```

For the immediate instructions (NI, XI) above, the second operand is not a storage address but is the data itself.

Operation Codes			Operand	Type
And	Or	Ex-Or		
NR	OR	XR	R1, R2	(RR)
N	O	X	R1, D2(X2, B2)	(RX)
NI	OI	XI	D1(B1), I2	(SI)
NC	OC	XC	D1(L1, B1), D2(B2)	(SS)

Fig. 5-3 Logical Instructions

Logical quantities may be compared using the compare instructions:

CLR	R1, R2	(RR)
CL	R1, D2(X2, B2)	(RX)
CLI	D1(B1, I2	(SI)
CLC	D1(L1, B1), D2(B2)	(SS)

The CLC was introduced in Chap. 2 and is included here for completeness. The comparison is binary with each operand treated as an unsigned binary quantity. We will discuss the condition code settings shortly.

Logical quantities can also be added or subtracted by the following instructions

ALR	R1, R2	(RR)
AL	R1, D2(X2, B2)	(RX)
SLR	R1, R2	(RR)
SL	R1, D2(X2, B2)	(RX)

Both operands are 32-bit unsigned quantities and all 32 bits participate in the operation. The result is stored in the first operand location. Subtraction is performed by adding the two's complement of the second operand to the first operand. The second operand is unchanged. The condition code setting is indicated in Fig. 5-4. The logical arithmetic instructions are useful when 32-bit numbers have to be added or subtracted such as in double-precision operations which we will illustrate in Problem 5-5. When logical quantities have to be shifted in the GPR, the following instructions may be used:

left-single	SLL	R1, D2(B2)	(RS)
left-double	SLDL	R1, D2(B2)	(RS)
right-single	SRL	R1, D2(B2)	(RS)
right-double	SRDL	R1, D2(B2)	(RS)

The logical shifts differ from the arithmetic shifts of Sec. 3. 9 in three important respects. First, the condition code is unchanged; second, all 32 or 64 bits participate in single or double shifts (for the arithmetic shifts, the sign bit is not shifted); third, zero bits are used when fill bits are required at either end of a GPR (the arithmetic shift instructions filled from the left according to the sign bit).

A useful instruction is the Test under Mask. Its format is

```
TM          D1(B1),I2          (SI)
```

The I2 field contains a byte of immediate data. The bytes of this mask are made to correspond one for one with the bits of the storage character specified by the first operand address. A mask bit of one causes the corresponding storage bit to be selected, a zero-bit causes the corresponding storage bit to be ignored. When all selected storage bits are zero, the condition code is set to 0; when all selected bits are one, the condition code is set to 3; if some of the selected bits are zero and some are one, the condition code is set

CONDITION CODE	0	1	2	3
BC INSTR. MASK BIT	8	9	10	11
MNEMONIC	BZ	BM	BP	BO
Add/Subtract (Arith.)	zero	minus	plus	overflow
Add/Subtract (Logical)	zero	not zero	zero*	not zero*
And/Or/Exclusive Or	zero	not zero	--	--
Compare Logical	equal	low	high	--
Test Under Mask	zero	mixed	--	one
Translate and Test	zero	incomplete	complete	--
* indicates carry out of sign position				

Fig. 5-4 Condition Code Settings

to 1. Figure 5-4 shows the condition code settings for a variety of logical and arithmetic instructions. In Chap. 3 we introduced mnemonics for some of the more frequent condition code settings, B2, BM, BP and BO. These nmemonies are principally useful for testing the results of arithmetic operations. The mnemonic value of BM, for instance, to test for mixed ones and zeros after a TM instruction is somewhat dubious. We recommend instead that the programmer use the conditional branch instructions covered in Sec. 3-8. For convenience their formats are repeated below.

```
BC          M1, D2(X2, B2)          (RX)
BCR         M1, R2                  (RR)
```

The M1 field is four bits long and comprises bits 8-11 of the branch instruction as indicated in Fig. 5-4. If any of the bits of the M1 field are matched by the condition code, the branch will be taken. As an example, if the following instruction is executed after a TM

```
                    BC      B'1001', LOCA
```

the branch will be taken, if the test field is either all zeros or all ones.

<u>Worked Example</u>

<u>5-1</u> Use the TM instruction for the personnel search criteria in Fig. 5-2. The test field is located in the four low order bits of a one-byte field, SEARCH. If a give test field meets the search criterion, branch to location OK, otherwise, branch to NEXTCASE. The coding follows:

```
*     SEARCH CRITERION   F + S
           TM            SEARCH, B'00001001'      SELECT F AND S
           BC            B'0101', OK
*     BRANCH IF SELECTED BITS ARE MIXED OR ALL ONES
           B             NEXTCASE
           --
*     SEARCH CRITERION FxS
           TM            SEARCH, B'00001001'
           BC            B'0100', OK
*     BRANCH IF SELECTED BITS ARE MIXED
           B             NEXTCASE
           --
*     SEARCH CRITERION  F· R̄
           TM            SEARCH, B'00001000'      SELECT F
           BC            B'0001', TESTR           BR IF F = 1
           B             NEXTCASE
   TESTR   TM            SEARCH,  B'00000010'     SELECT R
           BC            B'10001, OK              BR IF R = 0
           B             NEXTCASE
           --
*     SEARCH CRITERION F· (R + M)
*     TEST FOR F = 1 FIRST, THEN R + M
           TM            SEARCH,  B'00001000'     SELECT F
           BC            B'0001',  TESTRM         BR IF F = 1
           B             NEXTCASE
  TESTRM   TM            B'0101', OK
*     BRANCH IF SELECTED BITS ARE MIXED OR ALL ONES
           B             NEXTCASE
```

5.2 Data Packing

Very often the length of a data field is not a multiple of eight bits. When this occurs, the programmer has the option of either rounding the field length up to the next multiple of eight bits, or packing adjacent fields within a single byte. The first option wastes storage and input-output time, the second requires additional instructions for repacking the fields prior to processing them and then repacking for output. While a decision between the two depends on the specifics of a given situation, it is not unusual for byte packing to be favored because of the storage savings it allows.

As an example of the technique, assume that a 32-bit field is packed as follows in a word labeled DATA:

Bits	Label
0-11	NUMBER
12-14	DEPT
15-21	CLOCK
22-31	RATE

Unpack each field and store them in half-words. Each field contains a positive number. The code follows:

```
                LR      2,DATA
                SRDL    2,10
                SRSL    3,22
                STH     3,RATE
                SRDL    2,7
                SRSL    3,25
                STH     3,CLOCK
                SRDL    2,3
                SRSL    3,29
                STH     3,DEPT
                SRDL    2,12
                SRSL    3,20
                STH     3,NUMBER
                --
                --
                --
CLOCK           DS      H
RATE            DS      H
DEPT            DS      H
NUMBER          DS      H
DATA            DS      F
```

Logical shifts are used to avoid propagating sign bits through GPR3.

5.3 Byte Manipulation

The following two instructions are provided to load and store characters from the GPR to main storage:

STC	R1, D2(X2, B2)	(RX)
IC	R1, D2(X2, B2)	(RX)

The Store Character instruction (STC) stores the low order eight bits of R1 (bits 24-31) into the character located at the second operand address. The Insert Character instruction (IC) loads a character from the second operand address into the low order eight bits of R1. Instructions are also provided to allow for translation between byte codes. We may regard a given byte code as comprising a set of binary numbers from 0 to 255 (00000000 to 11111111). If we wish to translate from one code to another, we must set up a dictionary of 256 entries with each entry in the table giving the translated code corresponding to its position in the sequence. That is, the bytes of the source code (translate from) are used as indices to retrieve the corresponding byte from the target code (translate to).

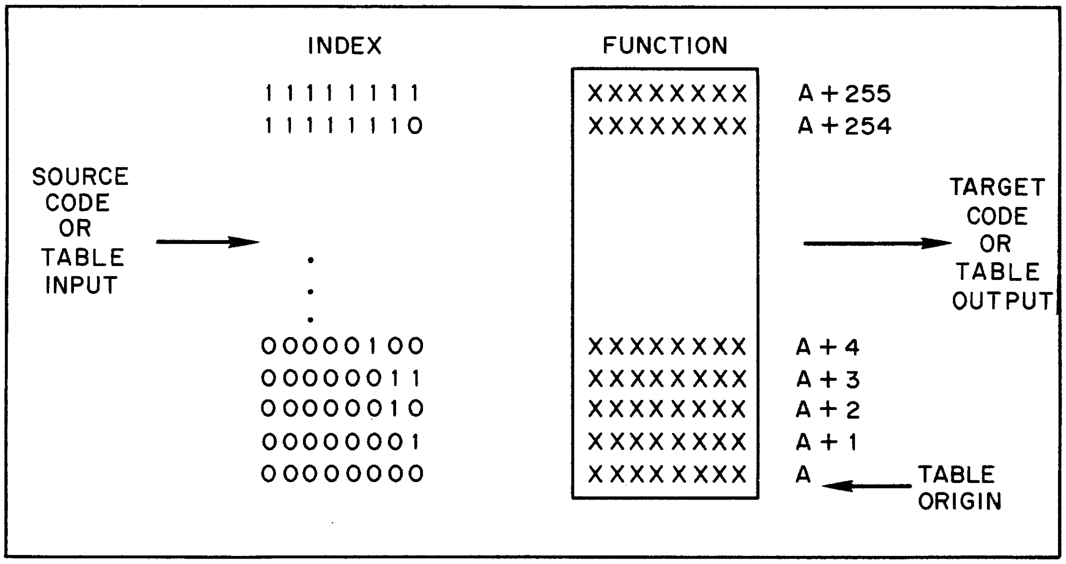


Fig. 5-5 Translate Table

As an example, for a code translation in which the code for %, 11001001, would be translated to the code for (, 11010001, the 78th entry (78 = 11001001) in the translation table would contain 11010001. There are two 360 instructions which do this table look-up automatically. The formats of these instructions are:

```
TR      SOURCE(L), TARGET     (SS)

TRT     SOURCE(L), TARGET     (SS)
```

The Translate (TR) instruction uses each byte in the SOURCE field to index a byte from the TARGET table. The selected byte from the TARGET table replaces the index from SOURCE. The operation proceeds from left to right, one byte at a time, until all L bytes of SOURCE have been translated. The translate table, TARGET, is addressed by adding each SOURCE byte to the address of the second operand, that is, to the initial address of the translation table. Since all 256 source codes are valid, a 256-byte target table is recommended. In the event that a particular source code contains less than 256 valid characters, we have two options if verification is necessary. Each source record may be scanned, a character at a time, to detect any invalid characters. However, this can be a lengthy operation if the invalid codes are scattered throughout all 256 possible combinations. The other alternative is to translate all invalid codes to a single target character and then scan the translated record using a Test Under Mask or Compare Logical Immediate instruction. The Translate and Test (TRT) instruction differs from the Translate instruction in that source bytes are not replaced. Instead, the target byte is examined after each source byte has been processed. If the target byte is zero, the process continues. If it is non-zero, the address of the source byte is loaded into bits 8-31 of GPR 1 and the target byte is loaded into bits 24-31 of GPR2. The instruction is then terminated. After the instruction is executed, the condition code is set as follows:

Condition Code	Meaning
0	All target bytes are zero
1	Non-zero target byte encountered before end of source field
2	Last target byte is non-zero
3	--

As an example, the TRT can be used to detect invalid characters in a data record or it can be used to detect delimiters between variable length records. For these two applications, the translation table would be filled with zeros except for the entries corresponding to invalid characters or delimiter characters. It should be clear that it is only the context of a particular application that determines if a character is invalid. As an illustration, if we knew that a data record contained only EBCDIC numbers and possibly blanks, then any other characters would be invalid.

Worked Example

5-3 A 10000-byte field is reserved for a variable length tape record. The last character in the record is 11111111. In addition, each tape record can contain up to 10 variable length fields separated by the delimiter 01010101; this delimiter also follows the last field. It is assumed that these two characters occur only as delimiters and not as data characters. Create a table of full-word entries which contains the number of variable length fields in the entire record (NUMBER), the size of the entire record (SIZE), and the length of each field in L(1) through as many entries as necessary up to L(10).

One of the interesting aspects of this problem is finding a way to use the Translate and Test instruction on a 10000-byte field - remember that the length field in the instruction is only 8 bits wide. One way is to use an operand of 0(256, 1) in the TRT to address the record with GPR1 initialized with its beginning location. This will then allow a scan of 256 bytes at a time. If a delimiter is found, its address will be stored in GPR1. We will store this in the appropriate entry in the L table since these addresses can be used to compute individual field lengths. We can get ready for the next scan by adding 1 to GPR 1 and branching back to the TRT instruction. (If we omitted the "add 1" step, the next scan would start on the previous delimiter which will give a bit immediately and cause the program to go into an endless loop.) The translate table will have 0's for its first 85 entries and 01010101 for its 86th since 01010101 = 85. The next 169 entries will be zero and the last entry, the 256th, will be 11111111. Actually, we need not use these specific characters, any two different, non-zero characters would do as well.

The coding follows:

```
            MVI     LAST,X'FF'    STORE 11111111 AT END OF VRECORD
            LA      1,VRECORD     GPR1 IS BASE REGISTER
            SR      3,3           CLEAR GRP3
SCAN        TRT     0(256,1),TABLE TEST FOR 01010101 OR 11111111
*   IF CONDITION CODE IS NON-ZERO, DELIMITER(01010101) OR
*   END OF RECORD (11111111) HAS BEEN FOUND, OTHERWISE
*   SCAN NEXT 256 CHAR OF VRECORD, NOTE THAT END
*   OF RECORD CHARACTER IS STORED AFTER VRECORD FIELD
*   A SAFEGUARD AGAINST MISSING END OF RECORD CHARACTER
            BC      B'1000', NXT256 IS CC = O
            CLI     0(2),B'11111111'  CHECK FOR END OF REC
            BE      ENDREC
            ST      1,L(3)         STORE DELIM LOC IN L
            LA      3,4(3)         ADD 4 TO GPR3 FOR NEXT STORE
            LA      1,1(1)         ADD 1 TO GPR1 FOR NEXT TEST
            B       SCAN           SCAN NEXT 256 BYTES
*   STARTING LOCATION FOR SCAN IS ADDR OF DELIMITER + 1
NXT256      LA      1,256(1)       ADD 256 TO GPR1 FOR NEXT SCAN
```

```
         B       SCAN
ENDREC   S       1,=A(VRECORD)     GPR1 CONTAINS LOC OF '11111111'
         ST      1,SIZE            SIZE = GPR1 - A(VRECORD)
         SRA     3,2               GPR3 CONTAINS 4*NO. OF RECS
         ST      3,NUMRECS
```

At this point, we will leave it to the reader to develop code to calculate the individual record lengths. Note that L(1) through L(NUMBREC) contain the location of the delimiter character which follows the last character of the record.

```
         --
         --
         --
TABLE    DC      85 x '00'
         DC      B'01010101'       01010101 = 85
         DC      169 x '00'
         DC      B'11111111'       11111111 = 255
VRECORD  DS      10000CL1
LAST     DS      CL1
SIZE     DS      F
NUMRECS  DS      F
L        DS      10F
```

5.4 Instruction Modification

Very often, a programmer will have occasion to modify an instruction, that is, to operate on the instruction as if it were data. Suppose, for instance, that we are dealing with variable length fields which have to be moved from an input area to a work area. This can be done by setting up a "dummy" move instruction such as MVC WORK(0), INPUT. Prior to each move, an instruction will be executed which stores the length (minus one) of the field in the length segment of the MVC instruction, bits 8-15. In this section, we will discuss two approaches to instruction modification: The direct approach above and an indirect approach which does not require the contents of storage to be changed.

Before getting into the details, let us pause to consider some of the implications of instruction modification. First, since direct instruction modification involves bit-level operations particularly if operation codes are being changed, the likelihood of programming errors is increased. Second, instruction modification usually complicates a program which makes it more difficult to test the program -- a point not to be treated lightly, since testing a program often takes more time than coding it. In addition, the documentation for such a program is more complex and hence difficult to read so that subsequent changes may be very troublesome to implement. As a final point, when a program contains instruction modification, it will be more difficult to translate that program for a different computer. It is possible to design a program which will translate an assembly language program for one computer to the assembly language of a different computer. In general, a high percentage of the code can be translated. However, the instruction modification portions in particular are usually impossible to translate by an automatic procedure. It is our conclusion that a programmer should avoid instruction modification, if at all possible. The result may be a less sophisticated program but more likely one that stands a better chance of being completed on time and in working order.

When instruction modification is unavoidable, the Execute instructions allows it to be done more cleanly than by bit manipulation and often, at a saving in instructions. The format of the Execute instruction is

EX	R1, D2(X2, B2)	(RX)

This instruction causes the instruction located at the second operand address to be executed. Unless the referenced instruction is a branch, control will be returned to the instruction immediately following the Execute. Branches will be executed properly. The referenced instruction will be modified by or'ing bits 24-31 of R1 with bits 8-15 of the referenced instruction. The or'ing is effective only during instruction execute time. The contents of storage are not changed by the or'ing nor are the contents of R1. The underlined portions of the instructions in Fig. 5-6 can be modified by Execute.

OP	R1, R2	(RR)
OP	D1(B1), I2	(SI)
OP	R1, R3, D2(B2)	(RS)
OP	R1, D2 (X2, B2)	(RX)
OP	D1 (L1, B1), D2(L2, B2)	(SS)
OP	D1(L, B1), D2(B2)	(SS)

Fig. 5-6 Instruction Modification by Execute

In each case, the sequence of the instruction bits through 8-15 are indicated properly. For instance, in the first SS instruction in Fig. 5-6, L1 occupies bits 8-11 and L2 bits 12-15. If only one element is underlined, then it occupies all eight bits, 8-15, of the modifiable portion of the instruction. Any instruction, except another Execute, may be the subject of an Execute. Note also that the lengths in SS-type instructions are one less than the field lengths of their operands. Up to this point, we did not have to be concerned with this since the assembler subtracted one from all implied or explicit lengths prior to assembling SS instructions. If the programmer modifies SS instructions, he must keep this in mind.

In the succeeding Worked Examples, we will show several illustrations of instruction modification, both by direct and by indirect methods.

Worked Example

5-4 A section of coding labeled INIT which is located in the middle of a program should be executed only once. On successive passes through that section, a branch should be executed to the instruction immediately following the INIT section which will be labeled NEXT. The following coding shows how this can be done by direct instruction modification.

```
START     --
          --
BRANCH    BC        B'0000', NEXT
          NI        BRANCH+1, B'1111'
INIT      --
          --
          --
          --
NEXT      --
          --
          B         START
```

On the first pass, the instruction at BRANCH will be treated as a NOP. Immediately after, it will be modified by the And Immediate (NI) to an unconditional branch. On the next pass, the INIT section will be branched over.

Replacing this logic with Execute instructions has no particular advantage. However, the same effect can be obtained with a compare instruction where one of the operands is modified during the first pass. In the interest of keeping program logic as straightforward as possible, this approach is to be preferred. As an illustration, assume that a one-byte field, CHAR, is set to zero by an instruction which is located above START and therefore, is executed only once. The coding follows:

```
          CLI       CHAR, X'00'
          BNE       NEXT
INIT      OI        CHAR, X'FF'
          --
          --
          --
NEXT      --
          --
```

Worked Example

5-5 Assume that a program contains a generalized routine to add together two variable length packed decimal fields, A and B, and then move the result at A to an output area, OUTPUT. Add the necessary coding to modify the AP and MVC instructions to allow different lengths of A and B to be inserted prior to each usage of the routine. The lengths (minus one) of A and B, LA and LB, are found, respectively, in bits 0-3 and 4-7 of a one-byte field, LGT. The coding follows

```
          MVC    ADD+1(1), LGT      MOVE LENGTHS
ADD       AP     A(0), B(0)         LA AND LB ARE BITS 8-11, 12-15
          IC     1, LGT             GPR1 CONTAINS LA AND LB
          SRA    1,4                SHIFT OUT LB
          STC    1, MOVE+1          STORE LA IN MOVE+1
MOVE      MVC    OUTPUT (0), A
```

The shift instruction is required to delete LB from GPR1 and put LA in the low-order four bits of the eight-bit length field of the MVC immediately following.

Worked Example

5-6 Redo the coding of Example 5-5 using Execute instructions wherever possible. The coding follows

```
          IC     1, LGT             LA AND LB IN GPR1
          EX     1, ADD             MODIFY ADD INSTRUCTION
          SRA    1,4                SHIFT OUT LB
          EX     1, MOVE            MODIFY MOVE INSTRUCTION
          --
          --
          --
          --
A         DS     16CL1
B         DS     16CL1
ADD       AP     A(0), B(0)
MOVE      MVC    OUTPUT(0), A
OUTPUT    DS     132CL1
```

These instructions will be executed in the sequence IC, EX, AP, SRA, EX, MVC followed by whatever instruction occurs immediately after the second Execute.

Answers to Exercise

5-1 a) $M + F + S$; $M\cdot(R + S)$; FxS; $Sx(M\cdot F)$.

b) $S = F = 0,\ M = R = 1$:

$M + F + S = 1$	(True)
$M\cdot S + F = 0$	(False)
$MxF + SxR = 1$	(True)
$Fx(S + M) = 1$	(True)
$M\cdot S + FxR = 1$	(True)

c) Manufacturing or finance or sales; either manufacturing and sales or finance; either manufacturing or finance, but not both or either sales or research, but not both; either finance or sales or manufacturing, but not both; manufacturing and sales or either finance or research but not both.

Problems

5-1 Using the bit numbering convention 01234567 for the eight bits in a byte, write instructions to perform the following operations on the one-byte field located at DATA: a) Set bits 0 and 6 to 1 without changing the other bits: b) Set bits 3 and 4 to 0 without changing the other bits. c) Reverse bits 0 - 3 and leave bits 4 - 7 unchanged.

5-2 If $F = 0$, $S = 1$, $M = 1$ and $R = 0$, what is the value of the following functions

a)	$F\cdot R + S$	d)	$S\cdot M + S\cdot R$
b)	$SxM\cdot R$	e)	$S\cdot(M + R)$
c)	$R\cdot(SxM)$	f)	$Mx\bar{R}$

5-3 With 1 and 0 replaced by True and False, respectively, express verbally the meanings of 5-2 a) through f). Specifically, what is the difference between b) and c).

5-4 With the bits corresponding to FMRS in bits 4-7 of the byte located at SEARCH, write instructions to branch to TRUE or FALSE according to the value of functions 5-2 b) and 5-2 c) above. Treat each as a separate case.

5-5 As we discussed in Chap. 3, double precision quantities(64-bit integers) can be considered as being made up of a positive 32-bit quantity in the low-order word and a signed 31-bit quantity in the high order word. Write a program to add together two 64-bit integers located in A, A + 4, B and B + 4. Store the result in C and C + 4 where A, B and C are full words. (Hint: Use the AL instruction to add the low-order 32-bit quantities. If an overflow occurs, add a 1 to the high order part. Should an arithmetic or logical add be used on the high-order quantities?) Also, if the sum is greater than 63 bits, branch to OFLO and print the message, OVERFLOW. If the addition can be handled successfully, print the result together with A and B. Make up enough test cases to try the various possibilities in your program and then run your program with this data.

5-6 Using the translate instructions, write code to translate from digits from ASCII to EBCDIC. If the input contains any characters other than digits or a blank, branch to ERROR.

5-7 The design and coding of a simulator presents some interesting programming challenges. A simulator is a program which will take a program written for one computer -- the source computer -- and allow it to be run on a different computer -- the target computer. The simulator is written in the language of the target computer. Essentially, the simulator and the source program to be simulated both reside in the core storage of the target machine (in their object code formats, not assembly language formats). The simulator performs the functions of the control circuitry of the source computer. It retrieves source instructions one at a time, deciphers them and carries out the intended operation. A key part of the simulation is operation code analysis. A typical way of doing this is to create a table of operation codes for the source machine and a corresponding table of locations of the simulation routines to carry out the indicated operation. Given a source instruction operation code, a table look-up can be performed and then a branch can be taken to the location of the simulation routine which will execute the operation. The translate instruction allows this to be done at a considerable saving in time, assuming that the source machine has an 8-bit operation code, design a routine using the Translate instruction to do the operation code analysis. Show how the output from the translate table can be used to set up a branch to an operation simulation routine.

5-8 Another challenging programming problem is a program which simulates programs written for the same machine. This is not as circular an exercise as you may think. Such a program can be very useful as a testing device for routines which prove difficult to debug by other means. Since the simulator program is, in effect, tracing its way through the source program an instruction at a time, it can display some very useful debugging information on the computer's Cathode Ray Tube or the printer, if a CRT is not available. It can output the location of the instruction being executed, the location and contents of its operands before and after execution. This gives the programmer detailed information about the flow of his program and the state of the machine at every step. He can then trace through the (voluminous) output and find the bugs. Very often, this type of simulator is called a trace program. Write a trace for 360 source programs which include only the instructions covered in

Chap. 2. Use the Translate instruction to produce assembly language mnemonics in the trace listing for the Operation codes (assume that all mnemonics are 32-bit words, padded with blanks, as necessary, for those which require less than four bytes).

Test your program on one of the worked examples in Chap. 2. If you are still in the mood for further intellectual challenge, add enough capability to your trace program's repertoire to allow it to trace itself.

5-9 Which logical instruction can be used to replace the second MVN (Move Numeric) in Worked Example 2-4?

5-10 Suppose that a particular data file contains only packed decimal data. How can the Translate instructions be used to detect invalid data, i. e. other than packed decimal? Describe the contents of the translation table. Remember that a byte of packed decimal data contains any of 10 decimal digits in its high-order part and the 10 decimal digits and possibly the + and - signs in the low-order part. As a result, there are 10 x 12 = 120 possible valid bytes.

5-11 In (partly) Worked Example 5-3, we put off calculating the field lengths until after the entire record had been scanned. However, it is somewhat simpler to calculate the lengths after each delimiter is encountered. Add the necessary coding to do this.

5-12 Example 5-4 gives the coding for a one-pass switch, that is, a branch which falls through on the first pass but transfers each subsequent time. Write the coding using direct instruction modification or an alternating switch which passes through or branches on alternate executions. (Hint: Consider exclusive or operations.)

5-13 Recode your solution to Prob. 5-12 using an approach other than direct instruction modification.

5-14 Can the shift operation in Example 5-5 be replaced by an appropriate Unpack instruction or by a Move with Offset?

5-15 In general, the Execute instruction is useful whenever a group of instructions require an identical modification such as an index register or a length. The desired modification can be stored in one of the GPR and then referred to as often as required by Execute instructions. This often produces a savings of instructions such as in Example 5-6. Make up an example where the Execute instruction can be used to save storage and then code it.

Chapter 6

FLOATING-POINT OPERATIONS

6.1 Floating-Point Numbers

In the examples we have considered so far, scaling or decimal point placement has presented no exceptional problems. The range of numbers in a calculation was known in advance and was usually fairly small. That is, in a typical division problem, the divisor might range from say, 0.XX to XXX.XX so that the quotient would require up to three additional digits. However, there is a large class of calculations where scaling is very difficult. The sequence of these calculations is usually long when compared to the kinds of problems we have discussed so far and the range of numbers involved is often very large. Sacling these calculations can be extremely difficult. With enough persistence and core storage and machine time, these problems could be programmed by using multiple precision techniques. That is, at every step of the calculation the largest possible result could be anticipated and enough words of storage set aside for it. An alternative solution is to let the computer manage all scaling. This can be done by storing each number in two parts: The digits making up the number itself and a quantity indicating how many decimal places the number contains. The result is called a floating-point number. The storage architecture of the computer will impose a limitation on the number of digits per floating point word. As we shall see, for the 360 this is about 7 or 16 decimal digits depending on which form of floating point number is used - single-precision or double-precision. If numbers larger than this maximum size are developed, say in multiplication, enough low order digits are dropped or truncated to bring the result within allowable size. This limitation on the number of significant digits restricts the use of floating point numbers principally to scientific computations. In addition, floating point operations take longer to execute than their equivalent fixed point operations - that is, operations on integers as in Chap. 3. This difference is more pronounced on the smaller models of the 360.

In the interests of readibility, we will introduce floating-point operations with decimal examples. Subsequently, the hexadecimal representation used by the 360 will be covered. Reverting to our hypothetical computer with a 10-digit word size, we will use the first two digits of a word to indicate the placement of the decimal point and the low order eight digits for the number itself. All numbers will be represented with their decimal point at the extreme left and multiplied by the appropriate power of 10 to reset the decimal point to its original location.

With the decimal point at the extreme left, floating-point numbers are in the normalized format. As examples,

$$
\begin{aligned}
21.75 &= .2175\text{x}100 = .2175 \text{ x } 10^2 \\
21234.5 &= .212345\text{x}100000 = .212345\text{x}10^5 \\
-.2122 &= -.2122\text{x}1 = -.2122\text{x}10^0 \\
.0123 &= .123 \div 10 = .123\text{x}10^{-1} \\
-.0012 &= -.12 \div 100 = -.12\text{x}10^{-2}
\end{aligned}
$$

These numbers will be represented in storage as a two digit exponent (the power of ten) and an eight digit fraction (the number itself). For the moment, assume that both the exponent and the number may have their own sign. While this is not the case in most computers -- specifically the 360 -- because there is only one algebraic sign per word, this limitation can be easily sidestepped as we will soon illustrate. The numbers above are then (an overscore indicates a negative quantity)

Number	Exponent	Fraction
21.75	02	21750000
21234.5	05	21234500
-.2122	00	$2122000\bar{0}$
.0123	$0\bar{1}$	12300000
-.0012	$0\bar{2}$	$1200000\bar{0}$

Addition or subtraction is performed by first comparing the exponents of both operands and then right-shifting the fraction with smaller exponent until it is properly aligned. The fractions are then added. As an example, consider the addition of A = 123456.75 and B = 23.456. In floating point we have

A	06	12345675
B	02	23456000
B shifted	06	00002345
A & B	06	12348020

Note that the low order digit of B has been shifted out of the sum. In fact, if B were small enough compared to A, the entire number would be shifted out of the sum. This circumstance is due to the fixed word size of floating point numbers. While the limitation to a fixed number of digits may not be objectionable in most scientific or engineering computations, the blind usage of floating point operations can create problems. Some of these will be discussed in this chapter. Additional references are given in the Bibliography. Floating point hardware on a computer will relieve the programmer from a lot of tedious work. However, his responsibility to know the sizes of numbers his program will treat is still unchanged.

Additional shifts will be used to normalize the result if a high order zero occurs or on an overflow out of the high order digit. As examples, consider the floating-point calculations

```
  02 61000000   ( = 61.0)          03 62500000   ( =  625.0)
+ 02 73000000   ( = 73.0)        - 03 61300000   ( =- 613.0)
  ───────────                      ───────────
  02 34000000   + overflow         03 01200000   ( =   12. )
  03 13400000   ( = 134.0)         02 12000000   ( =   12. )
```

In the first example, an overflow causes the fraction to be right shifted one place which increases the exponent by one. In the second, high order zeros have occurred and are eliminated by shifting the fraction to the left one place thereby reducing the exponent by one. This normalization after a calculation is termed post-normalization. It is interesting to consider what would happen if both operands in the second example were equal in their first five or six digits. The result would be a number which contains only two or three significant digits although the numbers which gave rise to it contained up to eight significant digits. This is a serious matter with very practical implications. If both terms represented physical measurements obtained with great difficulty to eight significant digits, we would have a result which contained only three significant digits. It is clear that the subtraction of nearly equal floating-point quantities should be avoided, if at all possible. Very often, alternative mathematical formulations can be developed. As an example, the expression

$$y = 1 - \cos x$$

is mathematically equivalent to

$$y = 2 \sin^2 (x / 2)$$

The second expression will take slightly more machine time but avoids the difficulties of the first one when x is very small since cos x is then very nearly equal to one.

Multiplication and division with floating point numbers proceeds by adding or subtracting the exponents and then performing the desired operation on the fractions. As an example, consider the product of 123.0 x 22.0. Expressing these numbers in normalized form, we have

$$\begin{aligned}(.123 \times 10^3) \times (.22 \times 10^2) &= .123 \times .22 \times 10^3 \times 10^2 \\ &= .123 \times .22 \times 10^{3+2} \\ &= .02706 \times 10^5 \\ &= .2706 \times 10^4\end{aligned}$$

The floating point result is

```
   03  12300000    (= 123.0)
 x 02  22000000    (=  22.0)
   05  02706000    (= 2706.0)
   04  27060000    (= 2706.0)
```

To divide 2706. 0 by 22. 0, we have

$$\begin{aligned}(.2706 \times 10^4) / (.22 \times 10^2) &= (.2706 / .22) \times 10^4 \times 10^{-2} \\ &= (.2706 / .22) \times 10^{4-2} \\ &= 1.23 \times 10^2 \\ &= .123 \times 10^3\end{aligned}$$

Let us return to the issue of how to handle floating point numbers when the computer has only one sign bit per word. The most straight forward solution is to use the sign bit for the algebraic sign of the number and then arrange to have all exponents represented as positive numbers. This can be done by restricting the range of exponents from -99 to +99 down to -50 to +49 and then adding 50 to the exponent. That is, numbers with exponents larger than +49 or smaller than -50 cannot be handled. If a calculation arises with numbers outside this range, it will have to be scaled. As examples of the excess-50 notation,

21.75	=	52 21750000
21234.5	=	55 21345000
-.2122	=	50 2122000$\bar{0}$
.0123	=	49 12300000
.0012	=	48 1200000$\bar{0}$

What we have said about addition and subtraction remains unchanged in excess -50 representation. There is a slight difference, however, with multiplication and division. In multiplication, the exponents of both factors are added and then 50 is subtracted from the sum. In division, the exponents are subtracted and 50 is added to the sum.

Exercise 6-1 Using the excess 50 notation, how would you represent the number zero? Justify your choice.

The 360 floating point instruction set operates in principle like the floating point feature of our hypothetical machine. The 360 allows two floating point forms, the short or single-precision form and the long or double-precision form. Their storage requirements are respectively 32 and 64 bits. For both, bit 0 is used as the sign bit for the fraction which is stored in its true form for negative numbers; two's complements are not used. The next seven bits are used for the exponent and the remaining 24 or 56 bits are used for the fraction. Figure 6-1 shows a schematic of these formats. The fraction is organized into 6 or 14 hexadecimal digits. All shifts are therefore done in increments of four bits. As a result, each bit in the exponent represents a power of 16. If the fraction were given as a binary number, each exponent bit would represent only a power of two. As we shall see, this would significantly reduce the exponent range

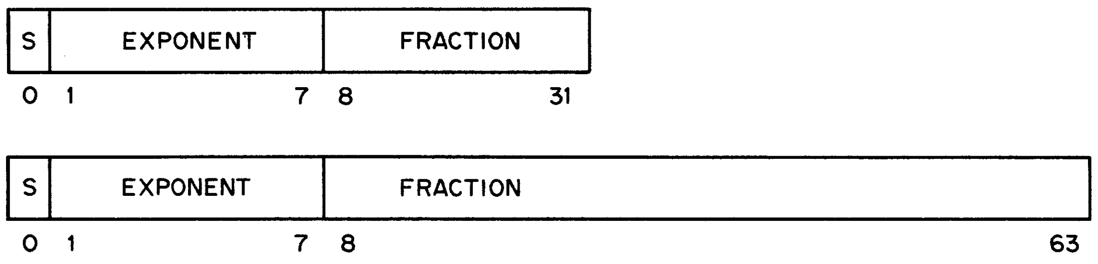


Fig. 6-1 Floating-Point Data Formats
Single and Double-Precision

With seven bits for the exponent, the mid range is $1000000_2 = 64$ so that an excess -64 representation will be used. This allows an exponent range of +63 to -64 which is 16^{63} to 16^{-64}. The largest number which can be represented in the normalized single precision form is

$$01111111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111 = .111111111111111111111111 \text{x} 16^{63}$$
$$= .7237006 \times 10^{76}$$

The smallest is

$$10000000\ 0001\ 1111\ 1111\ 1111\ 1111\ 1111 = -.000111111111111111111111 \text{x} 16^{-64}$$
$$= -.5397605 \times 10^{-78}$$

In double-precision, these numbers are essentially unchanged -- they will have 32 bits of either ones or zeros appended to the right. Their decimal equivalents would contain about 16 or 17 significant digits whereas in single-precision, we have seven to eight significant digits. If each fraction bit were used for a single binary digit, the exponent range would be reduced to 2^{63} and 2^{-64}, or approximately $.8 \times 10^{19}$ and $.5 \times 10^{-20}$. Zero is represented as a single or double word of all zero bits.

Conversions between decimal and floating-point are done using the same techniques for binary and decimal conversions we discussed in Chap. 3. The following worked examples illustrate the method.

Worked Example

6-1 Convert the number 1356.726 to single precision floating point.

We will first express the integer and fractional parts of the number in their hexadecimal equivalents.

```
         84/12 = C
 16 /1356                        .726
      128                         x16
       76                        4356
       64                        726
       12                      11.616       = .B
      5/4       = 4               x16
16 / 84                          3696
     80                          616
      4                         9.856       = .09
      /5        = 5               x16
16 / 5                           5136
                                 856
                               13.696       = .00D
```

$1356_{10} = 54C_{16}$

$.726_{10} = .B9D_{16}$

The fraction is carried out to only three hex digits because the single precision form allows only six hex digits per word and in this example, three have already been used for the integer portion. We now have

$$1356.726_{10} = 54C.B9D_{16}$$
$$= .54CB9D \times 16^3$$

The floating point exponent is then 64 + 3, and since the number is positive, the sign bit is zero. The complete floating point word is therefore

Sign	Exponent	Fraction
0	43	54CB9D
0	1000011	0101 0100 1100 1011 1001 1101

To convert this back to decimal, we subtract 64 from the exponent giving a result of 3. This indicates that the point is between the third and fourth hex digits. That is, the integer part lies in the first three hex digits and the remainder, the three low order hex digits, constitutes the fractional part. Converting these hex digits to decimal, we have

$$\begin{array}{r l}
5 & \\
\underline{\text{x}16} & \\
80 & \\
\underline{+4} & \\
84 & \\
\underline{\text{x}16} & \\
1344 & \\
\underline{+12} & (= C) \\
1356 & = 54C
\end{array}$$

$$\begin{aligned}
B/16 &= 11 \text{ x } .0625 \\
&= .6875 \\
9/16^2 &= 9 \text{ x } .00390625 \\
&= .03515625 \\
D/16^3 &= 13 \text{ x } .000244140625 \\
&= .003173828125 \\
.B9D &= .725830078125
\end{aligned}$$

The final result is

$$54C.B9D = 1356.725830078125$$

We have a deceptively large number of digits in the result. This is not due to the intrinsic precision of the number we started out with, but simply that we carried out the conversion to as many digits as possible. Note that the fourth fractional hex digit could be anything between O and F (if we didn't know in advance the initial decimal number). As a result, the decimal fraction could be off by as much as $F/16^4 = .0002$. In light of this, we are justified in regarding the result as valid to only three decimal places and rounding off accordingly which gives a result of .726, the decimal fraction we started out with. However, there is no guarantee that things will always work out this nicely. Because of the difficulties inherent in converting fractional numbers from one base to another, the reconversion will often give a result which is not exactly equal to the initial number.

Incidentally, note that the normalized form of a hexadecimal number may contain high order zeros. In the example above, one high order zero is present. This comes about because the fraction is normalized, not to the first non-zero bit, but to the first non-zero hex digit which may range from 1 to F. As a result, up to three high order zeros may occur. This is the slight tradeoff paid for hexadecimal normalization; for most applications it is more than offset by the much greater exponent range permitted by hexadecimal normalization when compared to the alternative of binary normalization.

6.2 Normalized Floating-Point Operations

Floating point operations on the 360 are performed in any one of four 64-bit floating-point registers (FPR) which have addresses 0, 2, 4 and 6. The formats of these registers are shown in Fig. 6-1. Double-precision operations use all 64 bits of a FPR. Single-precision operations, with the exception of multiplication use only the high order 32 bits of each FPR and do not change the contents of the low order 32 bits. Single-precision multiplication gives a 56-bit fraction. To allow for greater accuracy, single and double-precision addition and subtraction, which give a final result of six or 14 hex digits, will include an extra digit called a guard digit. The guard digit does not occupy space in the FPR's. If one of the operands is shifted right during the operation, a guard digit will be formed. If the subsequent result requires post-normalization, the guard digit will be left-shifted into the sixth or 14th hex digit result, otherwise the guard digit is not used. As an example, consider the subtraction of F. 11111_{16} from 10. 2222_{16}. The floating point representations of these numbers are

```
42  10222          (= .10222 x 16²)
41  F1111          (= .F11111 x 16)
```

Subtraction will cause the second number to be right shifted one hex digit creating a guard digit of 1. The result is

```
 42  102222
-42  0F1111       1     guard digit
 ----------
 42  011111       1
```

With post normalization this becomes

```
42  111111
```

where the guard digit has been shifted into the final result.

Four kinds of program interrupts can be generated by floating-point operations. They are exponent underflow, significance loss, exponent overflow and floating-point divide. These interrupts have the following causes:

Exponent Underflow: An arithmetic operation gives an exponent less than -64 (0 in excess -64 notation). The operation is completed by replacing the result with a true zero -- that is, 32 or 64 zero bits.

Significance Loss: The result fraction after an addition or subtraction is zero.

Exponent Overflow: An arithmetic operation gives an exponent greater than +63 (127 in excess -64 notation) and a non-zero fraction. This causes the operation to be terminated with the result data being unpredictable and therefore unsuitable for further calculation.

Floating-Point Divide: A division by a number with zero fraction is attempted. The operation is suppressed and the data in registers are unchanged.

Two of these interrupts, significance loss and exponent underflow, may be masked out by setting the appropriate bits of the Program Status word to zero. Depending on the problem, at hand, these conditions may not indicate difficulties in the calculation and so, interrupts need not be taken. However, when the significance mask bit in the PSW is zero (interrupt not taken), the result is set to true zero, otherwise the exponent and sign are unchanged. The other two conditions, exponent overflow and floating-point divide interrupt are much more likely to indicate serious program malfunctions when they occur. As a result, it is reasonable to have an interrupt when they occur. Depending on the operating systems option selected by the installation, when an interrupt occurs, the offending job may be terminated or an exit can be made to a user written routine to process the interrupt. The condition code is set by addition or subtraction, by compare instructions and by sign-control operations. Multiplication, division, loading and storing leave the condition code unchanged. The various condition code settings are summarized in Fig. 6-2 following the discussion of the floating-point instruction set.

The floating-point instruction set provides 44 different operations. As an assist to the reader's memory, we offer the following clues to deciphering floating point operation mnemonics. The first letter indicates the type of operations: A, S, M, D for addition, subtraction, multiplication and division, L for load, C for compare. Some mnemonics require two letters for the type such as store (ST) load and test (LT), load complement (LC), load positive and load negative (LP and LN). Apart from these exceptions, the second letter distinguishes between normalized double and single precision operands by a D or an E, respectively. The third letter, again with the exception of the two-letter mnemonics above, indicates the instruction class, an R signifies register-to-register (RR) operations and a blank indicates register-to-storage indexed operation. As examples, AER is the operation mnemonic for normalized single-precision register-to-register floating-point addition, and DD is an RX double-precision divide. Results replace the first operand, except for storing operations, where the second operand is replaced by the first. Also all storage locations of single-precision operands must be multiples of four and locations of double-precision operands must be multiples of eight, that is, they must be aligned on single or double word boundaries.

The following instructions load the first operand location with the contents of the second operand location. FPR1 and FPR2 are the addresses of floating-point registers and may have any value from 0, 2, 4, 6.

LE	FPR1, D2(X2, B2)	(RX)
LD	FPR1, D2(X2, B2)	(RX)
LER	FPR1, FPR2	(RR)
LDR	FPR1, FPR2	(RR)

These two instructions are used to store, the high order 32 bits (STE) or all 64 bits (STD) of a FPR into a storage location.

STE	FPR1, D2(X2, B2)	(RX)
STD	FPR1, D2(X2, B2)	(RX)

The following RR instructions allow sign control. Load and Test (LT) loads the first operand register and sets the condition code according to the sign of its contents. Load complement (LC) loads FPR1 and reverses the sign bit. Load Position (LP) loads FPR1 and sets the sign bit to zero and Load Negative performs the reverse operation, setting the sign bit to one.

LTER	FPR1, FPR2	(RR)
LTDR	FPR1, FPR2	(RR)
LCER	FPR1, FPR2	(RR)
LCDR	FPR1, FPR2	(RR)
LPER	FPR1, FPR2	(RR)
LPDR	FPR1, FPR2	(RR)
LNER	FPR1, FPR2	(RR)
LNDR	FPR1, FPR2	(RR)

Addition and subtraction are performed by the following instructions. The single-precision instructions operate only on the high order 32 bits of each FPR. The low-order 32 bits of each FPR are unchanged and do not enter into the operation. The result, which is placed in FPR1, is normalized.

AE	FPR1, D2(X2, B2)	(RX)
AER	FPR1, FPR2	(RR)
AD	FPR1, D2(X2, B2)	(RX)
ADR	FPR1, FPR2	(RR)
SE	FPR1, D2(X2, B2)	(RX)
SER	FPR1, FPR2	(RR)
SD	FPR1, D2(X2, B2)	(RX)
SDR	FPR1, FPR2	(RR)

Multiplication is provided for with the following instructions. Note that both single and double-precision products occupy all 14 fraction hex digits of a FPR. For single-precision products, the two low-order hex digits are always zero since no more than a 12-digit product can be developed from two six-digit operands. The result, which is stored in FPR1, is normalized. Also both operands are normalized, if necessary, prior to the multiplication. This

ME	FPR1, D2(X2, B2)	(RX)
MER	FPR1, FPR2	(RR)
MD	FPR1, D2(X2, B2)	(RX)
MDR	FPR1, FPR2	(RR)

is in effect only for the purposes of multiplication, it does not alter the storage representation of an operand. Division is accomplished by the following instructions. The single-precision operations yield a six hex digit quotient while double-precision gives a 14 hex digit quotient. In neither case is a remainder preserved. Note that single-precision operations use only the high order 32 bits of a FPR and ignore the remaining bits. For both types, the quotient will be normalized and both operands are treated as though they

DE	FPR1, D2(X2, B2)	(RX)
DER	FPR1, FPR2	(RR)
DD	FPR1, D2(X2, B2)	(RX)
DDR	FPR1, FPR2	(RR)

were pre-normalized, as with multiplication.

An operation analogous to division is provided for by the Halve operation. The second operand is moved to the first operand location and its fraction shifted right one bit. No postnormalization takes place. The formats for the Halve operation are

MER	FPR1, FPR2	(RR)
MDR	FPR1, FPR2	(RR)

Floating-point comparisons can be made through the following instructions

CE	FPR1, D2(X2, B2)
CER	FPR1, FPR2
CD	FPR1, D2(X2, B2)
CDR	FPR1, FPR2

The comparison is algebraic, taking into account the sign, exponent and fraction of each number. In effect, a floating-point subtraction is performed and the condition code is set according to the first operand being equal, lower than or greater than the second operand. A compare does not change either operand.

Figure 6-2 below gives a summary of condition code settings for floating point operations. The condition code may be tested using the BC instruction with appropriate mask field or by the equivalent mnemonics, BE, BH, BL, BP, BM, BZ, BO, BNH, BNL and BNE. Testing the condition code does not change it.

	Condition Code			
	0	1	2	3
BC Mask Bit	8	9	10	11
<u>Operation</u>				
Add/Subtract	zero	minus	plus	overflow
Compare	equal	low	high	--
Load and Test	zero	minus	plus	--
Load Complement	zero	minus	plus	--
Load Negative	zero	minus	--	--
Load Positive	zero	--	plus	--

Fig. 6-2 Condition Code Setting for Floating-Point Arithmetic. The condition code is set according to the sign of the result, which is in the first operand, or for Compare, if the first operand is equal, low or high relative to the second operand. Note that a true zero has a plus sign.

As an aid to reading the following examples, floating point constants may be defined using a DC with an operand E for single-precision and D for double precision. The constant is written in quote marks. If a decimal point is present, it is used to establish the exponent, if not, the number is assumed to be an integer. If necessary, the number may be written with an exponent in the form En which indicates the power of 10 the number is to be multiplied by prior to converting to the internal representation. As examples, consider the following (the comments field indicates the number being defined)

```
FLOAT      DC    E'-315.65'        = -315.65 SINGLE
CONA       DC    D'268'            =  268.0 DOUBLE
CONTBL     DC    E'2, 3, .5'       =  2.0, 3.0, 0.5 SINGLE
A          DC    E' -2.5E5'        = -250000 SINGLE
           AE    2,=E'1.75E-3'     =  .00175 SINGLE
```

Storage may be reserved by a DS statement using the operands E and D for single and double-precision data, respectively.

<u>Worked Examples</u>

6-2 Compute

$$Y = A + B*C/D$$

The coding for single precision operands is

```
         LE     0,B                 FPRO  =  B
         ME     0,C                 FPRO  =  B*C
         DE     0,D                 FPRO  =  B*C/D
         AE     0,A                 FPRO  =  A+B*C/D
         STE    0,Y
```

If there was a possibility that the product of B and C could produce an exponent overflow, the division should be placed before the multiplication.

6-3 Compute the expression in Example 6-2 above where Y is now a double precision number.

It will not suffice to replace the STE instruction with an STD. While the latter will store all 64 bits of FPRO in Y, the low order 32 bits are absolutely irrelevant as the preceding single precision instructions ignore these 32 bits. Instead, we must perform the division and addition in double precision. The multiplication can be done in single-precision because this will save some machine time, and more important, double-precision cannot improve on the accuracy of the product which is limited to 12 hex digits. In problems such as this, which involve mixing single and double-precision numbers, we can convert from the short to the long form in one of two ways. We can load the single-precision number into a FPR whose low-order 32 bits are zero and treat it thereafter as a double precision number. Or, if a FPR is not available, the single-precision number may be moved to the high-order part of a doubleword in core storage whose low order 32 bits are zero and thereafter, it may be treated as a double-precision word. The disadvantage of the latter technique is the storage to storage move it requires. The coding is then

```
           LE     0,B
           ME     0,C       B*C TO 12 HEX DIGITS
CLEAR      LD     2,=D'0'   CLEAR FPR2
           LE     2,D       LOW ORDER 32 BITS OF D = 0
           DDR    0,2       FPRO = B*C/D IN DOUBLE-P
           LE     2,A       LOW ORDER 32 BITS OF A = 0
           ADR    0,1       FPRO = A+B*C/D IN DOUBLE -P
           STD    0,Y
```

The instruction at CLEAR could be replaced by SDR 2,2 which would also set all 64 bits of FPR2 to zero. In some problems, it may be desirable to perform the calculations in double-precision to gain additional accuracy even though the final result is a single precision number. As an example, suppose the numbers 115, 11.5 and 1.5 were added in that order on a computer with three fractional digits per single-precision word. The result would be 127. Using double-precision addition, we get the correct sum, 128, which can then be stored as a single precision number. This illustration is an extreme one and we do not recommend that all floating-point operations be done in double-precision. However, where circumstances warrant, the use of double precision will give increased accuracy.

6-3 Compute

$$Y = A + \quad B-C \ /E$$

Where means "absolute value", that is, set the sign of the quantity in brackets to plus. All factors are double-precision.

The coding is

```
LD     4,B               FPR4  =  B
SD     4,C               FPR4  =  B-C
LPDR   4,4               FPR4  =    B-C
DD     4,E               FPR4  =   B-C   /E
AD     4,A               FPR4  =  Y
STD    4,Y
```

6-4 Compute

$$Y = (A+B) / (A+C) \quad + \quad (A+D) / (A-B)$$

where all factors are single precision.

The coding is

```
LE     6,A
AE     6,B               FPR6  =  A+B
LE     4,A
AE     4,C               FPR4  =  A+C
DER    6,4               FPR6  =  (A+B) / (A+C)
LE     4,A
SE     4,B               FPR4  =  A-B
LE     2,A
AE     2,D               FPR2  =  A+D
DER    2,4               FPR2  =  (A+D) / (A-B)
AER    2,6               FPR2  =  Y
STE    2,Y
```

6-5 FPR4 contains a double precision product. Round it to six hex digits and store in location PROD.

The rounding will be done by adding eight to the seventh hex digit of the product. If the seventh digit is eight or greater, the sixth digit will be rounded up by one. The exponent and sign will be loaded into the low-order eight bits of the roundoff constant and then added to the product. The coding is

```
                STD    4,PROD
                MVC    RCON(1),PROD   MOVE EXP, SIGN
                AD     4,RCON
                STE    4,PROD
                --
                --
                --
     RCON       DS     OD             BOUNDARY ALIGNMENT
                DC     4X'00'
                DC     X'80'          8 IN 7TH HEX DIGIT
                DC     3X'00'
     PROD       DS     E
                DS     E
```

Note that PROD is aligned on a double-word boundary and that the sign of the round-off constant will be the same as the product.

<u>6-6</u> A binary integer less than 2^{24} is stored in FIX. Convert it to a single-precision floating point number and store the result in FLOAT.

There are three elements to be either converted <u>or</u> calculated: the sign, exponent and fraction. The sign can be picked up easily, the fraction may have to be complemented and will require normalization. The exponent will not be greater than six (46 in excess -64) since the point is to be moved six hex places (24 bits) from the extreme right to the extreme left of the single-precision fraction. Subsequent normalization may reduce this exponent by one or more.

The coding is

```
            L      0,FIX       LOAD FIX IN GPRO
            LPR    1,0         GPR1 HAS ABSOLUTE VALUE
            N      0,MASK      GPRO CONTAINS SIGN
            O      0,EXP       ADD EXPONENT OF X'46'
            ALR    0,1         GPRO HAS UN-NORM NUMBER
            ST     0,TEMP
            LE     2,EXP
   ADD      AE     2,TEMP      NORMALIZE NUMBER
            ST     2,FLOAT
            --
            --
```

```
FIX      DS     E
MASK     DC     X'80000000'     1-BIT IN SIGN POSITION
EXP      DC     X'46000000'
TEMP     DS     E
FLOAT    EQU    TEMP            SAVES ONE WORD
```

The addition of a floating-point zero and the unnormalized number at ADD will give a normalized result since the AE instruction post-normalizes, if necessary.

6.3 Unnormalized Floating-Point Operations

The addition and subtraction instructions discussed in Sec. 6.2 have the property that their results are always in normalized form even though their operands may not be normalized. This is accomplished by post-normalization of the results. In this respect, they differ from multiplication and division which pre-normalize both operands and post-normalize the results, if necessary. Often, unnormalized addition and subtraction is a very useful facility. Perhaps the primary application of unnormalized operations is to non-arithmetic manipulation of floating-point numbers such as conversions between floating-point numbers and decimal and binary number bases. In addition, unnormalized operations do not force exponent underflow. As a result, the range of numbers which can be handled goes down from $.1_{16} \times 16^{-64}$ to $.000001_{16} \times 16^{-64}$ in single-precision and to $.00000000000001_{16} \times 16^{-64}$ in double-precision. This is significant only when numbers of this magnitude are being operated on. As an illustration of unnormalized addition, consider the sum of the following unnormalized hexadecimal numbers:

```
 42  024689   (A)
 41  001234   (B)
 42  000123   (B shifted)
------------
 42  0248BC   (A+B)
```

In normalized floating addition, the operation would be the same except that the sum would be normalized as 41 248BC4. As can be seen from this example, one of the disadvantages of unnormalized addition/subtraction is that it provides for less precision than normalized operations. With unnormalized operands, which are likely to contain high order zeros, there is a greater chance that significant digits will be shifted out of a FPR when exponent alignment occurs as in the example above. The worked examples following this section will discuss several applications of unnormalized floating-point operations.

The mnemonic scheme introduced in Sec. 6.2 needs slight modification to accommodate unnormalized operations. Single and double-precision operands are indicated by U and W respectively. Below are the unnormalized addition and subtraction mnemonics and their corresponding normalized operation mnemonics

Normalized	Unnormalized	
AE	AU	(RX)
AER	AUR	(RR)
AD	AW	(RX)
ADR	AWR	(RR)
SE	SU	(RX)
SER	SUR	(RR)
SD	SW	(RX)
SDR	SWR	(RR)

Their operation is similar to normalized addition and subtraction except that the result is not normalized, exponent underflow does not occur and the guard digit is not used. The condition code settings remain the same. Multiplication and division can be performed on unnormalized operands. However, the results are in normalized form.

Unnormalized constants may be defined by using a scale factor from 0-13 to cause right shifting. As examples

```
CONA        DC        ES4'13.75'
CONB        DC        DS7'216.364'
```

The scale factor for CONA, SA, causes the normalized equivalent of 13.75 to be right shifted four hex digits; CONB will be right shifted seven hex digits. When significant digits are shifted out of a word by the scale factor specification, the assembler will round-off prior to truncating the lost digits.

Worked Examples

6-7 A double-precision word is stored in FLOAT. Create two double-precision words containing the integer portion (INT) and the fractional portion (INT), both in unnormalized form, INT with its point at the extreme right and FRAC with its point at the extreme right of a double-precision word.

The coding is straightforward once we have the integer portion. This can then be subtracted from FLOAT giving as a result, the fractional part. Perhaps the easiest way to get at the integer part is to cause the fractional digits to be shifted out. This can be done by adding FLOAT and a floating-point zero with its points at the extreme right, that is, with an exponent of 64 + 14 = 78 = 4E. The coding is

```
              LD      0, FLOAT            FLOAT IN FPRO
              LDR     2, 0                        AND FPR2
ADD           AD      0, X14              INT IN FPR0, NORMALIZED
              SDR     2, 0                FRAC IN FPR2, NORMALIZED
              AW      0, X14              INT IN FPR0, UN-NORM.
              AW      2, X0               FRAC IN FPR2, UN-NORM.
              STD     0, INT
              STD     2, FRAC
              --
              DS      OD
X0            DC      X'0000000000000000'
X14           DC      X'4E00000000000000'
FLOAT         DS      D
FRAC          DS      D
INT           DS      D
```

Note the normalized addition at ADD. This is necessary since a normalized result is required at this stage. If the result was not normalized, subtraction from FLOAT would cause FLOAT to be right shifted precisely the amount necessary to truncate its fractional digits which would then give a zero result. As an example, suppose float had nine integer digits and five fractional digits; FLOAT would then have an exponent of 64+9. An unnormalized addition would give INT with an exponent of 64+14. When this was subtracted, in turn, from FLOAT which has a smaller exponent, FLOAT would be right shifted five places causing the subtraction to yeild zero, not FRAC. Using a normalized add, FLOAT will still be right shifted five places to align it properly with X14. However, after the addition, the result, which is INT, will be postnormalized with an exponent of nine.

6-8 Discuss how both parts of the floating point number in Example 6-7 can be converted to decimal.

Since INT is in the form of an integer, that is, its point is at the extreme right, it can be treated as a 54-bit binary integer and converted to decimal using a technique similar to the one in Example 3-14. As to the fractional part, let us assume that only a few significant digits are required so that we can consider it to be a single-precision number. This assumption does not restrict the method particularly, it does allow us to avoid certain complications as we shall see shortly. Treating FRAC as a short-form floating-point number, we have a 24-bit binary fraction with its point between bits seven and eight and its sign in bit zero. For our present purposes, bits 1-7 contain extraneous data and will have to be removed. For the moment, if we consider FRAC as an integer, its largest value is $2^{24} - 1$, or 16, 777, 215. If FRAC is converted to decimal and then divided by 16, 777, 215, we will have the decimal fractional equivalent of FRAC. Since FRAC contains six hex digits, the seventh hex digit would be at most $F/16^7 = 1/16^6 = .000000060$. This implies that the decimal equivalent of FRAC should not be carried out to more than seven places. That is, when a seven place number such as .1234567 is given, as far as the eight place is concerned, any number in the range of .12345665 to .12345674 could satisfy the original seven-place number. While we could convert the six hex digit number FRAC to any number of places, we know that the effect of dropping its low order half (digits seven through 14) could be an error of as much as .6 in the seventh decimal place. As a result, only seven decimal places will be calculated. To avoid this detailed analysis, a useful rule of thumb is 1.2 decimal digits per hex digit. The final point to be considered is the sign of FRAC. If it is negative, then FRAC should be complemented. The code follows

```
          L     3,FRAC
          N     3,MASK      CLEAR OUT EXPONENT,SIGN
          LT    0,FRAC      TEST SIGN
          BP    0,CONTINUE
          LCR   3,3         COMPLEMENT FRAC
CONTINUE  SLA   3,7         ALIGN POINT AT LEFT
          M     2,TEN7      MULTIPLY BY 10000000
SHIFT     SLDA  2,1         SHIFT SIGN OF GPR3 TO
                            BIT 31 OF GPR2
          --
          --
FRAC      DS    E
MASK      DC    X'00'
          DC    7X'FF'
TEN7      DC    F'10000000'
DFRAC     DS    D           00000000.XXXXXXXS
```

In order to decipher this coding more readily, the reader should review Example 3-15, which introduces the conversion method used above.

Answer to Exercise

6-1 The most sensible representation for zero in excess 50 notation for our hypothetical computer is 00 00000000. If we used 50 00000000, then zero in this representation when added to a number such as .0000000001, which is 41 10000000, would give a result of zero because prenormalization would shift the digit one out of range. In addition, using the latter representation for zero, we have zero comparing larger than .0000000001, again because of prenormalization which is necessary for exponent alignment. To avoid these anomalies, a full word of zeros is used.

Problems

6-1 Following are several floating point numbers in excess -50 notation.

Perform the indicated operations and show all results in normalized form.

A	=	52	12340012	A-B
B	=	52	12160011	C-A
C	=	53	01234321	C+D
D	=	50	00012121	C-E
E	=	55	01234321	E+D
F	=	53	00000000	F+D

6-2 a) Using the numbers in Prob. 6-1, perform the indicated operations with unnormalized results.

b) With the same numbers, perform the following calculations with unnormalized results: (C+B) + F; (C+F) + B. Repeat, normalizing the result after each step.

6-3 Convert the following numbers to floating-point using hexadecimal and excess -64 representation. Show your results to 6 hexadecimal digits: .5, 16, 2.4071, 1.1, .999, 53'5.31'.

6-4 Verify your results by converting your results in 6-3 back to decimal.

6-5 Are there any advantages to using SWR over SDR to clear a floating-point register?

6-6 Two single precision factors are multiplied to produce a double-precision product. If the result is stored using STE, the low order eight hex digits will be ignored. Construct a routine which will round this product to six hex digits prior to being stored by STE. Test your program on a 360 using positive and negative factors.

6-7 In Example 6-6, what would be the effect of replacing the ALR by an AR instruction? Will any other instruction perform the desired function?

6-8 Code the Indian problem using floating-point arithmetic. Convert the result to decimal and round-off to two places. Compare the code in the loop part of your program with the code in the loop of the fixed-point version of Sec. 4.1. While this is too small a sample to prove the point, it usually turns out that a given section of fixed-point code can be done in about half the number of instructions when floating-point arithmetic is used. The difference in the number of instructions required help to offset, to a degree, the extra time taken by floating point instructions.

6-9 Develop a technique for rounding off floating-point numbers to a fixed number of decimal places. Remember that the excess digits must be truncated after the roundoff constant is added.

6-10 Modify the code in Example 6-8 to round the high order part of the product to the nearest bit. Remember that the sign position of the odd register of an even-odd pair, here GPR3, contains data when both registers contain a double binary word. The sign bit for the double word is in the sign position of the even register. Remember, too, that the low-order word of a double word is considered positive, irrespective of the sign of the high-order word. See Example 3-14 for additional information on double-word binary data.

6-11 In general, floating-point arithmetic does not satisfy the fundamental axioms of arithmetic -- the Associative and Distributive axioms. Develop numerical examples to verify this statement. For reference, the Associative axioms for multiplication and addition are A(BC) = (AB) C and A +(B+C) = (A+B) + C; the Distributive axiom is A(B+C) = AB + AC. Floating-point arithmetic on the 360 does satisfy the Commutative axioms -- A+B = B+A and AB = BA -- that is, interchanging operands does not change the result of addition or multiplication.

6-12 Polynomial evaluation is frequently encountered in scientific computing. A polynormal is shown below using the notation * to multiply, A(N) for a_n and X**N for x^n:

A(0) + A(1)*X + A(2)*X**2 + . . . + A(N)*X**N

For the purposes of machine computation, this may be "nested" as

(((((A(N)*X + A(N-1))*X + A(N-2))*X + . . . + A(1))*X + A(0)

In the nested, or telescoped form, polynomial evaluation can be programmed more efficiently. Write a routine to evaluate a nested polynomial for N = 8. Concentrate on reducing execute speed as much as possible. Assume that X and the A's are single-precision numbers. Their sums should be carried out in double precision with the final result rounded and stored in single precision.

6-13 Square roots can be computed by using an algorithm such as the one below. If the number, X, whose square root is to be computed has exponent E and fraction F, then a first approximation to its square root, Yo, is given by

Yo = ((8F + 2)/9)*16** E/2 (E is even)

Yo = ((32F + 8)/9)*16** E/2 (E is odd)

where E/2 indicates the integer part of E/2. This approximation is then refined by four iterations of the form

$$Yn+1 = (Yn + X/Yn)/2$$

to better than 56-bit accuracy. To start the iteration, substitute Yo for Yn in the formula, thencompute Y, and substitute it, in turn, for Yn and compute Y2 and so forth. The final approximation will be Y4. As a test on your program, include the calculation of X - Y4*Y4 and run it on a 360. If your program is correct, the difference should be very nearly equal to zero. To avoid the complications of writing a general-purpose decimal/floating-point conversion program, for output, convert to decimal only the exponents; use hexadecimal notation for fractional values -- that is, convert each hex digit to one of the EBCDIC symbols: 0, 1, 2, ... , A, B, C, D, E, F. For each test case, show the original number, its square root and the difference above. Use DC statements to provide six different test cases.

<u>6-14</u> Give a conceptual description of a generalized routine for conversions between double-precision floating-point and decimal numbers in the form EE. XXXXXXXXXX where EE is a two digit signed number representing the exponent of ten and the X's are a sequence of 10 decimal digits representing the fractional part of the number in normalized form. The decimal input and output should be in this form.

Take as operating assumptions, first, the case where core storage for this routine is extremely tight so that the emphasis is on minimum storage requirements; second, assume that several thousand bytes of storage are available for the routine and any tables it may require -- in this case, emphasis is on minimum conversion time.

It is interesting to note that improvement in one or the other parameter, core storage or running time, is detrimental to the other. This is not an unusual circumstance in programming.

Chapter 7

MACROS

7.1 Introduction

In many programs there are functions, represented by a sequence of instructions, which are used over and over again without change or perhaps only slightly modified. If these instruction sequences could be represented by one assembly language statement, the programmer could save time by writing one statement instead of many everytime that set of instructions is used. This substitution of one statement for many is the essence of a macro instruction.

A macro instruction, as the prefix "macro" implies, is a collection of assembly language statements which when executed in a particular sequence perform a desired function. The function is specified by the programmer in one macro instruction even though a number of instructions may be necessary to accomplish it.

The macro instruction offers the programmer greater flexibility and ease in writing and maintaining his program. Once a macro instruction is defined it can be used in a program wherever the function is needed. The assembler will generate the appropriate sequence of instructions to perform the function. If the function is of general use it may be put in a library and used in any assembly language program that needs that function. If the function performs incorrectly or must subsequently be changed to incorporate additional facilities, then only the macro instruction definition must be changed and the program reassembled. This is considerably easier than correcting or modifying the sequence of instructions at every place they appear in a single program or in a series of programs.

In the Disk Operating System, a macro must be in the macro library before it can be used in a program. The catalogue function of the operating system's library program will allow a macro definition to be placed in the macro library. The larger operating systems allow a macro definition to be assembled with a program which requires it in addition to being included in the system's macro library.

Macro instructions provide the first step in allowing the assembly language programmer to call upon a function without coding all the necessary machine instructions everytime he requires it. In the next chapter, we will discuss an extension of this capability, subroutines.

7.2 Macro Definition and Expansion - Positional Macros

The heart of the macro instruction technique is the definition of the macro instruction. Once the definition of the macro is communicated to the assembler, the assembler will generate the necessary instructions according to the definition everytime the macro instruction is encountered in the user's program.

Up to this point, we have seen that the assembler recognizes two types of statements: The machine instruction statement which causes the assembler to generate the appropriate machine instruction and the assembler instruction statement which informs the assembler of a condition but does not cause a machine instruction to be generated. The macro definition is composed of both types of statements. The macro definition will contain both a macro header and a macro trailer statement. In between these two statements will be the macro prototype statement and the model statements. There are two types of macro prototype statements, positional and keyword. Macros using the positional prototype statement will be discussed first.

To define a macro the first statement used is the macro header statement whose format is

blank	MACRO	blank

This statement informs the assembler that a macro definition follows. As the blanks indicate, this statement is not labeled nor does it have operands.

The last statement in a macro definition is the trailer statement whose format is

blank	MEND	blank

The trailer statement informs the assembler that the macro definition finished.

The statements which are included between the macro header and trailer statements define the macro to the assembler. They do not cause the assembler to generate any machine instructions.

The macro positional prototype statement follows the header statement. Its format is

symbolic parameter or blank	a symbol	up to 49 symbolic parameters separated by commas

The symbol in the operation field may be any unique name. It cannot be the name of any instruction mnemonic which is already known to the assembler. This symbol is the name of the macro whose definition follows.

The symbolic parameters which may be used in the name and operand fields are symbols whose first character is an ampersand followed by one to seven alpha-numeric characters the first of which must be alphabetic. The symbolic parameters are used in the model statements which follow the prototype statement. When the macro is used in the program, the parameters which follow in the operand field are then substituted for the symbolic parameters when the code is generated. The generation of code by the assembler for a macro is termed a macro expansion. An example of a positional macro prototype statement is

```
&NAME  NET  &GROSS,&RATE
```

Net is the name of the macro and &NAME, &GROSS, &RATE are symbolic parameters. Following the prototype statement are the model statements which define the macro expansion. A model statement can be any machine or assembly language statement with the exception of END, ICTL, ISEQ, PRINT, and START. The model statement can have a symbolic parameter in any field except the operation field. That is, symbolic parameters can be used in the name, operand and comments fields of the model language statement.

The following example gives the macro definition for the macro NET which calculates net salary from gross income and a tax deduction rate. NETSAL and TEMP are assumed to be 5 and 6 byte fields, respectively.

```
        MACRO                          MACRO HEADER
&NAME   NET      &GROSS,&RATE          PROTOTYPE
&NAME   MVC      NETSAL,&GROSS         MODEL STATEMENTS
        MVC      TEMP,NETSAL
        MP       TEMP,&RATE
        NI       TEMP+3,X'F0'
        O1       TEMP+3,X'0A'
        SP       NETSAL,TEMP(4)

        MEND                           MACRO TRAILER
```

The model statements define the macro NET. These statements do not cause any code to be generated but merely serve to define the macro NET to the assembler.

If a programmer wished to use the macro NET in his program, he may write the macro statement as follows

```
INCOME     NET     SALGROSS, TAXRATE
```

When the assembler encounters this statement, and NET has been defined as above, the assembler would generate the following instructions in place of the macro instruction NET

```
INCOME     MVC     NETSAL, SALGROSS
           MVC     TEMP, NETSAL
           MP      TEMP, TAXRATE
           NI      TEMP+3, X'FO'
           OI      TEMP+3, X'OA'
           SP      NETSAL, TEMP(4)
```

The parameters INCOM, SALGROSS, and TAXRATE replace the symbolic parameters which were used in the macro definition. The reader can see why this prototype statement is positional since the replacement occurs by matching up corresponding symbols in the prototype and the macro statements. The assumptions made in the example are that NETSAL and TEMP are defined elsewhere in the program as 5 and 6 byte fields respectively. SALGROSS is a 6 digit packed field plus sign and TAXRATE is a four digit packed field plus sign. SALGROSS has two digits to the right of the implied decimal point. TAXRATE has four digits to right of its decimal point. The NI and OI instructions truncate the result of the multiplication to two digits to the right of the decimal point by putting a plus sign in the third digit.

Exercise 7-1 Using the following macro definition

```
             MACRO
&NAME        SUM        &TOTAL, &FLD1, &FLD2
             PACK       WORK1, &FLD1
             PACK       WORK2, &FLD2
&NAME        AP         WORK1, WORK2
             UNPK       &TOTAL, WORK1
             MEND
```

show the expansion of this macro statement:

```
          ADDER      SUM        GROSS,SALARY,COMM
```

It is possible to combine symbolic parameters with other characters to form names in the macro definition and hence, in the macro expansion. To combine any characters with a symbolic parameter the two fields are separated by a period. For example, suppose the symbolic parameter &MATRIX had the value A during a macro expansion the following terms in the definition would have the indicated generated names.

definitions	generated
&MATRIX.(1)	A(1)
MATR.&MATRIX	MATRA
&MATRIX.&MATRIX	AA

This combination of characters and symbolic parameters can be used in macro definitions to obtain more unique symbols in the expansion. For instance,

```
          MACRO
&NAME     NET        &SUF,&KIND,&AREA
&NAME     MVC        NET.&SUF,GROS.&SUF
          MVC        &AREA,NET.&SUF
          MP         &AREA,&KIND.RATE
          NI         &AREA.+3,X'F0'
          OI         &AREA.+3,X'0A'
          SP         NET.&SUF,&AREA.(4)
          MEND
```

The expansion for the macro statement

```
SALES     NET        SAL,DIS,HOLD
```

would be

```
SALES     MVC        NETSAL,GROSSAL
          MVC        HOLD,NETSAL
          MP         HOLD,DISRATE
          NI         HOLD+3,X'F0'
          OI         HOLD+3,X'0A'
          SP         NETSAL,HOLD(4)
```

This technique enables the user of the macro to supply only the unique letters for this expansion which allows the expansion to be readable each time it appears.

A macro definition can contain a previously defined macro statement. This combination is generally called a pair of nested macros or an outer and inner macros where the inner macro is the previously defined one. The inner macro can have another innermacro. The maximum nesting level is three so that the second inner macro could not have another macro imbedded in its definition.

To illustrate this let the macro SUM be defined as it appears in Exercise 7-1. The definition of the macro NET will use the macro SUM in its definition as follows

```
           MACRO
&NAME      NET         &TOTAL,&FLD1,&FLD2,&RATE
           SUM         &TOTAL,&FLD1,&FLD2
           MVC         NETSLS,WORK1
           MVC         TEMP,NETSLS
           MP          TEMP,&RATE
           NI          TEMP+3,X'F0'
           OI          TEMP+3,X'0A'
           SP          NETSLS,TEMP(4)
           MEND
```

The expansion for the macro statement

```
NETSAL     NET         GROSS,SALARY,COMM,FIT
```

would be

```
NETSAL     PACK        WORK1,SALARY
           PACK        WORK2,COMM
           AP          WORK1,WORK2
           UNPK        GROSS,WORK1
           MVC         NETSLS,WORK1
           MVC         TEMP,NETSLS
           MP          TEMP,FIT
           NI          TEMP+3,X'F0'
           OI          TEMP+3,X'0A'
           SP          NETSLS,TEMP(4)
```

The previous macro definitions and expansions were straight forward. Except for the parameter substitution, the same expansion was generated everytime the macro statement was used. However, there are programs which may require a variable expansion depending on the parameters used in the macro statement. This capability can be used to tailor the expansion to the needs of the user. In this way, a general macro definition can be written and many users who required either part or all of the function could use the same macro but supply only the parameters necessary for the particular function needed. In this way the macro expansion can be parameter sensitive. The variable macro expansion technique is used extensively in the systems generation process for operating systems which is discussed in Chapter 11. This is the process of generating an operating system for a particular installation from a generalized package.

In order to define variable expansion macros two types of assembler instructions are necessary. Remember that assembler instructions do not cause machine instructions to be generated but instruct and inform the assembler during assembly. The first of these assembly language statements are the SET statements. These statements enable the programmer to assign a value to a symbol which will be used by the assembler during the assembly. The variables which can have their value assigned by a SET statement are called set variables. There are three types of set variables and within each type there are two classifications, global and local. If a set variable is global, then whatever value was last assigned to it is used anywhere in the assembly when it is encountered. If the set variable is local, then the value it was last assigned is only valid within the macro in which it was defined. The three types of set variables are SETA for set arithmetic, SETB for set binary, and SETC for set character.

The SETA set variable can be assigned an integer number by a SETA language statement

a SETA variable symbol	SETA	an arithmetic expression

The SETA set variable can either be local or global. If it is local the symbol is one of the sixteen possible names of the form &ALn where n = 0-15. If the variable is global the name is of the form &AGn where n ranges from 0 to 15. These variables can be set to an integer value between 0 and 16,777,215 (2^{24}-1) by the arithmetic expression in the operand field.

Some examples:

			Results
&AL5	SETA	12	&AL5 = 12
&AG3	SETA	&AL5+4	&AG3 = 16
&AL7	SETA	&AG3-&AL5+&AL7	&AL7 = 4

(The assumption made above is that the SETA variable keep the values assigned to them previously.) If a variable has not been assigned a value, then a zero value is assumed. A macro definition using SETA variables follows

```
            MACRO
&NAME       NET         &GROSS,&LGTH1,&RATE,&LGTH2
&AL1        SETA        &LGTH1
&AL2        SETA        &LGTH2
&NAME       MVC         NETSAL,&GROSS.(&AL1)
            MVC         TEMP,NESAL.(&AL1)
&AL4        SETA        10-&AL1
            MP          TEMP+.&AL4.(&AL1),&RATA.(&AL2)
AL3         SETA        11&AL1-&AL2
            NI          TEMP+7,X'F0'
            OI          TEMP+7,X'0A'
            SP          NETSAL,TEMP+.&AL3.(&AL1)
            MEND
```

The expansion for the macro statement

```
SALCALC     NET         SALARY,6,FIT,3
```

would be

```
SALCALC     MVC         NETSAL,SALARY(6)
            MVC         TEMP,NETSAL(6)
            MP          TEMP+4(6),FIT(3)
            NI          TEMP+7,X'F0'
            OI          TEMP+7,X'0A'
            SP          NETSAL,TEMP+2(6)
```

Using the SETA variables, and having TEMP and NETSAL defined as 10 bytes in length, this macro can handle parameters of varying length.

The second type of set variable is a SETC variable. SETC variables are always global and may have symbols of the form &CGn where n ranges from 0 to 15. They can be assigned a character string value of up to eight characters by the SETC statement.

SETC variable symbol	SETC	a character string of up to 8 characters enclosed by ' marks

For example,

			Results
&CG3	SETC	'BP101'	&CG3 = BP101
&CG5	SETC	'A.&CG3'	&CG5 = ABP101
&CG15	SETC	'A.&CG5'	&CG15 = AABP101

When a SETC variable is not defined by a character string in its operand field it is considered a null string of zero bytes.

The third type of set variable is the SETB variable which is used to assign the value true or false to a SETB symbol. This variable can be either global or local. The symbol takes the form of &BLn or &BGn where n ranges from 0 to 127. The SETB variable is assigned a value of 0 for false or 1 for true by the SETB statement.

a SETB variable symbol	SETB	a logical or relational expression enclosed in parenthesis

The following logical and relational operators may be used in the operand's logical expression:

AND		
OR		LOGICAL operators
NOT		
EQ	equal	
NE	not equal	
LT	less than	RELATIONAL
GT	greater than	operators
LE	less than or equal	
GE	greater than or equal	

The logical operators have the same meaning here as in Chap. 5. The logical operators may connect SETB symbols. The relational operators may compare either SETA variables or SETC variables. Following are some examples; the comments field gives the value of each symbol.

```
&BG110     SETB    (1)                         &BG110 = 1(TRUE)
&BL3       SETB    (NOT&BG110)                 &BL3=0 (FALSE)
&BL5       SETB    (&BG110 AND NOT &BL3)       &BL5 = 1(TRUE)
&AL2       SETA    15
&AL3       SETA    10
&BG30      SETB    (&AL3 GT &AL2)              &BG30 = 0 (FALSE)
&BL20      SETB    (&BL20)                     &BL20 = 0 (FALSE)
```

When a SETB variable has not been assigned the value of zero (FALSE) is assumed.

With positional macros there are certain conventions which must be observed when a parameter in a macro statement is omitted. If the omitted parameter is to be followed by another parameter the commas which show the missing parameters must be included. If the omitted parameter is not followed by a parameter the remaining commas are not necessary. To illustrate, suppose the following macro prototype was part of a macro definition:

```
&NAME        MOVE        &P1,&P2,&P3,&P4
```

and the macro statement required omission of the parameter corresponding to &P2 and &P3 it would be written

```
HERE         MOVE        A,,,F
```

If the macro statement required omission of the last two parameters then it would be written

```
HERE         MOVE        A,B
```

If a macro definition required a test for null parameter it can do so by the following SET statement, assuming that the missing parameter corresponds to &P2 in the prototype statement:

```
&BL10        SETB        (&P2 EQ '')
```

If the parameter in the macro statement is omitted a null string will be substituted for it. The double quote marks indicate a null string. If the parameter corresponding to &P2 is present, & BL10 will equal zero (FALSE); if it not &BL10 will equal one (TRUE). The set variables alone do not give the variable macro expansion capability; they must be combined with the assembly conditional statements to do this. There are four conditional assembler instructions which can be used within a macro definition. They are provided to alter the sequence in which the assembler's macro generator processes statements in the macro definition. Again, these instructions do not produce object code; they are "executed" by the assembler, only.

The assembler conditional branch has the format

a sequence symbol or blank	AIF	a logical or relational expression followed a sequence symbol

The sequence symbol is a label in the name field of one of the model statements. Its first character must be a period (.) followed by an alphabetic character followed by up to six alphanumeric characters. For example

```
                    .A

                    .B43B

                    .LABEL1
```

are sequence symbols. The AIF statement will cause the assembler to branch forward to the statement labeled with the sequence symbol if the expression is true (1) or take the next statement following the AIF if the expression is false (0). As an illustration, consider the following macro definition

```
            MACRO
&NAME       NET         &GROSS, &RATE1, &RATE2
&NAME       MVC         NETSALS, &GROSS
            MVC         TEMP, NETSALS
            MP          TEMP, &RATE1
            NI          TEMP+3, X'F0'
            OI          TEMP+3, X'A0'
            SP          NETSALS, TEMP(4)
            AIF         (&RATE2 EQ ' ').END
            MVC         TEMP, NETSALS
            MP          TEMP, &RATE2
            NI          TEMP+3, X'F0'
            OI          TEMP+3, X'0A'
.END        MEND
```

If a macro statement was given which omitted the third parameter such as

```
HERE        NET         GRINCOM, TAXRATE
```

The following expansion would result

```
HERE        MVC         NETSALE, GRINCOM
            MVC         TEMP, NETSALS
            MP          TEMP, TAXRATE
            NI          TEMP+3, X'F0'
            OI          TEMP+3, X'0A'
            SP          NETSALS, TEMP(4)
```

At expansion time the assembler would have evaluated the logical expression in the AIF statement as being true and encounter the statement at .END which is MEND. The AIF instruction, in effect, directs the assembler to branch to .END if the third parameter (&RATE2) is missing, that is, blank (' '). If the third parameter was present and named DISRATE, the expansion would be

```
MVC       NETSALS, GRINCOM
MVC       TEMP, NETSALS
MP        TEMP, TAXRAT
NI        TEMP+3, X'F0'
OI        TEMP+3, X'0A'
SP        NETSALS, TEMP(4)
MVC       TEMP, NETSALS
MP        TEMP, DISRATE
NI        TEMP+3, X'F0'
OI        TEMP+3, X'0A'
SP        NETSALS, TEMP(4)
```

There are three other conditional statements, AIFB, AGO and AGOB. The AIFB statement operates analogously to the AIF statement except that the sequence symbol used in the branch appears in the name field of a model statement preceding the AIFB statement. When used with a SETA statement to modify a parameter, AIFB can be used to provide a looping capability so that a given section of code can be repeated a number of times. This could be applied in a character move macro where fields larger than 256 bytes may be encountered.

The AGO and AGOB conditional statements are used by the assembler in a similar manner as the AIF and AIFB statements. These statements, however, give an unconditional branch forward or backward, respectively, to the sequence symbol in the operand field. Their formats are

a sequence symbol or blank	AGO	a sequence symbol
a sequence symbol or blank	AGOB	a sequence symbol

The MEXIT statement can be used in addition to the MEND statement to terminate an expansion. Its format is

sequence symbol or blank	MEXIT	blank

A MEXIT is used when the conditional logic of the expansion will not take the assembler to the MEND statement without an unconditional branch. The MEXIT can be used instead of the branch.

There is a systems symbol, &SYSLIST(n), which can be used to refer to any parameter in a positional macro. The parameter desired is indicated by the integer n which may be a SETA variable. A macro definition which utilizes the condition statements, MEXIT, and &SYSLIST(n) follows

```
         MACRO
&NAME    NET      &INPUT, &LI, &PI, &RESULT, &LR, &UPR, &RATE1,      X
                  &RATE2, &RATE3, &RATE4
&AL1     SETA     &LI
&AL2     SETA     &LR
&AL4     SETA     6
         AIF      (&PI EQ ' ').PACKED
&AL3     SETA     &AL1/2+1
         PACK     &RESULT(&AL3), &INPUT(UAL1)
&AL1     SETA     &AL3
         AGO      .PACK1
.PACKED  MVC      &RESULT, &INPUT(&AL1)
.PACK1   ANOP[1]
&AL3     SETA     &AL1-1
.LOOP    MVC      TEMP+2, &RESULT(&AL3)
&NAME    MP       TEMP, &SYSLIST(&AL4)
         NI       TEMP+&AL3, X'F0'
         OI       TEMP+&AL3, X'0A'
         SP       &RESULT, TEMP(&AL1)
&AL4     SETA     &AL4+1
         AIFB     (&SYSLIST(SAL4)NE'').LOOP
         AIF      (&UPR NE'').UNPACK
         MEXIT
.UNPACK  UNPK     TEMP(&AL2), &RESULT(&AL1)
         MVC      &RESULT, TEMP(&AL2)
         MEND
```

&INPUT	Represents initial input field
&LI	Represents the length of the initial input field in bytes
&PI	If the parameter is present the input field is unpacked. If parameter is omitted, the input field is packed.

[1] ANOP is a conditional no operation statement and allows the sequence symbol .PACK1 to be located such that the succeeding SETA statement will be executed when .PACK1 is referenced. It has a similar effect to a no-operation.

&RESULT	Represents the resultant field
&LR	Represents the length of the resultant field in bytes
&UPR	If parameter is present the resultant field will be unpacked; if it is omitted the resultant field is packed.
&RATE1, &RATE2, &RATE3, &RATE4	Represent up to four parameters for which deductions from the input field will be made. If any are to be omitted they should be left off the end.

Exercise 7-2 Using the preceding macro definition, show the macro expansion for the following macro statements

```
a)  INCOME      NET     GROSALS, 7, P, NINCOM, 7, u,
                        FIT, FICA, INSUR, DISAB

b)  INCOME      NET     GROSALS, 7, P, NINCOM, 7, ,
                        FIT, FICA, , DISAP

c)  INCOME      NET     GROSALS, 5, , NINCOM, 7, u, FIT
```

7.3 Keyword Macros

When macro prototypes have a number of symbolic parameters and several of these are optional it becomes difficult to write the desired parameters in the proper corresponding positions. For macros of this type, where the number of possible parameters is large but are often not all used, the KEYWORD prototype statement may be used to advantage. The features discussed in the previous section on positional macros apply to the keyword macro except that the prototype statements are different.

A keyword prototype statement is written with the symbolic parameters equoted to either a parameter or a null field. As an illustration

```
&NAME       NET     &GROSS=, &RATE1=, &RATE2=,
                    &RESULT=NETSAL, &TEMP=WORK1
```

The symbolic parameters &GROSS, &RATE1, &RATE2, &RESULT, &TEMP are used in the definition of this keyword macro in much the same way as in the definition of a positional macro. However, the keyword macro statement does not have to specify the parameters in a particular order and if a parameter is omitted, the name on the right side of the equals side in the prototype is used in its place. As an illustration, consider the following keyword macro

```
           MACRO
&NAME      NET        &GROSS=,&RATE1=,&RATE2=                X
                      &RESULT=NETSALS,&TEMP=WORK1
           MVC        &RESULT,&GROSS
           MVC        &TEMP,&RESULT
           MP         &TEMP,&RATE1
           NI         &TEMP.+3,X'F0'
           OI         &TEMP.+3,X'0A'
           SP         &RESULT,&TEMP.(4)
           AIF        (&RATE2 EQ '').END
           MVC        &TEMP,&RESULT
           MP         &TEMP,&RATE2
           NI         &TEMP.+3 X'F0'
           OI         &TEMP.+3,X'0A'
           SP         &RESULT,&TEMP.(4)
.END       MEND
```

and the expansion resulting from

```
INCOME     NET        RATE1=FIT, TEMP=HOLD, GROSS=SALES
```

which is

```
           MVC        NETSALS, SALES
           MVC        HOLD, NETSALS
           MP         HOLD, FIT
           NI         HOLD+3, X'F0'
           OI         HOLD+3, X'0A'
           SP         NETSALS, HOLD
```

Note that the position of the parameters relative to each other is not important. Also if a parameter is not supplied (RESULT) the symbol in the prototype is used; if it is supplied it overrides the one given by the prototype.

7.4 Problems

1. Given the macro definition

```
              MACRO
&SYMBOL       TALOKUP       &TABLE,&NO ENTRY,&ENTADDR, &ARG,
                            &FLD,&FLD1LGT,&FLD2LGT,&FLD3, LGT,
                            &FLD3GT
&AL1          SETA          &FLD1LGT
              AIF           (&FLD2LGT EQ '').LOOKUP
&AL2          SETA          &FLD2LGT
              AIF           (&FLD3LGT EQ '').LOOKUP
```

```
&AL3         SETA     &FLD3LGT
             AIF      (&FLD4LGT EQ ''). LOOKUP
&AL4         SETA     &FLD4LGT
.LOOKUP      ANOP
&AG1         SET      &AL1+&AL2, +&AL3+&AL4
             AIF      (&FLD EQ ''). FIRST
&AG4         SETA     &AL1
&AG2         SETA     &FLD
&AG2         SETA     &AG2-1
             AIF      (&AG2 EQ 0).CONTIN
&AG3         SETA     &AL1
&AG2         SETA     &AG2-1
&AG4         SETA     &AL2
             AIF      (&AG2 EQ 0 ). CONTIN
&AG3         SETA     &AL2+&AG3
&AG2         SETA     &AG2-1
&AG4         SETA     &AL3
             AIF      (&AG2 EO 0).  CONTIN
&AG3         SETA     &AG3+&AL3
&AG4         SETA     &AL4
             AGO      .CONTIN
. FIRST      ANOP
&AG3         SETA     0
&AG4         SETA     &AL1
.CONTIN      ANOP
&AL5         SETA     &NO ENTRY
&AL6         SETA     &AL5*AG1-1
             LA       5,&TABLE+&AG3
             LA       6,&AG1 (0,0)
             LA       7,&TABLE+&AL6
L.&SYMBOL    CLC      &ARG,0(&AG4,5)
             BC       8, E.&SYMBOL
             BXL      5,6, L.&SYMBOL
             LA       1, 0(0, 0)
             BC       15,&SYMBOL
E.&SYMBOL    LR       1,5
             LA       5,&AG3(0,0)
             SR       1,5
&SYMBOL      ST       1,&ENTADDR
             MEND
```

and the macro statement

```
THERE        TALOKUP      INVENTY,2000, ENTRY, PARTNO,
                          3, 6, 4, 6, 4
```

What is the value of L.&SYMBOL and &A63? Also, describe the operation of the generated macro.

2. Using the macro definition in problem 1, what are the two macro statements that correspond to the following expansion

```
                LA          5,SALES+0
                LA          6, 15(0,0)
                LA          7,SALES+1499
LORDER          CLC         CUSTNO,0 (8,5)
                BC          8,EORDER
                BXL         5,6,LORDER
                LA          1,0(0,0)
                BC          15,ORDER
EORDER          LR          1,5
                LA          5,0(0,0)
                SR          1,5
                ST          1,ACCTST
```

3. Using the macro definition in problem 1, give the expansion for the following macro statement

```
PAY             TALOKUP     PAYROL,9000,ENTRY,MANNO
                            2,12,6,,10
```

4. Write the macro definition in problem 1 as a keyword macro.

5. When the macro statement in problem 3 has been expanded using the definition in problem 1, what is the value assigned to: &AG2,&AG4,&AL3.

6. Write a keyword macro that performs a table look up on a variable number of tables which have the same format; allow also for the number of entries and the length of their individual fields to be specified. The address of each entry found should be stored in a standard location for each table unless a particular location is specified.

Chapter 8

SUBROUTINES

8.1 Introduction

In Chapter 7, macro instructions were discussed. They provided the programmer with the capability of incorporating a predefined set of instruction in his program where needed by writing a macro statement. This saves programming time and once the macro is tested, the instructions that the macro statement causes to be generated are tested, thereby saving testing time.

However, even though the macro technique can reduce both coding and testing time, the instructions which are generated are in line. This means that each time a macro statement is written, the instructions generated take up CPU storage. If the same macro statement is used three times in a program, the same amount of storage is used three times. When the expansion (number of instructions generated) is small, this is acceptable, but when many instructions are generated per macro statement, it usually is not.

To enable the programmer to gain the advantages of the macro but not pay the penalty of duplicating the storage needed everytime the function is required, the subroutine may be employed. A subroutine is a set of instructions which perform a function when envoked by the user. The subroutine is actually a program which is branched to from another program and when completed branches back to the original program. The subroutine resides in storage once no matter how many times it is used. Therefore, it does not duplicate storage as a macro does.

The decision whether a function should be coded as a macro or a subroutine is dependent upon the performance vs. storage trade-off, the number of instructions in the function, and the variability of the function at the time it is involved. The mechanics involved in making the decision will be clearer after a discussion of the subroutine.

8.2 Linkages

Since a subroutine is essentially a program which is used by other programs there must be a way for both programs to communicate with each other. The program that wants to invoke a subroutine is usually referred to as the 'caller' or 'calling routine'. The subroutine which is being invoked is referred to as the 'called routine'. The execution of a program which invokes a subroutine is pictured below.

```
Start A               Start B
  -
  -
  -
  -
  -
  -
  -
  -
  -
Branch to B
  -
  -
  -
  -
  -
  -
  -
  -
  -
End A                 Return A


Program A           Subroutine B
```

Routine A starts to execute, then it needs the function provided by subroutine B. Routine A, the caller, branches to Routine B, the called routine. Routine B executes and when it is finished, it branches back to the caller, Route A. Therefore, Routine B is a program which is coded such that it can be branched to and will branch back to the caller. As an example

```
START     BALR      2,0
          USING     *,2
            .
            .
            .
          LA        15,SUBR   LOAD GPR 15 WITH
A         BALR      14, 15    ADDRESS OF
            .                 SUBROUTINE
            .
            .
          L         15,A(SUBR)
B         BALR      14,15
            .
            .
            .
          EOJ
SUBR      MVC       NAME,INPUT(15)
            .
            .
            .
          BCR       15,14
          END
```

The instructions which start at the label SUBR and end with the branch on condition 15constitute a subroutine, SUBR. The routine SUBR is called twice from within the same program that contains SUBR. Register 15 is loaded with the entry point (SUBR) of the subroutine and the branch and link register instruction following the BALR and branches to SUBR. Since register 14 has the proper address to which the subroutine should return, an unconditional branch to the address in GPR 14 occurs as the last instruction in the subroutine. In the example above, the first time the subroutine is called by the BALR at A it will return to the address A+2, the second time it is called by the BALR at B it will return to the address B+2.

The LA and BALR instruction which were used to call the subroutine are <u>linkage instructions.</u> They are used solely to set up the entry to and the return from the subroutine.

8.3 <u>Parameters</u>

Subroutines, like macros, require parameters. In the macro definition symbolic parameters are used which are replaced by the actual parameters when the macro is expanded at assembly time. Since a subroutine, unlike the macro, uses the same code to process multiple request at execution time the parameters are passed at that time. Expressed in another way, the macro, using one definition generates a separate set of instructions for each macro statement which reference the parameter supplied in that statement at <u>assembly</u> time. The instructions which make up the subroutine must be capable of referencing parameters which may be supplied with each request for its use at <u>execution</u> time. To illustrate how parameters can be passed between the caller and the called routine, consider the following program

```
BEGIN      BALR     2,0
           USING    *,2
            .
            .
            .
           LA       15,SQUROOT
           LA       14,RET1
           BALR     1,15
           DC       A(X)
           DC       A(XROOT)
RET1       L        5,XROOT
            .
            .
            .
           LA       15,SQUROOT
           LA       14,RET2
           BALR     1,15
           DC       A(Y)
           DC       A(YROOT)
RET2       L        5,YROOT
            .
            .
            .
           END
           USING    *,3
SQUROOT    LR,      3,15
```

```
            L         4,0(0,1)        ADDRESS OF INPUT
              .                       LOADED INTO REG. 4
              .                       DEVELOP SQUARE ROOT
              .
            L         4,4(0,1)        ADDRESS OF RESULT
            MVC       0(0,4)ROOT(4)   LOADED INTO REG 4
            BCR       15,14
   ROOT     DC        CL4
              .
              .
            END
```

The calling program invokes the square root subroutine (SQUROOT) twice. It uses a parameter list to pass two parameters to the subroutine. A parameter list is a string of constants which are the address of the parameters to be passed or the actual parameters themselves. In the above example the par parameter list follows the BALR instruction and contains two address constants. The address of the parameter list is loaded into register one by the BALR instruction. The return register 14 must be set up by an LA instruction since register 1 is set up by the BALR.

The subroutine sets up its addressability by moving its entry point address to register 3 from register 15 which was set up by the calling program. The subroutine can obtain the address of the variable of which it is to take the square root by using register 1 to access the parameter list in the calling routine. In the same manner it can obtain the address of where the result is to be stored. Once the format of the parameter list and what register will contain its address is decided, the subroutine can handle multiple requests as in the example above for the square root of X and Y to be returned in XROOT and YROOT respectively.

Parameters which are passed to a subroutine can be handled in two ways. If the subroutine uses the address of the parameters to reference it in its original location, the subroutine call is a call by name. If the subroutine uses the address of the parameters, to move the data to its own work areas and processes it there and later restores the data to its original location, this is a call by value. Examples of both types of subroutine calls are given below.

Worked Example

8.1 As an example of a call by value, consider a subroutine to calculate $A(X+1)^2+X=Y$ where the original value of X is changed to X+1 after the calculation of Y. The parameter list is in the calling routine, with its address in GPR1. The sequence of the parameter list is:

```
DC        A( S )
DC        A( T )
DC        A( C )
```

```
                USING     *,2
ENTPNT          LR        2.15          PROLOGUE START
                L         3,4(0,1)
                L         4,0(0,3)
                ST        4X
                L         3,8(0,1)
                ST        4,A
                BC        15,START
                CNOP      0,4
A               DS        CL4
X               DS        CL4
Y               DS        CL4           PROLOGUE END
START           LE        2,X
                LE        4,X
                AE        4,=E'1.0'
                STE       4,X
                ME        4,X
                ME        4,A
                AER       2,4
                STE       2,Y
                L         4,0(0,4)      EPILOGUE START
                MVC       0(4),Y(4)
                L         4,4(0,4)
                MVC       0(4),X(4)
                BCR       15,14         EPILOGUE END
                END
```

The parameter list in the calling program specifies the address of three parameters S, T, and C. The subroutine is to calculate S which equals $C(T+1)^2+T$. When the subroutine returns, the value of T will have been changed to T+1. Since this is a call by value the subroutine moves the values of S, T and C to locations in the subroutine Y, S and A using register 1 which has the address of the parameter list. This initialization of the subroutine is called the prologue. All calculations are done with the variables X, A and Y. If they are changed from their original values, the data in the calling program is not changed immediately to reflect it. When the subroutine has finished its calculations, it will then store the current values of those quantities which have changes in their original locations. This was done for S and T. This restoring and housekeeping is called the epilogue.

The subroutine to handle the same parameter list and function with a call by name follows

```
                USING     *,2
ENTPNT          LR        2.15
                L         3,4(0,1)      ADDRESS OF T IN 3
```

```
L       4,8(0,1)    ADDRESS OF C IN 4
LE      2,0(0,3)
LE      4,0(0,3)
AE      4,=E'1.0'
STE     4,0(0,3)
ME      4,0(0,3)
ME      4,0(0,4)
AER     2,4
L       4,0(0,1)
STE     4,0(0,4)
BCR     15,14
END
```

The trade off in a call by name versus a call by value is a trade off in register usage versus core storage usage. Call by name requires a register to address a parameter, whereas a call by value requires duplicate storage for the parameters. In practice, for subroutines with a large number of parameters, a call by value and a call by name require the same storage for data if the parameters are 4 bytes or less in length. Since the routine handles a large number of parameters it would store the address of the parameters in its own work area rather than picking them out of the parameter list everytime or holding each address in a register when handling a call by name.

In most cases a call by value will be used except when a parameter occupies considerable storage and duplicating the storage in the subroutine would be wasteful. When parameters are arrays or character strings they should be passed by name to save space. Parameters in one list may be passed by name, by value or by both.

8.4 Register Usage

If all routines are to be capable of communicating with each other and with the control program in an operating system, standard linkage conventions should be established. Once a standard linkage is established all programmers could use the standard and use of common or general subroutines becomes practical. The programming systems for the 360 have set up a linkage convention which is used by the components of the individual systems. If the programmer abides by this convention his programs will have no trouble in communicating with other programs.

A standard linkage convention consists of defining the function each register will serve and the responsibility of the caller and calling program. The following register conventions are used by 360 programming systems.

Register	Function
15	Entry point register--the address of the routine being branched to is loaded into 15 by the calling program.
	Return code register--the called routine may place a code in 15 before it returns to the caller to indicate the condition of the result.
14	Return register--the address of the instruction to which the subroutine is to branch to upon completion is loaded into 14 by the caller.
0	Parameter register--if the actual parameter(s) are to be passed, the first one is loaded into 0 by the caller.
1	Parameter register--if two actual parameters are passed the second one may be loaded into GPR1 by the caller.
	Parameter list register--if more than two parameters are to be passed, the actual parameters and/or their addresses may be arranged in a list and the address of the list is loaded into 1 by the calling program.

In general the called program has the responsibility to save and restore any registers it uses, usually 2-13. It may have to save 14-1 if it in turn calls another routine before it returns to the caller. In other words, the caller should not expect to find the contents of registers 2 - 13 different after the subroutine than before the call. The called routine is also responsible for maintaining the integrity of the return register 14.

Worked Example

8.2 An example to illustrate the register conventions, register saving and nested calls follows:

```
LEVEL1    BALR   2,0
          USING  *,2
            .
            .
            .
          L      15,=A(LEVEL2)
          LA     14,RETURN1
          BALR   1,15           LEVEL2    USING  *,15
          DC     A(L1P1)                  STM    1,14,SAVE2
          DC     A(L1P2)                  USING  *,2
```

```
         DC     A(L1P3)
RETURN1  L      7, CL4'0000'
         CR     15, 7
         BC     8, CONTIN
ERRRTN   .
         .
         .
CONTIN   .
         .
         .
         END
LEVEL3   USING  *, 15
         STM    2, 13, SAVE3
         USING  *, 5
         LR     5, 15
         DROP   15
         .
         .
         .
         L      15, =CL4'0005'
         LM     2, 13, SAVE3
         BCR    15, 14
SAVE3    DS     CL60
         END
```

```
         LR     2, 15
         DROP   15
         .
         .
         .
         L      15, =A(LEVEL3)
         LA     14, RETURN2
         BALR   1. 15
         DC     A(L2P1)
         DC     A(L2P2)
         DC     A(L2P3)
RETURN2  L      3, =CL4'0005'
                CR 15, 3
         BC     8, ERROR
         L      15, =CL4'0000'
         LM     1, 14, SAVE2
RETURN1  BCR    15, 14
SAVE2    DS     CL60
ERROR    .
         .
         .
         END
```

The example shows that a subroutine when it is called should only plan on using the entry point register (15) for addressing until it can save the contents of the registers it requires. Notice also that if each called routine saves the registers it will use at the beginning and restores them before it returns, each called routine can call another routine, and so on, indefinitely. Such a sequence is termed a set of nested subroutine calls.

Since register 15 is not needed once the branch to the subroutine has taken place and that routine has established its addressing, it can be used to return a code to indicate how the routine ended. These codes can indicate how the routine ended. These codes can indicate normal completion or if an error developed, the kind of error that occurred. The calling program can then test the completion code on return and if necessary take corrective action.

8.5 Types of Subroutines

So far, the discussions of subroutines has centered on communication between two or more routines. Now that the techniques have been laid out, we will consider how and where subroutines can be used profitably. There are five ways subroutines can be incorporated into a program. The subroutines are written in the same way regardless of the manner in which they are finally used with one exception which we will discuss.

The first and simplest way is to assemble the main routine and the subroutines as one program and one control section. A control section will be discussed in the second method of subroutines usage. When the main routine and subroutines are assembled as one program, the subroutine is coded just as another part of the program which is branched to when needed. For example

```
BEGIN      BALR     2,0
           USING    *,2
             .
             .
             .
           STE      6,X
           LA       15,SUBRTN
           BALR     14,15
             .
             .
             .
           STE      4,X
           LA       15,SUBRTN
           BALR     14,15
             .
             .
             .
SUBRTN     LE       2,X
           ME       2,A
           AE       2,B
           ME       2,X
           AE       2,D
           STE      2,Y
           BCR      15,14
           END
```

The subroutine when written and assembled with the main program has no problem in addressing the data directly. There is no need for a parameter list.

The second method a subroutine can be incorporated into a program is to write it as a separate control section in the same assembly. A control section is an independently relocatable section of code. In other words, control sections of a program can be loaded in any order and the program will run properly. In addition, control sections can be deleted or replaced by the Linkage Editor (discussed in chapter 11) without reassembly of the entire program. A control section can be independently relocated because all references to it from outside it and all references from it to outside are accomplished through the use of address constants. Since all address constants are relocated by the loader the position of a control section relative to other control sections is irrelevant. If the CSECT (control section) assembler statement is not used in an assembly, the program constitutes one control section. Each CSECT with a different name defines a new control section. The assembly program below has two control sections.

```
MAINPG   CSECT
BEGIN    BALR     2,0
         USING    *,2
           .
           .
           .
         STE      6,X
         L        15,=A(SUBRTN)
         BALR     14,15
           .
           .
           .
         STE      4,X
         L        15,=A(SUBRTN)
         BALR     14,15
           .
           .
           .
         EOJ
SUBRTN   CSECT
         BALR     4,0
         USING    *,4
         L        3,=A(X)
         L        4,=A(A)
         LE       2,0(0,3)
         ME       2,0(0,4)
         L        4,=A(B)
         AE       2,0(0,4)
         ME       2,0(0,3)
         L        4,=A(D)
         AE       2,0(0,4)
         L        4,=A(Y)
         STE      2,0(0,4)
         BCR      15,14
         END
```

The control section SUBRTN can be independently relocated relative to control section MAINPG since any references made in SUBRTN to symbols defined in MAINPG are by address constants.

The third method of incorporating subroutines into a program is to assemble the main program and the subroutine separately and have them joined together by the Linkage Editor. This method allows subroutines which were written and tested separately to be used in any program. The differences between two control sections assembled at once or individually and combined by Linkage Editor lie in the way the address constants are set up. The example below uses the same routines as previously except that the control sections are assembled separately.

```
MAINPG   CSECT
BEGIN    BALR     2,0
         USING    *,2

         STE      6,X
         L        15, VADRSUB
         BALR     14,15
           .
           .
           .
         STE      4,X
         L        15,VADRSUB
         BALR     14,15
           .
           .
           .
         EOJ
VADRSUB  DC       V(SUBRTN)
         ENTRY    A,B,D,X
         END

SUBRTN   CSECT
         BALR     5,0
         USING    *,5
         EXTERN   A,B,D,X
         L        3,ADDRX
         L        4,ADDRA
         LE       2,0(0,3)
         ME       2,0(0,4)
         L        4,ADDRB
         AE       2,0(0,4)
         ME       2,0(0,3)
         L        4,ADDRD
         AE       2,0(0,4)
         L        4,ADDRY
         STE      4,0(0,4)
         BCR      15,4
ADDRX    DC       A(X)
ADDRA    DC       A(A)
ADDRB    DC       A(B)
ADDRD    DC       A(D)
         END
```

Since the control sections were assembled separately, references to symbols which are not defined in an assembly must be indicated to the assembler. This is done either by a EXTERN statement as in SUBRTN or by a virtual address constant (VADRSUB) as in MAINPG. Also symbols which will be referenced from outside this assembly must also be indicated to the assembler. This is done by the ENTRY statement as in MAINPG. The virtual address constant, EXTERN statement and the ENTRY statement enable the assembler to generate the appropriate information for the linkage editor so it can match up the symbols between the separately compiled control sections.

The last three examples did not pass parameters to the subroutine. If a routine and subroutines were written for a specific purpose and are assembled or link edited together, the parameters may be passed implicitly as was done in the last three examples. This can be done if both the calling and the called programs know the name of the parameters and the assembler or linkage editor match up the symbols. The implicit passing of parameters is not used when a generalized subroutine is written.

The remaining two methods to incorporate subroutines are used where there is not enough storage to hold the entire program in storage at the same time. Both techniques are program segmentation processes. Overlay is used to set up segments of a program in a predefined structure by using the linkage editor. Each segment is composed of one or more control sections. Each control section and therefore, the segment, is written as a subroutine using the methods described previously. For example, let's take 12 control sections A through L and arrange a structure where the program can execute even though there is not enough storage to hold all twelve control sections in storage at one time.

Control section A and C are needed at all times. Therefore A and C will be made a segment or "phase". When B is used, E, H, and L are also required; when D is used, G and I are required; and when F is used, J and K are needed.

We have four segments or phases

Segment 1 consists of A, C

Segment 2 consists of B, E, H, L

Segment 3 consists of D, G, I

Segment 4 consists of F, J, K

By use of the PHASE control card (discussed in Chapter 11) the linkage editor will combine the control sections into the proper segments. The amount of storage necessary to run the program is the total required by Segment 1 plus the largest of the three remaining segments. The execution of the program starts with Segment 1 and proceeds to any segment called by Segment 1. Segments 2 through 4 are loaded when called and overlay any segment (except 1) which was called previously.

The dynamic call involves the calling of a subroutine which has been neither assembled or link edited with the calling program. This call is made during the execution of the calling program. Unlike all the previous methods there is no way that symbols can be matched up between routines in the dynamic call. Therefore, any parameters which are to be passed must be passed in registers or in a parameter list. The dynamic call makes use of the FETCH macro which is described in Chapter 11. When using the dynamic call, the routine is loaded at the time the call takes place. In the overlay example, instead of using the linkage editor to build the segments and the overlay structure, each control section could call any other control section at execution time without them having been assembled or link edited together.

8.6 Summary

The concept of the subroutine was discussed using the macro as a foundation. The key in writing subroutines is understanding the linkage techniques involved. The parameters, which are the data the subroutine uses, may be passed either implicitly or explicitly depending on the use of the subroutine and its relationship to the calling program. A standard convention for register usage makes the use of generalized routines more practical. How subroutines are incorporated into a program is very flexible and depends on the application. Finally, the macro and the subroutine provide similar capabilities. However, when the function required is lengthy and will be used several times the subroutine is more efficient. Remember that subroutine calls require in-line linkage instructions and prologue and epilogue instructions which take space and time. Therefore, the decision on macro or subroutine should consider the number of times the function is used, its length, and the set up linkage time and space for a subroutine. Also, the programmer should keep in mind that a macro can only be incorporated at assembly time whereas a subroutine can be incorporated at assembly time, link editor time and execution time.

8.7 Problems

1. Given the following linkage instructions and parameter list develop a prologue for a subroutine which processes the parameters by name

```
                LA          15,SUBRTN
                LA          14,RETURN1
                BALR        1,15
                DC          A(APARAM)
                DC          A(BPARAM)
                DC          A(CPARAM)
                DC          A(DPARAM)
                DC          A(EPARAM)
      RETURN1
```

2. Given the linkage instructions and parameter list in problem 1, develop a prologue for a subroutine which processes the parameters by value.

3. Develop a prologue and epilogue for a subroutine which processes the parameter list in problem 1 by name and uses registers 5, 6, 2, 4, and 8 for

processing and register 3 for addressing.

4. Develop an epilogue for a subroutine which processes the parameter list in problem 1 by value and changes the values of BPARAM and DPARAM.

5. A main program and a subroutine are to be assembled separately and link edited together. The parameters A, B, VELOC, TIME and DIST are to be passed implicitedly. Show the linkage instructions in the main program, the prologue in the subroutine (named ACCEL) and the use of the EXTERN and ENTRY statements.

6. Write the linkage, parameter list and subroutine which accepts as parameters a customer name which is 12 characters in length with the symbol CUSNAME, a discount rate table with a thousand entries each entry of which is 16 bytes, the first 12 bytes of each entry are the customer name and the last four bytes are the discount rate for this customer in packed decimal. The first entry is called DISCTBLE, a price which is in packed decimal four bytes in length called GRCOST; and a resultant field 4 bytes in length called NETCOST. The subroutine should look the customer name up in the discount table and compute GRCOST -- GRCOST x discount rate = NETCOST. Each parameter should be processed by the subroutine in the most efficient manner.

Chapter 9

SYSTEM/360 INPUT/OUTPUT OPERATION

9.1 Input-Output Devices

In previous chapters most of our attention was focused on the central processing unit (CPU) and the techniques used in making it perform particular functions. When these functions are viewed collectively, the result is that a set of data has been manipulated in some manner to produce a desired result.

In this chapter we will discuss how a program and the data it manipulates are brought into computer storage and how the results the program generates are transmitted from storage so that they may be displayed in a useable form.

An input-output (I/O) device consists of two parts -- the unit which records (writes) data to and retrieves (reads) data from a recording medium and the recording medium itself.

The system/360 is capable of accepting from and transmitting to many different input/output devices. Each unit and its recording medium evolved to suit particular needs. The most familiar recording medium is the punched card which was in wide use when the first computers came into existance. The punched card was the primary recording medium for electronic accounting equipment and therefore, it was natural that it should become one of the primary recording mediums for computers. Today, data destined for a computing system is still initially punched into cards in most instances. For data that is to be interpreted by human eyes the punched card alone was obviously not satisfactory. Printers were developed so the data could be recorded on the printed page.

Sometimes computer generated data does not necessarily have to be displayed for human viewing. It may be data which is to be read back into the computer either immediately or at some time in the future. In this situation, cards are not the most satisfactory recording media. The speed of card punching and reading is very slow relative to computing speed. Cards are bulky, they require manual handling and cannot be reused for different data.

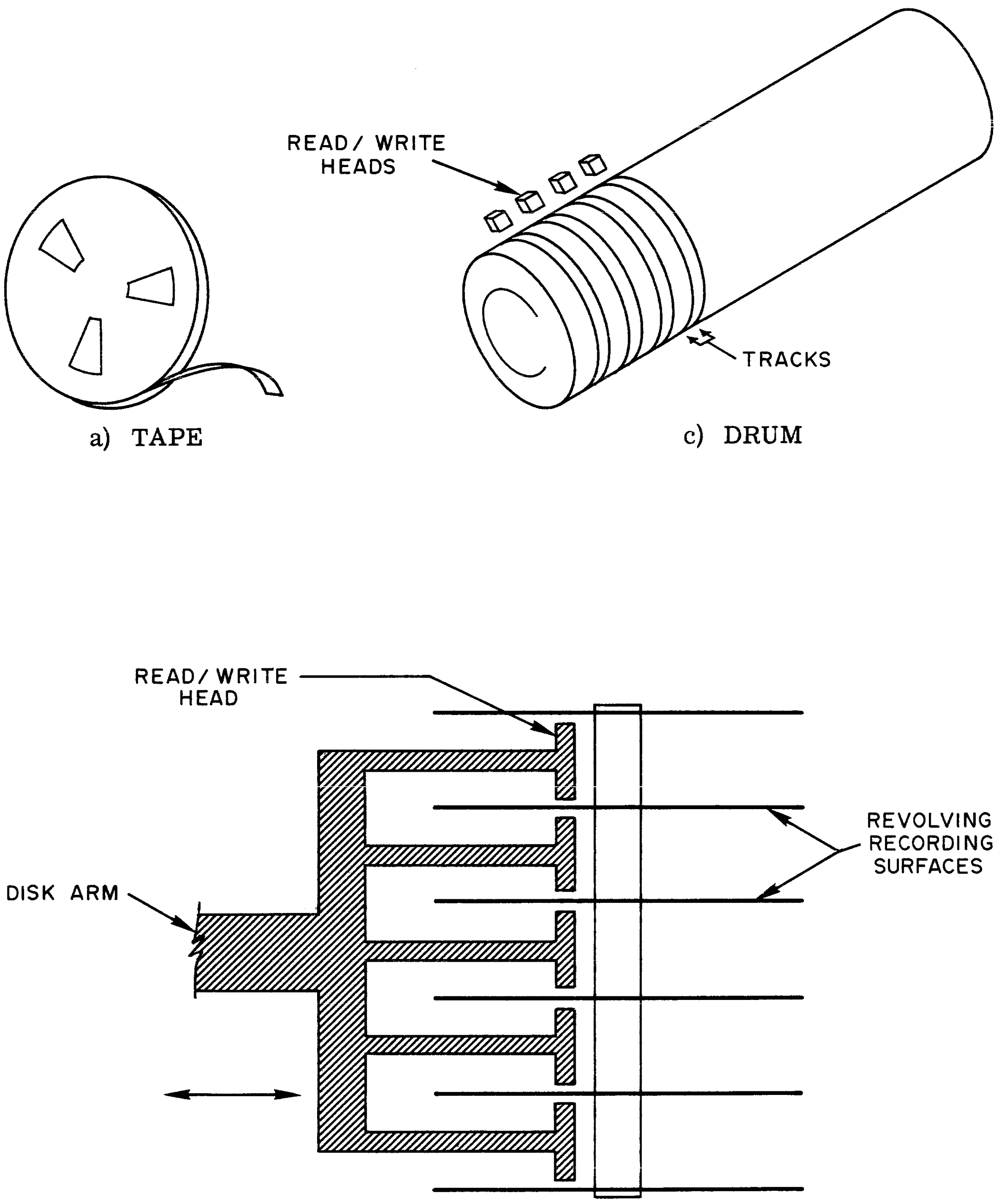
To overcome these problems magnetic tape (Fig. 9-1a) can be used. The tape is faster, the recorded data is more compact, it is reusable, and the same tape can be written and then read during a single execution of a program without manual intervention. Data is written on magnetic tape in a linear sequence, typically at a density of 800 bytes per inch (bpi). The data is arranged in units called records with each record separated from its successor by an inter-record gap of blank tape. A full 2400 ft. reel of tape can hold up to 24,000,000 characters depending on the number of inter-record gaps.

Magnetic tape, however, is a sequential recording media. After a complete file of data is written, the program may only need a particular data item. With tape, all the data recorded before that particular item would have to be skipped over to reach the desired item. Elimination of tape skipping time leads to the use of random access input/output devices, such as disks and drums (Fig. 9-1b, c). These devices enable a program to access a particular item of data directly, without the necessity of skipping over all preceding records.

This capability of being able to access directly a particular piece of data has significantly influenced the design of many application programs and made many new applications feasible. There are several types of direct access devices available with the 360. The two that will be discussed here are the magnetic disk and drum. Magnetic drums are usually faster and have less storage capacity than disks. The drum can be thought of as a cylinder which rotates at high speed. On the surface of the drum there are distinct circular tracks upon which data is recorded. These tracks can be considered as analogous to magnetic strips which encircle the cylinder and lie adjacent to each other. Above each track there is a read/write head which is capable of recording data on that track or reading data from it as the cylinder revolves under the head (Fig. 9-1c).

The disk input/output devices are designed for larger storage capacities than drums. To achieve this the access time - the time it takes to position the read/write to the desired record - is longer than that of drum devices. A disk resembles a stack of phonograph records which are fixed to a rotating spindle with a space between each disk. A comb like mechanism is arranged with one "tooth" of the comb in each space between the disks. At the end of each tooth there are two read/write heads. One services the disk surface immediately above it and the other services the disk surface below it. The comb-like mechanism can move in and out so that the read/write heads on the teeth can be positioned anywhere on the surface of the disks from the outer edge to the middle. The comb-like mechanism is called a disk arm (Fig. 9-1b).

Each disk surface has a set of concentric circular tracks upon which data can be recorded and retrived. Even though the outer tracks are physically longer than the inner ones they all hold the same amount of data. Whenever the disk arm is positioned to read or write data for a particular track on a specific disk surface all other read/write heads are positioned at the same track on the disks' surfaces above and below. Once the disk arm is positioned, the same track on each surface has a read/write head to service it, these vertically aligned tracks, one per disk surface, are called a cylinder. For example, suppose each disk surface had 200 concentric tracks and there are six disks in the stack. Further, the top and bottom disks use only their inside



a) TAPE c) DRUM

b) DISK

Fig. 9-1

recording surfaces. This means there are ten recording surfaces with two hundred tracks per surface or two thousand tracks. The disk arm has ten read/write heads (one per surface) on five arms and can be positioned at any one of the two hundred tracks. When the arm is positioned there are ten tracks that can be accessed without moving the arm. Therefore there are ten tracks per cylinder and two hundred cylinders per disk drive. Now assume that the cylinders are numbered from 000 to 199 and the read/write heads are 0 through 9. In order to access a particular track, say the fiftieth track on the third surface (the bottom side of the second disk from the top), the arm would be positioned at cylinder 049 and head number 2 would be selected. The positioning of the disk arm is called seeking. Once the read/write head is selected for either device there is a rotational delay until the particular price of data on that track is brought under the read/write head. This rotational delay is called latency time and can vary from the time it takes the drum or disk to make one full revolution to zero. On the average, latency time equals half the time for a full revolution. Therefore the access time for a drum is just the latency time and for a disk it is seek time plus latency time. The data storage capacities of a drum are in the range of several million characters while some disk devices can hold up to several hundred million characters.

The quantity and types of I/O devices attached to the computing system depend on the applications which the system will process. Magnetic tape and random access devices, particularly magnetic disks, are in widespread use today. In addition, many specialized I/O devices have been developed to ease man/computer communication. Typewriter-like terminals and graphic display devices fall in this category.

9.2 Input-Output Channels

After the brief discussion of input-output devices in Sec. 9.1, we will now examine the transmission of the data between an I/O device and the computer storage.

In the first computers, the transmission of data between I/O devices and storage was accomplished in a sequential fashion in relation to processing. That is, when a program needed to read or write data, processing would stop and the data transmission (Input-Output) would take place. At the end of the transmission the program would continue operation at the instruction following the one which requested the input-output operation (Fig. 9-2a).

This synchronous type of input-output was sufficient when the processing speed of the CPU for a single transaction was comparable to or greater than the I/O time for that transaction. However, as processing speeds increased, input-output operations could not keep up. More and more of the time to accomplish a task became I/O time. The central processing unit, almost irrespective of its speed, only required a small percentage of total program time. Therefore, increasing the speed of the CPU only reduced total program time by a small fraction (Fig. 9-2b). There were exceptions to this, especially for those scientific and engineering calculations which were characterized by a relatively small amount of input-output operations and a large amount of computing.

To decrease total execution time there are two alternatives - either increase the speed of input-output devices to be comparable to CPU process time, or have the data transmission occur in parallel with processing in the CPU. Although input-output devices have increased in speed considerably, they are still orders of magnitude slower than the CPU. This is because input-output devices are electromechanical whereas central processing units are electronic. The second alternative lead to the development of the data channel or I/O channel.

The I/O channel provides a path for input-output data transmissions between the I/O device and computer storage. This path permits data transmission operations to proceed concurrently with normal processing in the CPU. The maximum amount of concurrency or overlap of the two operations depends on the hardware implementation of the I/O channel.

Also, since a channel is designed as a general purpose unit, and since a wide variety of I/O units may be attached to it, an additional unit is required to interface each device type to a channel: a device control unit. Typical control units include the Tape Control Unit, Disk Control Unit, Drum Control Unit and Transmission Control Unit to permit computer I/O on communications equipment. Figure 9-2 gives a configuration schematic of a typical system.

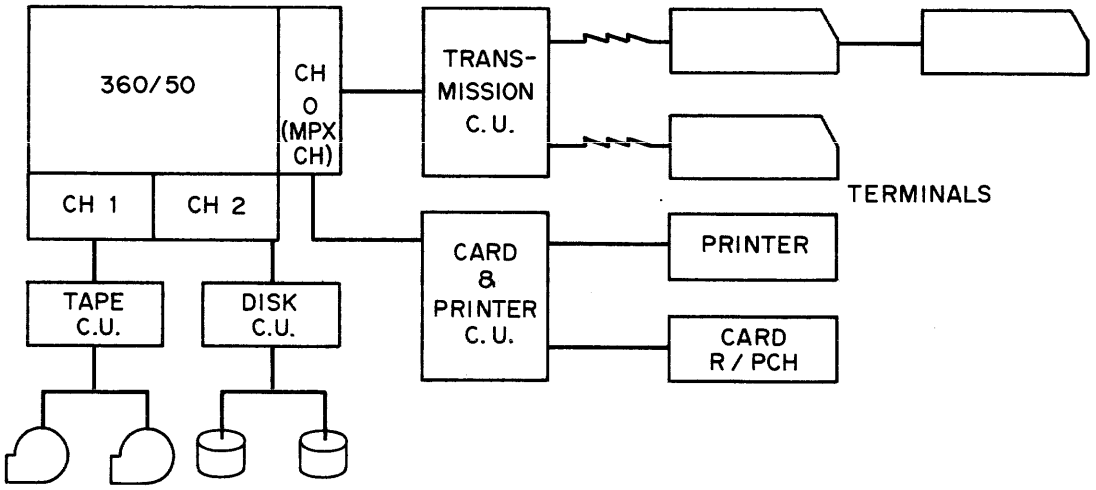


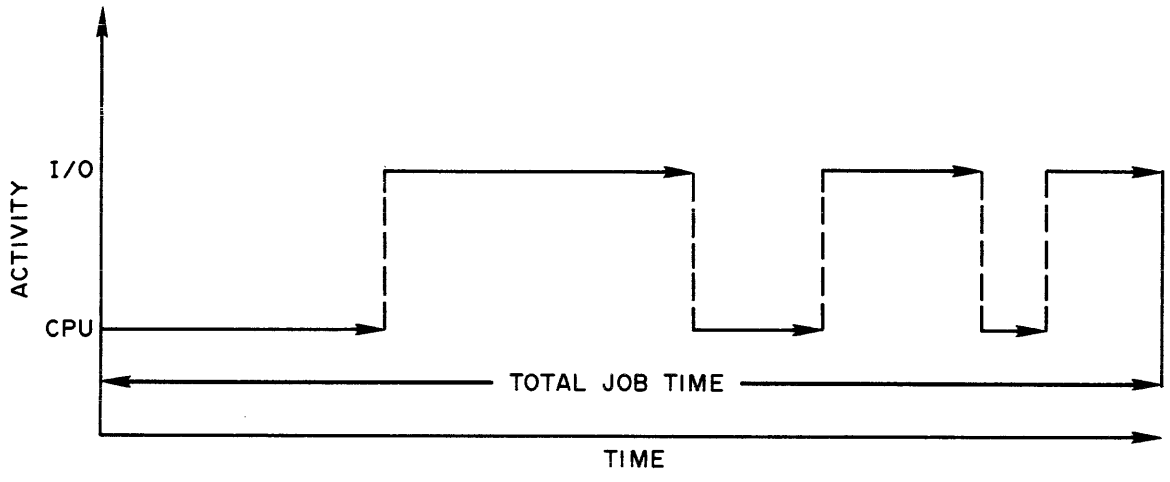
Fig. 9-2 System Configuration

Once the channel, the control unit and the device have been successfully started by the CPU, the channel and the CPU operate concurrently. The channel is able to fetch commands from storage, decode them and then execute them without the aid of the CPU. It is also able to store and access data from storage independently. However, in many systems there is only one path to storage which must be shared by the CPU and the channel. Since once an I/O device is started data must be accepted from it or supplied to it at a constant rate, the channel has priority for the shared path to storage. For example, suppose the channel is receiving data from a tape. The channel receives the data byte by byte from the control unit and stores it in a buffer in the channel. This buffer is just like the CPU core storage only it is used exclusively by the channel. When the channel has received enough bytes to fill this buffer it signals the CPU that it wants to store the data in the buffer into storage. If the CPU was about to access storage it would wait until the channel stores the data. If the CPU didn't require a storage access, it would continue processing while the channel stores data. The time taken by the channel to store its buffer must be smaller than the time required to enter a single byte into the channel buffer by the fastest I/O device connected to the system. This is so because the I/O recording medium is in motion and data is taken from it "on the fly". As a result, the channel buffer must be ready to accept the next data byte when it is presented.

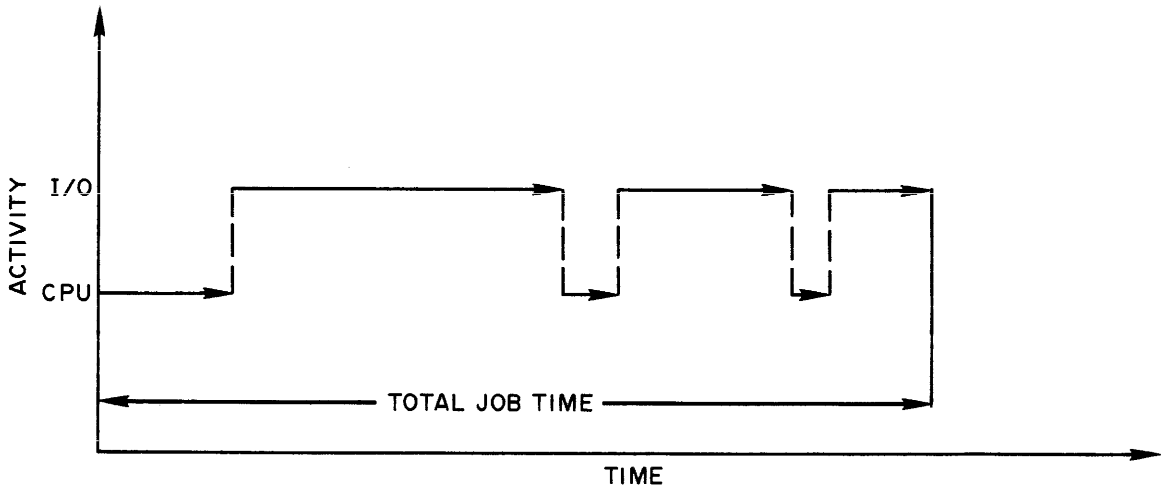
In the situation where an I/O operation may occur in parallel with processing, it takes some planning to make effective use of this capability. The request for data to be read should be given prior to the actual need for the data. In this way the program can continue processing while data transmission from the I/O device to storage takes place for the next transaction. Then when the need for the data arises, it can be accessed from storage rather than waiting for the complete input-output operation to finish (see Fig. 9-3b). However, for output operations no "look ahead" is necessary. The program starts the write operation when desired and continues processing. The I/O operation proceeds in parallel and will terminate at some point thereafter (Fig. 9-3c). For both input and output operations, care must be taken not to start another I/O operation on a channel until the previous one is completed since most channels can handle only one transmission between an I/O device and storage at a time. As programmers and programming systems have become more sophisticated in their input-output programming, the I/O channel's capabilities were increased. Today, one can think of the input-output channel as a small computer with a set of commands which are completely oriented to I/O processing.

9.3 <u>Input-Output Hardware Relationships</u>

The central processing unit controls all operations in the computing system. In the case of concurrent or overlapped input-output operations the CPU initiates the I/O operation. At this point the I/O channel goes into operation. The channel and the CPU are then operating concurrently. When the channel finishes, it signals the CPU. This signal is called an interrupt because it causes the central processing unit to stop its normal processing and take notice of the fact that the channel has ended its operation. If another I/O operation can be started on that channel, the CPU can then start it with a minimum of delay.



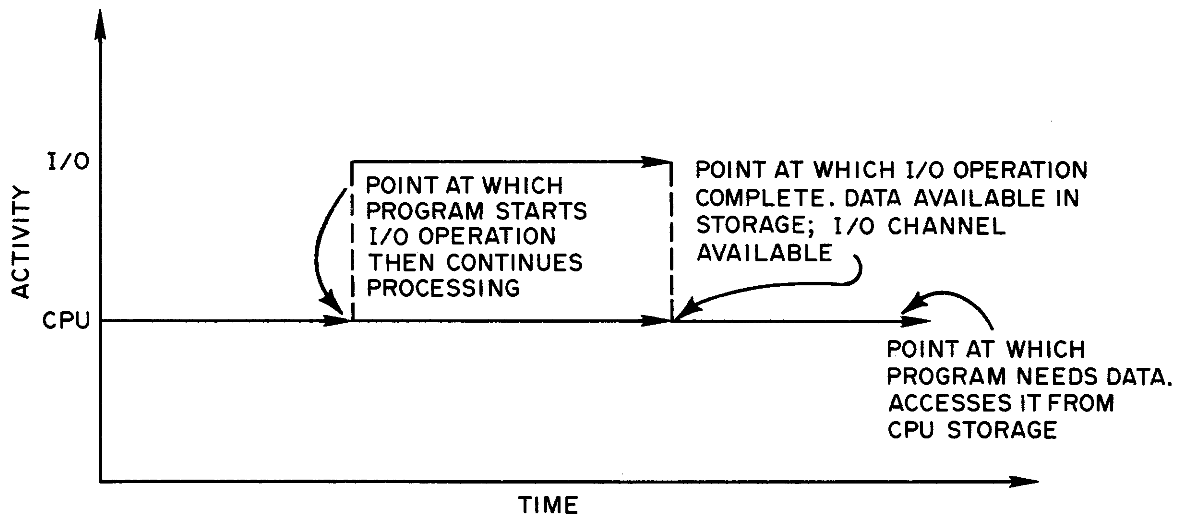
(a) CPU & I/O TIMES RELATIVELY EQUAL



(b) CPU TIME MUCH LESS THAN I/O TIME

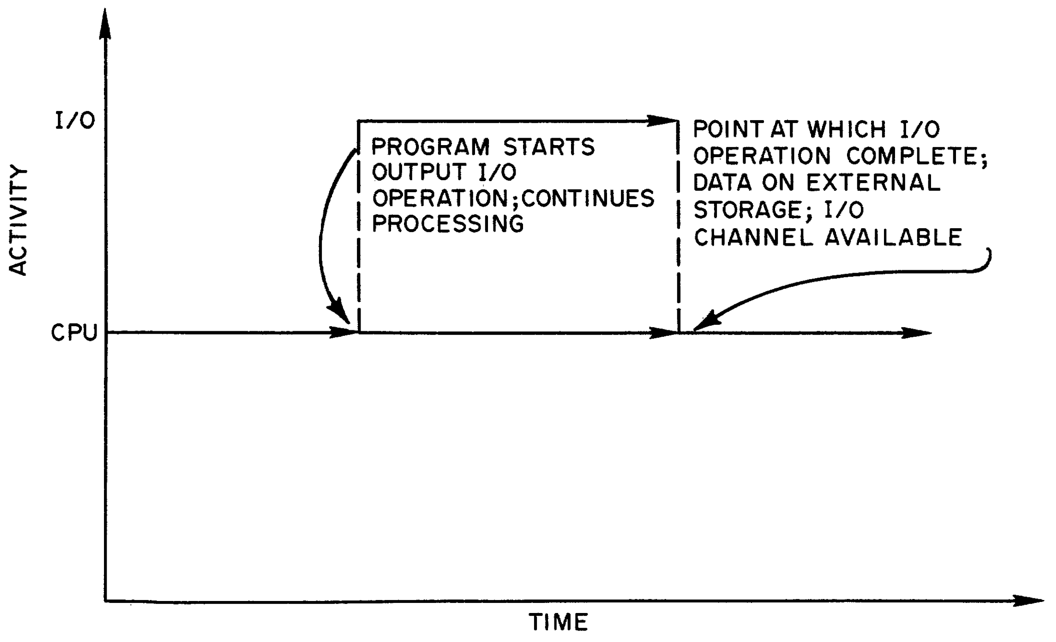
Fig. 9-3

OVERLAP OF INPUT-OUTPUT OPERATION WITH PROCESSING



ACTIVITY
I/O
CPU
POINT AT WHICH PROGRAM STARTS I/O OPERATION THEN CONTINUES PROCESSING
POINT AT WHICH I/O OPERATION COMPLETE. DATA AVAILABLE IN STORAGE; I/O CHANNEL AVAILABLE
POINT AT WHICH PROGRAM NEEDS DATA. ACCESSES IT FROM CPU STORAGE
TIME

(c) INPUT



ACTIVITY
I/O
CPU
PROGRAM STARTS OUTPUT I/O OPERATION;CONTINUES PROCESSING
POINT AT WHICH I/O OPERATION COMPLETE; DATA ON EXTERNAL STORAGE; I/O CHANNEL AVAILABLE
TIME

(d) OUTPUT

Fig. 9-3

Attached to a system/360 channel are up to eight control units. By using different control units various types of devices with different characteristics may be attached to the same S/360 channel. The control units themselves may have a maximum of eight I/O devices of similar operating characteristics attached to them. The path of control for I/O operation is then from the central processing unit to the channel, the channel to the control unit, and the control unit to the device. The data path between the device and storage also includes the control unit and the channel. Figure 9-4 gives a schematic of the process.

9.4 Type of I/O Channels

The system/360 has two types of input/output channels - the multiplexor and selector channels. The reason for two different channels stems from the wide variance in speed of the I/O devices that are available and the need to maximize data handling capacity of each channel.

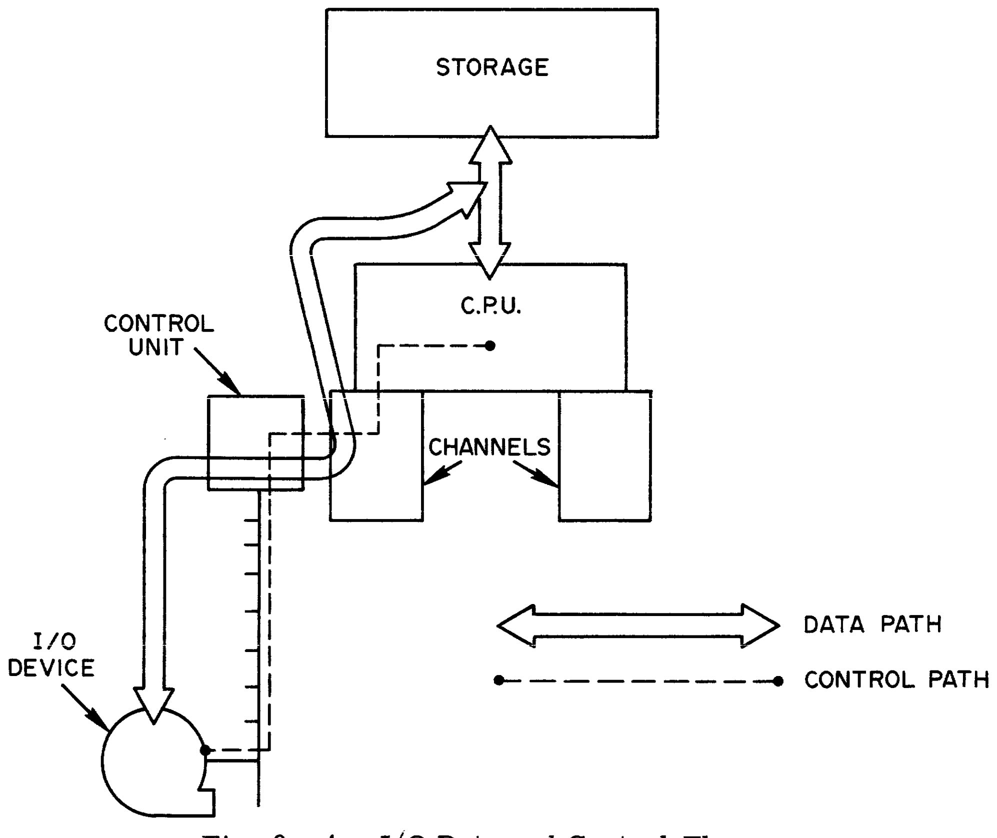


Fig. 9 - 4 I/O Data and Control Flow

The selector channel was designed to handle data transmissions of medium to high speed devices such as tape, disk and drum which have data transmission rates that range from 30 KB (30, 000 bytes/sec.) through 1200 KB respectively. The selector channel is designed to service one I/O device at a time. That is, once a data transmission to or from the device is started, the channel cannot transfer data for another device until the operation is completed. Since the devices serviced by a selector channel have a medium to high data rate, the channel is not tied up for an extended period of time on any single I/O operation. Therefore, its ability to service all the devices attached to it via their control units is not impaired. However, if this type of I/O channel were used to service the low speed devices such as card readers, punches, printers, and telecommunication lines which have data rates in the range of 15 bytes/sec to 5 KB; the elapsed time to service one device would be considerable and only a small fraction of the channels data rate capacity would be utilized.

To cope with this problem the multiplexor channel was developed. Since the capacity of the multiplexor channel is several orders of magnitude greater than the data rate of any of the slow speed I/O devices it was designed to service, it interleaves data transfers from several devices which operate concurrently so that the aggregate data rate approaches the capacity of the channel. Conceptually, the multiplexor channel combines bytes from different sources, transfers them over a common path and then separates them and stores them as though each source had transferred them separately. In fact, the multiplexor interrogates each device to find out if it has indicated that it needs service. If the device has something to send, it transfers one or more bytes across the channel. Then it continues to interrogate the other devices. The multiplexor can interrogate and service all the devices faster than the individual I/O devices can either generate the next byte to send or receive the next byte sent. Therefore, because the time between bytes transferred to or from a low speed I/O device is long compared to the time it takes the multiplexor to make a complete cycle of interrogation and service, and because the data rate of the device is low compared to aggregate rate of the multiplexor channel, this type of channel can service many low speed devices concurrently. The multiplexor channel can also service higher speed devices, but when doing so, it only services that one device. This is called operating in the burst mode.

Although there are two different channels on the 360, the programmer is not concerned with the complexities of each. This is handled by the hardware, so that the programmer is able to handle all devices in a similar manner using the same basic instructions.

9.5 Input-Output Operations

The input-output operations on the 360 are carried out by the I/O channel. However, the channel is started into operation by the central processing unit.

The CPU has as part of its instruction repertoire four instructions of the storage immediate (SI) format for controlling the channels. The channels execute commands in the same manner as the CPU executes instructions.

To get an I/O operation started a program issues a Start I/O instruction (SIO) which has the following format:

```
SIO     device-ADDR     (SI)
```

The operand device-ADDR is the address of the specific I/O device the programmer wishes to start into operation. It takes the form of an eleven bit binary number: CCC DDDD DDDD. Where CCC is the channel address which may range from 000 to 110. 000 is the multiplex channel address and 001 to 110 are the addresses of the one to six selector channels which may be attached to a 360. DDDD DDDD is the address of the device on the channel addressed. This can range from 0 to 255 depending on the number of devices attached to the channel.

When this instruction is executed by the CPU it causes the addressed channel to fetch from location 72 the channel address word (CAW). This word has the following bit structure:

KEY	0000	COMMAND ADDRESS
0 3	4 7	8 31

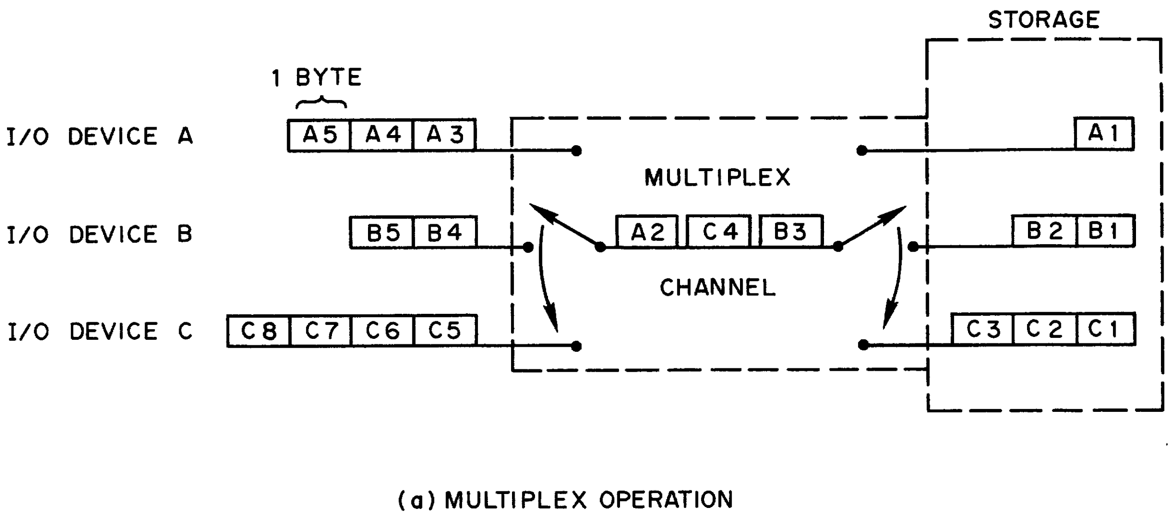
Here, **KEY** is related to the storage protection feature which will be discussed later in this section. The COMMAND ADDRESS is the location of the first I/O command (CCW) the channel is to execute. The CCW(channel command word)has the following format:

COMMAND CODE	DATA ADDRESS	FLAGS	0 0 0		COUNT
0 - 7	8 31	32 – 36	37 39	40 47	48 63

COMMAND CODE is the operation to be performed such as read or write. DATA ADDRESS is the location in storage from which the first byte of data is to be written or into which the first byte of data is to be read. FLATS will be discussed later and COUNT is the number of bytes to be read or written.

A channel command word can be defined in the assembler language by the CCW statement which has the format:

```
CCW     operand1, operand2, operand3, operand4
```



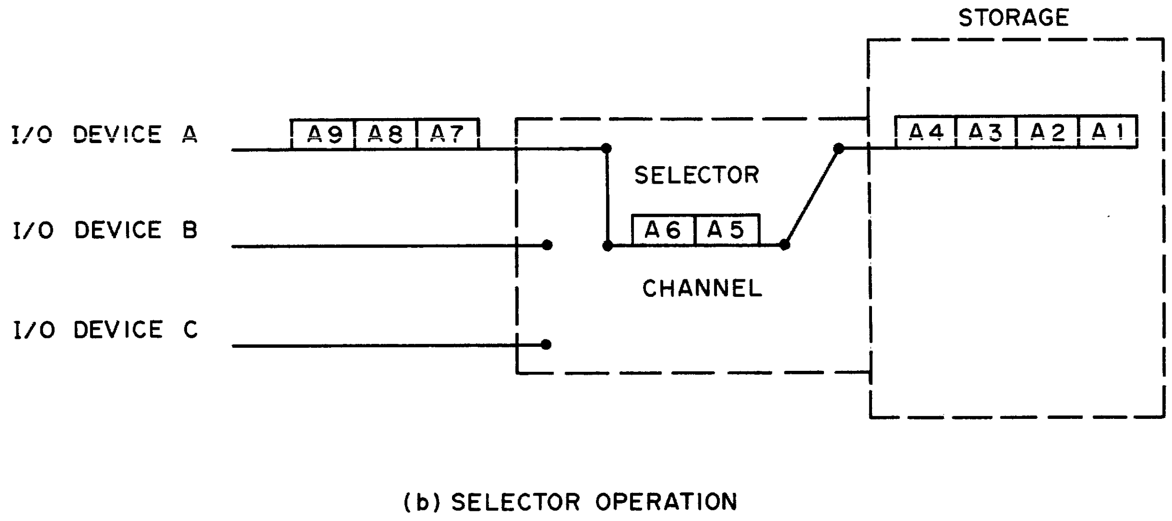


Fig. 9 - 5 S/360 Channel Operation

Here, operand1 is an absolute expression which defines the command code - read has a command code of 2, write a code of 1, additional codes will be introduced as needed; operand2 is a symbol which defines the data area; operand3 is an absolute expression which defines the flag bits and makes bits 37 through 39 zero; operand4 is an absolute expression which defines the count field.

Figure 9 - 6 gives a schematic overview of the steps necessary to start a channel in operation.

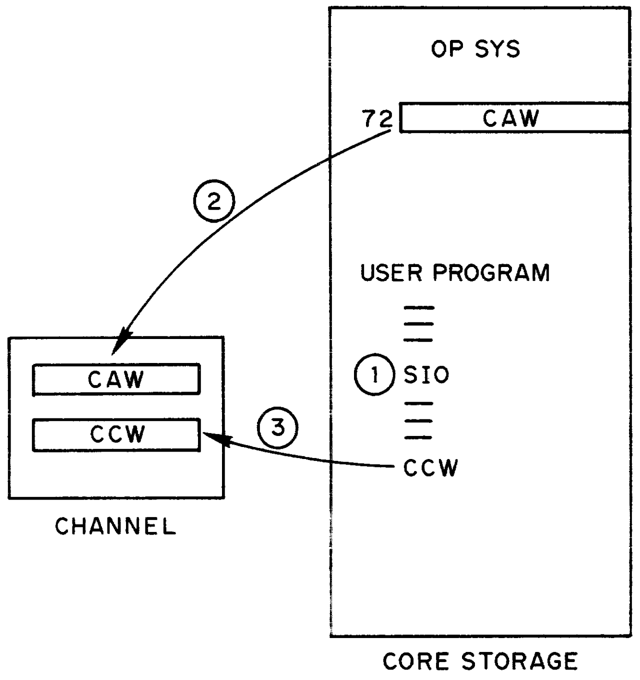


Fig. 9 - 6 Channel Start Up

First, the channel is activated by the start I/O instruction (see 1 in Fig. 9 - 6) given by the CPU. From this point, the CPU may continue to process instructions. Then, the channel fetches the CAW(2) and using the address in the CAW then fetches, in turn, the CCW from storage (3) and carries out the operation specified by it. At this point, the channel begins its I/O operation which goes on concurrently with whatever processing the CPU or other channels are engaged in.

Worked Example

9-1 To summarize the steps required to start an I/O operation, we will take as an example reading a card into storage. The card reader is on the multiplexor channel and has a device address of eight. The data on the 80-column card will require eighty bytes of storage. The program to accomplish this operation follows:

READ A CARD

```
BEGIN       BALR    2,0
            USING   *,2
ALPHA1      L       3,CAW
ALPHA2      ST      3,72
BETA        LA      3,CARDRDR
STARTIO     SIO     0(3)
             .
             .
             .
             .
             .
             .
CAW         DC      X'00',A(IOCOM)
IOCOM       CCW     2,RDAREA,X'00',80
RDAREA      DS      CL80
            DS      OF
CARDRDR     DC      X'00000008'
            END     BEGIN
```

The instructions at ALPHA1 and ALPHA2 load a channel address word (CAW) into location 72. BETA loads the device address of the card reader into register 3. The start I/O with a zero displacement uses the address in register 3. The channel fetches the word (CAW) from location 72; the address in the CAW then causes the CCW to be fetched from IOCOM. This CCW has an operation code of 2 which is a read. The CCW will cause the channel to transfer eighty bytes (80 is in the count field of the CCW) from the card reader to the location, RDAREA, in core storage. At the same time the CPU is executing whatever instructions follow the SIO instruction.

Although the program in Example 9 - 1 would accomplish the transfer of data desired, it is not the normal way a program would accomplish input-output operations on the S/360. In addition, nothing has been said about what happens when the channel completes the operation. In order to describe these processes it is necessary to examine the machine organization of the S/360 in greater detail.

The S/360 was designed to run under the control of a supervisory program. For this reason the system differentiates between two classes of programs. A program may execute in either of two states; the problem program state or the supervisor state. Most programs which a user will write will operate in the problem program state. However, problem programs will make use of programs in the supervisor state to perform certain services for them. This is because certain CPU instructions are privileged instructions and can only be executed by a program in the supervisor state. This was done to protect the supervisory program or monitor, which controls the system operation, from destruction by erroneous problem programs.

The S/360 also has an interrupt system incorporated into the system organization. An interrupt, as the name implies, causes whatever processing is in progress when the interrupt occurs to be stopped, the status of the system to be saved and another program, usually in the supervisory state, to be given control. There are 5 classes of interrupts that can occur. They are:

Program Check	(PC)
Machine Check	(MC)
External/Timer	(E/T)
Input/Output	(I/O)
Supervisor Call	(SVC)

The purpose of these interrupts is to bring to the immediate attention of the supervisory program the fact that one or more of these five conditions has occurred. A description of each interrupt type follows.

Program Check - A program has attempted to execute an invalid or privileged instruction, or has encounted a condition such as floating point overflow or divide by zero, or attempts to execute an instruction with an invalid operation code or invalid operand address.

Machine Check - A hardware malfunction has been detected.

External/Timer - The interval timer has expired or a device attached to the external interrupt needs service or the interruption key on the system control panel has been depressed.

I/O - An I/O operation which was previously initiated has ended. This interrupt signals the supervisor program that an I/O operation which had been proceeding concurrently with CPU processing has terminated. If another I/O operation requires the channel or device from which the interrupt was received the supervisor may then start the new I/O operation.

SVC - This interrupt is caused by a program executing an SVC instruction. This is the means that a program uses to request a service of a supervisory program such as an I/O operation.

The heart of the S/360 interrupt system is the Program Status Word (PSW). The format of this 8 byte double word is

SYSTEM MASK	KEY	AMWP	INTERCEPT CODE	ILC	CC	PROGRAM MASK	INSTRUCTION ADDRESS
0 7	8 11	12 15	16 31	32 33	34 35	36 39	40 63

System Mask - A 0 bit in bit positions 0-7 is used to mask out (prevent interrupts from happening) the interrupts from the multiplexor channel, each of the possible six selector channels and the External/Timer interrupt, respectively.

Key - The storage protect key that is assigned to a particular program.

A - Determines whether ASCII or extended binary code decimal (EBCDIC) character codes are used.

M - A 0 used to mask out the machine check interrupt.

W - Bit indicates if the CPU is idle(Wait state) or running.

P - Bit indicates if the program which is executing is in the problem state or the supervisor state.

Interruption Code - Indicates what caused the interrupt within one of the five categories of interrupts.

ILC - Indicates instruction length of the last instruction interpreted before an interrupt occurred.

CC - is the condition code setting before the interrupt occurred.

Program Mask - This field allows a program to mask out the following program interrupts:

Fixed-point overflow
Decimal overflow
Exponent underflow
Significance

Instruction Address - This is the location of the next instruction to be executed within the interrupted program.

When any program is executing it is under control of a PSW. When an interrupt occurs the program's PSW is stored and a new PSW is loaded which gives control to the new program. By saving the stored or old PSW all the information necessary to save the status of the interrupted program and continue it later is available.

The 360 has 10 double-word locations where PSW's are kept. Five of these are called old PSW locations and the other five are called new PSW locations. For each of the five classes of interrupt there is a new and old PSW location. When an interrupt occurs the PSW of the interrupted program is stored in the old location for that interrupt class and the PSW from the new location is used to start execution of the program (usually supervisory) which handles that class of interrupt. By this means, the supervisory program gains control and has the necessary information to let the interrupted program continue once the condition which caused the interrupt has been handled (Fig. 9 - 7). The supervisory program would move the old PSW from the old location and save it. When it is finished it will execute a load PSW instruction which loads the PSW it saved and thereby returns control to the problem program.

The 360 also has an optional memory protection feature as part of the hardware structure. This allows programs to be protected from each other. A program can only write in a storage location which has the same key (4 bits) that is in the key field of its PSW. Even I/O operations cannot transfer data into a storage location that has a key different from the key in the CAW. The storage key is assigned to storage in units of 2048 bytes.

Worked Example

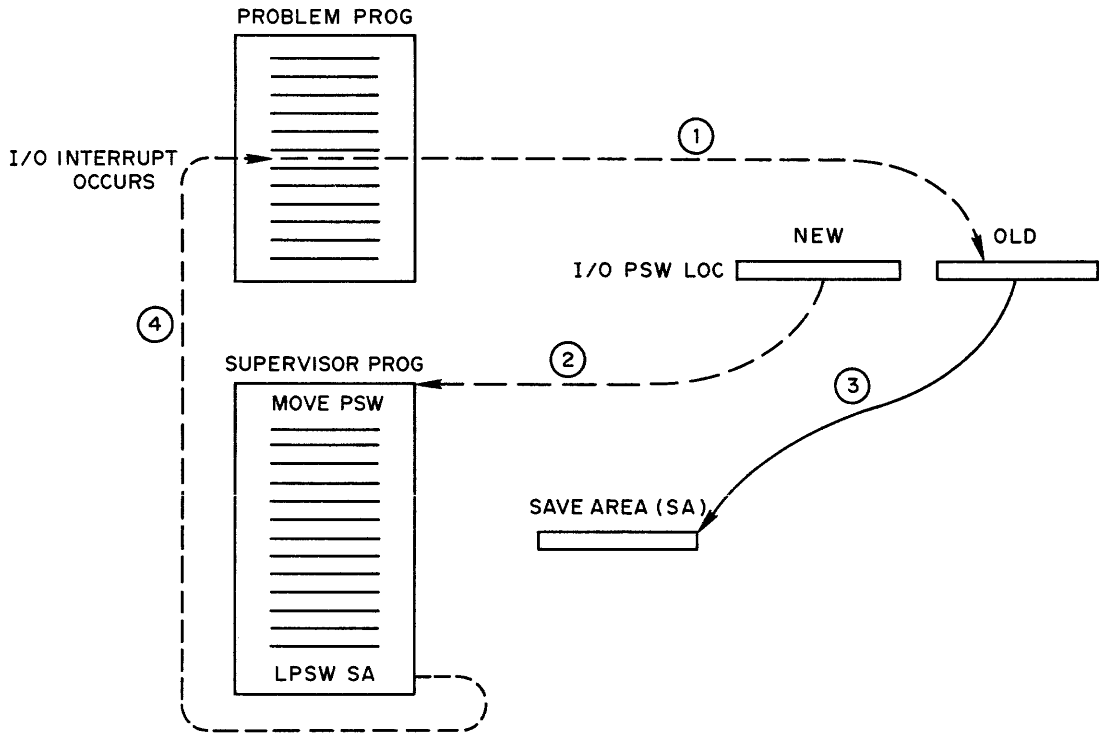
9-2 With this perspective on S/360 machine organization behind us, let's return to the problem of reading a data card into storage. Since the program which needs the data is usually in the problem program state, and cannot execute the start I/O (SIO) instruction itself because SIO is a priviledged instruction, it will have to request the supervisor program to do so. This is done by execution of a supervisor call instruction.

```
SVC     'code'
```

The operand, 'code', is a hexidecimal digit from 0 to 255 which is stored in the interrupt code of the old PSW which in turn is stored because the SVC instruction causes an interrupt.

Using the subroutine linkages discussed in Chapter 8 and the SVC instruction let's proceed with the problem.

```
BEGIN     BALR    2,0
          USING   *,2
            .
            .
            .
CAW       DC      X'00',A(IOCOM)
IOCOM     CCW     2,RDAREA,X'00',80
RDAREA    DS      CL80
          DS      OF
CARDRD    DC      X'00000008'
          END     BEGIN
```



(1) I/O INTERRUPT OCCURS - STORE OLD PSW

(2) HARDWARE GIVES CONTROL TO PROGRAM DEFINED BY NEW PSW

(3) SUPERVISORY PROGRAM SAVES OLD PSW

(4) SUPERVISORY PROGRAM LOADS OLD PSW FROM SAVE AREA WHICH RETURNS CONTROL TO PROBLEM PROGRAM

Fig. 9 - 7

In the code above, the channel address word (CAW) and the channel command word (CCW) are set up at CAW and IOCOM respectively also the address of the card reader and the storage to accept the data have been set up at CARDRDR and RDAREA respectively. What remains is to communicate to the supervisory program the channel address word, the I/O device (card reader) address, and the SVC code so it can determine what service is being requested. For a request to start I/O, assume a code of zero. This means that the supervisor code is constructed to recognize that a zero code means that the problem program is requesting the supervisor to start an I/O operation. The supervisor parameter registers are 0 and 1 and we will assume a standard such that the channel address word goes in 0 and the device address in 1. The problem program might be

```
BEGIN      BALR     2,0
           USING    *,2
             .
             .
             .
           L        0,CAW
           L        1,CARDRDR
           SVC      X'00'
             .
             .
             .
CAW        DC       X'00',A(IOCOM)
IOCOM      CCW      2,RDAREA,X'00',80
RDAREA     DS       CL80
           DS       OF
CARDRDR    DC       X'00000008'
           END      BEGIN
```

The SVC instruction will cause an interrupt and the supervisor program will gain control. The location of its first instruction is in the instruction address portion of the new PSW for an SVC interrupt. The program will first have to save the old PSW and then interrogate the code in the interruption code portion of the old PSW to find the service requested and branch to the appropriate routine to do the start I/O (SIO). The supervisory program can be written.

```
BEGIN      BALR     3,0
           USING    *,3
           MVC      SAVEPSW,32
           CL1      SAVEPSW+3,X'00'
           BC       X'00',SIORTN
             .
             .
             .
SIORTN     ST       0,72(0)
```

```
                    SIO         0(1)
                     .
                     .
                    LPSW        SAVEPSW
          SAVEPSW   DS          CL8
                    END         BEGIN
```

The supervisory program has started the input-output operation requested and returns control back to the problem program at the instruction following the supervisor call (SVC). At this point the I/O operation is proceeding concurrently with the execution of the problem program.

At some time later the input-output operation will complete and will cause an I/O interrupt. This will interrupt the problem program processing and control will be given to the supervisory program whose initial location is specified in the instruction address portion of the new I/O interrupt PSW. This routine will record the fact that the I/O operation is complete and return control to the problem program. This is

```
          BEGIN     BALR        3, 0
                    USING       *, 3
                    MVC         SVIOPSW, 56
                    MVC         IOCMPLT, SVIOPSW+2
                    LPSW        SVIOPSW LOAD PSW
          IOCMPLT   DS          CL2
          SVIOPSW   DS          CL8
                    END
```

This program saves the address of the I/O device which gave the interrupt in location IOCMPLT and then returns control back to the problem program via the load PSW instruction (LPSW). Since the problem program and the transmission of the data from the card were operating asynchronously, the problem program must check to see if the I/O operation is complete before it attempts to use the data. It does this by using another SVC instruction with a different code. The change of control to the supervisory program occurs as before by switching the PSW. The code in Fig. 9 - 8 is a composite of the entire operation of requesting and completing the input-output operation.

Let us review the complete operation. The problem program needs to read a card at some point. It loads register 0 with the channel address word and register 1 with the address of the card reader. Then it executes a supervisor call instruction with a code of, say, zero. This causes an interrupt and control is passed to the instruction at location BEGINS as indicated by arrow 1 in Fig. 9 - 8. The supervisor program saves the old PSW, checks the SVC code and branches to SIORTN. Here the start I/O instruction is issued and control is returned to the problem program as shown by arrow 1'. The program continues to execute until an I/O interrupt occurs. This causes

PROBLEM PROGRAM

```
          BEGIN     BALR      2, 0
                    USING     *, 3
                      .
                      .
                      .
                    L         0, CAW
                    L         1, CARDRDR
    *     IOREQ     SVC       X'00'   ①
                                      ①'
                      .             * ②
                      .               ②'
                      .
                    L         1, CARDRDR
    *     IOCHK     SVC       X'01'      ③
                      .                  ③'
                      .
                      .
          CAW       DC        X'00'A(IOCOM)
          IOCOM     CCW       2, RDAREA, X'00', 80
          RDAREA    DS        CL80
                    DS        OF
          CARDRD    DC        X'00000008'
                    END       BEGIN
```

SUPERVISORY PROGRAM

```
③①  BEGINS      BALR      3, 0
                 USING     *, 3
                 MVC       SAVEPSW, 32
                 CL1       SAVEPSW+3, X'00'
                 BC        8, SIORTN
                 CL1       SAVEPSW+3, X'01'
                 BC        8, CKIORTN
                   .
                   .
                   .
     SIORTN      ST        0, 72(0)
                 SIO       0(1)
                   .
                   .
                   .
③' ①' RETURN     LPSW      SAVEPSW

     CKIORTN     CL        1, IOCMPLT
                 BC        8, RETURN
                 MVC       SWITCH, SWTCONT
                 LPSW      WAIT
   ② IOINRPT     MVC       SUIOPSW, 56
                 MVC       IOCMPLT, SVIOPSW+2
                 CL1       SWITCH, X'FF'
                 BC        X'00', CKJORTN
   ②'            LPSW      SVIOPSW
     IOCMPLT     DS        CL2
     SVIOPSW     DS        CL8
     SAVEPSW     DS        CL8
     SWTCONT     DC        X'FF'
     WAIT        DC        X'0002000000000000'
     SWITCH      DS        CL1
                 END       BEGINS
```

Fig. 9 - 8 Input-Output Example

control to go to IOINRPT (arrow 2). This routine saves the address of the device which caused the interrupt at IOCMPLT and returns control to the problem program (arrow 2'). When the problem program needs the data that it requested with the SVC instruction at IOREQ, in order to make sure the data is at the location RDAREA, it requests the supervisor program to check if the operation is complete. The problem program loads into register 1 the address of the card reader and executes and SVC instruction with a code of one. Control passes to the supervisory program as indicated by arrow 3. It interrogates the code and branches to CKIORIN. Here it checks the device address stored by the previous I/O interrupt against that which the problem program loaded into register one. If it is equal, then the I/O operation is complete and control is returned to the problem program and the data is at RDAREA. However, if its not equal then the I/O operation is not complete and it loads a special PSW which puts the CPU into the wait state (idle) until the I/O interrupt does occur. Therefore, the problem program will continue processing if the I/O operation is complete when it makes the request at IOCHK, otherwise it waits at that point until it is complete.

The code in Fig. 9 - 8 is for illustrative purposes only. It is extremely elementary and does not take into account the fact that it may have to handle many interrupts and may not be able to service each request completely before handling the next one. The problem of both the supervisory and problem programs using the same registers is not considered. That is, the supervisory program would have to save and restore any registers it may use so that the problem programmer need not worry about the contents of the registers changing without positive action on his part. There are many more factors which also must be given consideration. However, for the remainder of the chapter let us assume that there is a program which operates in the supervisor state which will perform an I/O operation if we execute an SVC with a code of zero and give it the channel address word (CAW) and the device address. Also, we will assume that if an SVC with a code of one is executed, the supervisor will check for the completion of an I/O operation. If the operation is not complete, the supervisor will enter the wait state and remain there until an interruption occurs.

Worked Example

9-3 Consider a program which reads fifteen cards, processes the data and then writes the results on tape in card image form, that is, in records which are 80 bytes in length. Our object will be to overlap the input-output operations with the processing of the data. The program follows:

```
BEGIN     BALR    2,0
          USING   *,2
          L       0,CAWIN
          L       1,CARDER
READ1     SVC     X'00'        REQUEST FIRST CARD TO BE
                                READ USING CCW AT IOCOM
```

```
                L       3,=14
                L       5,=13                      INITIALIZE PROGRAM
                L       4,=0
                L       1,CARDRDR
IOCHKIN         SVC     X'01'                      CHECK READ OPERATION TO
                                                   MAKE SURE IT'S COMPLETE
MOVEIN          MVC     WORKARA,RDAREA             MOVE DATA FROM INPUT
*                                                  AREA TO WORK AREA
                L       0,CAWIN
RDNEXT          L       1,CARDRDR
IOCHKOU         SVC     X'00'                      READ NEXT CARD INTO
                 .                                 RDAREA PROCESS DATA
                 .                                 IN WORKARA AND LEAVE
                 .                                 RESULTS IN TEMP
                BXH     3,4,MOVEOU                 IF FIRST TIME, SKIP
                                                   CHECK ON WRITE
                L       1,TAPE
                SVC     X'01'                      CHECK PREVIOUS WRITE
MOVEOU          MVC     WRITARA,TEMP               MOVE DATA TO WRITARA
                L       0,CAWOUT
                L       1,TAPE
WRITE           SVC     X'00'                      REQUEST TO WRITE DATA
                                                   IN WRITARA TO TAPE
                BCT     3,IOCHK                    IF NOT FINISHED BRANCH
                                                   TO IOCHK TO PROCESS
CAWIN           DC      X'00',A(IOCOMIN)           NEXT CARD
CAWOUT          DC      X'00',A(IOCOMOU)
                DS      OF
CARDRDR         DC      X'00000008'                CARD READER ADDRESS
                                                   ON MULTIPLEX CHANNEL
TAPE            DC      X'00000108'                TAPE UNIT ADDRESS ON
*                                                  SELECTOR CHANNEL 1
WORKARA         DS      CL80                       80 BYTE STORAGE AREA
TEMP            DS      CL80                       80 BYTE STORAGE AREA
RDAREA          DS      CL80                       80 BYTE INPUT STORAGE AREA
WRITARA         DS      CL80                       80 BYTE OUTPUT STORAGE AREA
IOCOMIN         DC      2,RDAREA,X'00',80          CCW TO READ CARD
IOCOMOU         DC      1,WRITARA,X'00',80         CCW TO WRITE TAPE
                END     BEGIN
```

The program reads the first card at READ1 and while that operation is in progress it initializes registers 3, 4, and 5 for counting and a later test. Then it checks, at IOCHKIN, to determine if the first card has been read. If it has, it proceeds; if it hasn't, the program waits until input is completed and then proceeds. At MOVEIN, the data just read is moved to WORKARA. The program then reads the next card at RDNEXT. While this operation is in progress the program processes the data in WORKARA and leaves the results in TEMP.

The first time through the code, the branch to MOVEOU will be taken. At MOVEOU the data is moved from TEMP to WRITARA and is written on tape at WRITE. Then the count is checked to see if 15 cards have been read. If not, the branch to IOCHKIN is executed and the previous read operation is checked. The data is moved from RDAREA and the next card read. Anytime after the first time through the program, the branch to MOVEOU will not be taken and the previous write will be checked to make sure that the data in WRITARA has been written before the new data is moved in from TEMP.

As mentioned previously, the channel executes I/O commands as the central processing unit executes instructions. On the S/360, several CCW's can be connected together in three ways. The first two methods are called chaining and are accomplished by having a one bit set in the appropriate field of the flags portion of the CCW (positions 32-36). Bit positions 32 and 33 are used to indicate data chaining and command chaining, respectively. Bit position 34 is used to suppress the error indication that occurs when the data actually transmitted does not agree with the count that was given in the CCW. Bit position 35 is used to suppress transmission of data on an input operation. The operation is carried out at the device as a read operation but the data is not transmitted by the channel. This is called the skip flag because it enables data to be skipped over. Bit position 36 is used to cause an interrupt when the operation specified by this CCW is completed. It is used when several CCW will be executed for one SIO instruction. Normally the channel doesn't interrupt the CPU until it has finished the entire sequence of operations. The Program Controlled Interrupt flag (CCW bit 36) allows the channel to interrupt the CPU before the channel has completed the entire operation. It can be used to keep the CPU informed as the channel completes each CCW in a sequence.

Each time the channel completes an operation specified in a CCW, it checks the flag settings in that CCW to determine what to do next. Therefore, by use of the flags, the action of the channel at the completion of a CCW can be varied. The use of two of these flags, 32 and 33, will be discussed next.

A 1 bit in CCW position 32 indicates data chaining. That is, when the number of bytes in the count field of the present CCW is processed, the I/O operation will not be terminated; the next CCW in storage will be used to continue it. The type of operation,READ, WRITE,etc., remains the same. Data chaining can be used to read data into non-contiguous storage locations even though the bytes are contiguous on the external storage medium. The reverse is true when data chaining is used for writing. Data from several non-contiguous storage locations can be written into one contiguous stream of bytes on external storage.

Worked Example

9-4 For example, if the 80 byte data record from the card reader is to be stored with the first 30 bytes at location NAME, the next 20 bytes at location

DEPT, and the last 30 bytes at location TITLE. A program to do this can be written as follows:

```
BEGIN       BALR      2,0
            USING     *,2
            L         0,CAWIN
            L         1,CARDRDR
            SVC       X'00'
             .
             .
             .
            L         1,CARDRDR
            SVC       X'00'
            MVC       NAME,RDAREA(30)
            MVC       DEPT,RDAREA+30(20)
            MVC       TITLE,RDAREA+50(30)
TITLE       DS        CL80
CAWIN       DC        X'00',A(IOCOMIN)
            DS        OF
CARDRDR     DC        X'00000008'
DEPT        DS        CL80
IOCOMIN     DC        2,RDAREA,X'00',80
NAME        DS        CL80
            END       BEGIN
```

As an alternative,

```
BEGIN       BALR      2,0
            USING     *,2
            L         0,CAWIN
            L         1,CARDRDR
            SVC       X'00'
             .
             .
             .
            L         1,CARDRDR
            SVC       X'01'
TITLE       DS        CL80
CAWIN       DC        X'00',A(IOCOMIN)
            DS        OF
CARDRDR     DC        X'00000008'
DEPT        DS        CL80
IOCOMIN     CCW       2,NAME,X'80',30
            CCW       ,DEPT,X'80',20
            CCW       ,TITLE,X'00',30
NAME        DS        CL30
            END       BEGIN
```

The first program reads the data into 80 contiguous bytes and then uses CPU instructions to move it to the three separate locations. The second program reads data directly into the three separate locations by data chaining the three CCW's (bit 32 on). The process could be reversed on a write by changing the operation code in the first CCW from a 2 to a 1.

Command chaining allows several CCW's to be executed by the channel before terminating the I/O operation. However, unlike data chaining the operation code in the second CCW is interpreted and that operation is performed next. While in data chaining the chained CCW's applied to one record (a set of contiguous bytes on external storage), the CCW's which are command chained apply to separate records. Combinations of data and command chaining allow fairly intricate channel programs to be set up.

Worked Example

9-5 As an example of command chaining consider reading three records from tape. A program to do this follows:

```
BEGIN     BALR      2,0
          USING     *,2
          L         0,CAWIN
          L         1,TAPE
          SVC       X'00'
            .
            .
            .
          L         1,TAPE
          SVC       X'00'
          L         0,CAWIN1
          L         1,TAPE
            .
          SVC       X'00'
            .
            .
            .
          L         1,TAPE
          SVC       X'01'
          L         0,CAWIN2
          L         1,TAPE
          SVC       X'00'
            .
            .
            .
          L         1,TAPE3
          SVC       X'01'
```

```
CAWIN1      DC       X'00', A(IOCOMIN)
CAWIN2      DC       X'00', A(IOCOMIN+8)
CAWIN3      DC       X'00', A(IOCOMIN+16)
            DS       OF
TAPE        DC       X'00000108'
IOCOMIN     CCW      2, RDAREA, X'00', 80
            CCW      2, RDAREA+80, X'00', 80
            CCW      2, RDAREA+160, X'00'80
RDAREA      DS       CL240
            END      BEGIN
```

As an alternative,

```
BEGIN       BALR     2, 0
            USING    *, 2
            L        0, CAWIN
            L        1, TAPE
            SVC      X'00'
              .
              .
              .
            L        1, TAPE
            SVC      X'01'

CAWIN       DC       X'00', A(IOCOMIN)
            DS       OF
TAPE        DC       X'00000108'
IOCOMIN     CCW      2, RDAREA, X'40', 80
            CCW      2, RDAREA+80, X'40', 80
            CCW      2, RDAREA+160, X'00', 80
RDAREA      DS       CL240
            END      BEGIN
```

In the first program, the supervisor must issue three start I/O instruction to read the three records and check for the completion of each with the program possibly waiting at each check. In addition, the tape unit is started and stopped three times.

The second program which was command chained CCW's (bit 33 on) causes only one start I/O operation and one check to occur. It can run uninterrupted while the three records are brought into storage and the tape is only started once thereby saving time in comparison to the previous approach.

Worked Example

9-6 Data chaining and command chaining can be mixed. As an example of this, we will combine the previous examples. Suppose that it was necessary to read three records without stopping the tape and separate each of the 80 byte records into a 30 byte NAME location, 20 byte DEPT location and a 30 byte TITLE location. The channel program (CCW's) at IOCOMIN are:

```
IOCOMIN   CCW      2,NAME1,X'80',30
          CCW       ,DEPT1,X'80',20
          CCW       ,TITLE1,X'40',30
          CCW      2,NAME2,X'80',30
          CCW       ,DEPT2,X'80',20
          CCW       ,TITLE2,X'40',30
          CCW      2,NAME3,X'80',30
          CCW       ,DEPT3,X'80',20
```

The third facility for connecting CCW's to form a channel program is a channel command, Transfer In Channel (operation code 8). When the channel encounters a CCW with an operation code of 8, it fetches another CCW from the location specified in the data address field (bits 8 - 31) of the Transfer In Channel CCW and continues with the operation specified in the fetched CCW. It can be used to combine separately written CCW lists which are at different locations in storage into one channel program. This allows loops in a channel program much the same as the branch instructions allow the CPU to process instructions in sequence even though they do not occupy consecutive storage locations.

9.6 Problems

1. Using the conventions described in Chap. 9 to request an I/O operation, write a program which reads 80 byte cards from a card reader attached to the multiplex channel through a control unit with an address of OOC; and writes them on a tape attached to a selector channel through a control unit with an address of 184.

2. Use command chaining to read five cards from the card reader in problem 1 with one I/O request and write the 400 bytes to the tape with an CCW.

3. Use a combination of data chaining and command chaining to read five cards from the card reader with one I/O request and write one record of 80 bytes on tape which consists of the first 8 bytes from the first card, the second eight bytes of the second card, etc.

Chapter 10

I/O SOFTWARE

10.1 Introduction

In the effort to achieve faster and more efficient input/output operations, the complexity of the I/O hardware and the associated programming to make it operate has increased significantly. This becomes evident when we compare the execution of one CPU instruction to read a card in a system without overlapped I/O capability and what is necessary with the /360 and similar systems as shown in Chap. 9. Even when the programmer wrote the channel program himself, a supervisory program was assumed to start the channel in operation and to handle the interrupts when they occurred.

The need for a supervisory program for input/output operations is a widespread one. Most programmers are not interested in how the input/output functions of a system work, but merely in getting data in and out of storage. Therefore, a generalized Input/Output Control Supervisor (IOCS) is usually provided by the computer manufacture as part of the programming support for the computer system. This type of program in its basic form would schedule all input/output requests on the appropriate channels; handle input/output interrupts as they occur; keep track of which I/O requests have been completed; and when errors occur, take corrective action.

Given a basic IOCS the programmer does not have to concern himself with the actual execution of the input/output operations. He will be able to deal with the most advanced input/output hardware in a straightforward manner.

10.2 Data Formats

Since the purpose of an input/output system is to transfer data to and from storage, a discussion of the characteristics of that data and how it physically exists on external storage and in core storage is essential. The amount of data which each I/O device processes per I/O command is called a physical record. 'For a tape unit a physical record is the number of bytes which lie between two successive inter-record gaps on the tape (Fig. 10-1a). A tape unit has a very flexible physical record size. The maximum physical record could theoretically be the entire reel of tape. Devices like disk or drum on the S/360 have their physical data record defined in the same manner as tape, that is, the bytes contained between two consecutive inter-record gaps. However, the maximum physical record size is usually limited to the track size of the particular disk or drum device. For card equipment the physical record is defined as the card itself and for printers the physical record is the print line.

It is useful to think of a program processing a series of logically related data items, or a data set, related. For example, a payroll program deals with one employee at a time. All the data concerning each employee, such as, social security number, salary, number of exemptions, and so forth, are logically grouped. Therefore, the payroll program should have all this data at hand when processing each employee's payroll record. When related information is grouped together in a string of bytes, it is called a logical record. When most programs request an input/output operation they are either reading or writing a logical record.

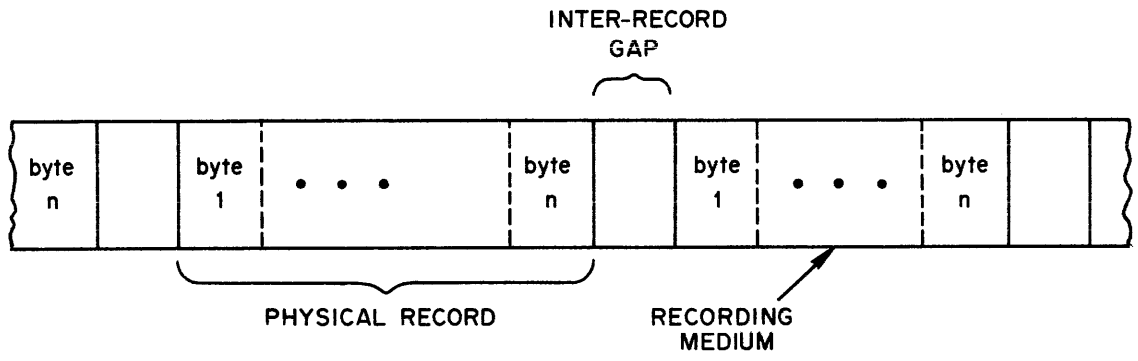
The logical record and physical record may be the same entity. But in many cases the logical records are small compared to the physical record size which is efficient for a particular I/O device. Most I/O devices require a specific amount of start up time before they are ready to transfer data. This time is fixed and therefore, is the same regardless of the size of the physical record. Hence, it is more efficient to transfer large blocks of information once the device is in operation than small ones.

A technique has been developed which satisfies the program's need to operate upon logical record and have the device transmit a physical record in a size which is efficient for that device. The technique is called blocking (Fig. 10-1b). A number of logical records are combined or blocked into one physical record. When this physical record is read into storage each logical record is separated from the physical record and used by the program as though only one logical record had been read into storage. The separation of the logical records from the physical record is called de-blocking. When a program is writing out logical records they are collected together and written as one physical record. The logical record is the collection of data the program deals with and the physical record is the data the I/O device transfers with one I/O command.

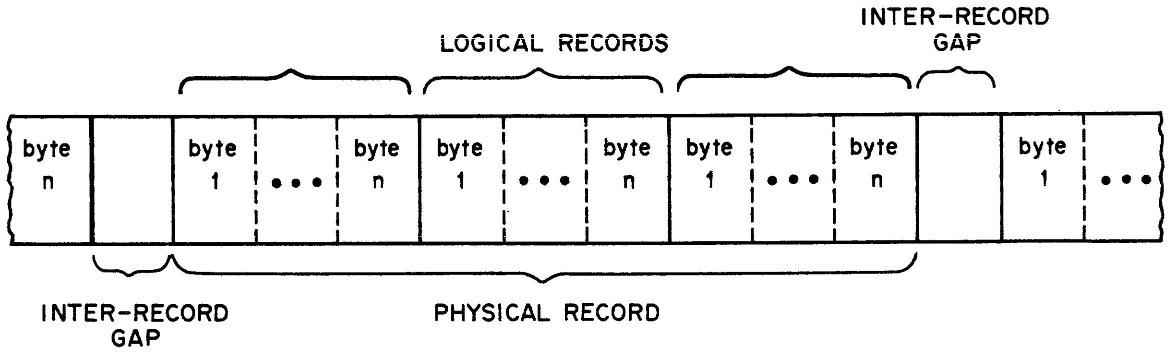
Logical records can have two formats. Fixed length logical records all have the same length in bytes. Variable length logical records vary in length from record to record. Some applications have data records where the amount of information in the record depends on the record type. The length of these records can vary by a considerable amount. If all records had to be the same length, they all would have to be the length of the maximum record possible for the application. Depending on the number of records and their distribution from the minimum to the maximum length, much external storage can be wasted if all records must be the same length.

For fixed length logical records it is simple to determine the number of logical records per physical record which is called the blocking factor. The physical record length is then an integral number of logical record

PHYSICAL AND LOGICAL RECORDS



a) PHYSICAL RECORD



b) 3 LOGICAL RECORDS/PHYSICAL RECORD

Fig. 10-1

lengths. All physical records that contain fixed length logical records are the same size. Therefore, once the blocking factor is known the blocking and deblocking of logical records into and from physical records is the same for each physical record or block.

Blocking and deblocking variable length logical records is more difficult than fixed length records. Since the length of each record can be different, there must be a way for the blocking/deblocking program to know the length of the record it is about to process. This is accomplished on the 360 by using the first four bytes of the logical record as a control word. The first two bytes contain the length of the record and the next two bytes contain blanks (See Fig. 10-2a).

The physical record which contains logical variable length records must itself be variable in length. The technique used to block variable length logical records is to specify the maximum physical length recorded desired and then place as many variable length logical records in a physical block of this size. When there is not sufficient room to place another logical record into the physical record it is then written with the unused space left off (See Fig. 10-2b). Since the physical record varies in size, there is a control word of four bytes in front which is in the same format as the control word for each logical record. As a result, the blocking factor for variable length records, unlike fixed length records, varies from physical record to physical record.

Worked Example

10-1 The idea of blocking and deblocking of logical records can be made clearer by two examples. We will first consider fixed length records. We wish to read from TAPE 1 logical records, each 80 bytes long, blocked three per physical record and write them on TAPE 2 blocked five per physical record. We will use the same conventions as in Chap. 9.

```
BEGIN    BALR   2,0
         USING  *,2
         L      4,0=F'80'
         L      5,=A(RDAREA+160)
         L      8,=F'80'
         L      9,=A(WTAREA+320)
         L      7,=A(WTAREA)          7 IS BLOCKING REGISTER
READ     L      0,CAWIN
         L      1,TAPE1
         SVC    X'00'
         L      1,TAPE1
         SVC    X'01'
         L      3,=A(RDAREA)          3 IS DEBLOCKING REGISTEF
```

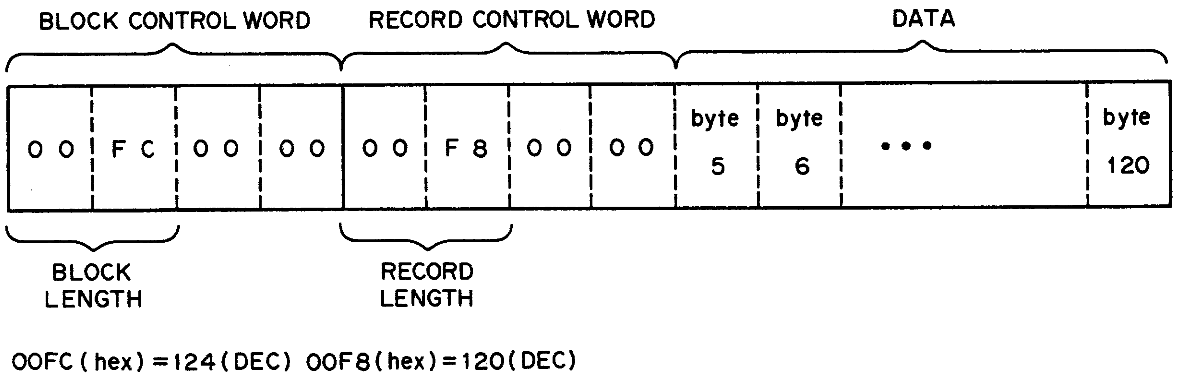
```
MOVE     MVC    0(80,3), 0(7)            SEE NOTE (*) BELOW
M        BXLE   3,4,READ                 3 LOGICAL RECORDS DE-
*                                          BLOCKED?
         BXLE   7,8,WRITE                5 LOGICAL RECORDS DE-
*                                          BLOCKED?
         BC     15,MOVEIN                NO
WRITE    L      0,CAWOUT
         L      1,TAPE2
         SVC    X'00'
         L      1,TAPE2
         SVC    X'01'
         L      7,A(WTAREA)
         BC     MOVE
CAWIN    DC     X'00',A(IOCOMIN)
CAWOUT   DC     X'00',A(IOCOMOU)
         DS     0F
TAPE1    DC     X'00000108'              TAPE ON CHANNEL1
TAPE2    DC     X'00000208'              TAPE ON CHANNEL2
IOCOMIN  CCW    2,RDAREA,X'00',240       PHYSICAL RECORD=3
*                                          LOGICAL RECORDS
IOCOMOU  CCW    1,WTAREA,X'00',400       PHYSICAL RECORD=5
*                                          LOGICAL RECORDS
RDAREA   DS     CL240
WTAREA   DS     CL400
         END    BEGIN
```

(*) If processing is needed code could be inserted between L and MVC. The address of a logical record in RDAREA is in register 3, the address of the next position in WTAREA is in register 7.

This program reads a physical record into RDAREA and using register 3 separates each 80 byte logical record from the 240 byte physical record. At MOVE, each logical record is moved from its location in RDAREA (GPR3) to the next available area in WTAREA. This location is maintained in GPR7. When three logical records have been moved from RDAREA, a new physical record is read. When five logical record have been moved into WTAREA that physical record is written and a new one begun.

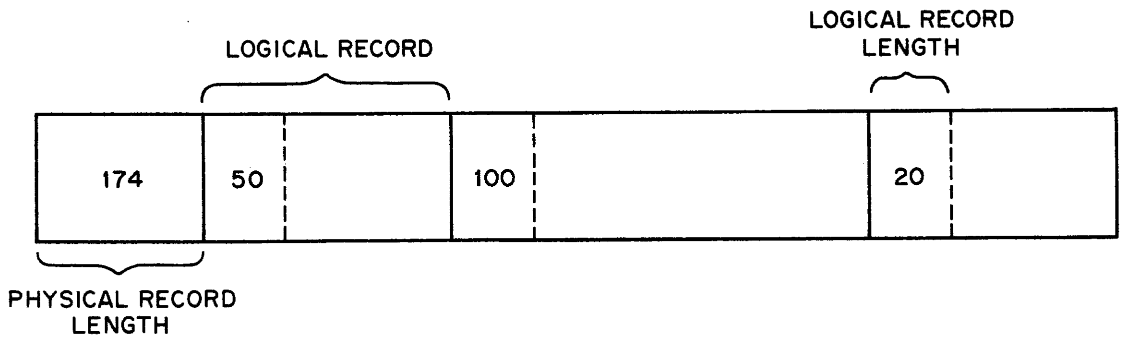
The second example is for variable length records. The program function is the same as before. Here the maximum physical record is 500 bytes on input and 1000 bytes on output.

Variable Length Logical Record



Unblocked: Physical Record = 1 Logical Record

a)



Blocked: 1 Physical Record = 3 Logical Records + Physical Record Control Word

b)

Fig. 10-2

```
BEGIN     BALR   2,0
          USING  *,2
          LH     10,=M'255'
          L      7,=A(WTAREA+4)          BLOCKING REGISTER IS 7
READ      L      0,CAWIN
          L      1,TAPE1
          SVC    X'00'
          L      1,TAPE1
          SVC    X'01'
          L      5,=A(RDAREA-4)
          AH     5,RDAREA                5 CONTAINS ADDRESS OF
*                                          LAST BYTE OF PR+1
          L      3,=A(RDAREA+4)          DEBLOCKING REGISTER IS 3
NEXT      LH     4,0(3)                  4 CONTAINS LENGTH OF LOGI-
*                                          CAL RECORD BEING MOVED
          L      11,=A(WTAREA+10000)
          SR     11,7
          CR     11,4
          BC     4,WRITE                 IS THERE ENOUGH SPACE
*                                          LEFT IN WTAREA?
COMP      CR     4,10                    IS RECORD BEING MOVED
*                                          255 BYTES LONG?
          BC     2,MOVE1                 YES, GO TO MOVE1
          STH    4,LENGTH                SAVE LENGTH
          MVC    MOVE+1(1),LENGTH+1      SET UP LENGTH FOR MOVE
MOVE      MVC    0(255,7),0(3)           MOVE RECORD
          AH     7,LENGTH                UPDATE BLOCKING REGIS-
*                                          TER
          AH     3,LENGTH                UPDATE DEBLOCKING
*                                          REGISTER
          SR     5,3                     ANY RECORDS LEFT
          BC     8,READ                  IN RDARER: NO
          BC     15,NEXT                            YES
MOVE1     MVC    0(255,7 ,0(3)
          SR     4,10                    UPDATE: LENGTH
          AR     7,10                       BLOCKING REGISTER
          AR     3,10                       DEBLOCKING REGISTER
          BC     15,COMP
WRITE     L      6,=A(WTAREA)
          SR     6,7                     DEVELOP ACTUAL LENGTH
          STH    5,WTAREA                  OF PHYS. REC. IN WTAREA
          L      0,CAWOUT                  AND STORE IT IN PHYSICAL
*                                          CONTROL WORD AND CCW
          L      1,TAPE2
          SVC    X'00'                   WRITE RECORD
          L      1,TAPE2
          SVC    X'01'
```

```
         L        7,=A(WTAREA+4)         INITIALIZE BLOCKING
                                          REGISTER
         BC       15,COMP
LENGTH   DS       CL2
         DS       0F
CAWIN    DC       X'00',A(IOCOMIN)
CAWOUT   DC       X'00',A(IOCOMOU)
TAPE1    DC       X'00000108'
TAPE2    DC       X'00000208'
IOCOMIN  CCW      2,RDAREA,X'20',500     COUNT 500 FOR MAXIMUM
                                          PHYSICAL RECORD.  IF
IOCOMOU  CCW      1,WTAREA,X'00',0        SHORTER NO ERROR BE-
RDAREA   DS       CL500                   CAUSE 34 ON COUNT TO
RTAREA   DS       CL1000                  BE INITIALIZED
         END      BEGIN
```

Unlike the program for deblocking fixed length records all the data concerning the actual physical record length and logical record length must be determined from the records themselves. The only sure thing is that the physical records on input will not exceed 500 bytes in length. The CCW used to read TAPE1 uses a count of 500, the maximum size expected. However, when a shorter physical record is encountered, the wrong length record error indication is suppressed by having bit 34 as one. On output, the actual size of the physical record to be written is placed into the count field of IOCOMOU and also into the control word of the physical record itself. Notice that although the maximum physical record length is 1000 bytes, the actual length depends on the length of the logical records that can be totally contained within the 1000 bytes.

10.3 Buffering

As we discussed in Chap. 9, efficient use of the hardware facility of overlapped input/output operations within a program requires advance planning. The program must anticipate what data it will need next and request this data to be read into storage prior to the time the data is needed. This enables the I/O operation to begin while the program continues execution on previous input data. When a record is needed in a program, it will already be in core storage if the need for it was anticipated far enough in advance.

There is a large class of applications which process records one after another. For example, a program may need all or almost all of the records on a reel of tape. In a payroll program, the employee master file will require processing of every record — either to pay an employee or to terminate him. If all records in a data set are to be read into storage in the order they exist on the external storage medium, we have sequential or consecutive processing. Since data records are processed in a sequential manner, there is no difficulty in anticipating what data record will be needed next. A technique has been developed which allows efficient use of overlapped I/O operations when a data set is being processed sequentially — buffering. Since each physical record in the data set will be read into storage, the I/O operation to read the next physical record can be given as soon as enough storage in the I/O area is available to hold that record. The idea is to set aside multiple I/O areas or buffers to hold physical records from the data set. As the program is processing the data from one I/O buffer it has requested I/O operations to fill the other areas with the succeeding physical records from the data set. When the program is finished using the data in one I/O buffer it initiates an input operation for it and starts work on the next buffer alternating through the set of buffers.

Worked Example

10-2 An example of a program to illustrate the buffering technique is a card to tape routine. The program will read cards and then write them on tape. Three buffers will be used for the data set being read from the card reader and two buffer areas will be set aside for the data set being written on tape.

```
BEGIN    BALR   2,0
         USING  *,2
         MVI    SWITCH+1,X'F0'  INITIALIZE SWITCH TO BRANCH
*                                 FIRST TIME THROUGH
         L      6,=F'0'
         L      3,=F'8'
START    L      0,CAWIN(3)      LOAD CAWIN INDEXED IN REVERSE
                                  BY GPR3
         L      1,CARDRDR
         SVC    X'00'
         S      3,4             HAVE 3 BUFFERS BEEN
*                                 SCHEDULED FOR READS
         BC     8,START
         BC     2,START         YES, INITIALIZE REGISTER 3
         L      3,=F'8'
READCHK  L      1,CARDRRR       CHECK READ ASSOCIATED WITH
                                  CAWIN(3)
         SVC    X'01'
         L      4,CAWIN(3)      LOAD ADDRESS OF CCW ASSOCIA-
*                                 TED WITH CAWIN(3) IN REG. 4
         L      5,0(4)          FIRST 4 BYTES OF CCW ASSOCIA-
*                                 TED WITH CAWIN(3) IN 5
         L      4,0(5)          ADDRESS OF INPUT BUFFER ASSO-
*                                 CIATED WITH CAWIN(3) IS IN 4
         MVC    0(80,4),TEMP    PHYSICAL RECORD IN TEMP
         L     0,CAWIN(3)             SCHEDULE BUFFER ASSOCIATED
*                                       WITH CAWIN(3) FOR READ
         L     1,CARDRDR
         SVC   X'00'
         S     3,=F'4'                UPDATE GPR3 TO
         BC    2,FORIN3               POINT TO NEXT
         BC    4,EHTIN3               CAWIN
         BC    15,SWITCH
FORIN3   L     3,=F'4'
         BC    15,SWITCH
EHTIN3   L     3,=F'8'
SWITCH   BC    15,HERE                BRANCH AROUND WRITE CHECK
*                                       UNTIL 2 WRITES HAVE BEEN
                                        GIVEN
         L     0,TAPE                 CHECK WRITE ASSOCIATED
                                        WITH CAWOU(6)
         SVC   X'01'
HERE     L     4,CAWOU(6)             LOAD ADDRESS OF CCW ASSO-
*                                       CIATED WITH CAWOU(6) GPR4
         L     5,0(4)                 LOAD FIRST 4 BYTES OF CCW
                                        INTO GPR5
         LA    4,0(5)                 LOAD ADDRESS OF BUFFOU
*                                       ASSOC. WITH CAWOU(6) IN TO
                                        GPR4
         MVC   0(4),TEMP              PHYSICAL RECORD IN OUT BUF-
*                                       FER ASSOC. WITH CAWOU(6)
         L     0,CAWOU(6)             ISSUE WRITE FOR BUFFOU
                                        ASSOCIATED WITH
         L     1,TAPE                 CAWOU(6)
         SVC   X'00'                  UPDATE GPR6
         A     6,=F'4'                TO POINT TO NEXT BUFFOU
         BC    8,READCHK
         MVI   SWITCH+1,X'00'         SET SWITCH TO NOP
         L     6,=F'4'                SECOND TIME THROUGH
         BC    15,READCHK
CAWIN    DC    X'00',A(IOCOIN1)
         DC    X'00',A(IOCOIN2)
         DC    X'00',A(IOCOIN3)
CAWOU    DC    X'00',A(IOCOOU1)
         DC    X'00',A(IOCOOU2)
IOCOIN1  CCW   2,BUFFIN1,X'00',80
IOCOIN2  CCW   2,BUFFIN2,X'00',80
IOCOIN3  CCW   2,BUFFIN2,X'00',80
IOCOOU1  CCW   1,BUFFOU1,X'00',80
IOCOOU1  CCW   1,BUFFOU2,X'00',80
TEMP     DS    CL80
BUFFIN1  DS    CL80
```

```
BUFFIN2  DS     CL80
BUFFIN3  DS     CL80
BUFFOU1  DS     CL80
         END    BEGIN
```

The program starts by issuing three read request to fill the three input buffers (BUFFIN1, BUFFIN2, BUFFIN3). When the BUFFIN3 is filled the code moves it to TEMP and schedules BUFFIN3 for another read. Then the record is moved from TEMP to BUFFOU1 and BUFFOU1 is scheduled for write. The next read is then checked (BUFFIN2, etc.). The read portion cycles through the 3 input buffers and the write code cyles through the 2 output buffers.

10.4 Input/Output Control System Functions

The input/output control system or data management section of system/360 operating systems usually provides functions at three levels. The fundamental level provides channel scheduling and error and interrupt handling. The programmer writes the channel program (CCW list) and must synchronize his program with the completion of the input/output operation. This level is very similar to what the previous examples have used. The intermediate level, in addition to the services performed at the fundamental level, relieves the programmer of writing the channel programs but he still must provide the synchronization between the I/O operations and the program execution. The third level of input/output capability provides in addition to the previous two, the synchronization capability automatically. Also, where the first two deal solely with a physical record and the program must anticipate the need for data in advance, this level provides both blocking and deblocking of logical records and automatic buffering. When choosing which level to use, the general rule is the higher the level the easier it is to use but it gives less flexibility. To describe each level and its capabilities, the macros of the 8K Basic Operating System will be used.

Since many of the examples presented previously assumed the function of the fundamental level, this level would be an appropriate place to start. In the /360 terminology this level is called Execute Channel Program (EXCP). There are three macros associated with the EXCP level. The first is the command control block macro

```
blockname  CCB  SYSnnn, command-list name
```

Here, blockname is the symbolic names associated with this command control block. There must be one command control block for each I/O device used in the program.

SYSnnn is the system symbolic name of the I/O device. The possible names are SYSRDR, SYSLIST, SYSIPT, SYSOPT, SYSLOG, SYS000-SYS254. The significance of these names will be discussed in the next chapter.

Command-list name is the symbolic name of the channel command or of the first channel command of a channel program that the program desires to have executed. This macro expands into a control block and takes the place of loading GPR0 with a CAW and GPR1 with the device address and having to construct those constants in the program as done in Chap. 9.

The next macro is Execute Channel Program

```
EXCP      block-name
```

EXCP is the macro operation which request the I/O supervisor to issue a start I/O operation and block-name is the symbolic name of the CCB macro which is associated with the I/O device involved. This macro replaces the supervisor call with a code of zero used in the previous examples.

The third macro at this level is the wait macro.

```
WAIT      block-name
```

Here, WAIT is macro used to synchronize the program execution with the completion of an I/O operation and blockname the name of the CCB macro associated with the I/O device involved. The use of this macro replaces the supervisor call instruction with a code of one.

Worked Example

10-3 Let's compare the example from Chap. 9 which read a punched card with a program using the EXCP level to do the same function. The coding using EXCP functions is shown in Fig. 10-3 alongside its assembly language equivalent.

```
           Chapter 9                                    EXCP

BEGIN      BALR    2,0                                  BALR    2,0
           USING   *,2                                  USING   *,2
             .                                            .
             .                                            .
             .                                            .
           L       0,CAW                                EXCP    CARDBLK
           L       1,CARDRDR                              .
           SVC     X'00'                                  .
             .                                            .
             .                                            .
             .
           L       1,CARDRDR                            WAIT    CARDBLK
           SVC     X'01'                                  .
             .                                            .
             .                                            .
             .
CAW        DC      X'00'A(IOCOM)               CARDBLK  CCB     SYS001,IOCOM
IOCOM      CCW     2,RDAREA,X'00',80           IOCOM    CCW     2,RDAREA,X'00',80
RDAREA     DS      CL80                        RDAREA   DS      CL80
           DS      0F                                   END     BEGIN
CARDRDR    DC      X'00000008'
           END     BEGIN
```

Fig. 10-3 Comparison of I/O Programs

Even though the registers which are used to pass the parameters are not explicitly stated when using macros, it should be remembered that registers 12, 13, 14, 15, 0, 1 are used by the supervisor and other programs for interrupt handling, linkage and parameter passing. These registers should not be used by the problem program for purposes other than mentioned above when communicating with the supervisor or when interrupts are expected. The CCB macro expands into a control block eight bytes in length. Two bytes hold the device address and 2 bytes for the address of the CCW(s). The remaining four bytes are used by the supervisor to communicate error and status information to the program and vice versa.

The intermediate level of input/output support, like EXCP, presents the programmer with a physical record in an area of storage specified by the program. This level usually called the READ/WRITE level and the highest level called the GET/PUT level enable the programmer to accomplish I/O without writing CCWs in his program. There are subroutines which are part of the I/O system which will do this for him. These routines are called access methods.

In order for the access method to construct a channel command or commands it needs certain information about the properties of the data set to be processed. This is accomplished by a DTFXX macro. The DTF stands for Define the File (file is equivalent to data set). The XX indicates the manner in which the records are accessed on external storage. The possible values for XX are:

SR	for serial or sequentially accessed data. Each record is accessed in a consecutive manner. This type of processing can be used with any type of I/O device.
DA	for direct access of records from disk or drum type I/O devices. Each record is accessed in a random manner. The records preceding the desired one in the data set do not have to be accessed previously in order to access the desired record.
IS	for indexed sequential files. Each record is identified by a key (tag). The records are accessed either sequentially by ascending key value or directly by a particular key value.
PH	for EXCP level. Applicable to all I/O devices when labels are employed. Labels will be discussed in the next section.

A DTFDA macro for a file where the records will be read on a random basis has the following format

```
name      DTFDA      operand1, operand2,  etc.
```

Here name is symbolic name of the DTFDA; the operands can be the following:

BLKSIZE=n	n is the size of the maximum physical record transferred. This is used to construct the count field in the CCW.
DEVICE=DISK11	DISK11 indicates that the file resides on a 2311 disk device.
ERRBYTE= ERRFLD	ERRFLD is the name of a two byte field where the supervisor can store status and error information.
IOAREA1= RDAREA	RDAREA is name of storage the location where the physical record is read into. It must be big enough to hold the largest record. It can be defined in the program by use of the DS operation.
READID= YES	The record is to be read will be located by an ID.
RECFORM= FIXONE	The records are physical records of fixed lengths. One logical record/ physical record is all that's allowed.
SEEKADR= NAME	Name is the location which contains the track address of the record to be accessed. This address is as follows:

M (1 byte)	symbolic unit number
BB (2 bytes)	00
CC (2 bytes)	0X X=0, 199
HH (2 bytes)	OX X=0, 9
R (1 byte)	record number

TYPEFLE= INPUT	the file will be read from but not written to.

There is one DTF macro in the program for each data set to be used. The DTF macros must occur immediately after the START statement in a user's program. If the program is to be executed on a disk system, the DTF macros must be preceded by a DTFBG (begin) macro statement. In any case, a DTFEN (end) statement must follow the last DTF macro.

Once the file or data set has been defined by a DTF a READ or WRITE macro may be issued any number of times to transfer a physical record from or to that file. The READ macro for a file with a DTFDA has the following format:

```
READ        filename, KEY

READ        filename, ID
```

filename	is the symbolic name of the DTFDA which defines this file.
KEY/ID	indicates whether the record is to be retrieved by a symbolic key or its location on the disk.

The WRITE macro has the following formats:

```
WRITE       filename,  KEY
WRITE       filename,  ID
WRITE       filename,  AFTERID
WRITE       filename,  AFTER
```

Here, _filename_ is the symbolic name of DTFDA;

KEY/ID   is the same as for READ macro;

AFTERID   indicates that this record is to be written after the record whose ID is supplied;

AFTER   indicates that this record is to be written following the last record written on this file.

Since the actual input/output operation is accomplished in parallel with processing, the wait macro is used for synchronization. This macro has the following format:

```
WAITF       filename
```

where _filename_ is the symbolic name of the DTFDA

Worked Example

10-4 To illustrate the use of the READ/WRITE level of I/O control let's take as an example a routine to update a direct access file. The routine will read a card, using EXCP, and will use the key in the card to locate the record on the file. Then read the record into core, update it with other information from the card and then write it back to the file.

The key is in positions 1 to 8 of the card and is a part number. Position 9 contains the transaction code, a plus or minus, which indicates if the data in positions 10 through 14 should be added to or subtracted from the inventory field of the record for this part number. The record in the direct access file has the following format:

Position	Item
1-8	part number
4-5	actual part number
6-8	TRACK ADDRESS (MCH)
	M Drive number
	C Cylinder number
	H Head number
9-14	inventory field
14-60	fields not pertinent to this routine

Although the record on the direct access file is 60 bytes, the key is 8 bytes and not transferred with the READ or WRITE therefore, an I/O area of 52 bytes is all that is needed. The program attempts to overlap the reading of the next card with the processing of the present record. Also the writing of the updated record is overlapped with the set up to read the next record. The complete program is shown in Fig. 10-4.

The use of the READ/WRITE level allows the user to perform I/O operations without the knowledge of how CCW's are written for a particular type of device such as a 2311 disk drive. He accomplishes this be defining his file in terms he knows in the DTF.

The highest level of IOCS is the GET/PUT level. The file is defined in a DTF just as in the READ/WRITE level. In addition the user doesn't have to plan ahead in order to achieve I/O overlap. To increase the I/O overlap efficiency, buffering if provided at this level. Blocking and deblocking are also provided.

The two macros used at this level in addition to the DTF are GET and PUT. The format of these macros are

```
         DTFBG
INVNTRY  DTFDA  BLKSIZE=52, DEVICE-DISK11,
*               ERRBYTE=ERRLOC, IDAREA=INVBUF
*               KEYARG=PARTNO, KEYLEN=8,
*               READKEY=YES, RECFORM=FIXUNB,
*               SEEKADR=TRACKNO, SRCHM=YES
*               TYPEFILE=INPUT
         DTFEN
BEGIN    BALR   2, 0
         USING  *, 2
         EXCP   CARDBLK                            READ 1ST CARD
         OPEN   INVNTRY                            DISCUSSED IN 10-5
CARD     WAIT   CARDBLK                            WAIT FOR DATA TO BE IN RDAREA
         MVC    TEMP, RDAREA                       SAVE DATA IN TEMP
         EXCP   CARDBLK                            READ NEXT CARD
         MVC    PARTNO, TEMP(7)                    MOVE KEY TO PARTNO
         MVC    TRACKNO, TEMP+5(1)                 MOVE M TO TRACK ADDRESS
         MVC    TRACKNO+4, TEMP+6(1)               MOVE C TO TRACK ADDRESS
         MVC    TRACKNO+6(1), TEMP+7(1)            MOVE H TO TRACK ADDRESS
SWITCH   BC     15, READD                          IF NOT 1ST TIME
         WAITF  INVNTRY                            THROUGH DON'T CHECK PREVIOUS WRITE
READD    READ   INVNTRY, KEY                       READ RECORD
         PACK   WORK+2(3), TEMP+13(5)              PACK INPUT DATA
         OC     WORK+2(1), TEMP+8                  OR IN TRANS ACTION CODE
         WAITF  INVNTRY                            CHECK READ
         AP     INVBUF+13(6), WORK+2(3)            UPDATE
         WRITE  INVNTRY, KEY                       WRITE
         BC     15, CARD
TRACKNO  DC     X'0000000000000000'
PARTNO   DS     CL8
INVBUF   DS     CL52
WORK     DS     CL3
CARDBLK  CCB    SYS001, IOCOM
IOCOM    CCW    2, RDAREA, X'00', 80
ERRLDC   DS     CL2
RDAREA   DS     CL80
```

Fig. 10-4 Program for Worked Example 10-4

```
GET          filename

GET          filename, workname
```

Here, GET presents the user with the next logical record; filename is the name of the DTF which describes this file; workname is the name of a storage area and indicates that if this parameter is supplied the next logical record is placed in that location. If it is not present the location of the next logical record in the buffer is placed in the IOREG.

```
PUT          filename

PUT          filename, workname
```

Here, PUT places the logical record in the output buffer; filename is the name of DTF which describes this file; workname, if present, gives the location of the logical record which is to be moved into the output buffer, if not present, the address of the next free byte in an output buffer is placed in IOREG.

Worked Example

10-5 To illustrate the GET/PUT level let's use a similar example as that used to illustrate blocking/deblocking and buffering in Sec. 10. 2 and 10. 3. The problem is to read from TAPE 1 blocked variable length records and write them on TAPE 2. The maximum physical input record is 1000 bytes long and 2500 bytes for output and there are two buffers for each tape. The OPEN and CLOSE macros in the following program will be discussed in Sec. 10. 5.

```
TAPE1    DTFSR   BLKSIZE=1000, DEVADDR=SYS001
*                DEVICE=TAPE, EOFADDR=CLOSEM
*                ERROPT=IGNORE, FILABLE=STD,
*                IOAREA1=BUFIN1, IOAREA2=BUFIN2,
*                RECFORM=VARBLK, TYPEFILE=INPUT,
*                WORKA=YES
TAPE2    DTFSR   BLKSIZE=2500, DEVADDR=SYS002
*                DEVICE=TAPE, FILABLE=STD, IOAREA1=BUFOUT1,
*                IOAREA2=BUFOUT2, RECFORM=VARBLK,
*                TYPEFILE=OUTPUT, WORKA=YES
         DTFEN
BEGIN    BALR    2,0
         USING   *,2
         OPEN    TAPE1, TAPE2
START    GET     TAPE1, TEMP
         PUT     TAPE2, TEMP
         BC      15,START
TEMP     DS      CL2500
BUFIN1   DS      CL1000
BUFIN2   DS      CL1000
BUFOUT1  DS      CL2500
BUFOUT2  DS      CL2500
CLOSEM   CLOSE   TAPE1,TAPE2
         END
```

If the program above is compared with those in Sec. 10.2 and 10.3, the ease of using the GET/PUT level with programming at the basic level is obvious. The system automatically schedules the input buffer for I/O as soon as the last logical record in the physical record is moved to TEMP by the GET. The system also automatically writes the output buffer when another logical record added would cause the physical record to exceed 2500 bytes. If the next logical record is not yet in storage when the GET is executed the program will wait until it is. If space is not available when the PUT is issued the program will also wait until space in an output buffer is available. Also, the system is placing one logical record at a time into TEMP (deblocking the physical input record) and placing the logical record in TEMP into the output buffer (blocking to create the output physical record).

The new parameters in the DTFSR are

EOFADDR=CLOSEM	when a tape mark is read on the input tape, this indicates the end of the records and the system will branch to CLOSEM
ERROPT=IGNORE	when an error occurs on the input tape ignore it.
FILABLE=STD	standard labels are used (discussed in 10-5)
BUFIN1, BUFIN2	are the two input buffers
BUFOUT1, BUFOUT2	are the two output buffers
RECFORM=VARBLK	the logical records are variable length and blocked

The GET/PUT level provides the user with the following functions automatically

1. buffering

2. blocking/deblocking

3. synchronization

4. channel command words

Depending on the flexibility desired, an appropriate level of IOCS can be chosen to ease the input/output programming.

10. 5 Labels

To insure that the proper tape reel or disk pack has been mounted so that the program doesn't read the wrong data or destroy valid data, a system of labels is used. Labels are usually written at the beginning and end of the data file. The label at the beginning is called a header and the one at the end a trailer. Tapes or disks may be unlabled, or use standard or non-standard label conventions. If the tapes are labeled according to the standards of the system, the system checks the labels. If the labels are of a non-standard format, it becomes the user's responsibility to check them.

For standard labels using the Disk Operating System there must be one volume label, 80 characters in length, and one data set header and trailer label also 80 characters in length. There may be additional volume labels and data set labels but they are not checked by IOCS. Fig. 10-5 illustrates.

There are two macros where appeared in Sec. 10-4, which are used to initialize IOCS for processing a particular file and to signal that all processing of the file is completed. These macros also cause the system to check the appropriate labels verifying that the proper disk pack or tape reels are mounted. These macros are OPEN and CLOSE. They have the following format

```
OPEN                filename, filename, etc.
```

This macro initializes the appropriate parts of the input/output routine so that the programs using either the Read/Write or Get/Put level of IOCS can process the filename(s) listed. It also causes the volume and data set header label to be verified at all three levels if labels are used.

The operand <u>filename</u> is the symbolic name of one or more DTF statements which will be processed by a READ, WRITE, GET, or PUT macro or when using EXCP and label checking is desired.

```
CLOSE               filename, filename, etc.
```

The CLOSE macro causes IOCS to restore any code affected by the OPEN for this filename or filenames to its condition prior to the OPEN. It also causes the system to check standard data set trailer labels when used and filename is one or more symbolic names of DTF statements.

When using READ/WRITE or GET/PUT an OPEN for the filenames must be executed before the first READ, WRITE, GET, or PUT is issued, whether label checking is desired or not. An OPEN is only necessary with EXCP when label checking is desired.

A CLOSE should be issued for every filename for which an OPEN was issued. This should be done when the processing of the file is complete.

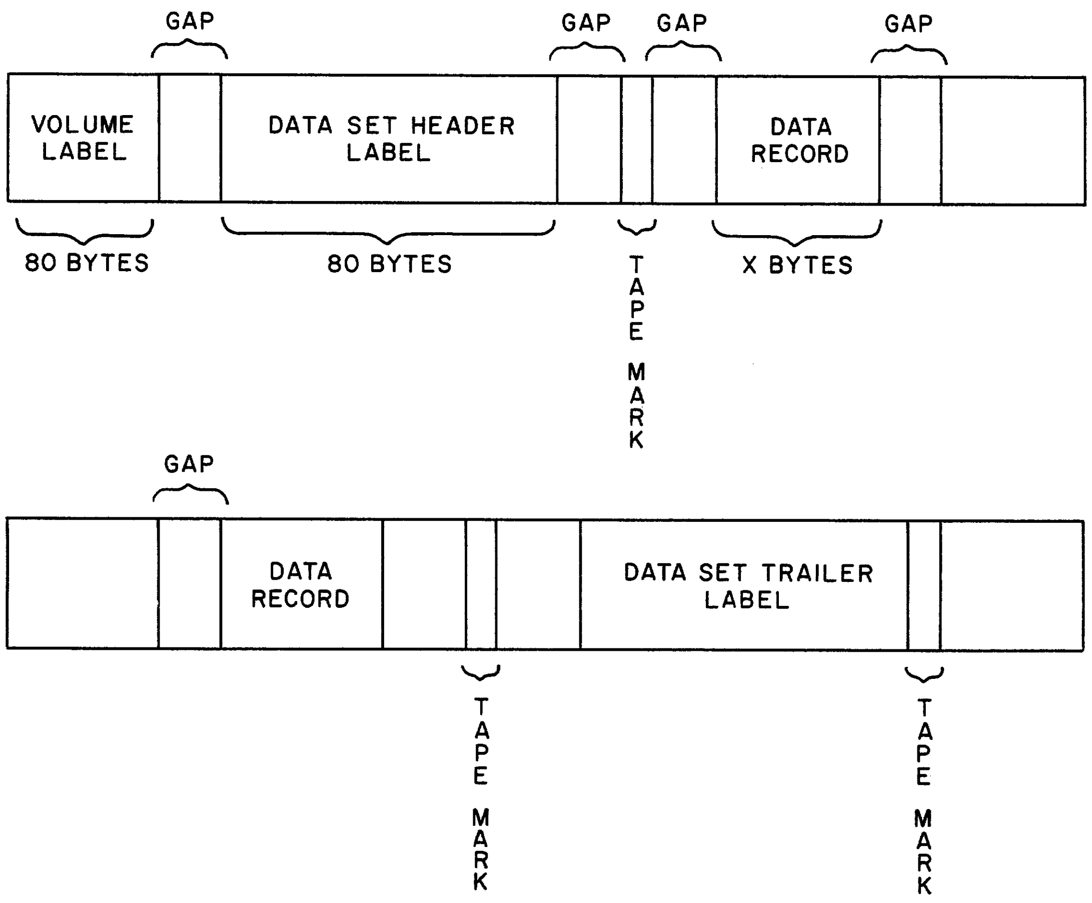


Fig. 10-5 Standard Header and Trailer Labels

Worked Example

10-6 Write an update program which reads presorted transaction cards in ascending order by customer number (cc 1-10) and adds the quantity in columns 11-15 to the corresponding record on the old master file and write the updated record on the new master file. If a record on the old master file does not have a corresponding transaction it should be written to the new master file unchanged. The records on the new and old master files are 120 bytes long, with a blocking factor of three. The customer number starts in position 31 of the 120 byte record. The field to be updated starts at byte 61 and is five bytes long and is packed decimal. When a transaction does not have a corresponding entry in the old master file the transaction should be printed. The old master and new master are on tape.

The coding is

```
         START
         BTFBGN
NEWMSTR  DTFSR    BLKSIZE=600, DEVADDR=SYS001,                X
                  DEVICE=TAPE, IOAREA1=NEW1,                  X
                  IOAREA2=NEW2, RECFORM=FIXBLK,               X
                  RECSIZE=120, REWIND=UNLOAD,                 X
                  TYPEFLE=OUTPUT, WORKA=YES
OLDMSTR  DTFSR    BLKSIZE=600, DEVADDR=SYS001,                X
                  DEVICE=TAPE, IOAREA1=OLD1,                  X
                  IOAREA2=OLD2, EOFADDR=OLDEND,               X
                  RECFORM=FIXBLK, RECSIZE=120,                X
                  REWIND=UNLOAD, TYPEFLE=INPUT,               X
                  WORKA=YES
TRANACT  DTFSR
                  BLKSIZE=80, DEVADDR=SYSIPT,                 X
                  DEVICE=READ04, IOAREA1=TRAN1,               X
                  IOAREA2=TRAN2, EOFADDR=TRANEND,             X
                  RECFORM=FIXUNB, TYPEFLE=INPUT,              X
                  WORKA=YES
ERRFILE  DTFSR    BLKSIZE=80, DEVADDR=SYSOPT,                 X
                  DEVICE=PRINTER, IOAREA1=ERR1,               X
                  IOAREA2=ERR2, RECFORM=FIXUNB,               X
                  TYPEFLE=OUTPUT, WORKA=YES
         DTFEN
         BALR     2, 0
         USING    *, 2
         OPEN     TRANACT, OLDMSTR, NEWMSTR, ERRFILE
START    GET      TRANACT, UPDATE
NEXT     GET      OLDMSTR, TEMP
COMP     CLC      TEMP+30, UPDATE(10)
         BC       8, RECUDTE           EQUAL?
         BC       2, ERROR             NO.MSTR HIGH?
         PUT      NEWMSTR, TEMP        NO.
         BC       15, NEXT
ERROR    PUT      ERRFILE, UPDATE      YES
         GET      TRANACT, UPDATE
         BC       15, COMP
RECUDTE  PACK     WORK, UPDATE+10(5)
         AP       TEMP+60(5), WORK
         PUT      NEWMSTR, TEMP
         BC       15, START
TEMP     DS       CL120
UPDATE   DS       CL80
WORK     DS       CL3
NEW1     DS       CL600
NEW2     DS       CL600
OLD1     DS       CL600
OLD2     DS       CL600
ERR1     DS       CL80
ERR2     DS       CL80
TRAN1    DS       CL80
TRAN2    DS       CL80
OLDEND   GET      TRANACT, UPDATE
         PUT      ERRFILE, UPDATE
         BC       15, OLDEND
TRANEND  CLOSE    OLDMSTR, NEWMSTR, TRANACT, ERRFILE
         EOJ
         END
```

Chapter 11

OPERATING SYSTEMS

11.1 Introduction

In Chapt. 10 an input/output control program (IOCS) was discussed. It was a generalized program which assisted the user to utilize the S/360 more efficiently for input/output operations. The idea of using a generalized program to increase the efficiency of a computing system was expanded and an operating system environment emerged. The areas in which an operating system attempts to improve the performance of the system are many. The one that stands out the most is the reduction of manual intervention on the part of the computer operation.

In most installations before the use of an operating system each job was set up by an operator. When a job was finished the operator cleaned up the system and set up the next job. This mode of operation was sufficient when computers were slower. However, as the speed of the computing system increased the speed of the operator did not. More and more time was spent waiting for the operator to complete his functions. During this time, the system was idle. If a good portion of the operator's tasks could be carried out by the system itself, a considerable amount of time could be saved and used for computing thereby increasing the capacity of the system.

What tasks did the operator have to perform for each program to be run? Depending on the program he had to:

Mount and dismount tapes
Mount and dismount disk packs
Reset the system
Set up the program to be loaded

In order to reduce manual intervention in these areas as much as possible and have these functions and other services available to the programmer, operating systems came into being.

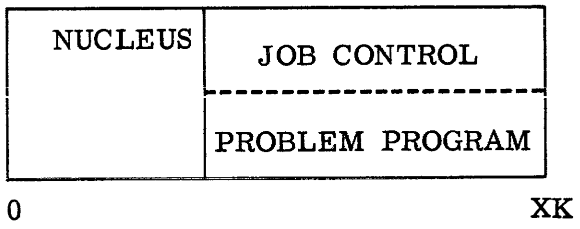
The goal of operating systems is to provide non-stop operation of the system. To do this a certain program remains resident in core storage at all times. This means that part of the storage of the central processing unit is not available to the programmer. The resident program is called the nucleus. However, the nucleus does not contain all the code that is necessary to perform the functions desired. Many routines are brought into core storage on an as needed basis. Since these routines are needed on a dynamic basis they must reside on external storage which is always available — that is, on-line. This is usually tape, disk, or drum. Therefore, a portion of the on-line external storage of the system is also reserved for the non-resident operating system routines. This external storage is usually called system's residence. One can see that there is a trade-off involved. The installation gives up some of the external storage and CPU storage of its computing system in order to gain the greater capacity which an operating system affords.

Operating systems have several components. The major two categories are Control Programs and Processing Programs. Control Programs comprise two major functions: scheduling and supervision. Processing Programs include the assembler, utilities, Sort/Merge programs and other language translators such as FORTRAN, COBOL and P/L1.

To describe the operation system functions the facilities of the Disk Operating System for System /360 will be used.

11.2 Scheduling

The purpose of the scheduler is to provide automatic job to job transition. That is, when one job which may be one or more programs finishes, the scheduler will get the next job going with a minimum of manual intervention. If the scheduling routines are to accomplish this they must receive instructions on what is to be done by the next job. This is accomplished by control cards which usually precede the job. The I/O device from which the cards and programs are read is the source of the job stream. The job stream is usually a combination of control cards, programs and data. Since the transition from job to job is automatic the jobs to be run are stacked one behind the other, separated by control cards. The Job Control (the scheduler) reads the control cards and performs the functions indicated by them. The program which follows the control cards is loaded into storage and executed. The program may even reside in an on-line storage device. In this case, through the use of certain control cards, the system will locate the program and bring it into core storage. When the program has finished, job control then reads the next set of control cards and continues. In 8K Basic Operating System the control cards are read from one I/O device, a card reader, and programs and data are read from another I/O device which may be a card reader, tape, or disk. The job control is not a resident routine. It is loaded from system residence between each job. When a program is loaded for execution it overlays the job control program. The core storage picture looks as follows:



The nucleus is always resident. The remaining storage is used alternately by job control and each program which represents a job.

Before we get into more detail concerning the scheduling function some terminology and certain concepts must be defined. In non-operating system environments most programs are assembled and then executed. The assembler assigns absolute storage locations and the program always executes from these locations. The procedure in an operating system environment is depicted in Fig. 11-1.

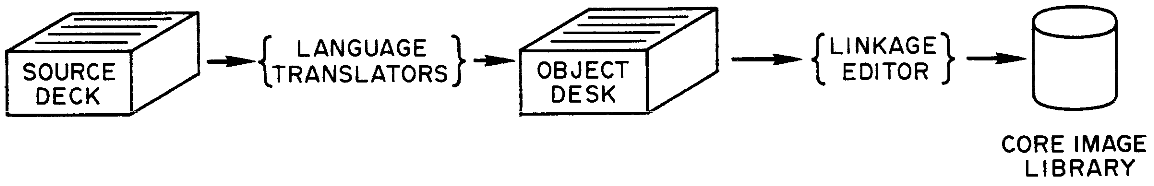


Fig. 11-1 Program Flow in an Operating System

The source deck is assembled or compiled in the usual manner. However, the resultant object deck is in relocatable form. That is, the program does not have absolute core address assigned to it. Another step has been added to the process of obtaining an executeable program. This step is called link editing and requires the services of the Linkage Editor program. Since all decks which come out of the language translators are relocatable, the Linkage Editor is used to assign absolute addresses to the programs and store them on the disk in core image form.

With this intermediate step of the linkage editor, several separately compiled or assembled routines can be combined together and executed as one program. Thus subroutines can be written (even in a different language than the main program), tested independently and then joined together by the linkage editor and executed as though they had been written as one program. The modular approach to programming allows the programming of different parts of an application to be written and tested in parallel thereby shortening the total time it takes to get the application running. Also, the maintenance of the entire program is easier since only the routine which is in error need be recompiled or reassembled instead of the whole program.

Many programs do not need special data sets when they are executing. They may use some tapes or disk during execution but the data is temporary and not needed once the program is finished. Programs in this category can be run under an operating system with the least operator intervention. The system has standard units set up from which the data is read, printed or punched; and scratch tapes and/or disk areas which are used for temporary storage. Therefore, these programs don't require the operator to mount or dismount tapes and disks and thus usage of the CPU is more productive.

As mentioned previously the scheduler is informed of what functions are to be performed by control cards. The control cards all have the same general format. All control cards are identified by two slashes (//) in columns 1 and 2 followed immediately by one or more blanks. Next is the operation which this control card performs. Following the operation but separated from it by at least one blank are the operands that belong with this operation. The individual operands are separated from each other by commas. This format is illustrated in the discussion of the particular control cards which follows. The first control card the scheduler usually encounters is the job card.

```
// JOB     program name,  program name
```

The job card defines the start of a new set of programs to perform a particular function. The scheduler reads all the control cards between a job card and the next job card. When it has sufficient information concerning this job it has the program whose name is specified as the first operand loaded. For example

```
// JOB     PAYROLL
```

would tell job control program that the payroll program should be loaded and executed.

In some cases more than one program may be necessary to accomplish a function. For example, suppose it was necessary to assemble a program and then execute it immediately. This is usually called "assemble and go" since the assembly and execution of the program appear as one operation to the user.

The job card for such an operation would be

```
// JOB        ASSEMBLER,  PAYROLL
```

This card tells job control that a source deck is to be assembled and the output of the assembler is to be link edited and executed.

Following the job card are the Assign cards which have the following format:

```
// ASSGN        SYSxxx, X'cuu', dd
```

The Assign card is used to link the symbolic name used in the program named in the job card to a physical input/output device. In Chap. 10 there was a symbolic unit name in the DTFXX's and CCB's. The ASSGN card links that symbolic name (SYSxxx) to a physical device whose address is cuu

c = 0 multiplex channel
    1 selector channel 1
    2 selector channel 2

uu = 00-FF in hexidecimal which is the units address on the channel.

dd = is a two digit code is to indicate the type of device (see Fig. 11-2)

xxx = is a three digit integer from 0 to 254

There should be one Assign card for each symbolic input/output device name used by the program named in the job card. If two programs are named in the job card the same symbolic names in each must use the same device.

An example of assign cards follows

```
// JOB        PAYROLL,
// ASSGN      SYSIPT, X'004', R1
// ASSGN      SYSOPT, X'007', L1
// ASSGN      SYS001, X'101', T2
// ASSGN      SYS002, X'201', T2
```

Anytime PAYROLL references SYSIPT it will be reading from a 2540 card reader on the multiplexor channel at device address 04. Similarly SYSOPT is assigned to a 1403 printer on the MPX channel at 07, SYS001 is assigned to a 2400 tape on the selector channel 1 at 01, and SYS002 is assigned to a 2400 tape on the selector channel 2 at 01. The symbolic assignment gives the programmer flexibility in the device assignment since actual devices are not assigned until the program is about to be run. This flexibility in I/O device assignment can be used in several ways to an installation's advantage. Since the actual address of the I/O units are not written into the program the program is I/O device independent. This means that if a program needs three tapes, it can use any three tapes which are available. They do not have to be the same three units that were used the last time the program was run. Therefore, if one tape drive is removed for maintenance, the configuration is changed, or the program is to be run on another 360 with a different configuration — as long as there are three tapes — only the ASSGN cards must be changed, the program remains unaltered. This gives the program greater useability and the installation more flexibility.

The card that is always the last control card is the Execute card

```
// EXEC        LOADER,        R
```

dd	DEVICE TYPE
C1	1052 Printer-Keyboard
D1	2311 Disk Storage Drive
L1	1403 or 1404 Printer
L2	1443 or 1445 Printer
P1	2540 Card Punch
P2	1442 Card Punch
P3	2520 Card Punch
R0	2671 Paper Tape Reader
R1	2540 Card Reader
R2	2540 Punch Teed Read
R3	1442 Card Reader/Punch
R4	2501 Card Reader
R5	2520 Card Reader/Punch
RR	1285 Optical Reader
T1	2400 Tape Drive 7 track
T2	2400 Tape Drive 9 track

Fig. 11-2 Device Type Codes

The execute card informs job control of two items. First, that this is the last control card for this job and secondly, the source where the program that is named in the first operand of the preceding job card is to be loaded from. If both operands in the EXEC card are left blank the program is stored in the core image library on disk and is to be loaded directly into core from there. If only LOADER is present, then the program is to be read from the device assigned to SYSIPT, link edited into the core image library and then loaded into storage for execution. In this case the program is in object module (relocatable) form on the device assigned to SYSIPT. If both operands are present then the linkage editor is to read the entire program from the relocatable library (object module form) on the disk and edit it into the core image library and then the system will load it into storage for execution.

There are additional control cards which are required but are not shown here because they do not have counterparts in other operating systems.

As mentioned previously in this chapter the aim of an operating system is to use the computing system more efficiently. One way to accomplish this is by reducing manual intervention, by replacing as many as possible of the functions performed by an operator on the job stream. This enabled the system to process one job after another with a minimum amount of operator intervention. Let's explore this concept using 8K Basic Operating System and the control cards just discussed.

Worked Example

11-1 The system is required to assemble two separate programs, execute a program in the core image library, assemble and execute a program, execute a program which is in the job stream in relocatable form and finally, to execute a program that is in relocatable form in the relocatable library.

The job stream, with control cards, is:

```
1      //     JOB ASSEMBLER
       //     ASSGN
                .
                .
2               .
       //     ASSGN
3      //     EXEC

4             SOURCE PROGRAM 1

5      //     JOB ASSEMBLER
       //     ASSGN
                .
                .
6               .
       //     ASSGN
```

```
7      //    EXEC

8      //    SOURCE PROGRAM 2

9      //    JOB PROGRAM 3
       //    ASSGN
               .
               .
10             .
       //    ASSGN
11     //    EXEC

12           DATA CARDS FOR PROGRAM 3

13     /*
14     //    JOB ASSEMBLER   PROGRAM 4
       //    ASSGN
               .
               .
15             .
       //    ASSGN
16     //    EXEC

17           SOURCE CARDS FOR PROGRAM 4

18           DATA FOR PROGRAM 4

19     /*
20     //    JOB PROGRAM 5
       //    ASSGN
               .
               .
21             .
       //    ASSGN
22     //    EXECUTE LOADER

23           OBJECT DECK FOR PROGRAM 5

24     //    JOB PROGRAM 6
       //    ASSGN
               .
               .
25             .
       //    ASSGN
26     //    EXEC LOADER, R

             data for PROGRAM

27     /*
28     //    PAUSE
```

Card number 1 is the job card which tells job control that the assembler is to be executed. The ASSGN cards (2) link the symbolic names the assembler uses to the physical I/O devices for this job. The execute card (3) tells job control that this is the end of the control cards for this job and that the assembler is to be loaded from the core image library on disk. Following the execute card are the source cards (4) which make up the program to be assembled. These source cards and the other non-control cards that appear later on could have been on any device assigned to SYSIPT. The control cards are on the device assigned to SYSRDR. In the example shown both SYSIPT and SYSRDR are assigned to the same device. Now the operating system will be able to process all the jobs mentioned in a continuous stream without manual intervention. Any temporary external storage and libraries can be assigned to disk storage which is on-line and doesn't need manual intervention to make it available.

Let's continue with the job stream. When the first assembly is completed, job control is automatically loaded to process the next set of control cards (5-7). These are similar to the first job and another assembly takes place. The next job which is automatically scheduled is PROGRAM 3. This program is also loaded from the core image library for execution. During its execution it will read the data cards (12) from the job stream. When the slash asterisk card (13) is read it tells PROGRAM 3 that all the data cards have been read. When PROGRAM 3 finishes, job control is loaded and continues to read the job stream. The next job card (14) indicates the assembler is to assemble the source cards (17) for PROGRAM 4, store the resultant object module temporarily in the relocatable library and call the linkage editor to process it, which leaves it in the core image library. Then the system loads the core image module of PROGRAM 4 for execution. During its execution it will read the data cards (18) in the input stream. Thus the operating system not only can pass control from one job to the next automatically, but it gives the programmer the capability to assemble and execute his program in one operation without manual intervention.

The next set of control cards (20-22) tell job control to call linkage editor to have the object deck for PROG5(23) edited into the core image library and then loaded into storage for execution. The last set of control cards cause the system to have the object module for PROG6 in the relocatable library to be link edited in to the core image library, loaded and executed using the data cards (26) in the input stream.

The system will pause after PROG6 finishes waiting for more control cards to be stacked on the device assigned to SYSRDR. Remember that the stack of six jobs, each defined by an appropriate set of control cards were executed one after another automatically without manual intervention.

There are many jobs which require large amounts of data in and out of the system. For jobs of this type the operator is usually required to mount tapes and disk packs before the job can begin execution. When this is necessary the system waits for him to perform this task and checks the labels on the disks and tapes, if present, to verify that the proper ones are mounted.

The system will process one job after another once it is in operation. To get the system started, the operator goes through an Initial Program Load (IPL) procedure which by a hardware generated input operation loads a program which initializes the system and loads the job control for the first time. After that, job control is brought into storage at the termination of each job to get the next job going.

11.3 Supervisor

Unlike job control, which was discussed in the preceding section, the supervisor program remains resident in the central processing unit storage as long as the operating system is being used. The programs which are resident constitute the nucleus. The nucleus resides in lower address storage. All the programs which are part of the supervisor do not remain in storage at all times. The programs which are used less frequently are stored on external storage called the systems residence device. When one of these programs is needed, it is loaded into an area within the nucleus. This area is used by all non-resident supervisory programs and is called the transient area. By using the concept of a transient area the amount of storage needed at all times by the nucleus can be reduced.

An example of transient routines would be the open and close routines which were discussed in Chap. 10 when describing the OPEN and CLOSE I/O macros. Whenever a file is opened for processing the open routines are loaded into the transient area in the nucleus. They perform their function and then the transient area is free to be used for other non-residence supervisor programs.

All supervisors contain at a minimum an Input/Output Supervisor which performs the functions described in Chaps. 9 and 10. As the nucleus size grows, additional services for the problem program are provided. In general as the size of the nucleus grows the more services it provides the user. These services usually satisfy a general need. Therefore, the user does not have to provide these functions in his own program and they ease the programming load. However, since these functions must serve a large number of users, they are generalized and usually take more storage than if the function were coded for a specific application. Here the trade-offs are between storage requirements and programming ease and time.

In the 8K Basic Operating System there are certain services available to the user when programming his application in addition to the I/O macros discussed in Chap. 10. These services are made available through macros which are coded in the application program and are assembled into supervisor call (SVC) instructions. When the SVC instruction is executed the supervisor will perform the appropriate service by using a resident routine or calling a non-resident one. Some of the specific services provided by the 8K system are described below by a discussion of each of the macros available.

The fetch macro has the following format.

```
FETCH        name
```

When this macro is used in the problem program, control is given to the systems loader which is resident in the nucleus to load either the program or the phase of a program whose six character name appears in the operand field from the core image library. This enables the programmer to have routines loaded on an as needed basis rather than have all the routines in storage at the same time. Routines are entered into the core image library by use of the Linkage Editor. Further details on the use of the Linkage Editor will be covered in the next section. Since programs in the core image library are not relocated when loaded, the origin of the program, that is, the location of the first storage location used, must be decided before it is stored in the core image library.

The message macro has the following format

```
MSG          code, REPLY
```

The macro allows the problem program to communicate with the operator. When this macro is used the one to four character code is displayed on the operator's console. If the second operand, REPLY, is used then the operator may type in a one character code to communicate back to the program that used the message macro. The reply if one was requested is stored in location name + 7.

For example, suppose the program wanted to tell the operator that the cards that had been read were incorrect. The program may use the message macro

```
CRDERR MSG CDER
```

The message CDER would appear on his console. However, in addition to informing him that there was an error, the programmer may want to know what the operator wants to do about it. Therefore, the message macro may be coded as

```
CRDERR MSG CDER,REPLY
```

Now the message on the console becomes CDERA. The A informs the operator that a one character reply is expected. The one character code he now types in is stored at CRDERR + 7 and the program may interrogate it to determine what action can be taken. This capability enables a program to be more flexible in handling unusual conditions and operator oriented functions.

There are certain conditions which arise in which it is more appropriate for a program to handle by itself instead of accepting the standard system action. This capability is given by the Set Exit macro which has the following format:

```
STXIT      n, pc-name, it-name, oc-name
```

Here, n is the number of a general purpose register the supervisor routine can use to refer to a communication region. The other three operands are optional and one or more can be included: pc-name is the symbolic name of the user's routine that he wants entered when a program check interrupt occurs; it-name is the symbolic name of the user's routine that gains control when an interval timer interrupt occurs; oc-name is the symbolic name of the user's routine that is given control when the operator wants to initiate communication with the system.

If an operand is omitted this must be indicated by the use of successive comas. For example, a program wants to gain control at location OPRTOR when the operator initiates communication and at TIMER when the interval timer interrupts. Register 6 can be used by the supervisor. The macro would look like

```
STXIT      6,, TIMER, OPRTOR
```

The program can issue the macro several times in the program. The last one issued is the one that's in effect at that time. The last two operands can be reset to the system standard by issuing the macro with the word CLOSE in those operands instead of the symbolic name.

When a user writes either an interval timer or operator communication routine he needs a way to return to his main program. This is provided by the exit macro which has the following format

```
EXIT      TR
EXIT      CR
```

Here, TR indicates a return from a timer routine and CR indicates a return from an operator routine.

There is a need in some programs to communicate with other parts of the program or another phase of the program which is not in storage at the time. In order to have this facility it is necessary to have an area of storage reserved for this purpose whose location is known to all programs. Most operating systems have such an area in the nucleus and therefore it is always available. This area is called a communications region or vector. It usually contains information concerning different parts of the supervisor and space for problem programs to leave a small amount of data to be referenced later.

Since the location of the communication region should be available to all programs, there is a macro which will give the requesting program the starting address of the communication region. The format of the macro is

```
COMRG
```

When this macro is issued the address of the communication region is returned in register 1. By using this address plus the appropriate displacement the program can obtain certain information which is stored in the communications region.

The COMRG macro enables the program to obtain data from the communications vector. There is also a macro which enables the program to store data in the region. This macro has the following format

```
MVCOM        byte, n, location
```

the MVCOM macro allows the user to modify bytes 12 to 23 in the communication region. The operand byte specifies the starting location in the communication region, from the 12th to the 24th byte, n is the number of bytes to be moved and location specifies the location in the user's program where the n bytes will be moved from.

Worked Example

11-2 To illustrate the use of these macros let's take the following example. There are two programs which are not in storage at the same time. PROG1 will build a table of data which PROG2 will use in its processing. PROG1 must tell PROG2 where the table it has created is located in storage. Consider the following coding:

```
         PROG1                               PROG2

BEGIN    BALR     2,0              BEGIN BALR      2,0
         USING    *,2                    USING     *,2
           .                               .
           .                               .
           .                               .
         MVCOM    12,4,TABLE1            COMRG
           .                             L         3,12(0,1)
           .                               .
           .                               .
           .                               .
         FETCH    PROG2                  END       BEGIN
TABLE1   DC       A(TABLE)
         END      BEGIN
```

PROG1 moves the 4 bytes at location TABLE1 which is the address of TABLE into the communication region starting at the twelfth byte of the region. Later PROG1 causes PROG2 to be loaded by using the FETCH macro. PROG2 uses the COMRG macro to get the starting address of the communications region into register 1. Then by using a displacement of it, 12 has the address of TABLE in register 3.

There are times when a program encounters an error condition which prevents it from proceeding. However before the program terminates it is useful to produce a picture of how core storage looks so that the programmer can determine what caused the error or unusual condition. To do this the program can issue a dump macro which has the following format

```
DUMP
```

This macro prints a picture of storage as it exists at that moment and then terminates the program.

A program is terminated when it tells the supervisor that it is finished. This can be done by the end of job macro with the format

```
EOJ
```

This macro informs the supervisor that the job is finished and to load job control to schedule the next job.

11.4 Service Programs

In addition to the control programs and language translators, operating systems usually provide a class of processing programs call service programs. These programs help the user operate and maintain the system. The Linkage Editor which was mentioned previously is one of these programs. Others include the utility programs and the Sort/Merge program. The latter provide functions which a large number of users would otherwise write programs for if a generalized program were not provided. The service programs run under the control programs in much the same manner that application programs do.

The Linkage Editor, discussed briefly in Section 11-2, is used to take the output of the language translators which is in relocatable format and edit it into the core image library. With the addition of this intermediate step there are some additional capabilities available to the user, particularly in the way he organizes, writes, tests and maintains his application programs.

Since the output of the language translators is a relocatable deck or object module, the Linkage Editor can combine separately compiled or assembled object modules into one executable program called a phase. This means that an application can be broken down into its basic parts and each part coded as a subroutine. The subroutines can be coded and tested separately. As a result, more than one person can work on coding and testing an application in a fairly independent manner once the ground rules are decided upon. Then, when each of the routines have been completed they can be combined, tested as a whole and run as though it had been written as one routine. The overlap obtained by several people working simultaneously on an application enables the job to be done in a shorter period of time. In addition maintenance time is reduced since when an error is found only the offending routine need be recompiled or reassembled and then link edited with the other routines.

To explore the functions of the Linkage Editor let's examine some examples of combining separate object modules into one executable program.

Worked Example

11-3 Our first example shows separate object modules link edited into a single executable module or phase.

The input to the Linkage Editor looks as follows

INPUT		OUTPUT
PHASE	P1, S, 1	PHASE P1 WITH STARTING POINT AT LOCATION A
OBJECT MODULE A		
OBJECT MODULE B		
OBJECT MODULE		
ENTRY	A	

The PHASE control card gives the name of the resultant phase, in this case P1, and also where P1 is to be loaded. S, 1 indicates that P1 is to be loaded at the first available location after the nucleus. The output from Linkage Editor is one module referred to as P1 in the core image library. The ENTRY card indicates the end of the object modules to be included in this phase and the entry point or starting address for execution for this phase.

Worked Example

11-4 In some cases the resultant phase is composed of object modules in the input stream and some from the relocatable library. The following example combines object modules A and C from the job stream with module B from the relocatable library into phase P1.

PHASE P1, S, 1

INCLUDE B

OBJECT MODULE A

OBJECT MODULE C

ENTRY A

The include card tells Linkage Editor to find object module B in the relocatable library and incorporate it in this edit at the point where the include card appears. The order of the programs in the resultant phase P1 is different.

However, the staring location is still A in the module in which that label appears.

<u>Worked Example</u>

11-5 The Linkage Editor can also process several object modules into several phases as the example below depicts.

```
PHASE              P1, S

OBJECT MODULE
      A

OBJECT MODULE
      B

ENTRY        A

PHASE              P2, L, , END B

OBJECT MODULE
      C

ENTRY        C

PHASE              P3, L, , ENDB

INCLUDE      D

ENTRY        D1
```

In the example the program needs object modules A, B, C, D but does not have enough storage to hold the four of them at the same time: as a result, the program is divided into three phases. Phase P3 is made up of module D called from the relocateable library. Phase P1 would be loaded and remain in storage. When P2 is needed it would be loaded from the core image library at the time P1 issues a fetch macro. P2 would be loaded at the location which is equivalent to the label ENDB which must be defined in P1. Assume ENDB defines the next location following the end of P1. Since P3 has the same load point, phases P2 and P3 would not be in storage at the same time. Only the one needed at that particular time would be present. P1 would determine which one is requested by the name used in the fetch macro. The L in the phase cards for P2 and P2 informs Linkage Editor that the starting location to be assigned for loading is represented by the following symbol ENDB which is defined in a preceding phase.

As we have shown in the preceding examples, the linkage editor function in an operating system gives the programmer flexibility in structuring his program. Another service program which is a necessary part of an operating system is the librarian program. This is usually one or more routines that enable the user to maintain the system libraries. In the 8K BOS system there are three basic libraries:

Core Image Library

Relocatable Library

Macro Library

The librarian routines enable the user to selectively print or punch the contents of these libraries. In addition, routines are provided to add, delete and update members of the libraries. These routines play a major part in keeping the whole system running efficiently.

11.5 System Generation

Since operating systems are composed of many programs which must satisfy a wide variety of users and equipment, they are generalized programs. This generalization requires core storage and processing time. However, each user is interested in his own requirements and not particularly in those of all other users of the system.

The first operating systems were designed for a small number of users and as new systems and input/output devices became available the size of a standard system grew to include all the desired functions and configurations and as a result, the size of these programs approached a size that was undesirably large in some instances.

To relieve each user from giving up storage for functions someone else needed, operating systems were designed to be adapted to a particular users needs. At first, this adaptation was in the input/output configuration area only. In other words, if an installation didn't have a particular type of device the code in the system for that device was dropped out. As operating systems provided more functions in other areas, these were provided in a selectable manner.

The process of taking a standard operating system and adapting it to a particular installation's needs is known as system generation. It is usually accomplished by using a starter system which is an operating system with an Assembler, Linkage Editor, and library routines. The installation then specifies its requirements via a set of parameters which are given to the starter system. Guided by these parameters the starter system assembles certain modules and link edits these with other object modules. The librarian then arranges the resultant modules into the appropriate libraries and the installations own version of the operating system is formed.

The technique of system generation enables one general system to be adapted to the needs of many different installations.

11.6 Summary

We have briefly discussed the evolvement of operating systems. The functions and facilities of the system such as scheduling, supervision and program structure were reviewed for an elementary system, 8K BOS. In the next chapter some of the more advanced concepts in operating systems will be surveyed.

As the computing system increases in speed and sophistication so will the programming systems that are employed to utilize this increased computing power. The evolution of operating systems was a major step in that direction.

Chapter 12

DISK OPERATING SYSTEM

12.1 Introduction

In Chapter 11 the concept of an operating system was introduced using the 8K Basic Operating System for illustration purposes. The 360 has several levels and types of operating systems which will control it. As the computing system increases in size and complexity, higher levels of operating system may be used to control its operation. As the level of the operating system increases, more function are provided and more resident core storage is required to operate it. The Basic Operating System is the first level of operating system provided. The Disk Operating System is an intermediate system and contains additional facilities not in the first level system.

12.2 Programming Languages

The Basic Operating system enabled the programmer to program in either assembly language or the Report Program Generator language. The Report Program Generator (RPG) language, unlike assembly language, is not tied to the instructions of the computing system. It is designed for use by a person who does not have to know the machine instructions of the computing system. It is primarily intended for writing programs which generate reports from input data. The program is written using statements from the RPG language and the Report Program Generator processor translates them into a machine language program. Since the source language does not resemble the machine instructions it is called a 'higher level language'. The higher level language also allows greater flexibility in the use of the program. Since the source program is not machine instruction dependent, it can be run on any computer that has a translator for that source language.

The Disk Operating System, in addition to assembly language and the Report Program Generator language, offers the user COBOL, FORTRAN, and PL/I. This gives the user a wider variety of languages from which to choose when programming an application. Each programmer or group which uses the computing system can use the language or languages which best suit the application yet the same operating system can run all the programs even though they were all written in different languages.

COBOL is a higher level language which is used in many commercial applications. The source statements for a program written in COBOL look very much like sentences in the English language. The COBOL processor reads the source statements and translates them into machine instructions which can be executed by the computing system. The COBOL language was developed jointly by the U. S. government and the computer manufactures in an effort to ease the conversion of applications from one computer to another and have a source language which is self documenting.

FORTRAN is also a higher level language and one of the oldest in use. The source statements of the FORTRAN language closely resemble mathematical notation. For this reason it has been widely used for scientific and engineering computing.

PL/I is another higher level language. It was designed to give the programmer a general purpose language which combines features of both FORTRAN and COBOL and adds new features which are in neither COBOL or FORTRAN. These new features enable the programmer to use a higher level language to program certain applications which previously required assembly language.

Using the subroutine technique discussed in Chap. 8, and the linkage editor concept covered in Chap. 11, parts of an application can be written in the language most appropriate and link edited into one executable program. Under the Disk Operating System the choice of languages is greater which gives the programmer more flexibility and the use of higher level languages makes programming easier.

12.3 Job Steps

In Chap. 11, the concept of a job and a job step were discussed but in the 8K Basic Operating System there is no way to define a job step formally. Whether a job required the use of several programs in a consecutive order or not was determined from the number of names in the job card. The example of a three step job of assemble-load-go used a job card with two names.

```
// JOB ASSEMBLER, program name
```

The presence of the program name as the second operand informed the scheduler that the linkage editor had to be called after the assembler before the program whose name appeared as the second operand could be called for execution. The Basic Operating System provided a form of job step capability but for a specific function (compile a go) and limited to not more than three steps.

The Disk Operating System provides a more general job step capability by expanded use of the EXEC control card. As in the basic system all assignments of input/output devices by the use of ASSG must follow the job card. These I/O assignments must be sufficient for all steps which will be executed as part of this job. However, there can be as many // EXEC cards as desired for each job and they can name any program needed. For example, suppose a job consisted of reading data and selecting certain records for output. These records are first sorted and from the sorted output a report will be written. The computation could be organized into a three step job. Assume that the first program which selects the data to be sorted has been previously tested and is stored in the core image library with the name of DATASEL. The report writing program was also previously tested and stored in the core image library with the name of REPORTB. The sorting of the data will be done by the Sort/Merge program which is one of the processors provided with the operating system. The job stream is shown in Fig. 12-1. The following operations will be performed:

Job Stream

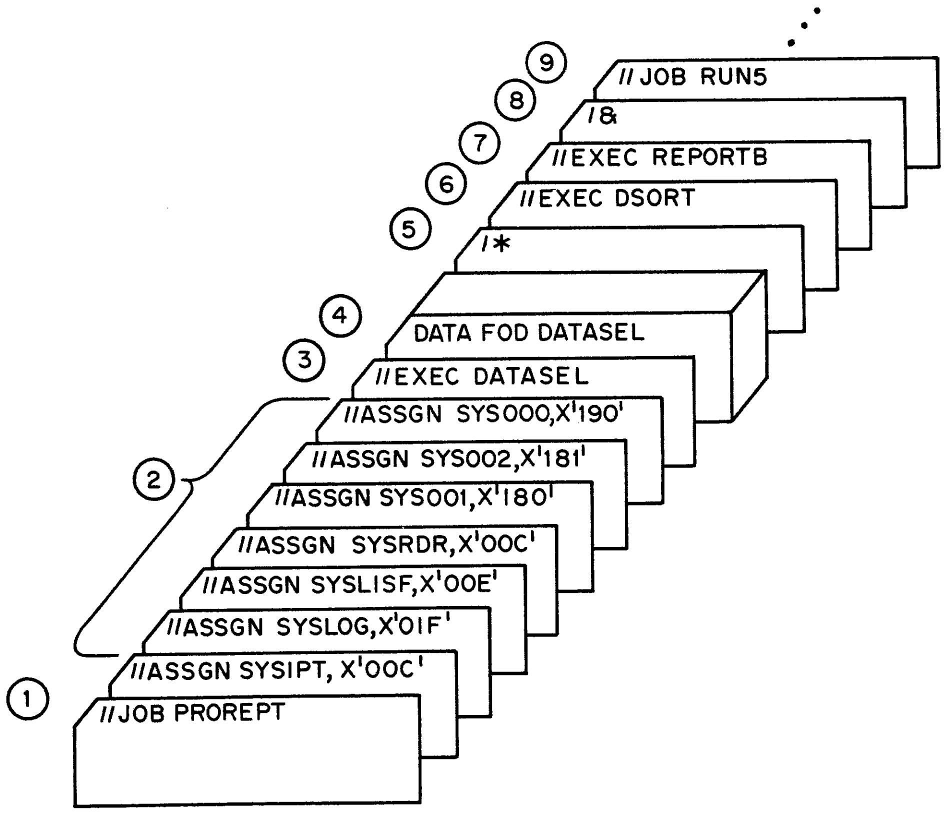


Fig. 12-1

1. The // JOB card defines the start of a new job with the name PROREPT. PROREPT is the name of the job and not the name of the program to be called as in the basic system.

2. The // ASSGN cards relate the symbolic names used in the program to the physical I/O devices. X'CUU' is the address of the physical input/output unit which is assigned to the symbolic name. SYSIPT is assigned to the card reader, the source of input for DATASEL. SYSLOG is assigned to the operator's console, the I/O device where operator messages will be printed. SYSLIST is assigned to the printer where error messages for the sort and the report produced by REPORTB will be printed. SVSRDR is also assigned to the card reader where the system reads the control cards. SYS001 is assigned to a tape where DATASEL will write its output data and DSORT will read that data as input to the sort. SYS002 is assigned to a tape also which is the output of sort and the input to REPORTB. SYS000 is assigned to a disk which will be used as a work area by the sort.

3. The // EXEC card tells the scheduler to call the program DATASEL for the first job step.

4. This is the data that DATASEL will read as input. It is in the input stream because SYSIPT and SYSRDR are assigned to the same device.

5. The /* indicates to DATASEL the end of data in the input stream. This card prevents DATASEL from reading as input data the control cards which follow and thereby destroying the next job.

6. // EXEC DSORT tells the scheduler to call the disk sort as the program to be executed as the second job-step.

7. // EXEC REPORTB informs the scheduler that the third job-step program REPORTB, is to be executed.

8. The /& card indicates the end of the job, that is, no more job-steps follow, and causes all I/O device assignment to be reset so that the I/O devices can be assigned for the next job.

9. The // JOB RUN5 indicates the start of the next job called RUN5.

Suppose now that both DATASEL and REPORTB had been written in COBOL and Report Program Generator, respectively and had to be compiled before they could be executed. The job stream would now look as shown in Fig. 12-2.

This job is a seven step job. The first four job steps perform a COBOL compilation of DATASEL and link edit it into the core image library follow by a Report Program Generator compilation of REPORTB and link edit it into the core image library. The next three steps are the same as in

Fig. 12-1. The input/output device assignments include one more tape unit assigned to SYS003. COBOL uses three work tapes which are assigned to

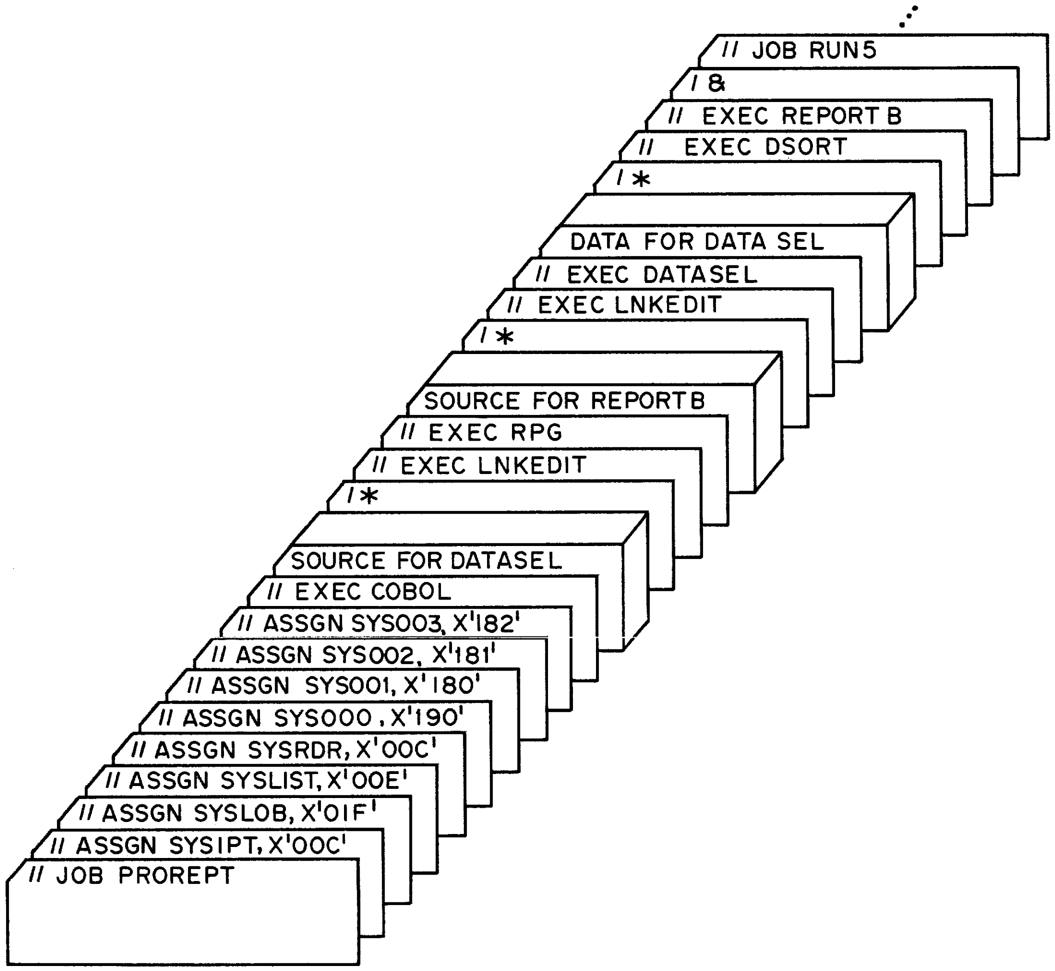


Fig. 12-2 Job Stream Illustration-2

SYS001, SYS002, and SYS003. It also outputs the compiled object deck to SYS000 which is assigned to disk. Linkage Editor reads its input, the object module, from SYS000 and use SYS001 as a work unit. The output of linkage editor goes into the core image library which doesn't need an assign card. The Report Program Generator use SYS001 as a work file and also puts its object module on SYS000. The I/O device usage for the last three job-steps is the same as the example shown in Fig. 12-1.

The more flexible job-step capability provided in the Disk Operating System allows the programmer to combine any number of programs that were previously separate runs into one job and have it run as though it were one program. This job-step capability also enables the application to be written in a modular manner and make use of existing programs.

12.4 Multiprogramming

In Chap. 9 the input/output channel was discussed. Its purpose is overlapping the processing and data transfer times in a program so that the central processing unit would not be idle while data transmission is taking place. The input/output channel reduced the total time that a program took to execute by overlapping the central processing unit time and input/output time when possible. In other words, the total execution time of a program before the use of the channel was the total of input/output time and processing time (Fig. 12-3a). After the use of the input/output channel the total program execution time is int sum of the total I/O time and processing time minus the overlapped time (Fig. 12-3b). There are certain cases where either the processing time or the I/O time is the most dominant factor in computing the total execution time. These programs are called process bound or I/O bound programs respectively. For instance, a program may have so much input/output time relative to processing time that if the overlap between I/O time and processing time were a hundred percent the total job time would be the input/output time and the processing unit would be idle for the time difference between I/O time and processing time. The converse is also true where processing time is much greater than I/O time. (Fig. 12-3a and Fig. 12-3d)

From Fig. 12-3 it can be seen that the most efficient use of the central processing unit is for case d), the processes bound job, and the worst case is a) where the CPU is idle whenever an I/O operation is required. In general, there are few programs which balance their I/O time and CPU time so that neither component is idle for a significant amount of time. Let

C_{idle} = idle CPU time

I_{idle} = idle I/O time

O = Overlapped time

P = total program time

Then

$$P = C_{idle} + I_{idle} + O$$

or

$$1 = \frac{C_{idle}}{P} + \frac{I_{idle}}{P} + \frac{O}{P}$$

For many programs, the average percentage of overlap is about 20 to 30% which means that the combination of CPU and I/O idle time is 70% of the total program execution time. Even in cases where the program is I/O bound or process bound one of the resources is idle part of the total program

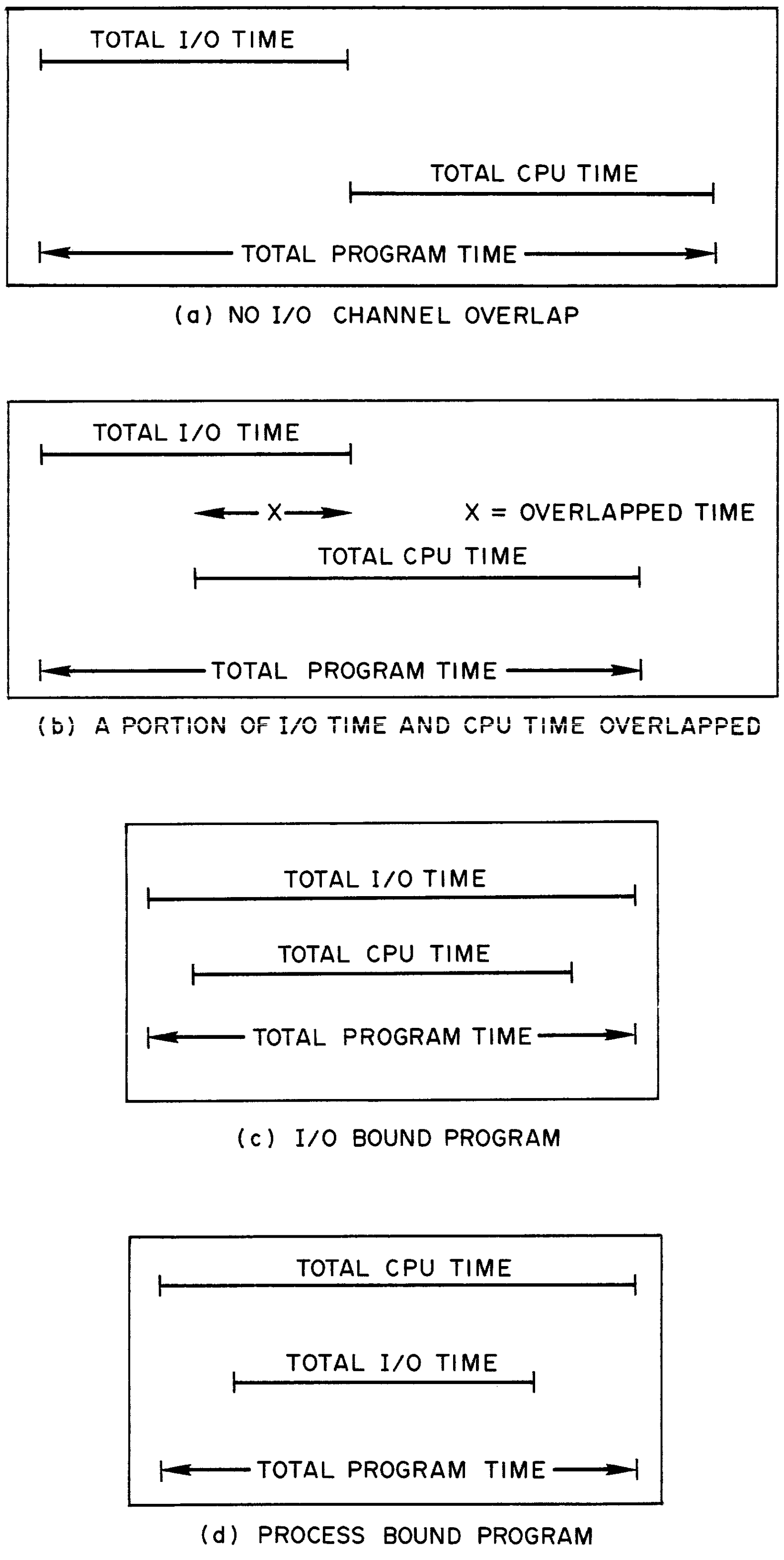


Fig. 12-3 Program Run Times

time. If this idle time could be used to process more programs, then the computing system whose cost is fixed could be better utilized. The result is increased thruput for approximately the same total cost which reduces the cost of getting a single job done.

Many computing systems have a number of input/output devices which are rarely used at the same time except by a few applications. The same is true for the storage of the central processing unit. It is generally a few major applications which dictate the total I/O device and storage requirements of a computing system. However, when other applications are run they tie up all these resources even though they do not use them. This is because a program is run from start to finish before the next program is run. If the device and storage that are not being used for the program being run could be brought into service for other programs, the total system could be used more efficiently.

There are some applications which do not utilize the system extensively but must be capable of supplying results on demand. Their demands for service are usually unpredictable and therefore cannot be scheduled. This usually requires the program to be run on the system on a continuous basis over a period of time. Other work cannot be done on that system because that program must supply its service on demand. This usually means either two systems, one for the continuous application and one for the other programs or running the other programs during the hours the continuous application is not needed. If the special program and other programs could utilize the same system concurrently, cost could be reduced and service to other users improved.

Because of the idle time of some of the computing systems components when it is running many applications, the problem of only a few applications using all the systems resources simultaneously and the need of a single system to perform multiple functions, the concept of multiprogramming evolved. The general concept is simple. Give a computing system under control of an operating system, multiprogramming is the ability to run multiple programs in an interleaved manner. The principle takes advantage of the fact that most programs have processing interspersed with input/output operations. If the program cannot compute during the data transfer, then the processing unit is idle. The idle time can be used by another program while the first one is waiting. In this way control of the central processing unit is given to another program whenever the first program requests an operation which would cause the CPU to be idle. The switching of control of the CPU between programs in this manner can happen so fast that it appears to the observer that the system is running both programs simultaneously. Remember this is not so. Each program is executed in a sequential manner but it may not go from start to finish before control of the CPU is given temporarily to another program.

To illustrate the multiprogramming concept, let's take an example where three individual programs have been loaded into storage. The fact that a program is in storage and ready to be executed is indicated by adding an entry in a list called a queue. The queue is a list of programs which are ready to execute but need control of the central processing unit. In our example, the queue has three entries, one for each of the three programs, A, B, C. The control program gives control of the CPU to the first program in the list, A, by a load PSW (program status word) instruction. The PSW has the starting address of program A, and therefore, program A has control of the CPU. Program A will continue to execute until it can no longer use the CPU for a continuous period of time. This situation can occur if program A issued an unconditional request for an input/output operation which means it does not want to continue until the I/O operation is complete. The CPU would then be idle while the I/O operation took place. The control program recognizes this fact and 1) flags program A's entry in the queue as waiting for I/O, 2) saves A's PSW in its entry in the queue, and 3) loads the PSW for program B because the entry for B is the highest in the queue that is not flagged as waiting. This gives control of the CPU to program B. The status of the three programs at this point is:

A	is in storage, has partially executed and is waiting for an I/O operation to complete;
B	is in storage, is in control of the CPU and is executing;
C	is in storage but has not started execution.

Now while B was executing, assume that an interrupt occurred which signaled the completion of the I/O operation for which A was waiting. The control program gains control because of the interrupt and 1) identifies it as the completion of A's I/O request, 2) removes the wait flag from A's entry in the queue, 3) saves B's PSW in its entry in the queue and 4) loads A's PSW because it is the highest entry in the queue that is not waiting. The status of the three programs at this point is:

A	is in storage and has resumed control of the CPU;
B	is in storage, partially executed and waiting to regain control of the CPU;
C	is in storage and waiting to gain control of the CPU

After a while A may request another input/output operation which cannot be satisfied immediately and its entry in the queue is flagged as waiting. B then is given control because it is the highest entry in the queue which is not

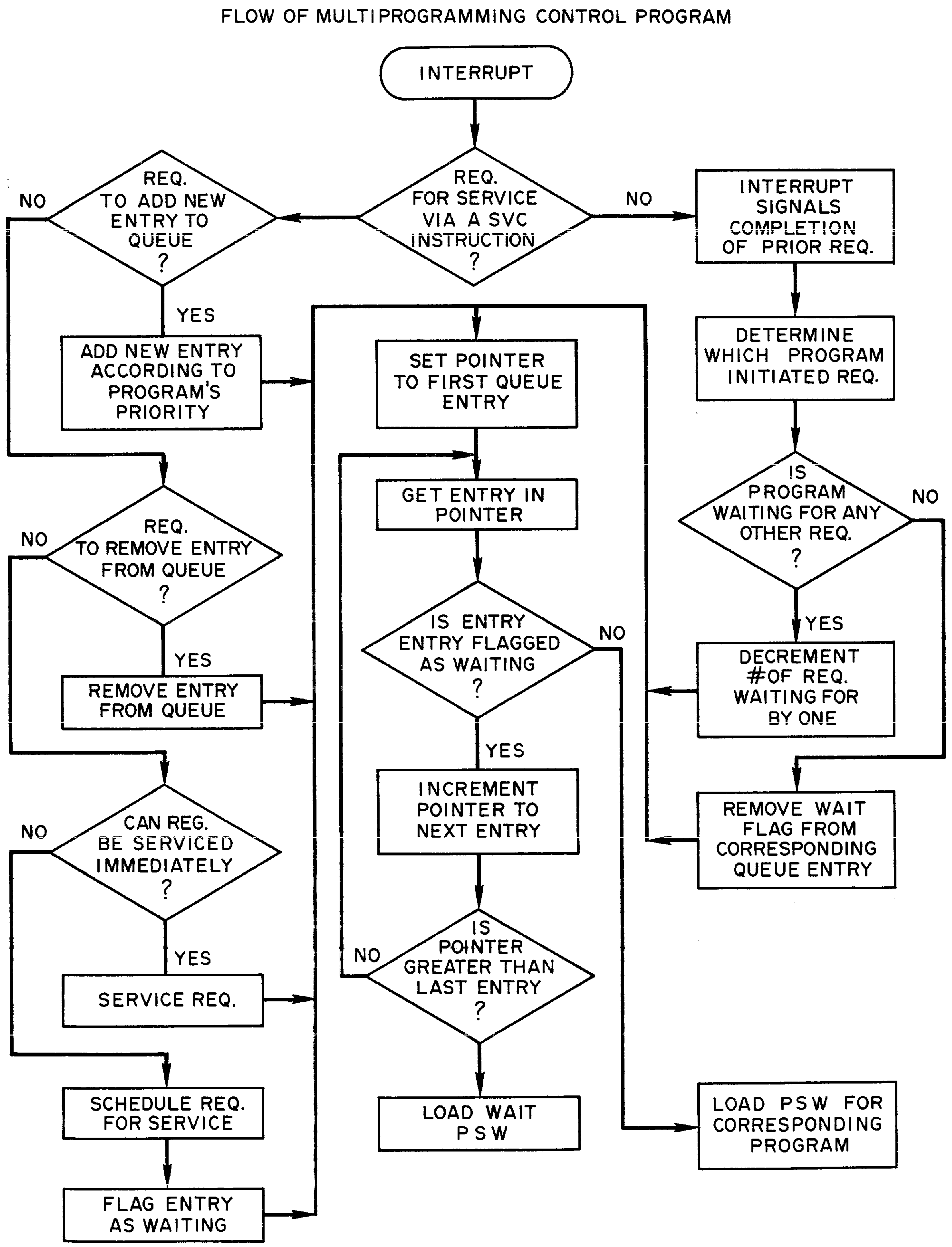

FLOW OF MULTIPROGRAMMING CONTROL PROGRAM
INTERRUPT
REQ. FOR SERVICE VIA A SVC INSTRUCTION ?
NO
INTERRUPT SIGNALS COMPLETION OF PRIOR REQ.
DETERMINE WHICH PROGRAM INITIATED REQ.
IS PROGRAM WAITING FOR ANY OTHER REQ. ?
NO
YES
DECREMENT #OF REQ. WAITING FOR BY ONE
REMOVE WAIT FLAG FROM CORRESPONDING QUEUE ENTRY
REQ. TO ADD NEW ENTRY TO QUEUE ?
NO
YES
ADD NEW ENTRY ACCORDING TO PROGRAM'S PRIORITY
REQ. TO REMOVE ENTRY FROM QUEUE ?
NO
YES
REMOVE ENTRY FROM QUEUE
CAN REQ. BE SERVICED IMMEDIATELY ?
NO
YES
SERVICE REQ.
SCHEDULE REQ. FOR SERVICE
FLAG ENTRY AS WAITING
SET POINTER TO FIRST QUEUE ENTRY
GET ENTRY IN POINTER
IS ENTRY ENTRY FLAGGED AS WAITING ?
NO
YES
INCREMENT POINTER TO NEXT ENTRY
IS POINTER GREATER THAN LAST ENTRY ?
NO
LOAD WAIT PSW
LOAD PSW FOR CORRESPONDING PROGRAM

Fig. 12-4 Multiprogramming Algorithm

flagged as waiting. B executes for a time and then requests a program to be loaded. Since this takes a finite time to be done the CPU can be used by another program for that interval of time. Program A is still waiting for the completion of the input/output operation, B is waiting for a program to be loaded, and so, C is given control of the central processing unit.

It can be seen from this example that multiprogramming a single central processing unit requires an algorithm to be developed which determines which one of a number of individual programs should get use of the CPU when the program which had control relinquishes it for a finite time period. The procedure used in the example and depicted in the flowchart in Fig. 12-4 takes advantage of the fact that most programs, if executed alone, have natural points throughout their run when the CPU is idle. The algorithm uses this idle time in one program to give control of the CPU to another program. If more than one program can use the CPU it gives control to the one whose entry in the queue is nearest the top. Therefore, the order of the entries in the queue affects the ability of the program to get CPU time. The program with the first entry in the queue will get the CPU whenever it can use it, whereas the program whose entry is last in the queue will only get the CPU when all programs whose entries are above its entry are flagged as waiting. The ordering of the queue, if all programs are considered equal, would be first in first out. In other words, the entries would be ordered by time of arrival at the system.

However, in many cases some programs are more important than others. If each program is assigned an 'importance factor' called a priority, then the queue can be ordered by priority with the more important programs towards the top and less important programs at the bottom. If programs have the same priority they are ordered first in first out within the same priority. As each program terminates, its entry is removed from the queue.

Our discussion of multiprogramming above does not treat other algorithms which are used to enable a CPU to work on multiple programs before any particular one is complete. This algorithm was used because it is similar to that used by the Disk Operating System and will be discussed in the context of that system in Sec. 12. 7.

12. 5 Teleprocessing

There is a large category of applications which are grouped under the name teleprocessing. Some of these applications are also called 'real-time'. This area of computing is of great interest and growing very rapidly today. The common thread in all teleprocessing applications is that the computing system receives and transmits data to a terminal device of some type over a communication line. In general, a teleprocessing application must run on the system for long periods of time. Many run for 8, 12, or 24 hours a day. This does not mean that they are executing instructions continuously for that period of time. The purpose of most teleprocessing programs is to be responsive. When a request is made the program must be there to receive that request and service it with a minimum delay. The terminal device which is connected to the computing system through a communication line can be of several types. Many are of the keyboard variety and operate much like a typewriter. They may be a visual display terminal using cathode ray tube, or another computer system. Many applications have been developed using the keyboard type terminal. This terminal allows a person to type information directly into the computer. In a sense, the person at the terminal is on-line or directly connected to the computing system. Some of the applications which utilizes this on-line capability follow are discussed below.

Inquiry: A person at a terminal enters a coded message from the terminal. The computer program receives the data, analyses it, usually retrieves information from a master file, and sends it to the terminal that requested it. Notice that the request was initiated from outside the computing system. The program responds when queried and therefore must be available whenever a request is possible. Since the requests for service are not scheduled by the program, most teleprocessing programs remain resident in storage for the period the service is being provided.

Message Switching: A computer is used to route messages from one terminal to another. Messages can also be routed by electro-mechanical switching systems. However since it is done in a computer by a program, the functions performed can be changed easily. The computer switching system is more flexible. In many cases it is also faster, thus being able to handle more messages. Messages are entered from a terminal with part of the message containing an identification of the destination terminals for this message. The program analyses this portion of the message, usually called the header, and then sends it to the appropriate terminals.

On-Line: In many applications there are master files of information which are updated on a periodic basis. This is usually accomplished by saving transactions which will change information in the master file until there is a sufficient number to justify on update run on the system. This process is called batching. This means that other programs that use the master file only have information which is current as of the last update. If the master file were always available to the update program and the transactions were entered through a terminal as they took place, the update program could process the transaction against the master file as they are received so that the file will always be current. This type of processing is called on line because the information in the master file is always accessable to the computing system and therefore to the person at the terminal.

The types of teleprocessing applications mentioned above often have real time constraints placed upon them. That is, the time that elapses between the point at which the message is entered and the answer is received is fixed or has a fixed lower limit. The interval that is considered reasonable depends on the application. In many applications the requirement is that the person at the terminal should feel that the system is working only for him even though it is servicing many terminals at the same time. That is, it is desirable that he perceive no delay in the computer's response to his inquiry. A satisfactory response time in this case may be of the order of several seconds. In other applications the system may be controlling a physical process via electronic equipment. In these cases the response time is often much more critical, ranging down to the order of thousandths of a second.

There are several ways terminals and the computer interract. These are:

Contention

Polling

Dial up

Contention occurs when the person at the terminal presses a button and an interrupt is caused in the CPU. The program then issues a read I/O instruction to read the data from the terminal. In many cases one communication line is shared by many terminals. This is called a multidrop line. If contention is being used on a multidrop line, the terminal that gets the line keeps it until it finishes and the other terminals wait in much the same fashion as party-line telephone service.

Polling is used in many cases for multidrop lines. Polling allows the computer program to control which terminal will be allowed to send data. If a terminal has something to send a button is depressed which sets a latch. The program sends a signal to a terminal on a line. If the latch has been set a circle Y signal (Y) is returned; if it has not a circle N (N) is returned. If the response is a Y the program the issues a read I/O command for the terminal and the data is transmitted. If a N is the response, the next terminal is polled. Polling is done by the program and therefore it controls the transmission of messages into the CPU.

Some terminals are connected to the computing system through a switching network, the same network that telephones use. In this case the person at the terminal dials the computing system phone number. The computer has instructions which enable it, in effect, to answer the phone and have the data sent. Also the computer can signal the terminal to disconnect the phone when it is finished sending data back to the terminal.

In the Disk Operating System there are input/output macros that enable the programmer to write programs that use terminals. These input/output macros are used in the same way as those for tape, cards, printing, and disk were used in Chap. 10 for input/output programming. The input/output support for teleprocessing is another access method which interfaces between the program and the I/O supervisor. There are two levels of I/O support for communication devices. They are the READ/WRITE, or intermediate level, and the GET/PUT, or highest level. These macros are functionally the same as those discussed in Chap. 10. Remember that with the READ/WRITE level macro it's the programmer's responsibility to synchronize the program with the I/O operation and to provide buffering, whereas at the GET/PUT level, these are provided automatically.

In a program using the READ/WRITE level, each communication line is defined with a DTF (Define the File) as each file was for tape or disk. In addition to the DTF a polling list for each line must be established. A polling list is a table that contains the addresses of all the terminals on the communication line in the order they should be polled. When the program issues a read request with the name of the DTF for the communication line it wants, the I/O system polls each terminal using the polling list for the line. If every terminal responds with a N (nothing to send) the read operation is complete and when the program sychronizes itself with the completion of the operation, a code which signifies nothing received will be returned to the problem program. If when polling a line the I/O system receives a X (something to send), it will read the data into a buffer completing the read operation. It is still the program's responsibility to synchronize the completion of the I/O operation with itself before using the data. Since in teleprocessing applications the main goal is responsiveness, it is necessary to read each line frequently. At the READ/WRITE level the scheduling of line reading is the responsibility of the user's program.

When sending a message to a terminal using the READ/WRITE level the program issues a write request and the I/O system will transmit the message when the line is free.

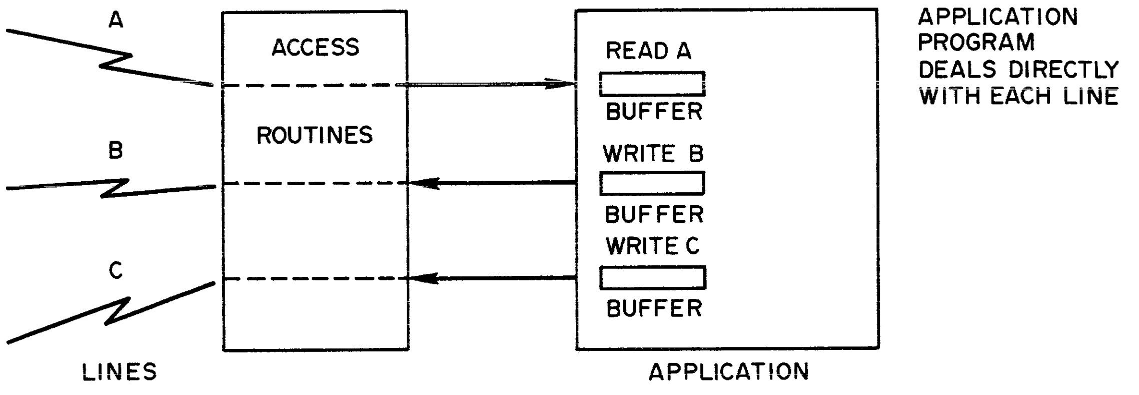
When using the GET/PUT level access method for teleprocessing applications, the operation of the access method and the user's program differ considerably from the READ/WRITE level. Since the GET/PUT level is responsible for line scheduling and sychronization in addition to the other functions there is more information that must be given the access method. This is done by using a set of macros which define: the line and terminal configuration, the format of the header label of the messages, the type of polling to be performed and other optional service desired. This collection of macros is then assembled by the assembler into a line control program. This program is responsible for all the functions associated with sending and receiving messages in the system. By polling the lines in a predetermined pattern it receives messages and stores them in an input queue or an output queue. Which queue is used is determined from the information in the header of the message. If there are any messages in the output queue it sends them to the appropriate destination when the desired line is free. Once the line control program is assembled, these functions are performed without the intervention of the application program.

The messages that are placed in the input queue need further processing. Those that were placed in the output queue by the line control program and will be sent by it are being switched as in the message switching application, by the line control program alone. When an application program wants to process a message in the input queue, it issues a GET macro which presents it with the next message in the queue. If the queue is empty the program waits until a message is placed there by the line control program. When the program wants to send a message it issues a PUT macro which will cause the message to be moved into the output queue. It will be transmitted by the line control program at a later time. The GET/PUT level handles most of the difficult functions in a teleprocessing application but is less flexible than the READ/WRITE level.

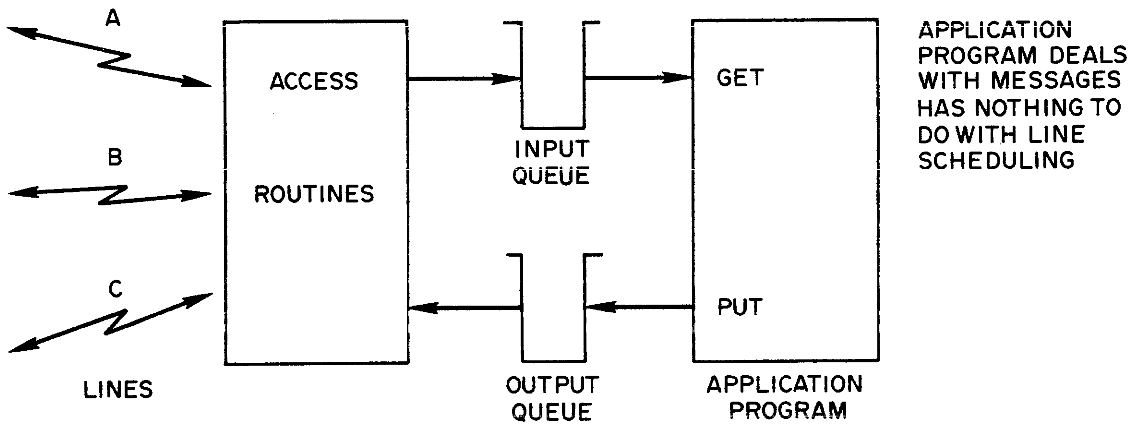
12.6 Spooling

As computers grew in size and speed the associated input/output devices, especially card readers, card punches, and printers could not keep pace as was discussed in Chap. 9. It became uneconomical to have large computing systems read data from a card reader and write output to a printer or punched card. The CPU was idle too much of the time while these operations where in process. It is economical to have the system read and write magnetic tape at speeds more nearly equal to the task than mechanical card readers and printers. However, the data comes in on punched card and the output must be printed or punched to be usable so the information must get to and from tape. Many installations use small computers to do the card to tape, tape to printer and tape to card operations. It is not unusual to have several such small systems supporting a large processing system. The operations performed by the small systems are called peripheral operations because they handled the data at both ends of the operation. These peripheral operations have the characteristics of being input/output bound. In fact in most cases they only use the CPU to issue the I/O commands. With the input/output channel being able to overlap I/O operations with CPU processing, a program with the characteristics of these peripheral programs would use very little CPU processing time and leave it free to work other programs. The technique of performing peripheral operations on the same system concurrently with running the major programs is called spooling. The acronym SPOOL stands for Simultaneous Peripheral Operations On Line. Spooling is a form of simple multiprogramming. In its early implementation, it was a restricted form in that it allowed the spool program to operate with only one other program rather than a number of programs. Usually the peripheral programs are given the CPU when they need it since they only require it to start an input/output operation and relinquish control immediately thereafter. The spool program was usually written as part of the control program and was designed to gain control when any of the I/O devices it was using completed an operation and interrupted processing. It would then determine if another I/O operation could be started and control would subsequently be returned to the interrupted program.

TELEPROCESSING ACCESS METHODS



(a) READ/ WRITE


A
B
C
LINES
ACCESS
ROUTINES
INPUT QUEUE
OUTPUT QUEUE
GET
PUT
APPLICATION PROGRAM
APPLICATION PROGRAM DEALS WITH MESSAGES HAS NOTHING TO DO WITH LINE SCHEDULING

(b) GET/ PUT

Fig. 12-5. Teleprocessing Access Methods.

Spooling enables an installation to have a large self contained system. It usually employs unused I/O devices and idle CPU time and permits a reduction in total system cost by not requiring as many peripheral computing systems. The spool operation usually degrades slightly the execution time of any particular job but permits an increase in total system performance.

12.7 Multiprogramming with the Disk Operating System

The Disk Operating System provides a multiprogramming capability which can be used with intermediate systems. The system allows up to three individual programs to be run concurrently. The storage of the CPU is partitioned into three areas, one for each program to be run concurrently. Each area is assigned a priority and the program that uses a given area has the priority associated with it. The three areas are designated as foreground one (F1), foreground two (F2) and background (B). F1 is the highest priority area and the program running in it has first claim on the CPU, followed by F2 and B, in that order.

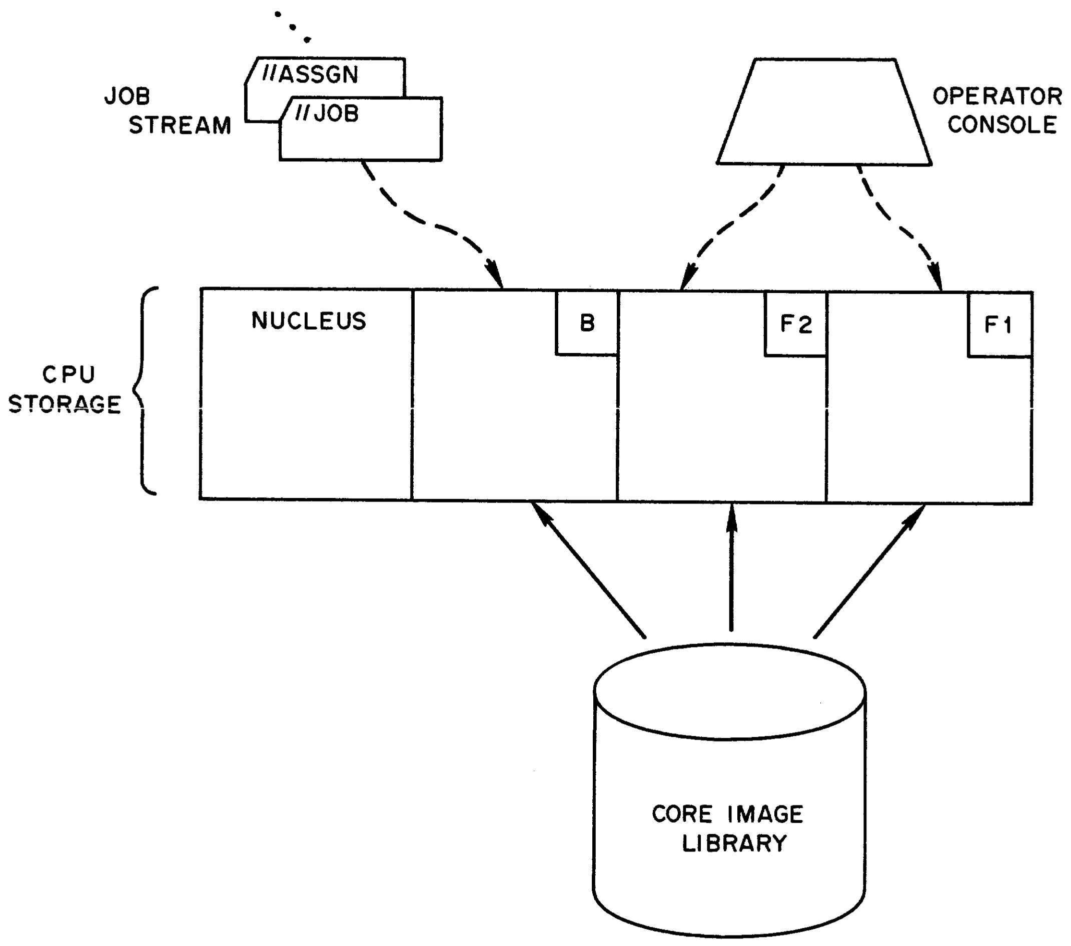
The jobs that are run in the background region are scheduled by job control based on the information in the job stream. This sequential succession of jobs is accomplished just as in a non-multiprogramming system. As one job step completes the next step is scheduled. When a job is complete, the first step of the next job is scheduled. The concept of scheduling the background area does not differ from the previous sequential systems we have discussed. At some point in time the operator may interrupt the system from his console and request a program to be loaded from the core image library into either F1 or F2 if no program is running in that area. The program that is loaded has the CPU priority of the area into which it is loaded. When a program in either foreground area terminates the operator may initiate another program immediately, wait until later, or leave the area unused. Let's summarize the scheduling procedure. The background area is scheduled by job control in conjunction with the job stream. The two foreground areas are scheduled by the operator. Therefore, each area is scheduled independently.

While programs are executing in the three areas, the CPU usage is determined by the priority of the respective programs which in turn is related to the area in which they are running. For instance the program in F1 gets control of the CPU whenever it needs it. If the program in F1 cannot use the CPU at that moment, the program in F2 is given control. When the programs in F1 and F2 cannot make use of the CPU, the program in the background area is given control. However, whenever a program in a higher priority area needs the CPU it will be given control. The programs coexist in storage together and are protected from each other and the control program is protected from all three by the storage protect feature.

The programs which run in the foreground areas are usually the type that must be present in core when needed such as teleprocessing and spooling programs. The programs processed in the background are the usual compiling, testing and production runs. An installation could use the Disk Operating System in a multiprogramming environment in the following manner: In F1, the teleprocessing application could be running. Since the teleprocessing application is usually critical in response time but not a major continuous CPU user, it could gain control of the CPU whenever it needs it and release it when it doesn't. In F2 the operator could schedule a program which does card to tape, tape to card, or tape to printer when those functions are needed. Since these peripheral programs are usually I/O bound they still leave sufficient CPU time to do the normal installation jobs in the background area.

By using multiprogramming an installation can accomplish teleprocessing, spooling and their other jobs on a single system instead of several systems. It must be kept in mind, however, that there must be enough computing power and I/O devices for all three applications. This usually costs less than three separate systems and eases scheduling problems of data interchange between applications, and permits ready use of common subroutines.

MULTIPROGRAMMING STORAGE LAYOUT



---- SOURCE OF CONTROL FOR EACH AREA

——— PROGRAM LOADING

Fig. 12-6

Appendix A

MACHINE-INSTRUCTION MNEMONIC OPERATION CODES

This appendix contains a table of the mnemonic operation codes for all machine instructions that can be represented in assembler language, including extended mnemonic operation codes. It is in alphabetic order by instruction. Indicated for each instruction are both the mnemonic and machine operation codes, explicit and implicit operand formats, program interruptions possible, and condition code set.

The column headings in this appendix and the information each column provides follow.

Instruction: This column contains the name of the instruction associated with the mnemonic operation code.

Mnemonic Operation Code: This column gives the mnemonic operation code for the machine instruction. This is written in the operation field when coding the instruction.

Machine Operation Code: This column contains the hexadecimal equivalent of the actual machine operation code. The operation code will appear in this form in most storage dumps and when displayed on the system control panel. For extended mnemonics, this column also contains the mnemonic code of the instruction from which the extended mnemonic is derived.

Operand Format: This column shows the symbolic format of the operand field in both explicit and implicit form. For both forms, R1, R2, and R3 indicate general registers in operands one, two, and three, respectively. X2 indicates a general register used as an index register in the second operand. Instructions which require an index register (X2) but are not to be indexed are shown with a 0 replacing X2. L, L1, and L2 indicate lengths for either operand, operand one, and operand two, respectively.

For the explicit format, D1 and D2 indicate a displacement and B1 and B2 indicate a base register for operands one and two.

For the implicit format, D1,B1 and D2,B2 are replaced by S1 and S2 which indicate a storage address in operands one and two.

Type of Instruction: This column gives the basic machine format of the instruction (RR, RX, SI, or SS). If an instruction is included in a special feature or is an extended mnemonic, this is also indicated.

Program Interruptions Possible: This column indicates the possible program interruptions for this instruction. The abbreviations used are: A - Addressing, S - Specification, Ov - Overflow, P - Protection, Op - Operation (if feature is not installed) and Other - other interruptions which are listed. The type of overflow is indicated by: D - Decimal, E - Exponent, or F - Floating Point.

Condition Code Set: The condition codes set as a result of this instruction are indicated in this column. (See legend following the table).

Instruction	Mnemonic Operation Code	Machine Operation Code	Operand Format Explicit	Operand Format Implicit
Add	A	5A	R1,D2(X2,B2) or R1, D2(,B2)	R1,S2(X2) or R1,S2
Add	AR	1A	R1,R2	
Add Decimal	AP	FA	D1(L1,B1),D2(L2,B2)	S1(L1),S2(L2)or S1,S2
Add Halfword	AH	4A	R1,D2(X2,B2)or R1,D2(,B2)	R1,S2(X2)or R1,S2
Add Logical	AL	5E	R1,D2(X2,B2)or R1,D2(,B2)	R1,S2(X2)or R1,S2
Add Logical	ALR	1E	R1,R2	
Add Normalized, Long	AD	6A	R1,D2(X2,B2)or R1,D2(,B2)	R1,S2(X2)or R1,S2
Add Normalized, Long	ADR	2A	R1,R2	
Add Normalized, Short	AE	7A	R1,D2(X2,B2)or R1,D2(,B2)	R1,S2(X2)or R1,S2
Add Normalized, Short	AER	3A	R1,R2	
Add Unnormalized, Long	AW	6E	R1,D2(X2,B2)or R1,D2(,B2)	R1,S2(X2)or R1,S2
Add Unnormalized, Long	AWR	2E	R1,R2	
Add Unnormalized, Short	AU	7E	R1,D2(X2,B2)or R1,D2(,B2)	R1,S2(X2)or R1,S2
Add Unnormalized, Short	AUR	3E	R1,R2	
And Logical	N	54	R1,D2(X2,B2)or R1,D2(,B2)	R1,S2(X2)or R1,S2
And Logical	NC	D4	D1(L,B1),D2(B2)	S1(L),S2 or S1,S2
And Logical	NR	14	R1,R2	
And Logical Immediate	NI	94	D1(B1),I2	S1,I2
Branch and Link	BAL	45	R1,D2(X2,B2)or R1,D2(,B2)	R1,S2(X2)or R1,S2
Branch and Link	BALR	05	R1,R2	
Branch on Condition	BC	47	R1,D2(X2,B2)or R1,D2(,B2)	R1,S2(X2)or R1,S2
Branch on Condition	BCR	07	R1,R2	
Branch on Count	BCT	46	R1,D2(X2,B2)or R1,D2(,B2)	R1,S2(X2)or R1,S2
Branch on Count	BCTR	06	R1,R2	
Branch on Equal	BE	47(BC 8)	D2(X2,B2)or D2(,B2)	S2(X2) or S2
Branch on High	BH	47(BC 2)	D2(X2,B2)or D2(,B2)	S2(X2) or S2
Branch on Index High	BXH	86	R1,R3,D2(B2)	R1,R3,S2
Branch on Index Low or Equal	BXLE	87	R1,R3,D2(B2)	R1,R3,S2
Branch on Low	BL	47(BC 4)	D2(X2,B2)or D2(,B2)	S2(X2) or S2
Branch if Mixed	BM	47(BC 4)	D2(X2,B2)or D2(,B2)	S2(X2) or S2
Branch on Minus	BM	47(BC 4)	D2(X2,B2)or D2(,B2)	S2(X2) or S2
Branch on Not Equal	BNE	47(BC 7)	D2(X2,B2)or D2(,B2)	S2(X2) or S2
Branch on Not High	BNH	47(BC 13)	D2(X2,B2)or D2(,B2)	S2(X2) or S2
Branch on Not Low	BNL	47(BC 11)	D2(X2,B2)or D2(,B2)	S2(X2) or S2
Branch if Ones	BO	47(BC 1)	D2(X2,B2)or D2(,B2)	S2(X2) or S2
Branch on Overflow	BO	47(BC 1)	D2(X2,B2)or D2(,B2)	S2(X2) or S2
Branch on Plus	BP	47(BC 2)	D2(X2,B2)or D2(,B2)	S2(X2) or S2
Branch if Zeros	BZ	47(BC 8)	D2(X2,B2)or D2(,B2)	S2(X2) or S2
Branch on Zero	BZ	47(BC 8)	D2(X2,B2)or D2(,B2)	S2(X2) or S2
Branch Unconditional	B	47(BC 15)	D2(X2,B2)or D2(,B2)	S2(X2) or S2
Branch Unconditional	BR	07(BCR 15)	R2	
Compare Algebraic	C	59	R1,D2(X2,B2)or R1,D2(,B2)	R1,S2(X2 or R1,S2
Compare Algebraic	CR	19	R1,R2	
Compare Decimal	CP	F9	D1(L1,B1),D2(L2,B2)	S1(L1),S2(L2)or S1,S2
Compare Halfword	CH	49	R1,D2(X2,B2)or R1,D2(,B2)	R1,S2(X2)or R1,S2
Compare Logical	CL	55	R1,D2(X2,B2)or R1,D2(,B2)	R1,S2(X2)or R1,S2
Compare Logical	CLC	D5	D1(L,B1),D2(B2)	S1(L),S2 or S1,S2
Compare Logical	CLR	15	R1,R2	
Compare Logical Immediate	CLI	95	D1(B1),I2	S1,I2
Compare, Long	CD	69	R1,D2(X2,B2)or R1,D2(,B2)	R1,S2(X2)or R1,S2
Compare, Long	CDR	29	R1,R2	
Compare, Short	CE	79	R1,D2(X2,B2)or R1,D2(,B2)	R1,S2(X2)or R1,S2
Compare, Short	CER	39	R1,R2	
Convert to Binary	CVB	4F	R1,D2(X2,B2)or R1,D2(,B2)	R1,S2(X2)or R1,S2
Convert to Decimal	CVD	4E	R1,D2(X2,B2)or R1,D2(,B2)	R1,S2(X2)or R1,S2

Instruction	Type of Instruction	Program Interruption Possible						Condition Code Set			
		A	S	Ov	P	Op	Other	00	01	10	11
Add	RX	x	x	F				Sum=0	Sum< 0	Sum >0	Overflow
Add	RR			F				Sum=0	Sum< 0	Sum >0	Overflow
Add Decimal	SS, Decimal	x		D	x	x	Data	Sum=0	Sum< 0	Sum >0	Overflow
Add Halfword	RX	x	x	F				Sum=0	Sum <0	Sum >0	Overflow
Add Logical	RX	x	x					Sum=0 Ⓗ	Sum 0 Ⓗ	Sum= 0 Ⓘ	Sum 0 Ⓘ
Add Logical	RR							Sum=0 Ⓗ	Sum= 0 Ⓗ	Sum= 0 Ⓘ	Sum 0 Ⓘ
Add Normalized, Long	RX, Floating Pt.	x	x	E		x	B, C	R	L	M	P
Add Normalized, Long	RR, Floating Pt.		x	E		x	B, C	R	L	M	P
Add Normalized, Short	RX, Floating Pt.	x	x	E		x	B, C	R	L	M	P
Add Normalized, Short	RR, Floating Pt.		x	E		x	B, C	R	L	M	P
Add Unnormalized, Long	RX, Floating Pt.	x	x	E		x	C	R	L	M	P
Add Unnormalized, Long	RR, Floating Pt.		x	E		x	C	R	L	M	P
Add Unnormalized, Short	RX, Floating Pt.	x	x	E		x	C	R	L	M	P
Add Unnormalized, Short	RR, Floating Pt.		x	E		x	C	R	L	M	P
Add Logical	RX	x	x					J	K		
And Logical	SS	x			x			J	K		
And Logical	RR							J	K		
And Logical Immediate	SI	x			x			J	K		
Branch and Link	RX							N	N	N	N
Branch and Link	RR							N	N	N	N
Branch on Condition	RX							N	N	N	N
Branch on Condition	RR							N	N	N	N
Branch on Count	RX							N	N	N	N
Branch on Count	RR							N	N	N	N
Branch on Equal	RX, Ext. Mnemonic							N	N	N	N
Branch on High	RX, Ext. Mnemonic							N	N	N	N
Branch on Index High	RX, Ext. Mnemonic							N	N	N	N
Branch on Index Low or Equal	RX, Ext. Mnemonic							N	N	N	N
Branch on Low	RX, Ext Mnemonic							N	N	N	N
Branch if Mixed	RX, Ext. Mnemonic							N	N	N	N
Branch on Minus	RX, Ext. Mnemonic							N	N	N	N
Branch on Not Equal	RX, Ext. Mnemonic							N	N	N	N
Branch on Not High	RX, Ext. Mnemonic							N	N	N	N
Branch on Not Low	RX, Ext. Mnemonic							N	N	N	N
Branch if Ones	RX, Ext. Mnemonic							N	N	N	N
Branch on Overflow	RX, Ext. Mnemonic							N	N	N	N
Branch on Plus	RX, Ext. Mnemonic							N	N	N	N
Branch if Zeros	RX, Ext. Mnemonic							N	N	N	N
Branch on Zero	RX, Ext. Mnemonic							N	N	N	N
Branch Unconditional	RX, Ext. Mnemonic							N	N	N	N
Branch Unconditional	RR, Ext. Mnemonic							N	N	N	N
Compare Algebraic	RX	x	x					Z	AA	BB	
Compare Algebraic	RR							Z	AA	BB	
Compare Decimal	SS, Decimal	x				x	Data	Z	AA	BB	
Compare Halfword	RX	x	x					Z	AA	BB	
Compare Logical	RX	x	x					Z	AA	BB	
Compare Logical	RX	x	x					Z	AA	BB	
Compare Logical	SS	x						Z	AA	BB	
Compare Logical Immediate	SI	x						Z	AA	BB	
Compare, Long	RX, Floating Pt.	x	x			x		Z	AA	BB	
Compare, Long	RR, Floating Pt.	x	x			x		Z	AA	BB	
Compare, Short	RX, Floating Pt.	x	x			x		Z	AA	BB	
Compare, Short	RR, Floating Pt.		x			x		Z	AA	BB	
Convert to Binary	RX	x	x				Data, F	N	N	N	N
Convert to Decimal	RX	x	x		x			N	N	N	N

Instruction	Mnemonic Operation Code	Machine Operation Code	Operand Format	
			Explicit	Implicit
Divide	D	5D	R1,D2(X2,B2) or R1,D2(,B2)	R1, S2(X2) or R1,S2
Divide	DR	1D	R1,R2	
Divide Decimal	DP	FD	D1,(L1,B1),D2(L2,B2)	S1(L1),S2(L2)or S1,S2
Divide, Long	DD	6D	R1,D2(X2,B2),or R1,D2(,B2)	R1,S2(X2) or R1,S2
Divide, Long	DDR	2D	R1,R2	
Divide, Short	DE	7D	R1,D2(X2,B2)or R1,D2(,B2)	R1,S2(X2) or R1,S2
Divide, Short	DER	3D	R1,R2	
Edit	ED	DE	D1(L,B1),D2(B2)	S1(L),S2 or S1,S2
Edit and Mark	EDMK	DF	D1(L,B1),D2(B2)	S1(L),S2 or S1,S2
Exclusive Or	X	57	R1,D2(X2,B2) or R1,D2(,B2)	R1,S2(X2) or R1,S2
Exclusive Or	XC	D7	D1(L,B1),D2(B2)	S1(L),S2 or S1,S2
Exclusive Or	XR	17	R1,R2	
Exclusive Or Immediate	X1	97	D1(B1),12	S1,12
Execute	EX	44	R1,D2(X2,B2) or R1,D2(,B2)	R1,S2(X2) R1,S2
Halve, Long	HDR	24	R1,R2	
Halve,Short	HER	34	R1,R2	
Halt I/O	HIO	9E	D1(B1)	
Insert Character	IC	43	R1,D2(X2,B2) or R1,D2(,B2)	R1,S2(X2) or R1,S2
Insert Storage Key	ISK	09	R1,R2	
Load	L	58	R1,D2(X2,B2) or R1,D2(,B2)	R1,S2(X2) or R1,S2
Load	LR	18	R1,R2	
Load Address	LA	41	R1,D2(X2,B2) or R1,D2(,B2)	R1,S2(X2) or R1,S2
Load and Test	LTR	12	R1,R2	
Load and Test, Long	LTDR	22	R1,R2	
Load and Test, Short	LTER	32	R1,R2	
Load Complement	LCR	13	R1,R2	
Load Complement, Long	LCDR	23	R1,R2	
Load Complement, Short	LCER	33	R1,R2	
Load Halfword	LH	48	R1,D2(X2,B2) or R1,D2(,B2)	R1,S2(X2) or R1,S2
Load, Long	LD	68	R1,D2(X2,B2) or R1,D2(,B2)	R1,S2(X2) or R1,S2
Load, Long	LDR	28	R1,R2	
Load Multiple	LM	98	R1,R3,D2(B2)	R1,R3,S2
Load Negative	LNR	11	R1,R2	
Load Negative, Long	LNDR	21	R1,R2	
Load Negative, Short	LNER	31	R1,R2	
Load Positive	LPR	10	R1,R2	
Load Positive, Long	LPDR	20	R1,R2	
Load Positive, Short	LPER	30	R1,R2	
Load PSW	LPSW	82	D1(B1)	
Load, Short	LE	78	R1,D2(X2,B2) or R1,D2(,B2)	R1,S2(X2) or R1,S2
Load, Short	LER	38	R1,R2	
Move Characters	MVC	D2	D1(L,B1),D2(B2)	S1(L),S2 or S1,S2
Move Immediate	MVI	92	D1(B1),12	S1,12
Move Numerics	MVN	D1	D1(L,B1),D2(B2)	S1(L),S2 or S1,S2
Move with Offset	MVO	F1	D1(L1,B1),D2(L2,B2)	S1(L1),S2(L2)or S1,S2
Move Zones	MVZ	D3	D1(L,B1),D2(B2)	S1(L),S2 or S1,S2
Multiply	M	5C	R1,D2(X2,B2)or R1,D2(,B2)	R1,S2(X2) or R1,S2
Multiply	MR	1C	R1,R2	
Multiply Decimal	MP	FC	D1(L1,B1),D2(L2,B2)	S1(L1),S2(L2) or S1,S2
Mulitply Halfword	MH	4C	R1,D2(X2,B2) or R1,D2(,B2)	R1,S2(X2) or R1,S2
Multiply, Long	MD	6C	R1,D2(X2,B2) or R1,D2(,B2)	R1,S2(X2) or R1,S2
Multiply, Long	MDR	2C	R1,R2	
Multiply, Short	ME	7C	R1,D2(X2,B2) or R1,D2(,B2)	R1,S2(X2) or R1,S2
Multiply, Short	MER	3C	R1,R2	
No Operation	NOP	47(BC 0)	D2(X2,B2) or D2(,B2)	S2(X2) or S2

Instruction	Type of Instruction	Program Interruptions Possible						Condition Code Set			
		A	S	Ov	P	Op	Other	00	01	10	11
Divide	RX	x	x				F	N	N	N	N
Divide	RR		x				F	N	N	N	N
Divide Decimal	SS, Decimal	x	x		x	x	D, Data	N	N	N	N
Divide, Long	RX,Floating Pt.	x	x	E		x	B,E	N	N	N	N
Divide, Long	RR, Floating Pt.		x	E		x	B,E	N	N	N	N
Divide, Short	RX, Floating Pt.	x	x	E		x	B,E	N	N	N	N
Divide, Short	RR, Floating Pt.		x	E		x	B,E	N	N	N	N
Edit	SS, Decimal	x			x	x	Data	S	T	U	
Edit and Mark	SS, Decimal	x			x	x	Data	S	T	U	
Exclusive Or	RX	x	x					J	K		
Exclusive Or	SS	x			x			J	K		
Exclusive Or	RR							J	K		
Exclusive Or Immediate	SI	x			x			J	K		
Execute	RX	x	x				G	(May be set by this instruction)			
Halve, Long	RR, Floating Pt.		x			x		N	N	N	N
Halve, Short	RR, Floating Pt.		x			x		N	N	N	N
Halt I/O	SI						A	DD	CC	GG	KK
Insert Character	RX	x						N	N	N	N
Insert Storage Key	RR	x	x			x	A	N	N	N	N
Load	RX	x	x					N	N	N	N
Load	RR							N	N	N	N
Load Address	RX							N	N	N	N
Load and Test	RR							J	L	M	
Load and Test, Long	RR, Floating Pt.		x			x		R	L	M	
Load and Test, Short	RR, Floating Pt.		x			x		R	L	M	
Load Complement	RR			F				P	L	M	O
Load Complement, Long	RR, Floating Pt.		x			x		R	L	M	
Load Complement, Short	RR, Floating Pt.		x			x		R	L	M	
Load Halfword	RX	x	x					N	N	N	N
Load, Long	RX, Floating Pt.	x	x			x		N	N	N	N
Load, Long	RR, Floating Pt.		x			x		N	N	N	N
Load Multiple	RS	x	x					N	N	N	N
Load Negative	RR							J	L		
Load Negative, Long	RR, Floating Pt.		x			x		R	L		
Load Negative, Short	RR, Floating Pt.		x			x		R	L		
Load Positive	RR			F				J		M	O
Load Positive, Long	RR, Floating Pt.		x			x		R	L	M	
Load Positive, Short	RR, Floating Pt.		x			x		R	L	M	
Load PSW	SI	x	x				A	QQ	QQ	QQ	QQ
Load, Short	RX, Floating Pt.	x	x			x		N	N	N	N
Load, Short	RR, Floating Pt.		x			x		N	N	N	N
Move Characters	SS	x			x			N	N	N	N
Move Immediate	SI	x			x			N	N	N	N
Move Numerics	SS	x			x			N	N	N	N
Move with Offset	SS	x			x			N	N	N	N
Move Zones	SS	x			x			N	N	N	N
Multiply	RX	x	x					N	N	N	N
Multiply	RR		x					N	N	N	N
Multiply Decimal	SS, Decimal	x	x		x	x	Data	N	N	N	N
Multiply Halfword	RX	x	x					N	N	N	N
Multiply, Long	RX, Floating Pt.	x	x	E		x	B	N	N	N	N
Multiply, Long	RR, Floating Pt.		x	E		x	B	N	N	N	N
Multiply, Short	RX, Floating Pt.	x	x	E		x	B	N	N	N	N
Multiply, Short	RR, Floating Pt.		x	E		x	B	N	N	N	N
No Operation	RX, Ext.Mnemonic							N	N	N	N

Instruction	Mnemonic Operation Code	Machine Operation Code	Operand Format	
			Explicit	Implicit
No Operation	NOPR	07(BCR 0)	R2	
Or Logical	O	56	R1,D2(X2,B2) or R1,D2(,B2)	R1,S2(X2) or R1,S2
Or Logical	OC	D6	D1(L,B1),D2(B2)	S1(L),S2 or S1,S2
Or Logical	OR	16	R1,R2	
Or Logical Immediate	OI	96	D1(B1),I2	S1,I2
Pack	PACK	F2	D1(L1,B1),D2(L2,B2)	S1(L1),S2(L2) or S1,S2
Read Direct	RDD	85	D1(B1),I2	S1,I2
Set Program Mask	SPM	04	R1	
Set System Key	SSK	08	R1,R2	
Set System Mask	SSM	80	D1(B1)	S1
Shift Left Double Algebraic	SLDA	8F	R1,D2(B2)	R1,S2
Shift Left Double Logical	SLDL	8D	R1,D2(B2)	R1,S2
Shift Left Single Algebraic	SLA	8B	R1,D2(B2)	R1,S2
Shift Left Single Logical	SLL	89	R1,D2(B2)	R1,S2
Shift Right Double Algebraic	SRDA	8E	R1,D2(B2)	R1,S2
Shift Right Double Logical	SRDL	8C	R1,D2(B2)	R1,S2
Shift Right Single Algebraic	SRA	8A	R1,D2(B2)	R1,S2
Shift Right Single Logical	SRL	88	R1,D2(B2)	R1,S2
Start I/O	SIO	9C	D1(B1)	S1
Store	ST	50	R1,D2(X2,B2) or R1,D2(,B2)	R1,S2(X2) or R1,S2
Store Character	STC	42	R1,D2(X2,B2) or R1,D2(,B2)	R1,D2(X2) or R1,S2
Store Halfword	STH	40	R1,D2(X2,B2) or R1,D2(,B2)	R1,S2(X2) or R1,S2
Store Long	STD	60	R1,D2(X2,B2)	R1,S2(X2) or R1,S2
Store Multiple	STM	90	R1,R2,D2(B2)	R1,R2,S2
Store Short	STE	70	R1,D2(X2,B2) or R1,D2(,B2)	R1,S2(X2) or R1,S2
Subtract	S	5B	R1,D2(X2	R1,S2(X2) or R1,S2
Subtract	SR	1B	R1,R2	
Subtract Decimal	SP	FB	D1(L1,B1),D2(L2,B2)	S1(L1),S2(L2) or S1,S2
Subtract Halfword	SH	4B	R1,D2(X2,B2) or R1,D2(,B2)	R1,S2(X2) or R1,S2
Subtract Logical	SL	5F	R1,D2(X2,B2) or R1,D2(,B2)	R1,S2(X2) or R1,S2
Subtract Logical	SLR	1F	R1,R2	
Subtract Normalized, Long	SD	6B	R1,D2(X2,B2) or R1,D2(,B2)	R1,S2(X2) or R1,S2
Subtract Normalized, Long	SDR	2B	R1,R2	
Subtract Normalized, Short	SE	7B	R1,D2(X2,B2) or R1,D2(,B2)	R1,S2(X2) or R1,S2
Subtract Normalized,	SER	3B	R1,R2	
Subtract Unnormalized, Long	SW	6F	R1,D2(X2,B2) or R1,D2(,B2)	R1,S2(X2) or R1,S2
Subtract Unnormalized, Long	SWR	2F	R1,R2	
Subtract Unnormalized, Short	SU	7F	R1,D2(X2,B2) or R1, D2(,B2)	R1,S2(X2) or R1,S2
Subtract Unnormalized, Short	SUR	3F	R1,R2	
Supervisor Call	SVC	0A	I	
Test and Set	TS	93	D1(B1)	S1
Test Channel	TCH	9F	D1(B1)	S1
Test I/O	TIO	9D	D1(B1)	S1
Test Under Mask	TM	91	D1(B1),I2	S1,I2
Translate	TR	DC	D1(L,B1),D2(B2)	S1(L),S2 orS1,S2
Translate and Test	TRT	DD	D1(L,B1),D2(B2)	S1(L),S2 orS1,S2
Unpack	UNPK	F3	D1(L1,B1),D2(L2,B2)	S1(L1),S2(L2)or S1,S2
Write Direct	WRD	84	D1(B1),I2	S1,I2
Zero and Add Decimal	ZAP	F8	D1(L1,B1),D2(L2,B2)	S1(L1),S2(L2)or S1,S2

Instruction	Type of Instruction	Program Interruptions Possible						Condition Code Set			
		A	S	Ov	P	Op	Other	00	01	10	11
No Operation	RR, Ext. Mnemonic							N	N	N	N
Or Logical	RX	x	x					J	K		
Or Logical	SS	x			x			J	K		
Or Logical	RR							J	K		
Or Logical Immediate	SI	x			x			J	K		
Pack	SS	x			x			N	N	N	N
Read Direct	SI	x			x	x	A	N	N	N	N
Set Program Mask	RR							RR	RR	RR	RR
Set Storage Key	RR	x	x			x	A	N	N	N	N
Set System Mask	SI	x					A	N	N	N	N
Shift Left Double Algebraic	RS		x	F				J	L	M	O
Shift Left Double Logical	RS		x					N	N	N	N
Shift Left Single Algebraic	RS			F				J	L	M	O
Shift Left Single Logical	RS							N	N	N	N
Shift Right Double Algebraic	RS		x					J	L	M	
Shift Right Double Logical	RS		x					N	N	N	N
Shift Right Single Algebraic	RS							J	L	M	
Shift Right Single Logical	RS							N	N	N	N
Start I/O	SI						A	MM	CC	EE	AA
Store	RX	x	x		x			N	N	N	N
Store Character	RX	x			x			N	N	N	N
Store Halfword	RX	x	x		x			N	N	N	N
Store Long	RX, Floating Pt.	x	x		x	x		N	N	N	N
Store Multiple	RS	x	x		x			N	N	N	N
Store Short	RX, Floating Pt.	x	x		x	x		N	N	N	N
Subtract	RX	x	x	F				V	X	Y	O
Subtract	RR			F				V	X	Y	O
Subtract Decimal	SS, Decimal	x		D	x	x	Data	V	X	Y	O
Subtract Halfword	RX	x	x	F				V	X	Y	O
Subtract Logical	RX	x	x						W,H	V,I	W,I
Subtract Logical	RR								W,H	V,I	W,I
Subtract Normalized, Long	RX, Floating Pt.	x	x	E		x	B,C	R	L	M	Q
Subtract Normalized, Long	RR, Floating Pt.		x	E		x	B,C	R	L	M	Q
Subtract Normalized, Short	RX, Floating Pt.	x	x	E		x	B,C	R	L	M	Q
Subtract Normalized, Short	RR, Floating Pt.		x	E		x	B,C	R	L	M	Q
Subtract Unnormalized, Long	RX, Floating Pt.	x	x	E		x	C	R	L	M	Q
Subtract Unnormalized, Long	RR, Floating Pt.		x	E		x	C	R	L	M	Q
Subtract Unnormalized, Short	RX, Floating Pt.	x	x	E		x	C	R	L	M	Q
Subtract Unnormalized, Short	RR, Floating Pt.		x	E		x	C	R	L	M	Q
Supervisor Call	RR							N	N	N	N
Test and Set	SI	x			x			SS	TT		
Test Channel	SI						A	JJ	II	FF	HH
Test I/O	SI						A	LL	CC	EE	KK
Test Under Mask	SI	x						UU	VV		WW
Translate	SS	x			x			N	N	N	N
Translate and Test	SS	x						PP	NN	OO	
Unpack	SS	x			x			N	N	N	N
Write Direct	SI	x				x	A	N	N	N	N
Zero and Add Decimal	SS, Decimal	x		D	x	x	Data	J	L	M	O

Program Interruptions Possible

Under Ov:

D = Decimal	
E = Exponent	
F = Fixed Point	

Under Other:

A	Privileged Operation
B	Exponent Underflow
C	Significance
D	Decimal Divide
E	Floating Point Divide
F	Fixed Point Divide
G	Execute

Condition Code Set

H	No Carry
I	Carry
J	Result = 0
K	Result is Not Equal to Zero
L	Result is Less Than Zero
M	Result is Greater Than Zero
N	Not Changed
O	Overflow
P	Result Exponent Underflows
Q	Result Exponent Overflows
R	Result Fraction = 0
S	Result Field Equals Zero
T	Result Field is Less Than Zero
U	Result Field is Greater Than Zero
V	Difference = 0
W	Difference is Not Equal to Zero
X	Difference is Less Than Zero
Y	Difference is Greater Than Zero
Z	First Operand Equals Second Operand
AA	First Operand is Less Than Second Operand
BB	First Operand is Greater Than Second Operand
CC	CSW Stored
DD	Channel and Subchannel not Working
EE	Channel or Subchannel Busy
FF	Channel Operating in Burst Mode
GG	Burst Operation Terminated
HH	Channel Not Operational
II	Interruption Pending in Channel
JJ	Channel Available
KK	Not Operational
LL	Available
MM	I/O Operation Initiated and Channel Proceeding With its Execution
NN	Nonzero Function Byte Found Before the First Operand Field is Exhausted
OO	Last Function Byte is Nonzero
PP	All Function Bytes Are Zero
QQ	Set According to Bits 34 and 35 of the New PSW Loaded
RR	Set According to Bits 2 and 3 of the Register Specified by R1
SS	Leftmost Bit of Byte Specified = 0
TT	Leftmost Bit of Byte Specified = 1
UU	Selected Bits Are All Zeros; Mask is All Zeros
VV	Selected Bits Are Mixed (zeros and ones)
WW	Selected Bits Are All Ones

Appendix B

SUMMARY OF CONSTANTS

TYPE	IMPLIED LENGTH (BYTES)	ALIGN-MENT	LENGTH MODI-FIER RANGE	SPECIFIED BY	NUMBER OF CON-STANTS PER OPERAND	RANGE FOR EX-PONENTS	RANGE FOR SCALE	TRUN-CATION/ PADDING SIDE
C	as needed	byte	1 to 256	characters	one			right
X	as needed	byte	1 to 256	hexadecimal digits	one			left
B	as needed	byte	1 to 256	binary digits	one			left
F	4	word	1 to 8	decimal digits	multi-ple	-85 to +75	-187 to +346	left
H	2	half word	1 to 8	decimal digits	multi-ple	-85 to +75	-187 to +346	left
E	4	word	1 to 8	decimal digits	multi-ple	-85 to +75	0-13	right
D	8	double word	1 to 8	decimal digits	multi-ple	-85 to +75	0-13	right
P	as needed	byte	1 to 16	decimal digits	multi-ple			left
Z	as needed	byte	1 to 16	decimal digits	multi-ple			left
A	4	word	1 to 4	any expression	one			left
V	4	word	3 or 4	relocatable symbol	one			left
S	2	half word	2 only	one absolute or relocatab-le expression or two absol-ute express-ions: exp(exp)	one			
Y	2	half word	1 to 4	any expression	one			left

Appendix C

POWERS OF TWO TABLE

2^n	n	2^{-n}
1	0	1.0
2	1	0.5
4	2	0.25
8	3	0.125
16	4	0.062 5
32	5	0.031 25
64	6	0.015 625
128	7	0.007 812 5
256	8	0.003 906 25
512	9	0.001 953 125
1 024	10	0.000 976 562 5
2 048	11	0.000 488 281 25
4 096	12	0.000 244 140 625
8 192	13	0.000 122 070 312 5
16 384	14	0.000 061 035 156 25
32 768	15	0.000 030 517 578 125
65 536	16	0.000 015 258 789 062 5
131 072	17	0.000 007 629 394 531 25
262 144	18	0.000 003 814 697 265 625
524 288	19	0.000 001 907 348 632 812 5
1 048 576	20	0.000 000 953 674 316 406 25
2 097 152	21	0.000 000 476 837 158 203 125
4 194 304	22	0.000 000 238 418 579 101 562 5
8 388 608	23	0.000 000 119 209 289 550 781 25
16 777 216	24	0.000 000 059 604 644 775 390 625
33 554 432	25	0.000 000 029 802 322 387 695 312 5
67 108 864	26	0.000 000 014 901 161 193 847 656 25
134 217 728	27	0.000 000 007 450 580 596 923 828 125
268 435 456	28	0.000 000 003 725 290 298 461 914 062 5
536 870 912	29	0.000 000 001 862 645 149 230 957 031 25
1 073 741 824	30	0.000 000 000 931 322 574 615 478 515 625
2 147 483 648	31	0.000 000 000 465 661 287 307 739 257 812 5
4 294 967 296	32	0.000 000 000 232 830 643 653 869 628 906 25
8 589 934 592	33	0.000 000 000 116 415 321 826 934 814 453 125
17 179 869 184	34	0.000 000 000 058 207 660 913 467 407 226 562 5
34 359 738 368	35	0.000 000 000 029 103 830 456 733 703 613 281 25

INDEX